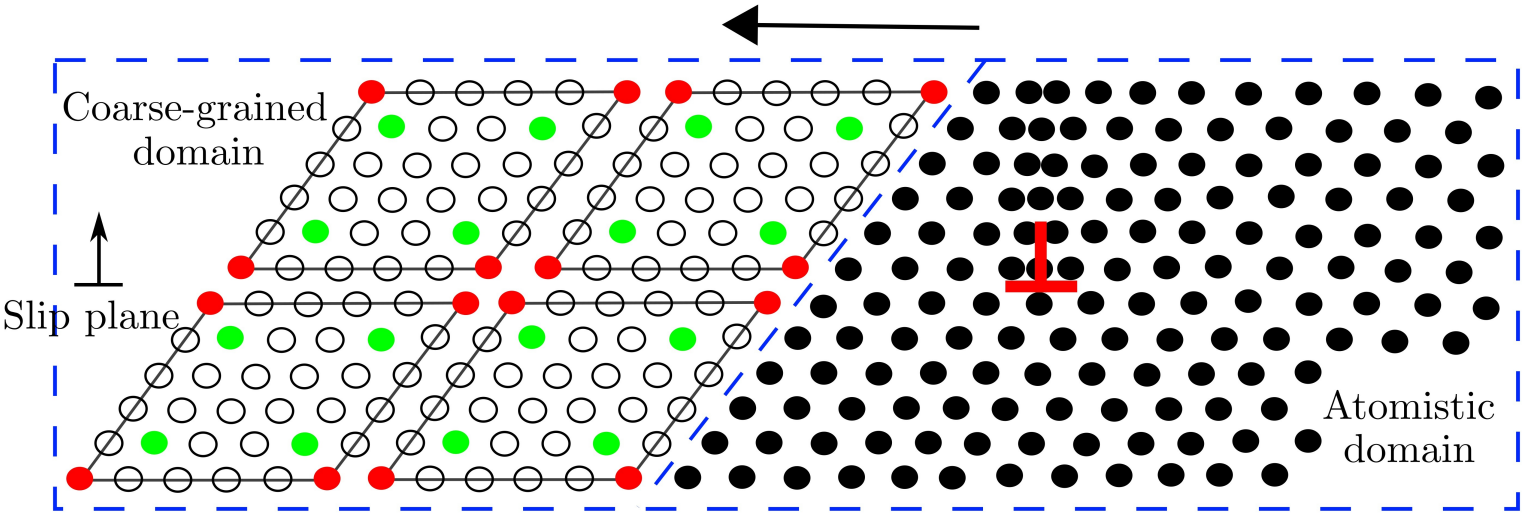
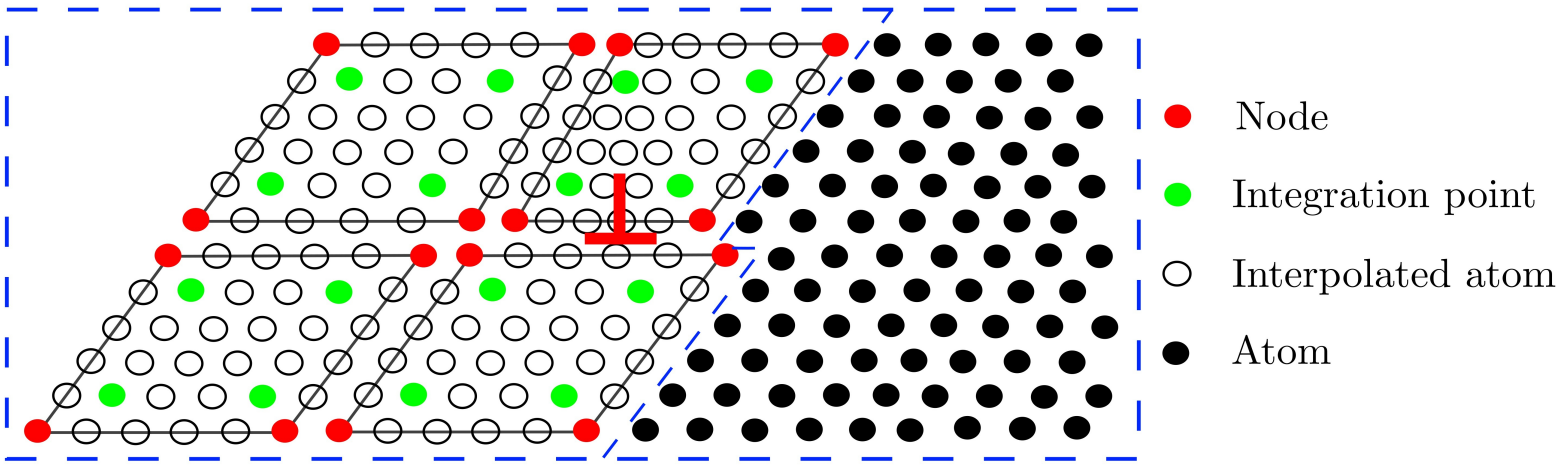
The Python-scripted Concurrent Atomistic-Continuum Simulator (PyCAC)

USER'S MANUAL

version 1.0



(a)



(b)

from Xu, Che, Xiong, Chen, McDowell. *Int. J. Plast.* 72 (2015) 91–126

Table of Contents

PyCAC User's Manual

June 14 2017 version

The PyCAC code, mainly written in Fortran and wrapped with a Python script, is designed to carry out concurrent atomistic-continuum (CAC) simulations.

A pdf version of this manual can be downloaded here.

Introduction

The concurrent atomistic-continuum (CAC) method is a partitioned-domain multiscale modeling technique that is applicable to nano/micron scale thermo/mechanical problems in a wide range of monoatomic and polyatomic crystalline materials. A CAC simulation model, in general, partitions the simulation cell into two domains: atomistic and coarse-grained domains. Differing from most concurrent multiscale methods in the literature, CAC employs a unified atomistic-continuum integral formulation with elements that have discontinuities between them and the underlying interatomic potential as the only constitutive relation in the system. As such, CAC admits propagation of displacement discontinuities (dislocations and associated intrinsic stacking faults) through a lattice in both atomistic and coarse-grained domains.

Differing from most concurrent multiscale materials modeling methods in the literature, CAC

- describes certain lattice defects and their interactions using fully resolved atomistics;
- preserves the net Burgers vector and associated long range stress fields of curved, mixed character dislocations in a sufficiently large continuum domain in a fully 3D model;
- employs the same governing equations and interatomic potentials in both domains to avoid the usage of phenomenological parameters, essential remeshing operations and *ad hoc* procedures for passing dislocation segments between atomistic and coarse-grained atomistic domains.

PyCAC features and non-features

Features

The PyCAC code can simulate thermo/mechanical problems in pure face-centered cubic (FCC) and body-centered cubic (BCC) metals using the Lennard-Jones (LJ) and embedded-atom method (EAM) potentials. In the coarse-grained domain, rhombohedral elements are employed to accommondate dislocations in 9 out of 12 sets of $\{111\}\langle 110\rangle$ slip systems in an FCC lattice, as well as 6 out of 12 sets of $\{110\}\langle 111\rangle$ slip systems in a BCC lattice.

Non-features

While the CAC method is applicable to thermo/mechanical problems in almost all crystalline materials, the current PyCAC code cannot simulate:

- dislocations in 12 sets of $\{112\}\langle 111\rangle$-type and 24 sets of $\{123\}\langle 111\rangle$-type slip systems in a BCC lattice;
- crystal structures other than FCC and BCC, e.g., simple cubic, diamond cubic, hexagonal close-packed;
- interatomic potentials other than LJ and EAM, e.g., Stillinger-Weber potential, Tersoff potential, modified EAM (MEAM) potential;
- multicomponent or multiphase materials, e.g., alloys, intermetallics;
- polyatomic crystalline materials, i.e., ceramic, mineral;
- adaptive mesh refinement.

Compilation and execution

MPI

The PyCAC code is fully parallelized with Message Passing Interface (MPI). Some functions in MPI-3 standard is provided. It works with Open MPI version 2.1, Intel MPI version 5.1, MPICH version 3.3, and MVAPICH2 version 2.3.

Compiler

Some intrinsic functions in Fortran 2003 is employed in the code, so compilers that fully support Fortran 2013 are preferred. For example, GNU Fortran version 7.0 and Intel Fortran version 17.0 work with the PyCAC code.

Module

In compilation, the first step is to create a static library `libcac.a` from the 54 module files `*_module.f90` in the `module` directory. There are five types of module files:

```
*_comm_module.f90
```

There is only one `*_comm_module.f90` file: `precision_comm_module.f90`. It controls the precision of integer and real numbers.

```
*_para_module.f90
```

There are 24 `*_para_module.f90` files. They define single value variables that can be used globally.

```
*_array_module.f90
```

There are 23 `*_array_module.f90` files. They define arrays that can be used globally. With a few exceptions, the `*_para_module.f90` and `*_array_module.f90` files come in pairs.

```
*_function_module.f90
```

There are 5 `*_function_module.f90` files. They define interatomic potential formulations, arithmetic/linear algerala calculations, unit conversion, etc.

```
*_tab_module.f90
```

There is only one `*_tab_module.f90` file: `eam_tab_module.f90`. It contains algorithms that extract EAM potential-based values from numerical tables.

Note that these module files should be compiled in this order, e.g., see the `install.sh` file, in creating the static library `libcac.a`. Otherwise, an error may be reported.

Subroutine

Then, an executale, named `CAC`, is compiled using one main program (`main.f90`) plus 171 subroutines (`*.f90`) in the `src` directory and linked with the static library.

In execution, the executable `CAC`, the input file `cac.in`, and the potential files are moved into the same directory. It follows that

```
mpirun -np num_of_proc ./CAC < cac.in
```

where `num_of_proc` is the number of processors to be used.

The users may run the PyCAC code on the MATerials Innovation Network (MATIN) at Georgia Tech when it is ready.

CAC Publications

Book chapters

1. Shengfeng Yang, Youping Chen, Concurrent atomistic-continuum simulation of defects in polyatomic ionic materials, in *Multiscale Materials Modeling for Nanomechanics* (ed: Christopher R. Weinberger, Garritt J. Tucker), Switzerland: Springer International Publishing, 2016
2. Y. P. Chen, J. D. Lee, Y. J. Lei, L. M. Xiong, A multiscale field theory: Nano/micro materials, in *Multiscaling in Molecular and Continuum Mechanics: Interaction of Time and Size from Macro to Nano* (ed: G. C. Sih), Netherlands: Springer, 2007

Dissertations and theses

1. Shuozhi Xu, The concurrent atomistic-continuum method: Advancements and applications in plasticity of face-centered cubic metals, *Ph.D. Dissertation*, Georgia Institute of Technology, 2016
2. Xiang Chen, A concurrent atomistic-continuum study of phonon transport in crystalline materials with microstructures, *Ph.D. Dissertation*, University of Florida, 2016
3. Shengfeng Yang, A concurrent atomistic-continuum method for simulating defects in ionic materials, *Ph.D. Dissertation*, University of Florida, 2014
4. Qian Deng, Coarse-graining atomistic dynamics of fracture by finite element method: Formulation, parallelization and applications, *Ph.D. Dissertation*, University of Florida, 2011
5. Liming Xiong, A concurrent atomistic-continuum methodology and its applications, *Ph.D. Dissertation*, University of Florida, 2011

Peer-reviewed journal articles on CAC simulations

(by acceptance date)

1. Xiang Chen, Liming Xiong, David L. McDowell, Youping Chen. Effects of phonons on mobility of dislocations and dislocation arrays, *Scr. Mater.* 137 (2017) 22-26
2. Shuozhi Xu, Liming Xiong, Youping Chen, David L. McDowell. Validation of the concurrent atomistic-continuum method on screw dislocation/stacking fault interactions,

Crystals 7 (2017) 120

3. Shuozhi Xu, Liming Xiong, Youping Chen, David L. McDowell. Comparing EAM potentials to model slip transfer of sequential mixed character dislocations across two symmetric tilt grain boundaries in Ni, *JOM* 69 (2017) 814-821
4. Shuozhi Xu, Liming Xiong, Youping Chen, David L. McDowell. Shear stress- and line length-dependent screw dislocation cross-slip in FCC Ni, *Acta Mater.* 122 (2017) 412-419
5. Shuozhi Xu, Liming Xiong, Youping Chen, David L. McDowell. An analysis of key characteristics of the Frank-Read source process in FCC metals, *J. Mech. Phys. Solids* 96 (2016) 460-476
6. Shuozhi Xu, Liming Xiong, Youping Chen, David L. McDowell. Edge dislocations bowing out from a row of collinear obstacles in Al, *Scr. Mater.* 123 (2016) 135-139
7. Shuozhi Xu, Liming Xiong, Qian Deng, David L. McDowell. Mesh refinement schemes for the concurrent atomistic-continuum method, *Int. J. Solids Struct.* 90 (2016) 144-152
8. Shuozhi Xu, Liming Xiong, Youping Chen, David L. McDowell. Sequential slip transfer of mixed character dislocations across $\Sigma 3$ coherent twin boundary in FCC metals: A concurrent atomistic-continuum study, *npj Comput. Mater.* 2 (2016) 15016
9. Liming Xiong, Ji Rigelesaiyin, Xiang Chen, Shuozhi Xu, David L. McDowell, Youping Chen. Coarse-grained elastodynamics of fast moving dislocations, *Acta Mater.* 104 (2016) 143-155
10. Shengfeng Yang, Ning Zhang, Youping Chen. Concurrent atomistic-continuum simulation of polycrystalline strontium titanate, *Philos. Mag.* 95 (2015) 2697-2716
11. Shuozhi Xu, Rui Che, Liming Xiong, Youping Chen, David L. McDowell. A quasistatic implementation of the concurrent atomistic-continuum method for FCC crystals, *Int. J. Plast.* 72 (2015) 91–126
12. Shengfeng Yang, Youping Chen. Concurrent atomistic and continuum simulation of bi-crystal strontium titanate with tilt grain boundary, *Proc. R. Soc. A* 471 (2015) 20140758
13. Liming Xiong, Shuozhi Xu, David L. McDowell, Youping Chen. Concurrent atomistic-continuum simulations of dislocation-void interactions in fcc crystals, *Int. J. Plast.* 65 (2015) 33-42
14. Liming Xiong, Xiang Chen, Ning Zhang, David L. McDowell, Youping Chen. Prediction of phonon properties of 1D polyatomic systems using concurrent atomistic-continuum simulation, *Arch. Appl. Mech.* 84 (2014) 1665-1675
15. Liming Xiong, David L. McDowell, Youping Chen. Sub-THz Phonon drag on dislocations by coarse-grained atomistic simulations, *Int. J. Plast.* 55 (2014) 268-278

16. Qian Deng, Youping Chen, A coarse-grained atomistic method for 3D dynamic fracture simulation, *J. Multiscale Comput. Eng.* 11 (2013) 227-237
17. Shengfeng Yang, Liming Xiong, Qian Deng, Youping Chen. Concurrent atomistic and continuum simulation of strontium titanate, *Acta Mater.* 61 (2013) 89–102
18. Liming Xiong, David L. McDowell, Youping Chen. Nucleation and growth of dislocation loops in Cu, Al and Si by a concurrent atomistic-continuum method, *Scr. Mater.* 67 (2012) 633–636
19. Liming Xiong, Qian Deng, Garritt Tucker, David L. McDowell, Youping Chen. Coarse-grained atomistic simulations of dislocations in Al, Ni and Cu crystals, *Int. J. Plast.* 38 (2012) 86–101
20. Liming Xiong, Qian Deng, Garritt Tucker, David L. McDowell, Youping Chen. A concurrent scheme for passing dislocations from atomistic to continuum domains, *Acta Mater.* 60 (2012) 899-913
21. Liming Xiong, Garritt Tucker, David L. McDowell, Youping Chen. Coarse-grained atomistic simulation of dislocations, *J. Mech. Phys. Solids* 59 (2011) 160-177
22. Qian Deng, Liming Xiong, Youping Chen. Coarse-graining atomistic dynamics of brittle fracture by finite element method, *Int. J. Plast.* 26 (2010) 1402-1414

Peer-reviewed journal articles on the theoretical foundations of CAC

(by acceptance date):

1. Youping Chen, Jonathan Zimmerman, Anton Krivtsov, David L. McDowell. Assessment of atomistic coarse-graining methods, *Int. J. Eng. Sci.* 49 (2011) 1337-1349
2. Youping Chen. Reformulation of microscopic balance equations for multiscale materials modeling, *J. Chem. Phys.* 130 (2009) 134706
3. Youping Chen. Local stress and heat flux in atomistic systems involving three-body forces, *J. Chem. Phys.* 124 (2006) 054113
4. Youping Chen, James Lee. Conservation laws at nano/micro scales, *J. Mech. Mater. Struct.* 1 (2006) 681-704
5. Youping Chen, James Lee, Liming Xiong. Stresses and strains at nano/micro scales, *J. Mech. Mater. Struct.* 1 (2006) 705-723

Acknowledgements and citations

PyCAC development has been funded by

- National Science Foundation
  - Georgia Institute of Technology, CMMI-1232878
  - University of Florida, CMMI-1233113
  - Iowa State University, CMMI-1536925
- Department of Energy, Office of Basic Energy Sciences
  - University of Florida, DE-SC0006539

If you use PyCAC results in your published work, please cite these papers

- Shuozhi Xu, Rui Che, Liming Xiong, Youping Chen, David L. McDowell. A quasistatic implementation of the concurrent atomistic-continuum method for FCC crystals, *Int. J. Plast.* 72 (2015) 91–126
- Liming Xiong, Garritt Tucker, David L. McDowell, Youping Chen. Coarse-grained atomistic simulation of dislocations, *J. Mech. Phys. Solids* 59 (2011) 160-177

as well as the website:

This user's manual is maintained by

- Shuozhi Xu (shuozhixu@gatech.edu)
- Thomas G. Payne (thomas.payne@gatech.edu)

Background

The theoretical foundation of CAC is the atomic field theory (AFT) proposed by Prof. Youping Chen and Prof. James D. Lee. AFT is based on the micromorphic field theory which, along with microstretch and micropolar field theory, begins to the more general microcontinuum field theory, a.k.a., generalized continuum theory, pioneered by numerous mechanicians, among whom the late Prof. Ahmed Cemal Eringen, who was Prof. James D. Lee's Ph.D. advisor.

Atomic field theory

In AFT, a crystalline material is viewed as a continuous collection of lattice points; embedded within each point is a unit cell with a group of discrete atoms. In this way, the micromorphic theory is connected with molecular dynamics and is expanded to the atomic scale. Here, the local density function is continuous at the level of the unit cell, but discrete in terms of the discrete atoms inside the unit cell. The AFT was originally designed with polyatomic crystalline materials in mind, but can also been applied to monoatomic crystals.

For more information, read papers first-authored by Prof. Youping Chen.

Governing equations of the CAC method

In Eulerian coordinates, the governing equations of the CAC method for a monoatomic crystal are

$$\frac{\partial \rho}{\partial t} + \nabla_{\mathbf{r}} \cdot (\rho \mathbf{v}) = 0$$

$$\frac{\partial (\rho \mathbf{v})}{\partial t} - \nabla_{\mathbf{r}} \cdot (\mathbf{t} - \rho \mathbf{v} \otimes \mathbf{v}) - \mathbf{f}_{\text{ext}} = \mathbf{0}$$

$$\frac{\partial (\rho e)}{\partial t} - \nabla_{\mathbf{r}} \cdot (\mathbf{q} + \mathbf{t} \cdot \mathbf{v} - \rho e \mathbf{v}) - \mathbf{f}_{\text{ext}} \cdot \mathbf{v} = 0$$

where ρ is the microscopic local mass density, t is the time, $\mathbf{r}$ is the physical space coordinates, $\mathbf{v}$ is the local velocity field, $\mathbf{f}_{\text{ext}}$ is the external body force field, $\mathbf{t}$ is the 2^{nd} rank momentum flux tensor, e is the energy, and $\mathbf{q}$ is the heat flux. Assuming that the temperature gradient is negligible and there is no external force density, the governing equations in CAC is reduced to

$$\rho \ddot{\mathbf{r}} - \mathbf{f}_{\text{int}} = \mathbf{0}$$

where $\mathbf{f}_{\text{int}}$ is the internal force density and the superposed dots denote the material time second derivative. The last equation is solved directly in dynamic CAC while is used to derive the equivalent nodal force/energy in quasistatic CAC.

For more information, read chapter 2 of Shuozhi Xu's Ph.D. dissertation.

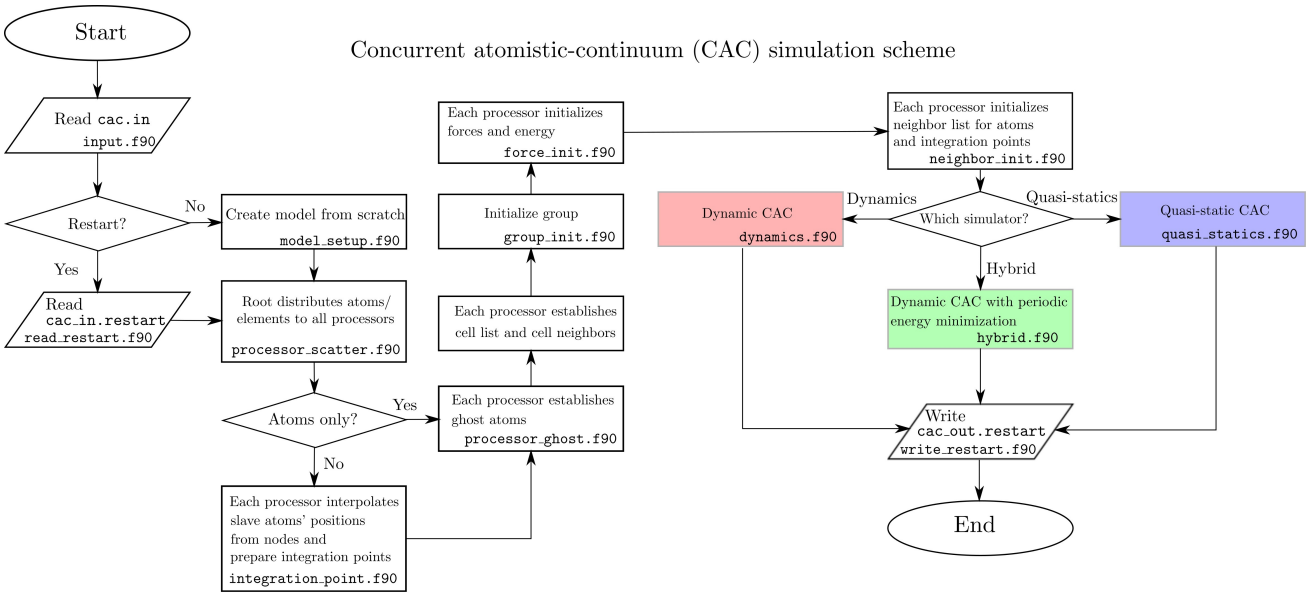
Algorithm

Due to the similarity between CAC and atomistic simulations regarding lattice structure and force/energy calculations, the CAC algorithm adopts common atomistic techniques. Newton's third law is employed in the atomistic domain to promote efficiency in calculating the force, pair potential, local electron density, and stress. The short-range neighbor search adopts a combined Verlet list and link-cell methods. There are, however, two major issues regarding the imposition of periodic boundary conditions (PBCs) in CAC simulations with coarse-graining that do not exist in standard atomistic simulations.

For more information, read chapter 3 of Shuozhi Xu's Ph.D. dissertation.

Scheme

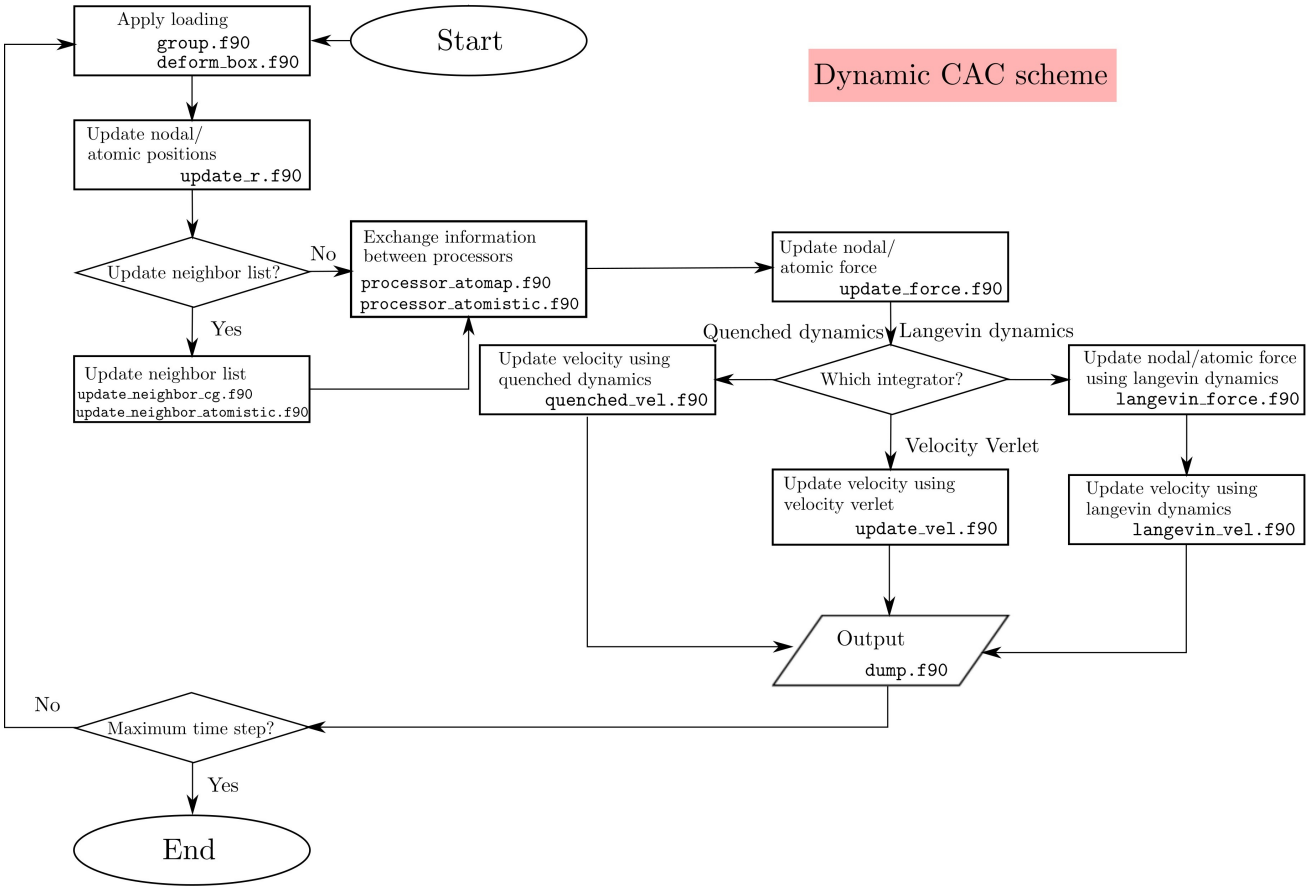
A flowchart of the CAC simulation algorithm based on spatial decomposition is presented below:



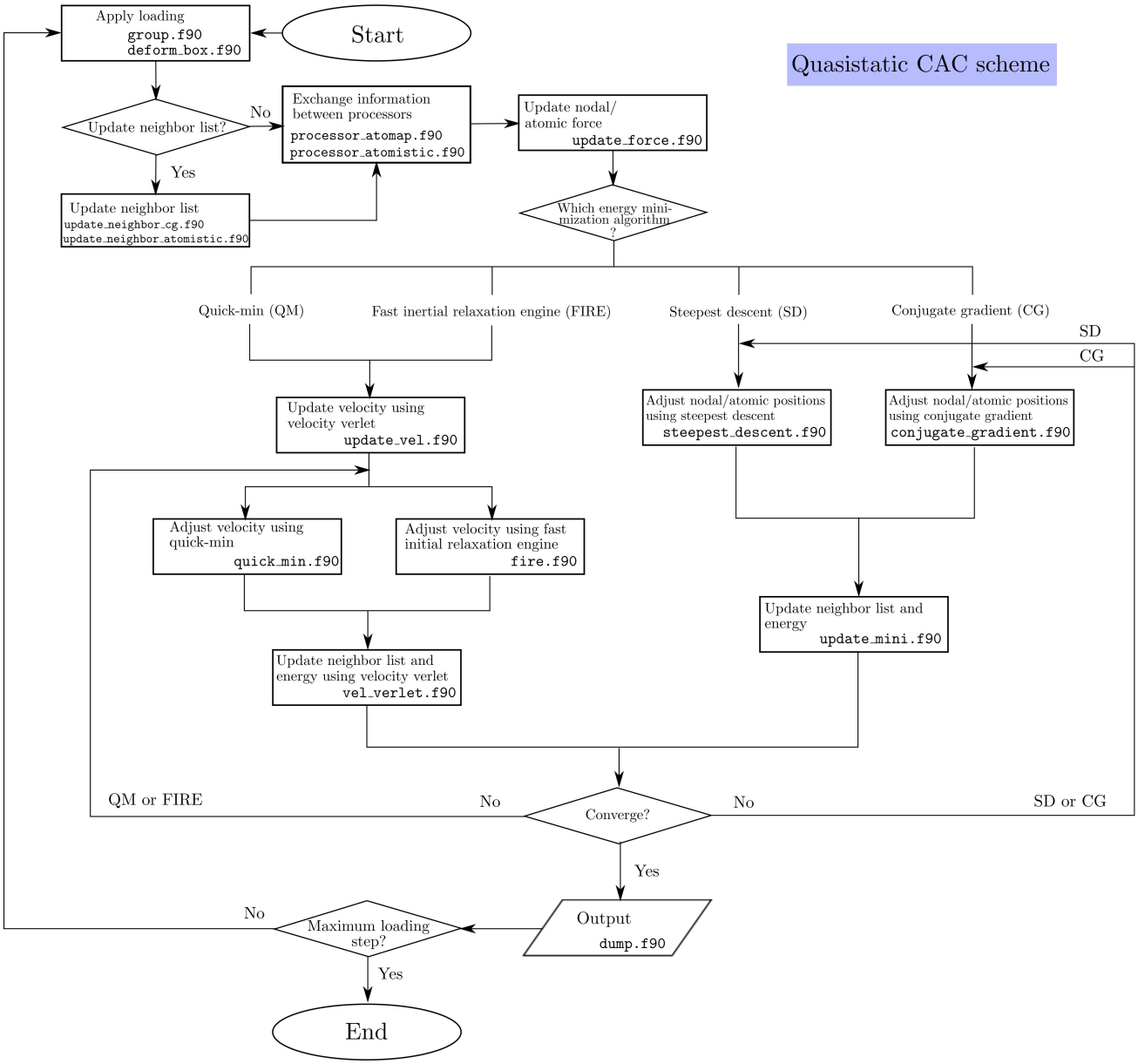
where there are three types of CAC simulations: dynamics, quasistatics, and hybrid, specified by the simulator.

In CAC simulations, the elements/nodes/atoms information can either be created from scratch (`model_setup.f90`) or read from the `cac_in.restart` file (`read_restart.f90`), depending on the parameters in the restart command.

The dynamic CAC scheme is



The quasistatic CAC scheme is


Quasistatic CAC scheme
Start
Apply loading
group.f90
deform_box.f90
Update neighbor list?
No
Yes
Exchange information between processors
processor_atomap.f90
processor_atomistic.f90
Update neighbor list
update_neighbor_cg.f90
update_neighbor_atomistic.f90
Update nodal/ atomic force
update_force.f90
Which energy minimization algorithm ?
Quick-min (QM)
Fast inertial relaxation engine (FIRE)
Steepest descent (SD)
Conjugate gradient (CG)
SD
CG
Update velocity using velocity verlet
update_vel.f90
Adjust nodal/atomic positions using steepest descent
steepest_descent.f90
Adjust nodal/atomic positions using conjugate gradient
conjugate_gradient.f90
Adjust velocity using quick-min
quick_min.f90
Adjust velocity using fast initial relaxation engine
fire.f90
Update neighbor list and energy
update_mini.f90
Update neighbor list and energy using velocity verlet
vel_verlet.f90
QM or FIRE
No
Converge?
No
SD or CG
Yes
No
Maximum loading step?
Output
dump.f90
Yes
End

Parallelization

Among the three parallel algorithms commonly employed in atomistic simulations — atom decomposition (AD), force decomposition (FD), and spatial decomposition (SD), SD yields the best scalability and the smallest communication overhead between processors. Unlike AD and FD, the workload of each processor in SD, which is proportional to the number of interactions, is unfortunately not guaranteed to be the same. In CAC, the simulation cell has nonuniformly distributed integration points (in the coarse-grained domain) and atoms (in the atomistic domain), such that the workload is poorly balanced if one assigns each processor an equally-sized cubic domain as in full atomistics. This workload balance issue is not unique to CAC, but also encountered by other concurrent multiscale modeling methods.

The PyCAC code employs the SD algorithm in which the load balance is optimized. For more information, read chapter 3 of Shuozhi Xu's Ph.D. dissertation.

Arithmetic precision

To ensure the processor-independent precision, the working precision (`wp`) is defined in the `precision_comm_module.f90` module file.

The default precision is 64-bit reals, the users can opt for 128-bit reals by modifying `wp` .

Units

PyCAC assumes the use of the following defined molecular units:

- The unit of time is 10^{-12} seconds (i.e., picoseconds)
- The unit of length is 10^{-10} meters (i.e., Angstroms)
- The unit of mass is $1.66053904 \times 10^{-27}$ kilograms (i.e., Daltons - unified atomic mass units)
- The unit of energy $1.602176565 \times 10^{-19}$ Joules (i.e., eV)
- The unit of pressure is 10^{9} Pascales (i.e., GPa)

Input

To run a PyCAC simulation, one may choose to do one of the following:

1. create/modify `pycac.in`, which is then read by the Python interface to create `cac.in`
2. create/modify `cac.in`, in which the commands provide all input parameters for a CAC simulation.

The `cac.in` file, along with the potential files (`embed.tab`, `pair.tab`, and `edens.tab` for the EAM potential; `lj.para` for the LJ potential), are read by the Fortran CAC code to run the CAC simulation.

The potential files for some FCC metals are provided in the `potentials` directory.

EAM potential

The EAM formulation for potential energy is

$$E = \frac{1}{2}\sum_i \sum_{j \neq i} V(r^{ij}) + \sum_i F(\bar{\rho}^i)$$

where

$$\bar{\rho}^i = \sum_{i \neq j} \rho^{ij}(r^{ij})$$

The first line of each `*.tab` file is

```
N first_val last_val
```

where `N` is a positive integer that equals the number of data pair (each line starting from the second line), `first_val` and `last_val` are non-negative real numbers suggesting the first and the last datum in the first column (starting from the second line), respectively.

- In `embed.tab`, the first column is the unitless host electron energy $\bar{\rho}$; the second column is the embedded energy F, in unit of eV.
- In `pair.tab`, the first column is the interatomic distance r, in unit of Angstrom; the second column is the pair potential V, in unit of eV.

- In `edens.tab`, the first column is the interatomic distance r, in unit of Angstrom; the second column is the unitless local electron density.

For example, the first few lines of `potentials/eam/Ag/williams/edens.tab` are

```
3000 0.5018316703334310 5.995011000293092
0.5018316703334310       8.9800288540000004E-002
0.5036633406668621       9.0604138970000001E-002
0.5054950110002930       9.1404200869999990E-002
0.5073266813337241       9.2200486049999988E-002
```

In the PyCAC code, an approximation is introduced to calculate the host electron density of the integration points in the coarse-grained domain. For more information, read chapter 3 of Shuozhi Xu's Ph.D. dissertation.

LJ potential

The LJ formulation for potential energy is

$E = \frac{1}{2}\sum_i \sum_{j \neq i} 4\epsilon \left[\left(\frac{\sigma}{r^{ij}} \right)^{12} - \left(\frac{\sigma}{r^{ij}} \right)^{6} \right]$

where ϵ and σ are two parameters. In the PyCAC code, the interatomic force, not the energy, is shifted such that the force goes ccontinuously to zero at the cut-off distance r_{c}, i.e., if $r < r_{\mathrm{c}}$, $f = f(r) - f(r_{\mathrm{c}})$; otherwise, $f = 0$.

In `lj.para`, a blank line or a line with the "#" character in the beginning is the comment; four parameters, ϵ, σ, r_0, and r_{c} are presented as positive real numbers (except r_0, which is non-negative) in any sequence, where r_0 is a place holder that should always be 0.0 for the LJ potential. Note that for the EAM potential, r_0 equals the minimum interatomic distance, i.e., the smallest `first_val` given in `pair.tab` and `edens.tab`.

For example, `potentials/lj/Cu/kluge/lj.para` reads

```
# parameters for the LJ potential

epsilon   0.167
sigma     2.315
```

```
rcmin     0.
rcoff     5.38784
```

where `epsilon` = ϵ, `sigma` = σ, `rcmin` = r_0, and `rcoff` = r_c.

Other files

When `boolean_restart` = *t*, a `cac_in.restart` file needs to be provided. This file is renamed from one of the `cac_out_#.restart` files, where `#` is a positive integer.

When `boolean_restart_group` = *t* and `restart_group_num` > 0, or when `boolean_restart_refine` = *t* and `refine_style` = *group*, one or more `group_in_#.id` files need to be provided, where `#` is a positive integer. These files are renamed from `group_out_#.id` files, which are created automatically when the total number of new group, restart group, and boundary group > 0.

Output

A series of vtk and dump files created on-the-fly

The main output of a CAC simulation contains two types of files, with information of the elements, nodes, and atoms. In these files, `#`, a non-negative integer, is the simulation step at which a file is created:

- `cac_cg_#.vtk` and `cac_atom_#.vtk` files, created by `vtk_legacy.f90` and read by ParaView, contain elemental/nodal information and atomic information in the coarse-grained and atomistic domains, respectively. Note that besides the nodal/atomic positions, the energy scalar, force vector, and stress tensor of each node/atom is also recorded in these `*.vtk` files.
- `dump.#` files, created by `atomp_plot.f90` and read by OVITO, are standard LAMMPS dump files containing postions of the real atoms in the atomistic domain and the interpolated atoms in the coarse-grained domain. Each file can be read by LAMMPS to carry out an equivalent fully-resolved atomistic simulation.

One-time vtk and dump files

Besides these files that are created on-the-fly (with a frequency of `output_freq`), in the beginning of a simulation, a `model_atom.vtk` file containing atomic positions in the atomistic domain, a `model_cg.vtk` file containing nodal positions in the coarse-grained domain, and a `model_intpo.vtk` file containing integration point positions and weights in the coarse-grained domain are also created, by `vtk_legacy_model.f90`. A `dump.lammps` file which, in addition to the nodal/atomic positions, also contain the nodal/atomic velocities if `simulation_style` = *dynamics* or *hybrid*, is created by `atomp_plot_lammps.f90`. When the total number of new group, restart group, and boundary group > 0, multiple `group_cg_#.vtk` and `group_atom_#.vtk` files, where `#`, a positive integer, is the group ID, are created by `vtk_legacy_group.f90`. These files are used to show whether the initial simulation cell and group settings are correct. Different from the `cac_cg_#.vtk` and `cac_atom_#.vtk` files, the `*.vtk` files here do not contain the energy/force/stress information, but only the nodal/atomic positions.

All `*.vtk` and `dump.*` files are then post-processed for visualization purposes.

Other files

`cac.log` is the log file of a CAC simulation, containing information mostly written by `cac_log.f90`.

`stress_strain` and `temperature`, with a frequency of `log_freq`, record the 3×3 stress/strain tensors and the temperature, respectively, at each simulation step.

If `boolean_debug` = *t*, a writable `debug` file is created by `debug_init.f90`. The user can then write to it whatever he/she wants using unit number 13, i.e.,

```
write(13, format) output
```

When the total number of new group, restart group, and boundary group > 0, a series of `group_out_#.id` files are created, where `#` is a postive integer starting from 1. These files can then be renamed to `group_in_#.id` for restart group and refinement purposes.

A series of `cac_out_#.restart files` are also created, where `#` is a positive integer starting from 1. One of these files can then be renamed to `cac_in.restart` to restart a previous simulation when boolean_restart = *t*.

Python interface

how python works with CAC

Command

This chapter describes how a CAC input script `cac.in` is formatted and the input script commands used to define a CAC simulation.

Note that the PyCAC input script has a different format.

In a CAC simulation, default settings for some commands are first established by `defaults.f90`, then the entire `cac.in` is read to override some of the default settings: (i) a blank line or a line with the "#" character in the beginning is discarded, and (ii) each command should contain no more than 350 characters. Subsequently, `input_checker.f90` is run to check whether all commands that do not have default settings are provided in `cac.in`. In preparing `cac.in`, it is important to follow the syntax and to distinguish between an interger and a real number, e.g., a real number must be written as *2.* or *2.0*, instead of *2*.

The sequence of commands in `cac.in` does not matter, except for the cal, group, and modify commands, in which case extra commands that (i) appear later and (ii) exceed the numbers in cal_num, group_num, and modify_num, respectively, will be ignored. For example, if `cal_number` = *2*, the last cal command below will be ignored:

```
cal group_1 energy
cal group_2 force
cal group_3 stress
```

During the CAC simulation, a self-explanatory error message, followed by termination of the program by:

```
call mpi_abort(mpi_comm_world, 1, ierr)
```

or a warning message may be issued if something is potentially wrong.

When `boolean_restart` = *t*, the elements/nodes/atoms are read from the `cac_in.restart` file, in which case all commands in the *Simulation Cell* category below become irrelevant; otherwise, the simulation cell is built from scratch.

Below is a list of all 36 CACS commands, grouped by category.

Simulation Cell:

boundary, box_dir, grain_dir, grain_mat, grain_move, grain_num, modify_num, modify, subdomain, unit_num, unit_type, zigzag

Materials:

lattice, mass, potential

Settings:

bd_group, cal_num, cal, constrain, simulator, dump_num, ensemble, group_num, group, limit, mass_mat neighbor, temperature

Actions:

deform, dynamics, minimize, refine, restart, run

Miscellanies:

convert, debug

bd_group

Syntax

```
bd_group x boolean_l boolean_u style_cg style_at depth boolean_def time_start time_end
         y boolean_l boolean_u style_cg style_at depth boolean_def time_start time_end
         z boolean_l boolean_u style_cg style_at depth boolean_def time_start time_end
```

- `boolean_l` , `boolean_u` , `boolean_def` = *t* or *f*

  ```
  t is true
  f is false
  ```

- `style_cg` = *null* or *element* or *node*
- `style_at` = *null* or *atom*
- `depth` = postive real number
- `time_start` , `time_end` = non-negative integer

Examples

```
bd_group x f f null atom 2. t 200 1000 y t f node atom 3. t 0 1000 z t t element null
1. f 500 1000
```

Description

This command provides a shortcut to create groups for the elements/nodes/atoms within a certain distance from each simulation cell boundary (six in total). The IDs of these groups follow the regular groups created or read (from `group_in_#.id`) by the group command. In groups created using this command, the elements/nodes/atoms are not displaced subject to the interatomic forces. In other words, equivalently in the group command,

- `boolean_move` , `boolean_release` = *t*

- `vel_x` , `vel_y` , `vel_z` = *0.0*
- `group_name` = group_# (where # is an integer starting from `new_group_number` + `restart_group_number` + 1)

Along a certain axis, `boolean_l` and `boolean_u` decide whether a group at the corresponding lower and upper boundaries is created, respectively, as illustrated in the figure below.

If a group is to be created, `style_cg` and `style_at` become non-trivial. `style_cg` decides whether the group contains elements (*element*), nodes (*node*), or nothing (*null*) in the coarse-grained domain. The differences between *element* and *node* are also important in the group command. `style_at` decides whether the group contains atoms (*atom*) or nothing (*null*) in the atomistic domain.

All groups defined by the `bd_group` command have a block shape, equivalently, `group_shape` = *block* in the group command. Along the y axis, for example, the groups at the lower and upper boundaries are respectively bounded by

```
x inf inf y inf depth z inf inf
x inf inf y upper_b-depth inf z inf inf
```

where `upper_b` is the upper bound of the simulation cell, similar to that in the group command. The `depth` is in unit of lattice_space_max along the corresponding axis.

`boolean_def` decides whether the group is deformed along with the simulation cell, the same as the one in the group command.

`time_start` and `time_end` , in unit of time step, decides when the groups begin to take effect and become unrestricted (i.e., `boolean_move` = *f* in the group command), respectively.

Related commands

Since this command provides a shortcut to create groups, all of its function can be realized by the group command.

Related files

`group_init.f90`

Default

None.

boundary

Syntax

```
boundary x y z
```

- `x`, `y`, `z` = *p* or *s*

  ```
  p is periodic
  s is non-periodic and shrink-wrapped
  ```

Examples

```
boundary p s s
```

Description

This command sets the boundary conditions of the simulation cell along the *x*, *y*, and *z* directions. Along each axis, the same condition is applied to both the lower and upper face of the box.

p sets periodic boundary conditions (PBCs). The nodes/atoms interact across the boundary and can exit one end of the box and re-enter the other end. For more information of the PBCs in the coarse-grained domain, read chapter 3 of Shuozhi Xu's Ph.D. dissertation.

s sets non-periodic boundary conditions, where nodes/atoms do not interact across the boundary and do not move from one side of the box to the other. The positions of both faces are set so as to encompass the atoms in that dimension, no matter how far they move.

Under either boundary condition, no nodes/atoms will be lost during a simulation.

Related commands

When *p* is set along a certain direction, the corresponding zigzag is set to *f*. In other words, a boundary has to be flat to apply the periodic boundary condition.

This command becomes irrelevant when `boolean_restart` = *t*.

Default

```
boundary p p p
```

box_dir

Syntax

```
box_dir x i j k y i j k z i j k
```

- `i`, `j`, `k` = real number

Examples

```
box_dir x 1. 0. 0. y 0. 1. 0. z 0. 0. 1.
box_dir x 1. 0. 0. y 0. 0.94281 -0.33333 z 0. 0. 1.
```

Description

Decide the grain boundary (GB) plane or the atomistic/coarse-grained domain interface orientation with respect to the simulation cell when there is more than one grain, i.e., `grain_num` > 1 in the grain_num command. When `grain_num` = 1, this command does not take effect.

Assume that `direction` = 2 in the grain_dir command, i.e., the grains are stacked along the *y* direction, the `box` command in the first example results in a GB plane normal to the *y* axis; the `box` command in the second example, however, results in a GB plane inclined with respect to the *y* axis, as shown in the figure below.

The [`ijk`] vector here is similar to those in the group and modify commands.

In the literature, this command was used to create the $\Sigma 3\{111\}$ coherent twin boundary in Fig. 1 of Xu et al. 2016 and Fig. 1(a) of Xu et al. 2017 and the $\Sigma 11\{113\}$ symmetric tilt grain boundary in Fig. 1(b) of Xu et al. 2017.

Related commands

As opposed to the grain_mat command whose orientations are for the lattice, the orientations in this command are with respect to the simulation cell. One may use the convert command to convert the lattice-based orientation to the simulation cell-based orientation.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`model_init.f90` , among many

Default

```
box_dir x 1. 0. 0. y 0. 1. 0. z 0. 0. 1.
```

cal_num

Syntax

```
cal_num cal_number
```

- `cal_number` = non-negative integer (<= 19))

Examples

```
cal_num 0
cal_num 3
```

Description

This command sets the number of group-based calculations.

Related commands

Calculations are based on group.

Related files

`dump_init.f90` and `group_cal.f90`

Default

```
cal_num 0
```

cal

Syntax

```
cal group_name cal_variable
```

- `group_name` = a string (length <= 30)
- `cal_variable` = *energy* or *force* or *stress*

Examples

```
cal group_1 force
cal group_3 stress
```

Description

This commands calculates certain quantities associated with a group.

energy is the total potential energy in a group divided by the number of nodes/atoms in the group. It is a scalar.

force and *stress* are the total force and stress in a group, respectively. *force* is a 3×1 vector while *stress* is a 3×3 tensor.

Results of this command are written to `group_cal_#` at certain simulation step, where `#` is the ID of calculation. For *stress*, a 3×3 strain tensor of the simulation box is appended right after the stress tensor.

Related commands

There cannot be fewer `cal` commands than `cal_number`. When there are too many `cal` commands in `cac.in`, those appearing later will be ignored. The `group_name` must match one for the groups set in the group command.

Related files

`group_cal.f90`

Default

None.

constrain

Syntax

```
constrain boolean i j k
```

- `boolean` = *t* or *f*

  ```
  t is true
  f is faulse
  ```

- `i` , `j` , `k` = real number

Examples

```
constrain f 1. 1. 0.
constrain t 0. 0. 1.
```

Description

The command decides whether and how a force constrain is added to the system. When `boolean` is *t*, the nodal/atomic force vector is projected onto the [`ijk`] direction such that they can only move along that direction, either in dynamic or quasi-static simulations.

Note that the direction is with respect to the simulation cell. For example, the second example projects the force vector onto the *z* axis of the simulation cell.

Related commands

None.

Related files

`constraint.f90`

Default

```
constrain f 0. 0. 1.
```

convert

Syntax

```
convert i j k
```

- `i` , `j` , `k` = real number

Examples

```
convert -1. 1. 2.
convert 1. -1. 0.
```

Description

This command converts the lattice orientation [`i``j``k` ] of each grain to the orientation with respect to the simulation cell [ `i'``j'``k'`]. Results of this conversion will be shown on the screen as

```
Converted box direction of grain # is i' j' k'
```

where `#` is the grain ID.

For example, if the lattice orientation of the second grain along the *x* axis is [211], this command will convert [211] into [100] and output

```
 Converted box direction of 2 grain is 1.0000 0.0000 0.0000
```

Related commands

This command is useful when the user has a set of lattice orientations in mind and wants to find the orientation with respect to the simulation cell, e.g., to be used in the box_dir command.

Related files

`convert_direction.f90`

Default

```
convert 0. 0. 0.
```

debug

Syntax

```
debug boolean_debug boolean_mpi
```

- `boolean_debug`, `boolean_mpi` = *t* or *f*

  ```
  t is true
  f is faulse
  ```

Examples

```
debug t f
debug t t
```

Description

This command generates a writable file named `debug` for debugging purpose. The file is created only when `boolean_debug` is *t*; the unit number is 13. The user can then write whatever he/she wants to `debug` using unit number 13, i.e.,

```
write(13, format) output
```

When `boolean_mpi` is *t*, all processors have access to `debug`, otherwise only the root does.

Related commands

None.

Related files

`debug_init.f90`

Default

```
debug f f
```

deform

Syntax

```
deform boolean_def def_num
       {ij boolean_cg boolean_at def_rate stress_l stress_u flip_frequency}
       time time_start time_always_flip time_end
```

- `boolean_def`, `boolean_cg`, `boolean_at` = *t* or *f*

  ```
  t is true
  f is false
  ```

- `def_num` = non-negative integer (<= 9)
- `ij` = *xx* or *yy* or *zz* or *xy* or *yz* or *yz* or *zy* or *xz* or *zx*
- `def_rate` = real number
- `stress_l`, `stress_u` = positive real number
- `flip_frequency` = positive integer
- `time_start`, `time_always_flip`, `time_end` = non-negative integer

Examples

```
deform t 1 {zx t t 0.05 0.6 0.7 10} time 500 1000 2500
deform t 2 {xx t f 0.01 1. 1.2 20} {yz f t 0.02 0.8 0.9 30} time 400 600 1900
```

Description

This command sets up the homogeneous deformation of the simulation box. Note that the curly brackets `{` and `}` in the syntax/examples are to separate different deformation modes, the number of which is `def_num`; all brackets should not be included in preparing `cac.in`.

The deformation is applied only if `boolean_def` is *t*. The coarse-grained and atomistic domains are deformed only if `boolean_cg` and `boolean_at` are *t*, respectively.

`def_num` sets the number of superimposed deformation modes.

`ij` decides each deformation mode, i.e., how the strain is applied. Following the standard indexes ϵ_{ij} in continuum mechanics, `i` and `j` are the face on which and the direction along which the strain is applied. When `i` and `j` are the same, a uniaxial strain is applied; otherwise, a shear strain is applied.

`def_rate` is the strain rate, in the unit of ps^{-1}.

`stress_l` and `stress_u` are the lower and upper bounds of the stress tensor component (designated by `ij`) of the simulation cell, respectively, in unit of GPa. In most PyCAC simulations, all stress components are usually initially very small. Subject to the strain, most stress tensor components increase until one of them is higher than the corresponding `stress_u`, at which point the strain rate tensor changes sign, i.e., the deformation is reversed. Subject to the newly reversed strain, most stress tensor components decrease until one of them is lower than the corresponding `stress_l`, in which case the strain rate tensor changes sign again, i.e., the deformation is applied as the initial setting. Whether the stress component is out of bounds is monitored not at every step, but at every `flip_frequency` step.

The deformation begins when the total step equals `time_start` and stops when the total step exceeds `time_end`.

When (i) the total step is larger than `time_always_flip` and (ii) the total step does not exceed `time_end` and (iii) the strain rate tensor has not changed sign previously, the strain rate tensor changes sign at every step, regardless of the stress bounds defined by `stress_l` and `stress_u`. This is used, e.g., to keep a quasi-constant strain while the nodes and atoms adjust their positions in dynamics or quasi-static equilibrium. To disable this option, the user may set `time_always_flip` to be larger than `time_end`.

Related commands

Groups defined by the group and bd_group commands may be homogeneously deformed along with the simulation cell, depending on the value of `boolean_def` in these two commands.

Related files

`deform_init.f90` and `deform_box.f90`

Default

```
deform f 1 xx f f 0. 0. 0. 1 time 0 0 0
```

dump

Syntax

```
dump output_freq reduce_freq restart_freq log_freq
```

- `output_freq`, `reduce_freq`, `restart_freq`, `log_freq` = positive integer

Examples

```
dump 500 300 1000 10
```

Description

This command sets the frequency with which the output is performed. For example, when a certain frequency is 100, the corresponding output is conducted when the total step is divisible by 100.

`output_freq` sets the frequency with which the `dump.#` files (readable by OVITO) and the `*.vtk` files (readable by ParaView) are written to the disk system. The user may then post-process these files for visualization and further analysis.

`reduce_freq` sets the frequency with which certain quantities are written to the `cac.log` file by root, which MPI_Reduces relevant information from other processors.

`restart_freq` sets the frequency with which the `cac_out_#.restart` files are written to the disk system. These files can be read to restart simulations.

`log_freq` sets the frequency with which one line is written to the `cac.log` file to monitor the simulation progress.

Related commands

None.

Related files

`dump_init.f90` and `dump.f90`

Default

```
dump 1000 1000 5000 50
```

ensemble

Syntax

```
ensemble boolean_t boolean_p
```

- `boolean_t` , `boolean_p` = *t* or *f*

  ```
  t is true
  f is false
  ```

Examples

```
ensemble t f
```

Description

This command decides whether the temperature (`boolean_t`) and the pressure (`boolean_p`) are kept a constant in a PyCAC simulation.

Related commands

The temperature is kept a constant only when `dyn_style` = *ld*. A warning will be issued if other `dyn_style` is used. The pressure cannot be kept a constant in the current PyCAC code.

Related files

`thermostat.f90`

Default

```
ensemble f f
```

grain_dir

Syntax

```
grain_dir direction overlap
```

- `direction` = 1 or 2 or 3
- `overlap` = real number

Examples

```
grain_dir 1 0.1
grain_dir 2 0.2
```

Description

This command sets the grain stack direction and the overlap between adjacent grains along that direction, as shown in the figure below:

`direction` can only be 1, 2, or 3, corresponding to the *x*, *y*, or *z* directions, respectively.

`overlap`, in unit of the periodic length of the lattice, sets the overlap between adjacent grains along the `direction`. It is used to adjust the relative position along a certain direction between adjacent grains to find the energy minimized grain boundary structure. If `overlap` is a large positive real number, some atoms from adjacent grains may be too close to each other. In this case, one may use the *cutoff* style in the modify command to delete some atoms that are within a certain distance from others.

Related commands

This command is only relevant when `grain_number` is more than one.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`box_init.f90` and `model_init.f90`

Default

```
grain_dir 3 0.
```

grain_mat

Syntax

```
grain_mat {grain_id x i j k y i j k z i j k}
```

- `i`, `j`, `k` = real number

Examples

```
grain_mat {1 x -1. 1. -2. y 1. 1. 0. z 1. -1. -1.}
grain_mat {1 x 1. 1. 0. y -1. 1. 2. z 1. -1. 1.} {2 x 1. 1. 0. y -1. 1. -2. z -1. 1. 1
.}
```

Description

This command sets the lattice orientations in each grain, along the *x*, *y*, and *z* directions, respectively. Note that the curly brackets `{` and `}` in the syntax/examples are to separate different grains, the number of which is `grain_number`; all brackets should not be included in preparing `cac.in`.

Any two sets of vector must be normal to each other, i.e.,

$\mathbf{x} \cdot \mathbf{y} = 0$

$\mathbf{y} \cdot \mathbf{z} = 0$

$\mathbf{x} \cdot \mathbf{z} = 0$

The right hand rule must also be obeyed, i.e.,

$\mathbf{x} \times \mathbf{y} \parallel \mathbf{z}$

$\mathbf{y} \times \mathbf{z} \parallel \mathbf{x}$

$\mathbf{z} \times \mathbf{x} \parallel \mathbf{y}$

An error will be issued if any of these requirements is not satisfied.

The maximum `grain_id` must be larger than or equal to `grain_number`. All information related to `grain_id` that is larger than `grain_number` is discarded.

Related commands

The number of grain is specified in the grain_num command.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`grain.f90`

Default

```
grain_mat 1 x 1. 0. 0. y 0. 1. 0. z 0. 0. 1.
```

grain_move

Syntax

```
grain_move {grain_id move_x move_y move_z}
```

- `grain_id` = positive integer
- `move_x`, `move_y`, `move_z` = real number

Examples

```
grain_move {1 0. 0. 0.} {2 0.5 -0.301 0.001}
```

Description

This command sets the displacements of the origin of each grain along the *x*, *y*, and *z* axis. Note that the curly brackets `{` and `}` in the syntax/examples are to separate different grains, the number of which is `grain_number`; all brackets should not be included in preparing `cac.in`.

The maximum `grain_id` must be larger than or equal to `grain_number`. All information related to `grain_id` that is larger than `grain_number` is discarded.

Related commands

When the displacement vector is along the group stack direction, result by this command may be equivalent to setting the `overlap` between adjacent grains. Note that the same `overlap` is applied between all adjacent grains, while this command sets the displacement vector for each grain independently.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`box_init.f90`

Default

```
grain_move 1 0. 0. 0.
```

grain_num

Syntax

```
grain_num grain_number
```

- `grain_number` = positive integer

Examples

```
grain_num 2
```

Description

This command sets the number of grains in the simulation cell. When `grain_number` > 1, grains are stacked along the `direction`. Each grain has its own lattice orientations, origin displacements, and number of subdomains.

Related commands

In commands grain_mat, grain_move, subdomain, unit_num, and unit_type, all information related to `grain_id` that is larger than `grain_number` in this command will be discarded.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`box_init.f90` and `grain.f90`

Default

```
grain_num 1
```

group_num

Syntax

```
group_num new_group_number restart_group_number
```

- `new_group_number`, `restart_group_number` = non-negative integer

Examples

```
group_num 3 0
group_num 2 1
```

Description

This command sets the numbers of new groups and restart groups. In CAC, a group is a collection of elements/nodes/atoms. There are two purposes of having groups: (i) to apply a controlled displacement to a group, (ii) to calculate certain mechanical quantities such as energy, force, and stress.

The new groups are defined in the group command. The elements/nodes/atoms contained in restart groups are read from the `group_in_#.id` files, yet the displacement information is set in the group command. Note that in the file name, the `#` is an integer starting from `new_group_number` + 1.

Note that the total number of groups, i.e., `new_group_number` + `restart_group_number` + the number of boundary groups set in the bd_group, cannot be larger than 39. Note that for all groups, CAC outputs `group_out_#.id` files containing corresponding elements/nodes/atoms information, where `#` is the group id starting from 1. One may rename `group_out_#.id` to `group_in_#.id` and use the latter for the restart groups.

Related commands

The controlled displacement information of each group is set in the group command.

Related files

`group_init.f90` and `group.f90`

Default

```
group_num 0 0
```

group

Syntax

```
group group_name style_cg style_at group_shape
      x lower_b upper_b i j k
      y lower_b upper_b i j k
      z lower_b upper_b i j k
      boolean_in group_axis
      group_centroid_x group_centroid_y group_centroid_z
      group_radius_large group_radius_small
      boolean_move boolean_release boolean_def
      vel vel_x vel_y vel_z
      time time_start time_end
      disp disp_lim
      boolean_grad boolean_switch
      grad_ref_axis grad_vel_axis
      grad_ref_l grad_ref_u
```

- `group_name` = a string (length <= 30)
- `style_cg` = *element* or *node* or *null*
- `style_at` = *atom* or *null*
- `group_shape` = *block* or *cylinder* or *cone* or *tube* or *sphere*
- `lower_b`, `upper_b` = real number or *inf*
- `i`, `j`, `k` = real number
- `boolean_in`, `boolean_move`, `boolean_release`, `boolean_def`, `boolean_grad`, `boolean_switch` = *t* or *f*

  ```
  t is true
  f is false
  ```

- `group_axis`, `grad_ref_axis`, `grad_vel_axis` = *1* or *2* or *3*
- `group_centroid_x`, `group_centroid_y`, `group_centroid_z` = real number

- `group_radius_large` , `group_radius_small` = positive real number
- `vel_x` , `vel_y` , `vel_z` = real number
- `disp_lim` = non-negative real number
- `time_start` , `time_end` = non-negative integer
- `grad_ref_l` , `grad_ref_u` = real number or *inf*

Examples

```
group group_1 null atom block x inf inf 1. 0. 0. y inf inf 0. 1. 0. z 14.4 inf 0. 0. 1
. t 3 20. 5. 0. 10. 10. f
group group_2 node null cylinder x inf inf 1. 0. 0. y inf inf 0. 1. 0. z 14.4 inf 0. 0
. 1. f 3 20. 5. 0. 10. 10. t t t vel 0. 0. 0. time 0 2500 disp 5. f
group group_3 element atom cone x inf inf 1. 0. 0. y inf inf 0. 1. 0. z 14.4 inf 0. 0.
 1. t 3 20. 5. 0. 10. 5. t t t vel 0. 0. 0. time 0 2500 disp 10. t f 2 1 50. 60.
group group_4 element null sphere x inf inf 1. 0. 0. y inf inf 0. 1. 0. z 14.4 inf 0.
0. 1. t 3 20. 5. 0. 10. 10. t t t vel 0. 0. 0. time 0 2500 disp 3. t t 3 2 10. 100.
group group_5 t t t vel 0. 0. 0. time 0 100 disp 3. f
```

Description

This command sets controlled displacements for new groups and restart groups, the numbers of which are provided in the group_num command. The elements/nodes/atoms in a group are displaced at each simulation step (when `boolean_move` = *t*), deformed with the simulation cell deformation (when `boolean_def` = *t*), or not displaced/deformed. In any case, when the `total_step` is between `time_start` and `time_end` , the force on the group calculated by the interatomic potential is discarded in `constraint.f90` and so does not take effect. The syntax, to some extent, is similar to the first one of the modify command.

The new groups are created by first providing the elements/nodes/atoms information (by options from `style_cg` to `group_radius_small`) while the same information for the restart groups is read from `group_in_#.id` , where `#` is an positive integer starting from `new_group_number` + 1. The `group_in_#.id` files are renamed from the `group_out_#.id` files that were created automatically in previous CAC simulations when the total number of groups > 0.

For the restart groups, which are introduced when `boolean_restart_group` = *t* and `restart_group_number` > 0, the syntax of this command becomes (e.g., the fifth example)

```
group group_name boolean_move boolean_release boolean_def
      vel vel_x vel_y vel_z
      time time_start time_end
      disp disp_lim
      boolean_grad boolean_switch
      grad_ref_axis grad_vel_axis
      grad_ref_l grad_ref_u
```

`style_cg` decides whether the group contains elements (*element*), nodes (*node*), or nothing (*null*) in the coarse-grained domain. The differences between *element* and *node* are also important in the bd_group command. `style_at` decides whether the group contains atoms (*atom*) or nothing (*null*) in the atomistic domain.

There are currently five `group_shape` : *block*, *cylinder*, *cone*, *tube*, and *sphere*. The groups introduced in the bd_group command has `group_shape` = *block*.

`lower_b` and `upper_b` are the lower and upper boundariess of the `group_shape` , respectively, in unit of the lattice periodic length, for the corresponding direction. When `lower_b` or `upper_b` is *inf*, the corresponding lower or upper simulation cell boundaries are taken as the `group_shape` boundaries, respectively.

`lower_b` and `upper_b` are their plane boundaries normal to the central axis `group_axis` direction. Note that `group_axis` is irrelevant when `group_shape` = *sphere*.

`i` , `j` , and `k` decide the `group_shape` boundary plane orientations with respect to the simulation cell, similar to those in the box_dir and modify commands.

Note that these five options (`lower_b` , `upper_b` , `i` , `j` , and `k`) are irrelevant when `group_shape` = *sphere*, and when `group_shape` = *cylinder* or *cone* or *tube* if the corresponding direction is not `group_axis` . However, they need to be provided regardless.

When `boolean_in` = *t*, elements/nodes/atoms inside the `group_shape` belong to the group; otherwise, those outside do.

`group_centroid_x` , `group_centroid_y` , and `group_centroid_z` , in unit of the lattice periodic length, are the coordinates of the center of the base plane of a *cylinder* or *cone* or *tube*, or the center of a *sphere*. When `group_shape` = *cylinder* or *cone* or *tube*, the

`group_centroid_*` that corresponds to the `group_axis` becomes irrelevant. For example, when `group_axis` = *2*, `group_centroid_y` can take any real number without affecting the results.

`group_radius_large` is the base radius of a *cylinder*, the large base radius of a *cone*, the outer base radius of a *tube*, or the radius of a *sphere*. `group_radius_small`, the small base radius of a *cone* or the inner base radius of a *tube*, is irrelevant for other `group_shape`.

Note that these six options (`group_axis`, `group_centroid_*`, and `group_radius_*`) are not relevant when `group_shape` = *block*. Yet, they need to be provided regardless.

When `boolean_move` = *t*, the group is assigned a displacement at each simulation step; otherwise, no controlled displacement is applied and all following options become irrelevant, as in the first example. In the latter case, the purpose of having a group is to calculate certain mechanical quantities of this group such as energy, force, and stress.

When `boolean_release` = *t*, the group is no longer assigned a displacement at each simulation step when `total_step` > `time_end`; otherwise, the group is assigned a zero displacment vector, i.e., fixed, when `total_step` > `time_end`.

When `boolean_def` = *t*, the group is deformed along with the simulation box, the same as that in the bd_group command. Note that in both commands, the group is deformed only when `total_step` is between `time_start` and `time_end`. This option takes effect regardless of the controlled displacement vector.

[`vel_x`, `vel_y`, `vel_z`] is the displacement vector assigned to the group at each simulation step, in unit of ps/Angstrom.

`time_start` and `time_end` are the starting and ending simulation steps, respectively, for the controlled displacement.

`disp_lim` is the upper bounds of the magnitude of the total group displacement, in unit of the lattice constant. For example, if a group is displaced first by vector $\mathbf{a}$ then by vector $\mathbf{b}$ that is not parallel to $\mathbf{a}$, the total displacement is defined as $|\mathbf{a}| + |\mathbf{b}|$, instead of $|\mathbf{a} + \mathbf{b}|$. If the total displacement is larger than `disp_lim`, the displacement vector is zeroed.

When `boolean_grad` = *t*, the displacement is assigned to the group gradiently, i.e., different elements/nodes/atoms in the group may have different [`vel_x`, `vel_y`, `vel_z`]. The `grad_vel_axis` component of the displacement vector is linearly applied to the group based on the positions of elements/nodes/atoms along the `grad_ref_axis` direction. `grad_ref_l`

and `grad_ref_u` are the lower and upper bounds of the graded displacement, in unit of the lattice priodic length, with *inf* referring to the relevant simulation cell boundaries. The elements/nodes/atoms located at or below `grad_ref_l` are assigned a zero displacement, i.e., fixed; those located at or above `grad_ref_u` are assigned [`vel_x` , `vel_y` , `vel_z`]; those located between `grad_ref_l` and `grad_ref_u` are assigned a vector whose `grad_vel_axis` component is linearly graded while the other two components remain the same with respect to [`vel_x` , `vel_y` , `vel_z`].

In the third example, the elements/nodes/atoms which are located below $50.0 \times \mathrm{latticeperiodiclength}$ along the *y* axis (because `grad_ref_axis` = *2*) are assigned a zero displacement vector; those located above $60.0 \times \mathrm{latticeperiodiclength}$ along the *y* axis are assigned [`vel_x` , `vel_y` , `vel_z`]; those in between are assigned a linearly graded displacement vector whose *x* component (because `grad_vel_axis` = *1*) is varied between zero and `vel_x` while its *y* and *z* components are `vel_y` and `vel_z` , respectively.

When `boolean_switch` = *t*, the lower and upper bounds of the graded displacement are switched. In the fourth example, the elements/nodes/atoms which are located below $10.0 \times \mathrm{latticeperiodiclength}$ along the *z* axis (because `grad_ref_axis` = *3*) are assigned a displacement vector [`vel_x` , `vel_y` , `vel_z`]; those located above $100.0 \times \mathrm{latticeperiodiclength}$ along the *z* axis are assigned a zero displacement; those in between are assigned a linearly graded displacement vector whose *y* component (because `grad_vel_axis` = *2*) is varied between zero and `vel_y` while its *x* and *z* components are `vel_x` and `vel_z` , respectively.

Related commands

There cannot be fewer `group` commands than `new_group_number` + `restart_group_number` . When there are too many `group` commands in `cac.in` , those appearing later will be ignored. The `group_name` in the cal command must match that in the current command.

The `group_name` of groups defined in the bd_group command are group_#, where # is an integer starting from `new_group_number` + `restart_group_number` + 1.

This command becomes irrelevant when `new_group_number` + `restart_group_number` = 0.

Related files

`group.f90`

Default

None.

lattice

Syntax

```
lattice chemical_element lattice_structure lattice_constant
```

- `chemical_element` = a string (length <= 30)
- `lattice_structure` = *fcc* or *bcc*
- `lattice_constant` = positive real number

Examples

```
lattice Cu fcc 3.615
lattice Al fcc 4.05
lattice Fe bcc 2.8553
```

Description

This command sets the lattice.

`lattice_constant` is in unit of Angstrom.

Note that (i) the current CAC code can only simulate pure metals with single chemical element, (ii) `lattice_structure` must be either *fcc* or *bcc*, yielding rhombohedral elements with {111} and {110} surfaces, respectively. An error will be issued if other lattice structures are provided.

Related commands

The `atomic_mass` is provided separately in the mass command.

`lattice_structure` becomes irrelevant when `boolean_restart` = *t*.

Related files

`box_init.f90` and `lattice.f90`

Default

None.

limit

Syntax

```
limit atom_per_cell_number atomic_neighbor_number
```

- `atom_per_cell_number`, `atomic_neighbor_number` = positive integer

Examples

```
limit 100 100
limit 120 140
```

Description

This command sets the initial number of atoms per cell (`atom_per_cell_number`) and the number of neighboring atoms per atom (`atomic_neighbor_number`). The numbers are used to allocate initial arrays for atoms in cells and neighbors of atoms. If, during the simulation, the array sizes become larger than those initially allocated, the numbers set in this command will increase by 20 to enlarge the arrays, until the array sizes exceed the increased numbers, in which case the numbers are increased in increments of 20 again..

Related commands

These two numbers depend on the cutoff distance and bin_size of the interatomic potential.

Related files

`neighbor_init.f90` , `update_neighbor.f90` , `cell_neighbor_list.f90` , `update_cell_neighbor.f90` , and `update_cell.f90`

Default

```
limit 100 100
```

mass

Syntax

```
mass atomic_mass
```

- `atomic_mass` = positive real number

Examples

```
mass 63.546
mass 26.9815
mass 55.845
```

Description

This command sets the atomic mass, in unit of g/mol. The three examples are for Cu, Al, and Fe, respectively, corresponding to those in the lattice command. Note the current PyCAC code can only simulate pure metals.

Related commands

The mass matrix type in the finite element calculation is specified in the mass_mat command.

Related files

`crystal.f90` and `mass_matrix.f90`

Default

None.

mass_mat

Syntax

```
mass_mat mass_matrix
```

- `mass_matrix` = *lumped* or *consistent*

Examples

```
mass_mat lumped
mass_mat consistent
```

Description

This command sets the mass matrix type used in the finite element calculation. The lumped mass matrix approximates the mass of each element and distributes it to the nodes. The consistent mass matrix distributes the exact mass over the entire element.

Related commands

The atomic mass is defined by the mass command.

Related files

`mass_matrix.f90` and `update_equiv.f90`

Default

```
mass_mat lumped
```

minimize

Syntax

```
minimize mini_style max_iteration tolerance
```

- `mini_style` = *cg* or *sd* or *fire* or *qm*
- `max_iteration` = positive integer
- `tolerance` = positive real number

Examples

```
minimize cg 1000 1d-5
minimize fire 100 1d-6
```

Description

This command sets the style and two parameters for the energy minimization in quasistatic CAC. At each simulation step (loading increment), the energy minimization usually consists of two levels of iterations: in the outer iteration, the nodes/atoms are moved along a certain direction; in the inner iteration, a line search is conducted to determin the magnitude of the movement to minimize the energy.

There are four `mini_style`: congjugate gradient (*cg*), steepest descent (*sd*), fast inertial relaxation engine (*fire*), and quick min (*qm*).

Both *cg* and *sd* use the negative gradient of potential energy as the initial direction; from the second step, however, the *sd* style uses the current negative gradient while the *cg* style uses the negative gradient conjugated to the current potential surface. The line search is used to find the length along which the nodes/atoms need to move along the designated direction to find the minimized energy. For more information of the energy minimization with these two styles, read chapter 3 of Shuozhi Xu's Ph.D. dissertation.

The *fire* style is based on Bitzek et al., 2006 while the *qm* style is based on quenched dynamics which is used also in dynamic CAC. The difference is that only one quenched dynamics iteration is carried out at each simulation step in dynamic CAC while many quenched dynamics iterations are performed at each simulation step in quasistatic CAC until the energy converges at that step. For the *fire* and *qm* styles, the inner iteration is irrelevant.

The energy minimiztion is considered to converge when either the outer iteration reaches `max_iteraction` or the energy variation between successive outer iterations divided by the energy of the current iteration is less than the `tolerance`.

Related commands

This command is relevant only when simulator = *statics* or *hybrid*.

Related files

`quasi_statics.f90`, `mini_init.f90`, `update_mini.f90`, `mini_energy.f90`, `hybrid.f90`, `conjugate_gradient.f90`, `steepest_descent.f90`, `quick_mini.f90`, `fire.f90`

Default

```
minimize cg 1000 1d-6
```

modify_num

Syntax

```
modify_num modify_number
```

- `modify_number` = non-negative integer (<= 19)

Examples

```
modify_num 2
```

Description

This command sets the number of modifications that are made to the elements/nodes/atoms that are built from scratch, i.e., when `boolean_restart` = *f*.

Related commands

The modification style is set by the modify command.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`model_modify.f90`

Default

```
modify_num 0
```

modify

Syntax

```
modify modify_name modify_style modify_shape
       x lower_b upper_b i j k
       y lower_b upper_b i j k
       z lower_b upper_b i j k
       boolean_in boolean_delete_filled modify_axis
       modify_centroid_x modify_centroid_y modify_centroid_z
       modify_radius_large modify_radius_small

modify modify_name modify_style depth tolerance
```

- `modify_name` = a string (length <= 30)
- `modify_style` = *delete* or *cg2at* or *cutoff*
- `modify_shape` = *block* or *cylinder* or *cone* or *tube* or *sphere*
- `lower_b` , `upper_b` = real number or *inf*
- `i` , `j` , `k` = real number
- `boolean_in` , `boolean_delete_filled` = *t* or *f*

  ```
  t is true
  f is false
  ```

- `modify_axis` = *1* or *2* or *3*
- `modify_centroid_x` , `modify_centroid_y` , `modify_centroid_z` = real number
- `modify_radius_large` , `modify_radius_small` , `depth` , `tolerance` = positive real number

Examples

```
modify modify_1 delete cylinder x 0. 1. 0.94281 0. -0.33333 y inf inf 0. 1. 0. z inf i
```

```
nf 0. 0. 1. t t 3 50. 50. 1. 2. 5.
modify modify_2 cg2at block x inf inf 1. 0. 0. y 1. 12. 0. 0.94281 -0.33333 z inf inf
0. 0. 1. t f 1 20. 4. 5. 17. 13.
modify modify_3 cutoff 0.1 0.01
```

Description

This command sets the modifications made to the elements/nodes/atoms that are built from scratch, i.e., when `boolean_restart` = *f*. The first syntax, to some extent, is similar to that of the group command for the new group.

There are currently three `modify_style` : *delete*, *cg2at*, and *cutoff*. When `modify_style` = *delete* or *cg2at*, the first syntax is used; otherwise, the second syntax with *depth* and *tolerance* is used.

In the first syntax, there are five `modify_shape` : *block*, *cylinder*, *cone*, *tube*, and *sphere*. *delete* removes some elements/atoms from the preliminary simulation cell, and *cg2at* refines some elements into atomic scale.

`lower_b` and `upper_b` are the lower and upper boundaries of the `modify_shape` , respectively, in unit of the lattice periodic length, for the corresponding direction. When `lower_b` or `upper_b` is *inf*, the corresponding lower or upper simulation cell boundaries are taken as the `modify_shape` boundaries, respectively.

`lower_b` and `upper_b` are their plane boundaries normal to the central axis `modify_axis` direction. Note that `modify_axis` is irrelevant when `modify_shape` = *sphere*.

`i` , `j` , and `k` decide the `modify_shape` boundary plane orientations with respect to the simulation cell, similar to those in the box_dir and group commands.

Note that these five options (`lower_b` , `upper_b` , `i` , `j` , and `k`) are irrelevant when `modify_shape` = *sphere*, and when `modify_shape` = *cylinder* or *cone* or *tube* if the corresponding direction is not `group_axis` . However, they need to be provided regardless.

When `boolean_in` = *t*, elements with any of their parts (in the coarse-grained domain) and atoms (in the atomistic domain) inside the `modify_shape` are deleted (*delete*) or refined to atomic scale (*cg2at*); otherwise, those outside are. In the coarse-grained domain, an element might have some part inside and the remaining part outside `modify_shape` ; for this element, with *delete*, the region that is left behind due to the deletion may not have the

shape specified by `modify_shape`. In this case, if `boolean_delete_filled` = *t*, atoms (that are linearly interpolated from the original element) will be filled in to maintain the `modify_shape`. E.g., if `boolean_in` = *t*, the interpolated atoms of the deleted elements that are outside `modify_shape` are filled in; otherwise, those inside are, as shown in the figure below. Note that `boolean_delete_filled` is irrelevant when `modify_style` = *cg2at*.

`modify_centroid_x`, `modify_centroid_y`, and `modify_centroid_z`, in unit of the lattice periodic length, are the coordinates of the center of the base plane of a *cylinder* or *cone* or *tube*, or the center of a *sphere*. When `modify_shape` = *cylinder* or *cone* or *tube*, the `modify_centroid_*` that corresponds to the `modify_axis` becomes irrelevant. For example, when `modify_axis` = *3*, `modify_centroid_z` can take any real number without affecting the results.

`modify_radius_large` is the base radius of a *cylinder*, the large base radius of a *cone*, the outer base radius of a *tube*, or the radius of a *sphere*. `modify_radius_small`, the small base radius of a *cone* or the inner base radius of a *tube*, is irrelevant for other `group_shape`.

Note that these six options (`modify_axis`, `modify_centroid_*`, and `modify_radius_*`) are not relevant when `modify_shape` = *block*. Yet, they need to be provided regardless.

In the second syntax, which is for `modify_style` = *cutoff*, `depth` and `tolerance`, in unit of the lattice periodic length along the grain stack direction, specify the size of the target region and the cutoff distance, respectively, as shown in the figure below.

This is useful when the interatomic distance < `tolerance` or the distance between a node and an atom < `tolerance`, e.g., at the grain boundary, because of the `overlap` or grain origin displacements. At first, a check is conducted, within the region set by `depth`, on both the real atoms in the atomistic domain or the interpolated atoms in the coarse-grained doain. Within a pair, if both are real atoms, the one associated with a smaller `grain_id` is deleted; if one is a real atom and the other is an interpolated atom, the real atom is deleted; if both are interpolated atoms, an error message is issued because it is impossible to delete an interpolated atom from an element.

Related commands

There cannot be fewer `modify` commands than `modify_number`. When there are too many `modify` commands in `cac.in`, those appearing later will be ignored.

This command becomes irrelevant when `boolean_restart` = *t* or `modify_number` = 0.

Related files

`model_modify.f90` and `model_modify_interpo.f90`

Default

None.

neighbor

Syntax

```
neighbor bin_size neighbor_freq
```

- `bin_size` = non-negative real number
- `neighbor_freq` = positive integer

Examples

```
neighbor 1. 100
neighbor 2. 200
```

Description

This command sets parameters for updating the neighbor list. In PyCAC simulatoins, each atom in the atomistic domain and each integration point in the coarse-grained domain maintain neighbor lists. Note that the non-integration point interpolated atoms in the coarse-grained domain do not have neighbor lists.

`bin_size`, in unit of Angstrom, sets the length of the bin, which adds to the cutoff distance r_c of the interatomic potential. All atoms within cutoff distance + `bin_size` from an atom are the neighbors of this atom.

`neighbor_freq` is the frequency with which a check of whether the neighbor list should be updated is issued. The neighbor list is updated if, with respect to the nodal/atomic positions recorded at the last check, any node or atom has a displacement larger than half the `bin_size`, the neighbor lists of all integration points and atoms are updated.

Related commands

The initial number of neighboring atoms per atom is set in the limit command.

Related files

`neighbor_init.f90` and `update_neighbor.f90`

Default

```
neighbor 1. 200
```

potential

Syntax

```
potential potential_type cohesive_energy
```

- `potential_type` = *lj* or *eam*

  ```
  lj is the Lennard-Johns potential
  eam is the embedded-atom method potential
  ```

- `cohesive_energy` = negative real number

Examples

```
potential lj -3.54
potential eam -4.45
```

Description

This command sets the interatomic potentials. Currently, the PyCAC simulations accept two `potential_style`: Lennard-Johns (*lj*) and embedded-atom method (*eam*) potentials. One file for the *lj* potential and four files for the *eam* potential, respectively, should be provided as input.

`cohesive_energy` is the cohesive energy of one atom in a perfect lattice given by the interatomic potential, in unit of eV.

Related commands

None.

Related files

`potential.f90`, `eam_tab.f90`, `deriv_tab.f90`, and `lj_para.f90`.

Default

None.

refine

Syntax

```
refine refine_style refine_group_number unitype
```

- `refine_style` = *all* or *group*
- `refine_group_number`, `unitype` = positive integer

Examples

```
refine all 1 6
refine group 1 12
refine group 2 6
```

Description

This command sets refinement styles when `boolean_restart_refine` = *t*.

There are two `refine_style` : *all* or *group*.

When `refine_style` = *all*, all elements in the coarse-grained domain are refined into atomic scale. This is used when, e.g., the users want to perform an equivalent full atomistic simulation with the CAC code. Currently, this option is correctly trigered only when all elements have the same size, i.e., the same `unitype` had been used in all coarse-grained subdomains based on which the `cac_in.restart` file was created. In the first example, the `cac_in.restart` file refers to a simulation cell with elements each of which has $(6+1)^3 = 343$ atoms. `refine_group_number` is irrevelant for this `refine_style`.

When `refine_style` = *group*, selected elements in the `group_in_#.id` files (where `#` is a positive integer starting from 1) in the coarse-grained domain are refined into atomic scale. The `group_in_#.id` files are renamed from the `group_out_#.id` files that were created automatically in previous CAC simulations when the total number of groups > 0. `unitype` is irrevelant for this `refine_style`.

Related commands

This command becomes irrelevant when `boolean_restart_refine` = *f*.

Related files

`refine_init.f90`

Default

None.

restart

Syntax

```
restart boolean_restart boolean_restart_refine boolean_restart_group
```

- `boolean_restart` , `boolean_restart_refine` , `boolean_restart_group` = *t* or *f*

  ```
  t is true
  f is false
  ```

Examples

```
restart f f f
restart t f f
restart t t f
```

Description

This command sets the restart styles.

When `boolean_restart` = *t*, the code reads the elements/nodes/atoms information from the `cac_in.restart` file; otherwise, the simulation cell is built from scratch and both `boolean_restart_refine` and `boolean_restart_group` become *f* regardless of their styles in this command.

When `boolean_restart_refine` = *t*, some elements in the coarse-grained domain, are refined to atomic scale by linear interpolation from the nodal positions. Which elements to be refined depend on the `refine_style` .

When `boolean_restart_group` = *t*, elements/nodes/atoms information of the restart group is read from `group_in_#.id` files, where `#` is an positive integer starting from `new_group_number` + 1. On the one hand, there cannot be fewer `group_in_#.id` files than `restart_group_number` ; on the other hand, any `group_in_#.id` file with `#` > `new_group_number` + `restart_group_number` is not read. Note that for the restart groups, the

controlled displacement is set in the group command, in which a syntax different from that for the new groups is used. When `boolean_restart_group` = *f*, `restart_group_number` becomes 0.

Related commands

When `boolean_restart_refine` = *f*, the refine command becomes irrelevant.

When `boolean_restart_group` = *t*, the group_num and group commands provide the restart group number and the controlled displacement information, respectively.

Related files

`read_restart.f90` and `write_restart.f90`

Default

```
restart f f f
```

run

Syntax

```
run total_step time_step
```

- `total_step` = non-negative integer
- `time_step` = positive real number

Examples

```
run 10000 0.002
```

Description

This command sets the total step and time step of a CAC simulation.

`total_step` is the total time step of dynamic CAC, the total loading increment of quasistatic CAC, or the total time step of the dynamic part in hybrid CAC.

`time_step`, in unit of ps, is the time step in dynamic CAC simulations or the dynamic part in hybrid CAC simulations.

Related commands

`time_step` becomes irrelevant when simulation_style = *statics*.

When `boolean_restart` = *t*, the `total_step` is added to the time stamp read from the `cac_in.restart` file, instead of overriding it.

Related files

`dynamics_init.f90`, `dynamics.f90`, and `hybrid.f90`.

Default

```
run 0 0.002
```

simulator

Syntax

```
simulator simulation_style
```

- `simulation_style` = *dynamics* or *statics* or *hybrid*

Examples

```
simulator dynamics
simulator hybrid
```

Description

This command sets the `simulation_style` in PyCAC simulations: *dynamics* (dynamic CAC), *statics* (quasistatic CAC), or *hybrid* (dynamic CAC with periodic energy minimization). The former two `simulation_style` have different schemes.

Related commands

More style information for the PyCAC are set in the dynamics and minimize commands.

Related files

`dynamics.f90`, `quasi_statics.f90`, and `hybrid.f90`

Default

```
simulator dynamics
```

subdomain

Syntax

```
subdomain {grain_id subdomain_number}
```

- `grain_id`, `subdomain_number` = positve integer

Examples

```
subdomain {1 1}
subdomain {1 2} {2 3}
subdomain {1 1 2 1 3 1}
```

Description

This command sets the number of subdomains in each grain.

In CAC, a unit is either the primitive unit cell of the lattice (for the atomistic domain) or a finite element (for the coarse-grained domain). Finite elements of different sizes are different types of unit. In a CAC simulation cell, each spatial region consisting of the same type of unit is a subdomain, as illustrated in the figure below:

The size of and the unit type in each subdomain in each grain is specified in the unit_num and unit_type commands, respectively. The three examples above correspond to the three examples in the unit_num and unit_type commands:

- In the first example, there is one grain designated by the first *1*, which has one subdomain designated by the second *1*.
- In the second example, there are two grains: the first grain has two subdomains designated by the first *2*, the second grain has three subdomains designated by *3*.
- In the third example, there are three grains, each of which has one subdomain, designated by the second *1*, the third *1*, and the fourth *1*, respectively.

The maximum `grain_id` must be larger than or equal to `grain_number`. All information related to `grain_id` that is larger than `grain_number` is discarded.

Related commands

In the unit_num and unit_type commands, the maximum `subdomain_id` in each grain must equal the corresponding `subdomain_number`.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`box_init.f90`

Default

```
subdomain 1 1
```

temperature

Syntax

```
temperature temp
```

- `temp` = positive real number

Examples

```
temperature 10.
temperature 300.
```

Description

This command sets the temperature for the dynamic and hybrid PyCAC simulations, in unit of K. In quasi-static simulations, the temperature is effectively 0.

Related commands

A constant temperature is maintained in the system only when `dyn_style` = *ld*. A warning will be issued if other `dyn_style` are used.

Related files

`ensemble.f90` , `langevin_dynamics.f90` , and `langevin_vel.f90`

Default

```
temperature 10.
```

dynamics

Syntax

```
dynamics dyn_style energy_min_freq damping_coefficient
```

- `dyn_style` = *ld* or *qd* or *vv*

  ```
  ld is Langevin dynamics
  qd is quenched dynamics
  vv is velocity Verlet
  ```

- `energy_min_freq` = positive integer
- `damping_coefficient` = positive real number

Examples

```
dynamics ld 300 1.
dynamics qd 500 5.
```

Description

This command sets the style of dynamic run in PyCAC simulations.

When `dyn_style` = *ld*, the Langevin dynamics is performed, i.e.,

$$m\ddot{\mathbf{R}} = \mathbf{F} - \gamma\dot{\mathbf{R}}$$

where m is the normalized lumped mass or the atomic mass, $\mathbf{R}$ is the nodal/atomic position, $\mathbf{F}$ is the equivalent nodal/atomic force, and γ is the `damping_coefficient`, in unit of ps^{-1}. In the PyCAC code, the Velocity Verlet form is employed, as given in Eqs. 1-3 in Xu et al., 2016. The velocity $\dot{\mathbf{R}}$ is updated in `langevin_vel.f90`.

Note that the *ld* style is used to keep a constant temperature in PyCAC simulations by adding to the force a normal random variable with the mean zero and the deviation $\sqrt{2m\gamma k_{\mathrm{B}}T/\Delta t}$, where m is the atomic mass, k_{B} is the Boltzmann constant ($8.6173324 \times 10^{-5}\mathrm{eV/K}$), T is the temperature in unit of K, and Δt is the time step in unit of ps. The random variable is calculated and added to the force in `langevin_force.f90`.

When `dyn_style` = *qd*, the quenched dynamics is performed, in which

- if the force and velocity point in opposite directions, the velocity is zeroed, i.e.,

$$\text{if } \dot{\mathbf{R}} \cdot \mathbf{F} < 0, \dot{\mathbf{R}} = 0$$

- otherwise, the velocity is projected along the direction of the force, such that only the component of velocity parallel to the force vector is used, i.e.,

$$\text{if } \dot{\mathbf{R}} \cdot \mathbf{F} \geq 0, \dot{\mathbf{R}} = \frac{(\dot{\mathbf{R}} \cdot \mathbf{F})\mathbf{F}}{|\mathbf{F}|^2}$$

Note that with the *qd* style, which was first used in Xu et al., 2016, the temperature is considered 0 K or very nearly so.

When `dyn_style` = *vv*, dynamic simulation follows the Velocity Verlet scheme. Note that the *vv* style cannot be used to keep a constant temperature; a warning will be issued if the user tries to do so.

The `energy_min_freq` is the frequency with which the energy minimization is performed during a dynamic run. This is relevant only if the simulator_style is `hybrid`.

Related commands

run and simulator.

Related files

`dynamics_init.f90`, `dynamics.f90`, `langevin_dynamics.f90`, `quenched_dynamics.f90`, `hybrid.f90`, among many

Default

```
dynamics vv 500 1.
```

unit_num

Syntax

```
unit_num {grain_id [subdomain_id x unit_num_x y unit_num_y z unit_num_z]}
```

- `grain_id`, `subdomoain_id` = positive integer
- `unit_num_x`, `unit_num_y`, `unit_num_z` = positive integer

Examples

```
unit_num {1 [1 x 2 y 3 z 4]}
unit_num {1 [1 x 8 y 20 z 12] [2 x 40 y 2 z 60]} {2 [1 x 40 y 1 z 60] [2 x 8 y 25 z 12
] [3 x 6 y 7 z 10]}
unit_num {1 [1 x 2 y 3 z 4]} {2 [1 x 6 y 1 z 2]} {3 [1 x 10 y 2 z 3]}
```

Description

This command sets the size of each subdomain along three directions in each grain. The `unit_num_x`, `unit_num_y`, and `unit_num_z` are in unit of the `x`, `y`, and `z` length of the projection of the unit (primitive unit cell in the atomistic domain or the finite element in the coarse-grained domain) on the `yz`, `xz`, and `xy` planes, respectively.

Similar to the unit_type command, this command consists of two loops. The outer loop, illustrated by `{}`, is based on grain; the inner loop, illustrated by `[]`, is based on subdomain. Note that the curly brackets `{` and `}` as well as the square brackets `[` and `]` in the syntax/examples are to separate different subdomains and grains, the number of which are `subdomain_number` and `grain_number`, respectively; all brackets should not be included in preparing `cac.in`.

When `grain_number` > 1 and/or `subdomain_number` > 1, the size of each subdomain set directly by this command is most likely not the same, which may be problematic in some cases, e.g., in a bicrystal, as shown in Fig. (a) below. Assume the grain stack direction is *x*,

the CAC code will then increase the size of all subdomains along both *y* and *z* directions to match the subdomain(s) with the largest *y* and *z* length, respectively, as shown in Fig. (b) below.

The three examples above correspond to the three examples in the subdomain command.

The maximum `grain_id` must be larger than or equal to `grain_number`. All information related to `grain_id` that is larger than `grain_number` is discarded.

Related commands

Within each grain, the maximum `subdomain_id` must equal the corresponding subdomain_number.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`box_init.f90` and `model_init.f90`

Default

None.

unit_type

Syntax

```
unit_type {grain_id [subdomain_id unitype]}
```

- `grain_id`, `subdomain_id` = positive integer
- `unitype` = 1 or positive even integer (>= 4)

Examples

```
unit_type {1 [1 12]}
unit_type {1 [1 1] [2 8]} {2 [1 6] [2 16] [3 10]}
unit_type {1 [1 14]} {2 [1 1]} {3 [1 6]}
```

Description

The command sets the `unitype` in each subdomain in each grain.

Similar to the unit_num command, this command consists of two loops. The outer loop, illustrated by `{}`, is based on grain; the inner loop, illustrated by `[]`, is based on subdomain. Note that the curly brackets `{` and `}` as well as the square brackets `[` and `]` in the syntax/examples are to separate different subdomains and grains, the number of which are `subdomain_number` and `grain_number`, respectively; all brackets should not be included in preparing `cac.in`.

The number of atoms per unit is $(\mathrm{unitype}+1)^3$, where `unitype` must be either 1 (atomistic domain) or an even integer that is no less than 4 (coarse-grained domain): in the latter case, (i) it must be even because the first order Gaussian quadrature is employed in the PyCAC code to solve the governing equations, (ii) it must be >= 4 because the second nearest neighbor (2NN) element with 125 integration points is employed and so there cannot be fewer than 125 atoms in one element. For more information of the 2NN element and the Gaussian quadrature implementation, read Appendices A and B of Xu et al., 2015.

The three examples above correspond to the three examples in the subdomain command:

- In the first example, there is only one grain, designated by the first *1*, having only one subdomain, designated by the second *1*, with the `unitype` = *12*.
- In the second example, there are two grains, designated by the first *1* and the second *2*, respectively. The first grain has two subdomains: the first is atomistics because `unitype` = *1*; the second contains elements each of which has $(8+1)^3 = 729$ atoms. The second grain has three subdomains: the first contains elements each of which has $(6+1)^3 = 343$ atoms; the second contains elements each of which has $(16+1)^3 = 4913$ atoms; the third contains elements each of which has $(10+1)^3 = 1331$ atoms.
- In the third example, there are three grains, each of which contains one unit type. Note that the second grain is atomistics because `unitype` = *1*.

The maximum `grain_id` must be larger than or equal to `grain_number`. All information related to `grain_id` that is larger than `grain_number` is discarded.

Related commands

Within each grain, the maximum `subdomain_id` must equal the corresponding subdomain_number.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`model_init.f90`

Default

None.

zigzag

Syntax

```
zigzag boolean_x boolean_y boolean_z
```

- `boolean_x`, `boolean_y`, `boolean_z` = *t* or *f*

  ```
  t is true
  f is false
  ```

Examples

```
zigzag t f f
zigzag t t t
```

Description

This command decides whether the simulation cell boundaries are left zigzagged along the *x*, *y*, and *z* directions, respectively.

Due to the rhombohedral shape of the finite elements in the coarse-grained domain, the simulation cell mostly likely has zigzagged boundaries, as shown in Fig. C27(a) of Xu et al., 2015. On the other hand, flat boundaries are sometimes desirable to enforce the periodic boundary conditions or to lower the aphysical stress concentrations at the boundaries.

If one of the three booleans in this command is *f*, atoms will be filled in the jagged interstices, resulting in flat boundaries for the corresponding direction, as shown in ig. C27(b) of Xu et al., 2015, unless the boundaries were already flat with rhomboheral elements, e.g., on {111} planes in an FCC and on {110} planes in a BCC lattice. If a certain boolean is *t*, no atoms will be filled in at the boundaries.

Related commands

When a boundary is periodic, the corresponding zigzag boolean becomes *f*, regardless of what is set in this command, because the periodic boundaries must be flat in the current code.

This command becomes irrelevant when `boolean_restart` = *t*.

Related files

`model_init.f90`

Default

```
zigzag t t t
```

Post-processing

A CAC simulation outputs a lot of files, most of which are `*.vtk` and `dump.*` files that can be visualized and analyzed using OVITO and ParaView, respectively. As of June 2017, the latest versions of these two software, OVITO 2.8.2 and ParaView 5.4, are compatiable with the CAC results.

The stress-strain curve and the simulation step-temperature curve can be plotted by processing the `stress_strain` and `temperature` files, respectively, using common graphing software such as MATLAB, Octave, Origin, SigmaPlot, and gnuplot.

OVITO

In a CAC simulation, a series of `dump.#` files, containing the positions of the atoms (both the real atoms in the atomistic domain and the interpolated atoms in the coarse-grained domain), are created on-the-fly, with a frequency of `output_freq`. A `dump.lammps` file which, in addition to the nodal/atomic positions, may also contain the nodal/atomic velocities information if `simulation_style` = *dynamics* or *hybrid*, is also created in the beginning of the simulation. All these `dump.*` files can be read and analyzed by OVITO --- The Open Visualization Tool, which provides a variety of analyses.

A common usage of OVITO to process the `dump.*` files is to visualize the dislocations. First, import any `dump.#` file into OVITO. Then load the Dislocation analysis (DXA) modifier and deselect the Particles in Display. This approach applies to both the FCC and BCC metals.

To visualize lattice defects other than dislocations, e.g., stacking faults, twin boundaries, other modifiers. For FCC metals, the Common neighbor analysis modifier can be loaded, followed by that selected FCC particles are deleted to visualize the defects. For BCC metals, the Centrosymmetry parameter modifier can be loaded, then atoms with a large Centrosymmetry parameter are selected and deleted to visualize the defects.

ParaView

In a CAC simulation, a series of `cac_cg_#.vtk` and `cac_atom_#.vtk` files, containing the nodal/atomic position/energy/force/stress information, are created on-the-fly, with a frequency of `output_freq`. A `model_cg.vtk` file, a `model_atom.vtk`, and possibly some `group_cg_#.vtk` and `group_atom_#.vtk` files (when the total number of new group, restart group, and boundary group > 0) are also created in the beginning of the simulation. All these `*.vtk` files, with the legacy formats as opposed to the XML formats, can be read and analyzed by ParaView, which provides a variety of analyses. In most cases, a CAC simulation cell contains both the atomistic and coarse-grained domain, and so a pair of `cac_cg_#.vtk` and `cac_atom_#.vtk` files (with the same integer `#`) should be loaded into ParaView at the same time.

Example problems

The PyCAC distribution includes an examples sub-directory with five sample problems:

- Dislocation migration across the atomistic/coarse-grained domain interface
- Screw dislocation cross-slip
- Dislocation multiplication from a Frank-Read source
- Dislocation/obstacle interactions
- Dislocation/stacking fault interactions
- Dislocation/coherent twin boundary interactions

Dislocation migration across the atomistic/coarse-grained domain interface

FCC Cu, Mishin EAM potential, 2197 atoms per element in the coarse-grained domain. Results published in Fig. 15 of Xu et al., 2015. A smaller model of is adopted here.

60° mixed type dislocation migration from the atomistic domain to the coarse-grained domain

The movie below is produced using CAC input file and rendered by OVITO. log file log/at2cg.in

60° mixed type dislocation migration from the coarse-grained domain to the atomistic domain

The movie below is produced using CAC input file and rendered by OVITO.

Screw dislocation cross-slip

In PyCAC, global variables are defined by 54 module files `*_module.f90` in the `module` directory. There are five types of module files:

Dislocation multiplication from a Frank-Read source

In PyCAC, global variables are defined by 54 module files `*_module.f90` in the `module` directory. There are five types of module files:

Dislocation/obstacle interactions

FCC Ni, Mishin EAM potential, 2197 atoms per element in the coarse-grained domain.

Dislocation/void interactions

Void radius = 5 nm. The movie below is produced using CAC Input file and rendered by OVITO.

Dislocation/precipitate interactions

Precipitate radius = 5 nm. The movie below is produced using CAC Input file and rendered by OVITO.


00:00 / 00:00

Dislocation/stacking fault interactions

In PyCAC, global variables are defined by 54 module files `*_module.f90` in the `module` directory. There are five types of module files:

Dislocation/coherent twin boundary interactions

In PyCAC, global variables are defined by 54 module files `*_module.f90` in the `module` directory. There are five types of module files:

Miscellanies

This chapter provides miscellaneous information that is important but does not fit into other chapters.

ParaView

In PyCAC, global variables are defined by 54 module files `*_module.f90` in the `module` directory. There are five types of module files:

ParaView

In PyCAC, global variables are defined by 54 module files `*_module.f90` in the `module` directory. There are five types of module files:

ParaView

In PyCAC, global variables are defined by 54 module files `*_module.f90` in the `module` directory. There are five types of module files: