

PyCAC User's Manual

The concurrent atomistic-continuum simulation environment

<http://www.pycac.org>



Copyright (c) 2017 Georgia Institute of Technology. All rights reserved.

Table of Contents

Cover	1.1
Introduction	1.2
PyCAC features and non-features	1.2.1
Compilation and execution	1.2.2
Running PyCAC on MATIN	1.2.3
Publications	1.2.4
Acknowledgements and citations	1.2.5
Background	1.3
Atomistic field theory	1.3.1
A brief history of CAC	1.3.2
Algorithm	1.4
Scheme	1.4.1
Parallelization	1.4.2
Arithmetic precision	1.4.3
Units	1.4.4
Input	1.4.5
Output	1.4.6
Python scripting interface	1.5
input.py()	1.5.1
output.py()	1.5.2
visualization.py()	1.5.3
Command	1.6
boundary	1.6.1
box_dir	1.6.2
cal	1.6.3

constrain	1.6.4
convert	1.6.5
debug	1.6.6
deform	1.6.7
dump	1.6.8
dynamics	1.6.9
element	1.6.10
fix	1.6.11
grain_dir	1.6.12
grain_mat	1.6.13
grain_move	1.6.14
grain_num	1.6.15
group_num	1.6.16
group	1.6.17
lattice	1.6.18
limit	1.6.19
mass	1.6.20
minimize	1.6.21
modify_num	1.6.22
modify	1.6.23
neighbor	1.6.24
potential	1.6.25
refine	1.6.26
restart	1.6.27
run	1.6.28
simulator	1.6.29
subdomain	1.6.30
temperature	1.6.31

unit_num	1.6.32
unit_type	1.6.33
zigzag	1.6.34
Post-processing	1.7
OVITO	1.7.1
ParaView	1.7.2
Example problems	1.8
Stationary dislocations	1.8.1
Dislocation migration	1.8.2
Screw dislocation cross-slip	1.8.3
Dislocation multiplication	1.8.4
Dislocation/obstacle interactions	1.8.5
Dislocation/stacking fault interactions	1.8.6
Dislocation/coherent twin boundary interactions	1.8.7
Miscellanies	1.9
element vs node	1.9.1
lattice periodicity length	1.9.2
processor rank	1.9.3
shared elements	1.9.4
EAM potential	1.9.5
Code developer guide	1.10
atom and atomap	1.10.1

PyCAC User's Manual

Aug 28 2017 version

Copyright (c) 2017-2018 Georgia Institute of Technology. All Rights Reserved.

PyCAC, the concurrent atomistic-continuum (CAC) simulation environment, is a software suite that allows users to run CAC simulations and analyze data.

Currently, the CAC simulator and analyzer are written in Fortran 2008, with different parts of the workflow glued by a Python scripting interface.

A pdf version of this manual can be downloaded [here](#).

This user's manual is maintained by Shuzhi Xu, a former Ph.D. student (08/2011-12/2016) and Postdoctoral Fellow (01/2017-06/2017) with [Prof. David L. McDowell](#) in the [School of Mechanical Engineering](#) at the [Georgia Institute of Technology](#). Shuzhi is [now at UC Santa Barbara](#) and can be reached at shuzhixu@ucsb.edu for any questions about this manual.

If you are interested in the PyCAC source code, please [email Prof. David L. McDowell](#).

Introduction

The concurrent atomistic-continuum (CAC) method is a partitioned-domain multiscale modeling technique that is applicable to nano/micro-scale thermo/mechanical problems in a wide range of monatomic and polyatomic crystalline materials. A CAC simulation model, in general, partitions the simulation cell into two domains: a coarse-grained domain and an atomistic domain. Distinct from most concurrent multiscale methods in the literature, CAC employs a unified atomistic-continuum integral formulation with elements that have discontinuities between them; also, the underlying interatomic potential is 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, as shown in the figure below.



Figure 1. A 2-D CAC simulation domain consisting of an atomistic domain (right) and a

coarse-grained domain (left) (Xu et al.). The atomistic domain is composed of atoms (black circles), which follow the same governing equations in the atomistic simulation. The coarse-grained domain consists of discontinuous elements of varying size, each of which contains a large number of underlying atoms with the nodes (red circles) as the only degrees of freedom. Only the force/energy on integration points (green circles) and nodes are calculated. In (a), an edge dislocation (red \perp) is located in the atomistic domain. Upon applying a shear stress on the simulation cell, the dislocation migrates into the coarse-grained domain in (b), where the Burgers vector spreads out between discontinuous elements.



Figure 2. In 3-D, elements have faces on $\{111\}$ planes and $\{110\}$ planes in an FCC and a BCC lattice, respectively. The positions of atoms within each element (open circles) are interpolated from the nodal positions (red filled circles).

In a (big) nutshell, 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 the atomistic and coarse-grained domains.

Introduction

PyCAC features and non-features

Features

The PyCAC code can simulate monatomic pure face-centered cubic (FCC) or pure body-centered cubic (BCC) metals using the Lennard-Jones (LJ) or the embedded-atom method (EAM) potentials in a constant temperature field. In the coarse-grained domain, 3D trilinear rhombohedral elements are employed to accommodate 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, current version of the PyCAC code has not yet been extended to 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, or modified EAM (MEAM) potential;
- 1D or 2D materials that require 1D or 2D elements, respectively, as well as materials requiring 3D elements different from the rhombohedral ones;
- multicomponent, multi-constituent, multiphase, or polyatomic crystalline materials, e.g., alloys, intermetallics, composite materials, ceramic, mineral;
- materials in a non-constant temperature field.

Moreover, the [adaptive mesh refinement scheme](#) is not implemented in the current PyCAC code.

Compilation and execution

MPI

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

Fortran compiler

Some intrinsic functions in Fortran 2008 is employed in the code, so compilers that fully support Fortran 2008 are preferred. For example, [GNU Fortran](#) version 7.0 and [Intel Fortran](#) version 17.0 work with the PyCAC code.

To compile the code, simply run the `install.sh` file in the PyCAC code package, i.e.,

```
./install.sh
```

Note that the compilation process has not been tested on [Microsoft Windows](#). On [macOS](#), a message

```
/opt/local/bin/ranlib: file: libcac.a(constant_para_module.o) has no symbols
```

may occur. The users are suggested to compile and run the PyCAC code on [Linux](#), which [dominates the high performance computing systems](#).

Compilation

Module

In compilation, the first step is to create a static library `libcac.a` from the 56 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 the integer and real numbers.

```
*_para_module.f90
```

There are 25 `*_para_module.f90` files. They define single value variables that may be used globally.

```
*_array_module.f90
```

There are 24 `*_array_module.f90` files. They define arrays that may 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 algebra calculations, unit conversion, etc.

```
*_tab_module.f90
```

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

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

Subroutine

Then, an executable, named `cac`, is compiled using one main program (`main.f90`) plus 175 subroutines (`*.f90`) in the `src` directory and linked with the static library `libcac.a`.

Execution

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 positive integer `num_of_proc` is the number of processors to be used. As an example, see the `run.sh` file in the PyCAC code package.

The users may run the PyCAC code on the [MATerials Innovation Network \(MATIN\)](#) at Georgia Tech. Instructions will be added later.

Running PyCAC on MATIN

The users may run the PyCAC code on the [MATerials Innovation Network \(MATIN\)](#) at Georgia Tech. Instructions will be added later.

Publications

Book chapters

1. Shuzhi Xu, Ji Rigelesaiyin, Liming Xiong, Youping Chen, David L. McDowell, Generalized continua concepts in coarse-graining atomistic simulations, in *Generalized Models and Non-Classical Approaches in Complex Materials* (ed: Holm Altenbach, Joël Pouget, Martine Rousseau, Bernard Collet, Thomas Michelitsch), Switzerland: Springer International Publishing, 2018 (accepted)
2. Shengfeng Yang, Youping Chen, [Concurrent atomistic-continuum simulation of defects in polyatomic ionic materials](#), in *Multiscale Materials Modeling for Nanomechanics* (ed: Christopher R. Weinberger, Garrett J. Tucker), Switzerland: Springer International Publishing, 2016
3. 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. Xiang Chen, [A concurrent atomistic-continuum study of phonon transport in crystalline materials with microstructures](#), *Ph.D. Dissertation*, University of Florida, 2016
2. Shuzhi Xu, [The concurrent atomistic-continuum method: Advancements and applications in plasticity of face-centered cubic metals](#), *Ph.D. Dissertation*, Georgia Institute of Technology, 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. Shuzhi Xu, Thomas G. Payne, Hao Chen, Yongchao Liu, Liming Xiong, Youping Chen, David L. McDowell. PyCAC: The concurrent atomistic-continuum simulation environment, *J. Mater. Res.* (accepted)
2. Hao Chen, Shuzhi Xu, Weixuan Li, Ji Rigelesaiyin, Thanh Phan, Liming Xiong, A spatial decomposition parallel algorithm for a concurrent atomistic-continuum simulator and its preliminary applications, *Comput. Mater. Sci.* 144 (2018) 1-10
3. Xiang Chen, Adrian Diaz, Liming Xiong, David L. McDowell, Youping Chen. Passing waves from atomistic to continuum, *J. Comput. Phys.* 354 (2018) 393-402
4. Xiang Chen, Weixuan Li, Adrian Diaz, Yang Li, Youping Chen, David L. McDowell. Recent progress in the concurrent atomistic-continuum method and its application in phonon transport, *MRS Comm.* 7 (2017) 785-797
5. Xiang Chen, Weixuan Li, Liming Xiong, Yang Li, Shengfeng Yang, Zexi Zheng, David L. McDowell, Youping Chen. Ballistic-diffusive phonon heat transport across grain boundaries, *Acta Mater.* 136 (2017) 355-365
6. 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
7. Shuzhi 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
8. Shuzhi 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
9. Shuzhi 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
10. Shuzhi 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
11. Shuzhi 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
12. Shuzhi 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
13. Shuzhi 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
14. Liming Xiong, Ji Rigelesaiyin, Xiang Chen, Shuzhi Xu, David L. McDowell, Youping Chen. Coarse-grained elastodynamics of fast moving dislocations, *Acta Mater.* 104 (2016) 143-155
15. Shengfeng Yang, Ning Zhang, Youping Chen. Concurrent atomistic-continuum simulation of polycrystalline strontium titanate, *Philos. Mag.* 95 (2015) 2697-2716
16. Shuzhi 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
17. Shengfeng Yang, Youping Chen. Concurrent atomistic and continuum simulation of bicrystal strontium titanate with tilt grain boundary, *Proc. R. Soc. A* 471 (2015) 20140758
18. Liming Xiong, Shuzhi Xu, David L. McDowell, Youping Chen. Concurrent atomistic-continuum simulations of dislocation-void interactions in fcc crystals, *Int. J. Plast.* 65 (2015) 33-42
19. 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
20. 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
21. Qian Deng, Youping Chen, A coarse-grained atomistic method for 3D dynamic fracture simulation, *J. Multiscale Comput. Eng.* 11 (2013) 227-237
22. Shengfeng Yang, Liming Xiong, Qian Deng, Youping Chen. Concurrent atomistic and continuum simulation of strontium titanate, *Acta Mater.* 61 (2013) 89–102
23. 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
24. Liming Xiong, Qian Deng, Garrett 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
25. Liming Xiong, Youping Chen. Coarse-grained atomistic modeling and simulation of inelastic material behavior, *Acta Mech. Solida Sin.* 25 (2012) 244-261
26. Liming Xiong, Qian Deng, Garrett Tucker, David L. McDowell, Youping Chen. A concurrent scheme for passing dislocations from atomistic to continuum domains, *Acta Mater.* 60 (2012) 899-913
27. Liming Xiong, Garrett Tucker, David L. McDowell, Youping Chen. Coarse-grained

- atomistic simulation of dislocations, *J. Mech. Phys. Solids* 59 (2011) 160-177
28. 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. The origin of the distinction between microscopic formulas for stress and Cauchy stress, *Europhys. Lett.* 116 (2016) 34003
2. Youping Chen and Adrian Diaz. Local momentum and heat fluxes in transient transport processes and inhomogeneous systems, *Phys. Rev. E* 94 (2016) 053309
3. Youping Chen, Jonathan Zimmerman, Anton Krivtsov, David L. McDowell. Assessment of atomistic coarse-graining methods, *Int. J. Eng. Sci.* 49 (2011) 1337-1349
4. Youping Chen. Reformulation of microscopic balance equations for multiscale materials modeling, *J. Chem. Phys.* 130 (2009) 134706
5. Youping Chen, James Lee, Liming Xiong. A generalized continuum theory and its relation to micromorphic theory, *J. Eng. Mech.* 135 (2009) 149-155
6. Youping Chen. Local stress and heat flux in atomistic systems involving three-body forces, *J. Chem. Phys.* 124 (2006) 054113
7. Youping Chen, James Lee. Conservation laws at nano/micro scales, *J. Mech. Mater. Struct.* 1 (2006) 681-704
8. Youping Chen, James Lee, Liming Xiong. Stresses and strains at nano/micro scales, *J. Mech. Mater. Struct.* 1 (2006) 705-723
9. Youping Chen, James Lee. Atomistic formulation of a multiscale theory for nano/micro physics, *Philos. Mag.* 85 (2005) 4095-4126
10. Youpig Chen, James D. Lee, Azim Eskandarian. Micropolar theory and its applications to mesoscopic and microscopic problems, *Comput. Modeling Eng. Sci.* 5 (2004) 35-43
11. Youpig Chen, James D. Lee, Azim Eskandarian, Atomistic counterpart of micromorphic theory, *Acta Mech.* 161 (2003) 81-102

Acknowledgements and citations

The CAC simulator and analyser was written in Fortran by [Shuzhi Xu](#); the [Python scripting interface](#) was written by [Alex Selimov](#) and [Thomas G. Payne](#); the code in its present form is a [culmination of developments over the last decades](#).

The PyCAC code development was sponsored 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
- [Institute for Materials, Georgia Institute of Technology](#)

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

- Shuzhi Xu, Thomas G. Payne, Hao Chen, Yongchao Liu, Liming Xiong, Youping Chen, David L. McDowell. PyCAC: The concurrent atomistic-continuum simulation environment, *J. Mater. Res.* (accepted)
- Shuzhi 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, Garrett Tucker, David L. McDowell, Youping Chen. [Coarse-grained atomistic simulation of dislocations](#), *J. Mech. Phys. Solids* 59 (2011) 160-177
- Youping Chen. [Reformulation of microscopic balance equations for multiscale materials modeling](#), *J. Chem. Phys.* 130 (2009) 134706

as well as the website www.pycac.org.

Background

This chapter, written by [Prof. Youping Chen](#), presents the atomistic field theory and a brief history of CAC.

Atomistic field theory

The theoretical foundation of the CAC method is the atomistic field theory (AFT) [1,2], which is an extension of the Irving Kirkwood (IK)'s non-equilibrium statistical mechanical formulation of "*the hydrodynamics equations for a single component, single phase system*" [3] to a two-level structural description of crystalline materials. It employs the two-level structural description of all crystals in solid state physics, i.e., the well known equation of "crystal structure = lattice + basis" [4]. As a result of the bottom-up atomistic formulation, all the essential atomistic information of the material, including the crystal structure and the interaction between atoms, are built in the formulation. A schematic of micromorphic theory and AFT is given below.



Macro- and micro-motions of a material particle P in (a) micromorphic theory and (b) AFT. Left in (a) and (b) is the reference state at time 0 while right is the deformation state at time t . \mathbf{X} and \mathbf{x} are the positions of the mass center of the unit cell, Ξ and ξ are internal positions, \mathbf{Y} and \mathbf{y}^α are positions of atom α with respect to \mathbf{X} and \mathbf{x} , respectively, N_a is the number of atoms in a unit cell.

The governing equations for conservative systems

The result is a concurrent atomistic-continuum representation of balance laws for both atomistic and continuum coarse-grained domains in the following form [1,2]:

$$\frac{d\rho^\alpha}{dt} + \rho^\alpha (\nabla_{\mathbf{x}} \cdot \mathbf{v} + \nabla_{\mathbf{y}^\alpha} \cdot \Delta \mathbf{v}^\alpha) = 0$$

$$\rho^\alpha \frac{d}{dt} (\mathbf{v} + \Delta \mathbf{v}^\alpha) = \nabla_{\mathbf{x}} \cdot \mathbf{t}^\alpha + \nabla_{\mathbf{y}^\alpha} \cdot \boldsymbol{\tau}^\alpha + \mathbf{f}_{\text{ext}}^\alpha$$

$$\rho^\alpha \frac{d\mathbf{e}^\alpha}{dt} = \nabla_{\mathbf{x}} \cdot \mathbf{q}^\alpha + \nabla_{\mathbf{y}^\alpha} \cdot \mathbf{j}^\alpha + \mathbf{t}^\alpha : \nabla_{\mathbf{x}} (\mathbf{v} + \Delta \mathbf{v}^\alpha) + \boldsymbol{\tau}^\alpha : \nabla_{\mathbf{y}^\alpha} (\mathbf{v} + \Delta \mathbf{v}^\alpha)$$

where \mathbf{x} is the physical space coordinate; $\mathbf{y}^\alpha (\alpha = 1, 2, \dots, N_a)$ with N_a being the total number of atoms in a unit cell) are the internal variables describing the position of atom α relative to the mass center of the lattice cell located at \mathbf{x} ; ρ^α , $\rho^\alpha(\mathbf{v} + \Delta\mathbf{v}^\alpha)$, and $\rho^\alpha e^\alpha$ are the local densities of mass, linear momentum and total energy, respectively; $\mathbf{v} + \Delta\mathbf{v}^\alpha$ is the atomic-level velocity and \mathbf{v} is the velocity field; $\mathbf{f}_{\text{ext}}^\alpha$ is the external force field; \mathbf{t}^α and \mathbf{q}^α are the stress and heat flux due to the homogeneous deformation of lattice, respectively; τ^α and \mathbf{j}^α are the stress and heat flux due to the reorganizations of atoms within the lattice cells, respectively.

For monatomic crystals, which PyCAC can simulate, $\mathbf{y}^\alpha = \mathbf{0}$ and $N_a = 1$; the governing equations reduce to

$$\frac{d\rho}{dt} + \rho \nabla_{\mathbf{x}} \cdot \mathbf{v} = 0$$

$$\rho \frac{d\mathbf{v}}{dt} = \nabla_{\mathbf{x}} \cdot \mathbf{t} + \mathbf{f}_{\text{ext}}$$

$$\rho \frac{de}{dt} = \nabla_{\mathbf{x}} \cdot \mathbf{q} + \mathbf{t} : \nabla_{\mathbf{x}} \mathbf{v}.$$

For conservative systems, i.e., a system in the absence of an internal source that generates or dissipates energy, the AFT energy equation is equivalent to the AFT linear momentum equation. Because of its [current features](#), only the first two governing equations are explicitly implemented into PyCAC. Employing the classical definition of kinetic temperature, which is proportional to the kinetic part of the atomistic stress, the linear momentum equations can be expressed in a form that involves the internal force density $\mathbf{f}_{\text{int}}^\alpha$ and temperature T [5-7],

$$\rho^\alpha \ddot{\mathbf{u}}^\alpha + \frac{\gamma^\alpha k_B}{\Delta V} \nabla_{\mathbf{x}} T = \mathbf{f}_{\text{int}}^\alpha + \mathbf{f}_{\text{ext}}^\alpha, \quad \alpha = 1, 2, \dots, N_a$$

where \mathbf{u}^α is the displacement of the α th atom at point \mathbf{x} ; the superposed dots denote the material time derivative; ΔV is the volume of the finite-sized material particle (the primitive unit cell for crystalline materials) at \mathbf{x} ; k_B is the Boltzmann constant; $\gamma^\alpha = m^\alpha / \sum_{\alpha=1}^{N_a} m^\alpha$,

and $\mathbf{f}_{\text{int}}^{\alpha}$ is the internal force density and is a nonlinear nonlocal function of relative atomic displacements.

For systems with a constant temperature field or a constant temperature gradient, the temperature term has the effect of a surface traction on the boundary or a body force in the interior of the material [6]. Denoting the temperature term in the above equation as $\mathbf{f}_T^{\alpha}(\mathbf{x})$ and the finite element shape function as $\Phi_{\xi}(\mathbf{x})$, the Galerkin weak form of the above equation can be written as

$$\int_{\Omega(\mathbf{x})} \Phi_{\xi}(\mathbf{x}) (\rho^{\alpha} \ddot{\mathbf{u}}^{\alpha}(\mathbf{x}) + \mathbf{f}_T^{\alpha}(\mathbf{x}) - \mathbf{f}_{\text{int}}^{\alpha}(\mathbf{x}) - \mathbf{f}_{\text{ext}}^{\alpha}(\mathbf{x})) d\mathbf{x} = 0$$

where $\Omega(\mathbf{x})$ is the simulation domain; the integrals can be evaluated using numerical integration methods such as Gaussian quadrature, leading to a set of discretized governing equations with the finite element nodal displacements as the unknowns to be solved. Note that in PyCAC, the $\mathbf{f}_T^{\alpha}(\mathbf{x})$ term has not yet been implemented as (i) the effect on mechanical properties in a constant temperature field is small and (ii) work is underway to compare different descriptions of temperature in the coarse-grained domain.

The accuracy, efficiency, and stability of the CAC simulator are then determined by the two approximations: the shape function and the numerical integration. Simulation results can be displayed in terms of finite elements, which can also be mapped back to atomic positions and be used to plot the atomic trajectories. With the only constitutive relation being the nonlocal atomic force-displacement relation, continuity between elements in the usual finite element method is not required. Consequently, nucleation and propagation of dislocations and/or cracks can be simulated via sliding and separation between finite elements.

AFT and the equilibrium ensembles

The local densities defined in the Irving and Kirkwood formulations are ensemble averaged point functions. The ensemble averaging was described by Irving and Kirkwood as "*repeating the observations many times*" [3]. In the early version of the AFT formulation [1], the local densities were also defined as ensemble averages and hence the governing equations were written in terms of ensemble-averaged local densities. In the later version of the AFT formulation [2], the local densities are instantaneous quantities, according to argument by Evans and Morris [8], who wrote "... *the reason for considering instantaneous*

expressions is two-fold. The fluxes are based upon conservation laws and these laws are valid instantaneously for every member of the ensemble. They do not require ensemble averaging to be true. Secondly, most computer simulation involves calculating system properties from a single system trajectory. Ensemble averaging is almost never used because it is relatively expensive in computer time".

Note that the AFT local densities and governing equations were derived as an extension of the Irving and Kirkwood's formulation of the equations of hydrodynamics. Consequently, they differ from other statistical mechanical formulations that follow the Gibbs' equilibrium statistical theory of ensembles. Popular equilibrium ensembles include (i) the microcanonical ensemble, which describes a systems isolated from its surroundings and governed by Hamilton's equations of motion (NVE), (ii) the canonical ensemble, which describe a system in constant contact with a heat bath of constant temperature (NVT), and (iii) the isothermal-isobaric ensembles, which describes systems in contact with a thermostat at temperature T and a barostat at pressure P (NPT) [9]. These ensembles, known as equilibrium ensembles and allowing a wide variety of thermodynamic and structural properties of systems to be computed, can be realized in [dynamic CAC](#), in which a constant temperature is kept via a [Langevin thermostat](#) while a constant pressure is maintained via a Berendsen barostat.

References

1. Youping Chen, James Lee. [Atomistic formulation of a multiscale theory for nano/micro physics](#), *Philos. Mag.* 85 (2005) 4095-4126
2. Youping Chen. [Reformulation of microscopic balance equations for multiscale materials modeling](#), *J. Chem. Phys.* 130 (2009) 134706
3. J.H. Irving, Jhon G. Kirkwood. [The statistical mechanical theory of transport processes. IV. The equations of hydrodynamics](#), *J. Chem. Phys.* 18 (1950) 817-829
4. Charles Kittel. [Introduction to Solid State Physics](#), 1956: John Wiley & Sons, Inc
5. Qian Deng, [Coarse-graining atomistic dynamics of fracture by finite element method: Formulation, parallelization and applications](#), *Ph.D. Dissertation*, University of Florida, 2011
6. Liming Xiong, Youping Chen. [Coarse-grained simulations of single-crystal silicon. Modelling Simul. Mater. Sci. Eng.](#) 17 (2009) 035002
7. Liming Xiong, Youping Chen, James D. Lee. [Atomistic simulation of mechanical properties of diamond and silicon carbide by a field theory](#), *Modelling Simul. Mater. Sci.*

Eng. 15 (2007) 535-551

8. Denis J. Evans, Gary P. Morriss. *Statistical Mechanics of Nonequilibrium Liquids*, 2008: Cambridge University Press
9. Mark E. Tuckerman. *Statistical Mechanics: Theory and Molecular Simulation*, 2010: Oxford University Press

A brief history of CAC

The CAC method is based on a concurrent atomistic-continuum formulation of balance laws [1,2] that are implemented using a finite element (FE) strategy, with the interatomic potential as the only constitutive relation. The formulation of the CAC balance equations, originally referred to as an atomic micromorphic theory or [atomistic field theory \(AFT\)](#) by [Youping Chen and James D. Lee](#) [1,3-8], is an extension of the Irving Kirkwood's formulation of "*the hydrodynamics equations for a single component, single phase system*" [9] to a two-level structural description of crystalline materials. The CAC formulation differs from that of classical continuum mechanics in that it employs a two-level structural description of materials. It is also distinct from the well-established theories of generalized continuum mechanics such as the Cosserat theory [10], micropolar theory [11,12], and micromorphic theory [13,14] in that the sub-level structural description is not continuous but discrete.

The first version of the CAC numerical tool was developed by [Liming Xiong \(Ph.D. 2011\)](#) and [Qian Deng \(Ph.D. 2011\)](#). The reformulated balance equations were numerically implemented using FE method with trilinear FE shape functions and nodal integration, and the simulation tool was demonstrated to be able to capture the phenomenon of phase transition in Si [15] and the dynamic processes of fracture, including crack initiation, propagation and branching (Fig. 1) [16,17].



Figure 1. Time sequence of CAC simulations of a brittle material ($2.24 \mu\text{m}$ by $1.4 \mu\text{m}$) showing (a) stress waves emitting from a propagating crack; (b) and (c) crack branching as a result of the interactions between waves propagating from the crack tip and those reflected from the specimen boundaries [17].

The form and capabilities of the CAC method were extended substantially as a direct result of collaborative efforts between [University of Florida](#) and [Georgia Tech](#) in modeling and simulations of the dynamics of dislocations: elements that have discontinuities between them were employed, and the Gaussian quadrature was used for integration in the coarse-grained domain. Nucleation and propagation of dislocations in the coarse-grained domain [18,19], passing dislocations from the atomistic domain to the coarse-grained domain [20], the growth of dislocation loops in Cu, Al and Si (Fig. 2(a)) [21], fast moving dislocations (Fig. 2(b)) [22], and other progresses [23-26], have been successfully simulated without special numerical treatment or supplemental constitutive relations. The name "CAC" for the methodology was coined by [David L. McDowell](#) in 2010.



Figure 2. (a) CAC simulation results of the nucleation and growth of dislocation loops in Cu, Al, Si [21]. (b) Time sequences of dislocation motions in MD and fully coarse-grained CAC simulations [22].

The CAC code was rewritten using Fortran 90 by [Shengfeng Yang \(Ph.D. 2014\)](#) for multiscale simulation of polycrystalline ionic materials. This is the second-generation of the CAC code. It employs the Wolf method to calculate the long-range Columbic force and a special type of element (i.e., an incomplete element) to model regions with defects such as

grain boundaries (GBs) in polyatomic materials. This version of the CAC code enables multiple meshing resolutions and simulation of two or more materials (e.g., Si and Ge), with multiple types of interatomic potentials, including the Buckingham and the Stillinger-Weber potentials. The CAC code was demonstrated to reproduce the equilibrium structures and energies of GBs in SrTiO_3 , in good agreement with those obtained from existing experiments and density functional theory calculations. The code has been used to study the dynamic processes of crack initiation as well as the evolution of dislocation in single crystalline [27], bicrystalline [28], and polycrystalline SrTiO_3 (Fig. 3) [29,30].

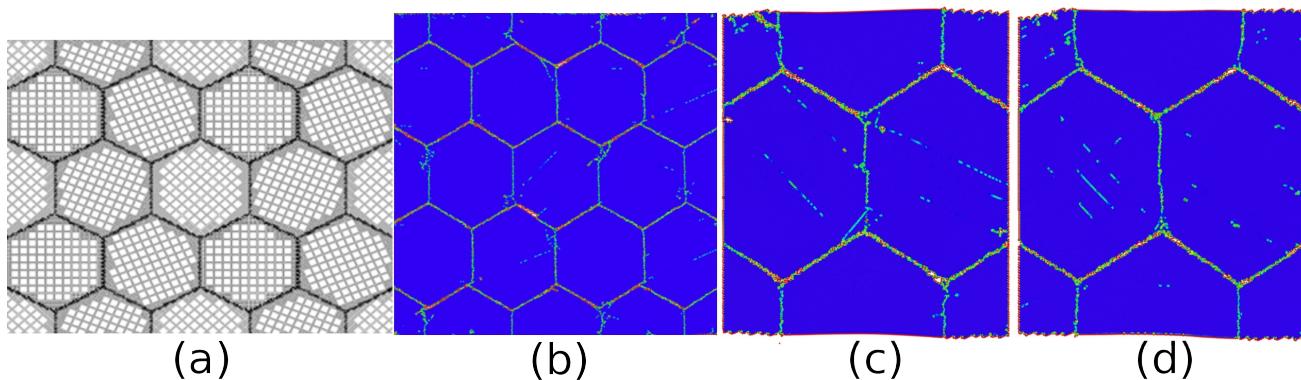


Figure 3. (a) A CAC model of polycrystalline SrTiO_3 (2D view) in which the GBs are modeled with atomic resolution and the grains with coarse-scale finite elements; (b) Central symmetry parameter plot of the deformed model showing the nucleation and propagation of many dislocations and their interactions with the GBs; The comparison between (c) CAC and (d) MD simulation results is shown at strain 8.7% [30].

The CAC code was also rewritten by [Shuzhi Xu \(Ph.D. 2016\)](#) using Fortran 2008. The code was optimized and the efficiency was significantly improved. The code includes the quasistatic version of CAC to carry out quasistatic simulations so as to obtain energy minimized atomic and nodal structures, in addition to dynamic simulations. Equipped with the [Python scripting interface](#) written by [Thomas G. Payne](#), this version of code, termed [PyCAC](#), has been well tested for ductile fracture [31], quasistatic dislocation migration [32], screw dislocation cross-slip [33], edge dislocations bowing out from obstacles [34], dislocation multiplication from Frank-Read sources [35], dislocation/stacking fault interactions [36], and dislocation/GB interactions (Fig. 4) [37,38].



Figure 4. Snapshots of dislocation pile-up with dominant leading screw character impinging against a $\Sigma 3\{111\}$ coherent twin boundary (CTB) [38]. In (a), five incoming dislocations approach the CTB subject to an applied shear stress. In (b), the leading dislocation is constricted at the CTB, where two Shockley partial dislocations are recombined into a full dislocation. In (c), with certain interatomic potentials, the dislocation effectively cross-slips into the outgoing twinned grain via redissociation into two partials. In (d), with different potentials, the redissociated dislocation is absorbed by the CTB, with two partials gliding on the twin plane in opposite directions.

Based on the Fortran 90 code, [Xiang Chen \(Ph.D. 2016\)](#) extended the CAC method for space- and time-resolved simulation of the transient processes of the propagation of heat pulses in single crystals and across GBs [39] as well as the interactions between heat pulses and moving dislocations (Fig. 5) [40]. A phonon representation of the heat pulses, termed a coherent phonon pulse model [41], was created to mimic the coherent lattice excitation achieved via ultrashort laser pulses, and was incorporated into the framework of CAC to provide a coupled treatment for defect dynamics and phonon thermal transport. A first attempt was made to pass full phonon spectrum from the atomistic domain to the coarse-grained domain by introducing a wave-based interpolation scheme [42].



Figure 5. Normalized kinetic energy distribution in simulations of the propagation of dislocations and a heat pulse: (a-c) a moving dislocation before meeting a heat pulse, showing that the motion of the dislocation is accompanied by radial-shaped wavefronts of phonons ahead of the moving dislocation and V-shaped wave tails in the wake of the dislocation; (d) the dislocation meeting with a propagating heat pulse; (e-f) an array of moving dislocations meeting with the heat pulse showing partially coherent partially diffuse scattering of the phonons by the moving dislocations [40].

The groups of Profs. [McDowell](#), [Chen](#), and [Xiong](#) are still actively advancing the CAC method. Keep an eye on the [CAC publications](#) for the latest progress!

References

1. Youping Chen, James Lee. [Atomistic formulation of a multiscale theory for nano/micro physics](#), *Philos. Mag.* 85 (2005) 4095-4126
2. Youping Chen. [Reformulation of microscopic balance equations for multiscale materials modeling](#), *J. Chem. Phys.* 130 (2009) 134706
3. Youpig Chen, James D. Lee, Azim Eskandarian, [Atomistic counterpart of micromorphic theory](#), *Acta Mech.* 161 (2003) 81-102
4. Youping Chen, Jonathan Zimmerman, Anton Krivtsov, David L. McDowell. [Assessment of atomistic coarse-graining methods](#), *Int. J. Eng. Sci.* 49 (2011) 1337-1349
5. Youping Chen, James Lee, Liming Xiong. [A generalized continuum theory and its relation to micromorphic theory](#), *J. Eng. Mech.* 135 (2009) 149-155
6. Liming Xiong, Youping Chen, James D. Lee. [Modeling and simulation of boron-doped nanocrystalline silicon carbide thin film by a field theory](#), *J. Nanosci. Nanotech.* 9 (2009) 1034-1037
7. Liming Xiong, Youping Chen, James Lee. [Simulation of dislocation nucleation and motion in single crystal magnesium oxide by a field theory](#), *Comput. Mater. Sci.* 42 (2008) 168-177
8. Liming Xiong, Youping Chen, James D. Lee. [Atomistic simulation of mechanical](#)

- properties of diamond and silicon carbide by a field theory, *Modelling Simul. Mater. Sci. Eng.* 15 (2007) 535-551
9. J.H. Irving, Jhon G. Kirkwood. *The statistical mechanical theory of transport processes. IV. The equations of hydrodynamics*, *J. Chem. Phys.* 18 (1950) 817-829
10. Eugène Cosserat, François Cosserat. *Théorie des corps déformables*, Paris, (1909) 17-29
11. Youpig Chen, James D. Lee, Azim Eskandarian. *Micropolar theory and its applications to mesoscopic and microscopic problems*, *Comput. Modeling Eng. Sci.* 5 (2004) 35-43
12. A. Cemal Eringen. *Theory of micropolar elasticity*, in *Microcontinuum Field Theories*, Springer (1999) 101-248
13. A. Cemal Eringen. *Microcontinuum Field Theories: I. Foundations and Solids*, Springer (1999)
14. A. Cemal Eringen. *Mechanics of micromorphic continua*, in: *Mechanics of Generalized Continua. IUTAM Symposia (International Union of Theoretical and Applied Mechanics)*, Springer (1968)
15. Liming Xiong, Youping Chen. *Coarse-grained simulations of single-crystal silicon*. *Modelling Simul. Mater. Sci. Eng.* 17 (2009) 035002
16. Qian Deng, Youping Chen, *A coarse-grained atomistic method for 3D dynamic fracture simulation*, *J. Multiscale Comput. Eng.* 11 (2013) 227-237
17. Qian Deng, Liming Xiong, Youping Chen. *Coarse-graining atomistic dynamics of brittle fracture by finite element method*, *Int. J. Plast.* 26 (2010) 1402-1414
18. Liming Xiong, Garrett Tucker, David L. McDowell, Youping Chen. *Coarse-grained atomistic simulation of dislocations*, *J. Mech. Phys. Solids* 59 (2011) 160-177
19. Liming Xiong, Qian Deng, Garrett 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, Garrett 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, 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
22. Liming Xiong, Ji Rigelesaiyin, Xiang Chen, Shuzhi Xu, David L. McDowell, Youping Chen. *Coarse-grained elastodynamics of fast moving dislocations*, *Acta Mater.* 104 (2016) 143-155

23. Liming Xiong, Shuzhi Xu, David L. McDowell, Youping Chen. [Concurrent atomistic-continuum simulations of dislocation-void interactions in fcc crystals](#), *Int. J. Plast.* 65 (2015) 33-42
24. 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
25. 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
26. Liming Xiong, Youping Chen. [Effects of dopants on the mechanical properties of nanocrystalline silicon carbide thin film](#), *Comput. Modeling Eng. Sci.* 24 (2008) 203-214
27. Shengfeng Yang, Liming Xiong, Qian Deng, Youping Chen. [Concurrent atomistic and continuum simulation of strontium titanate](#), *Acta Mater.* 61 (2013) 89–102
28. Shengfeng Yang, Youping Chen. [Concurrent atomistic and continuum simulation of bicrystal strontium titanate with tilt grain boundary](#), *Proc. R. Soc. A* 471 (2015) 20140758
29. Shengfeng Yang, Youping Chen, [Concurrent atomistic-continuum simulation of defects in polyatomic ionic materials](#), in *Multiscale Materials Modeling for Nanomechanics* (ed: Christopher R. Weinberger, Garrett J. Tucker), Switzerland: Springer International Publishing, 2016
30. Shengfeng Yang, Ning Zhang, Youping Chen. [Concurrent atomistic-continuum simulation of polycrystalline strontium titanate](#), *Philos. Mag.* 95 (2015) 2697-2716
31. Shuzhi 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
32. Shuzhi 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
33. Shuzhi 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
34. Shuzhi 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
35. Shuzhi 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
36. Shuzhi 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

37. Shuzhi 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
38. Shuzhi 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
39. Xiang Chen, Weixuan Li, Liming Xiong, Yang Li, Shengfeng Yang, Zexi Zheng, David L. McDowell, Youping Chen. [Ballistic-diffusive phonon heat transport across grain boundaries](#), *Acta Mater.* 136 (2017) 355-365
40. 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
41. Xiang Chen, Aleksandr Chernatynskiy, Liming Xiong, Youping Chen. [A coherent phonon pulse model for transient phonon thermal transport](#), *Comput. Phys. Comm* 195 (2015) 112–116
42. Xiang Chen, Adrian Diaz, Liming Xiong, David L. McDowell, Youping Chen. [Passing waves from atomistic to continuum](#), *J. Comput. Phys.* 354 (2018) 393-402

Algorithm

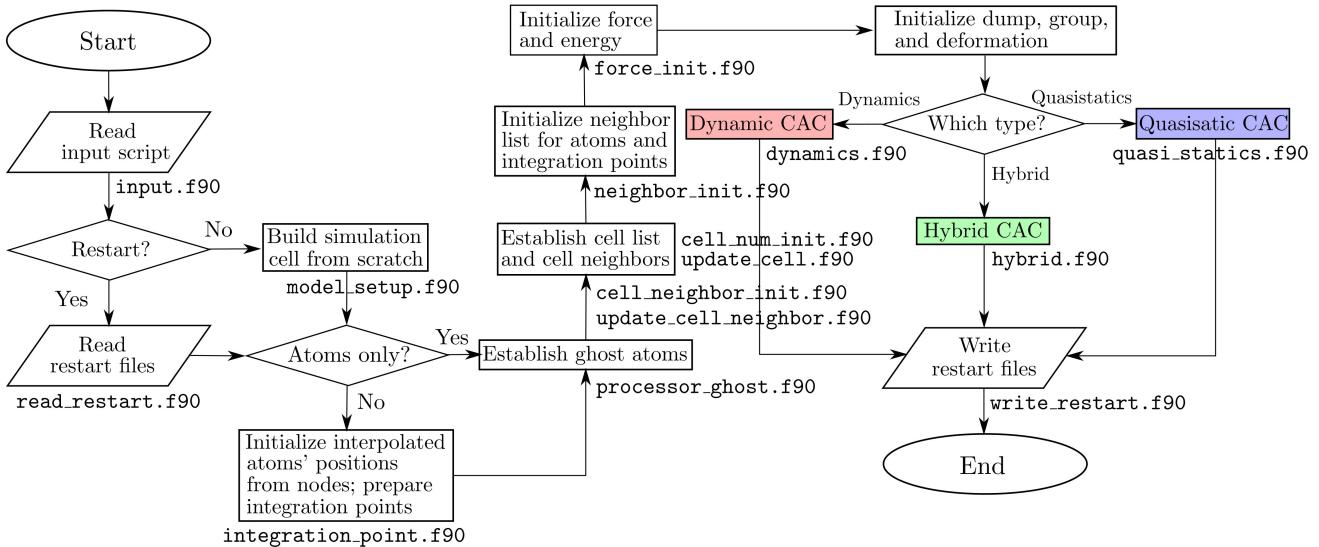
A framework for mixed atomistic/continuum modeling, the CAC algorithm adopts common atomistic modeling and finite element techniques. In the atomistic domain, Newton's third law is employed to promote efficiency in calculating the force, pair potential, local electron density, and stress. The short-range neighbor search employs a combined cell list and Verlet list method. In the coarse-grained domain, the Garlekin method and Gaussian quadrature are employed to solve the [governing equations](#).

There are, however, several issues in CAC simulations with coarse-graining that do not exist in standard atomistic and finite element method simulations.

For more information, read chapter 3 of [Shuzhi Xu's Ph.D. dissertation](#).

Scheme

A flowchart of the CAC simulation scheme based on [spatial decomposition](#) is presented below:

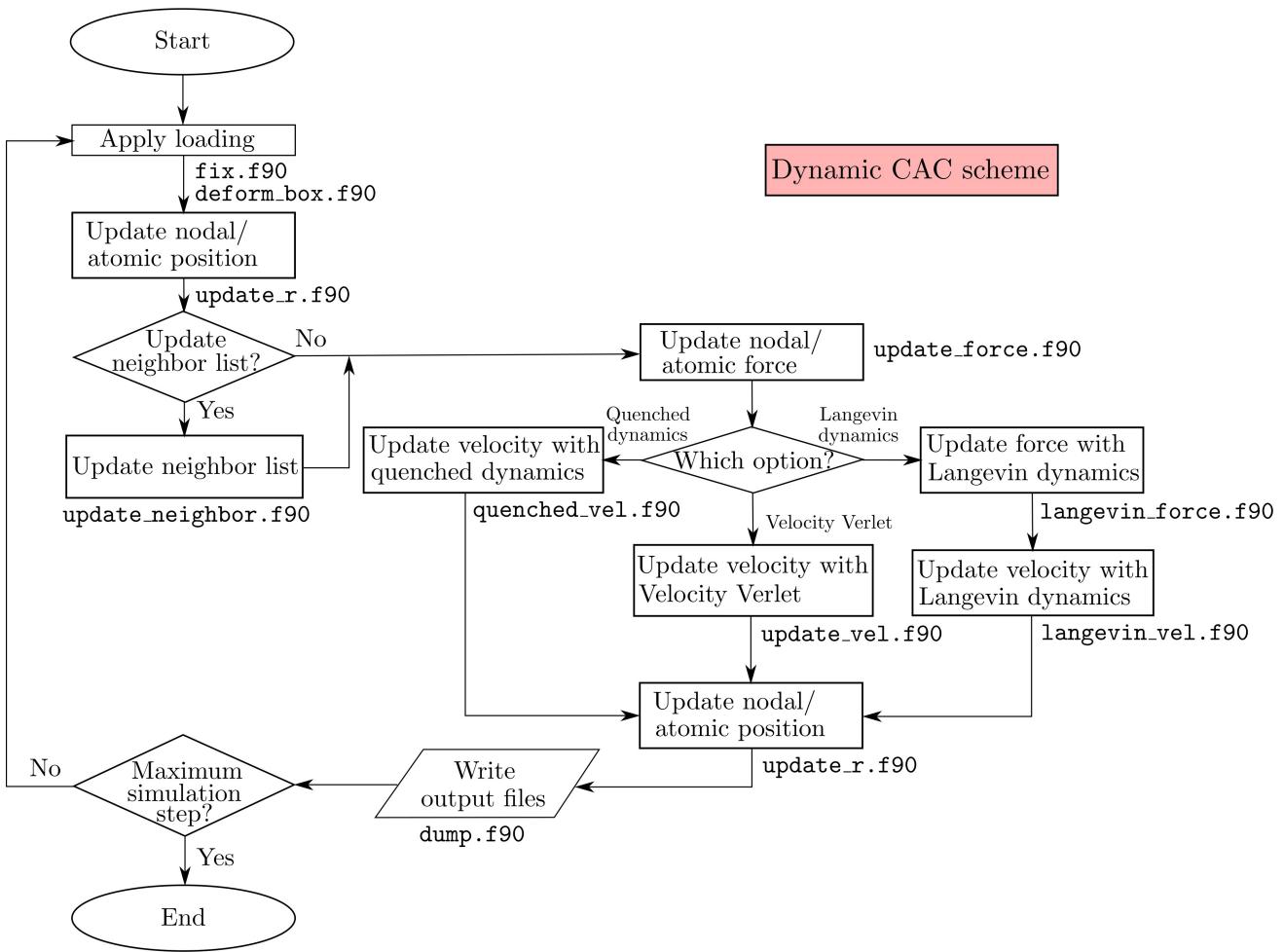


where there are three types of CAC simulations: dynamics, quasistatics, and hybrid, specified by the [simulator](#).

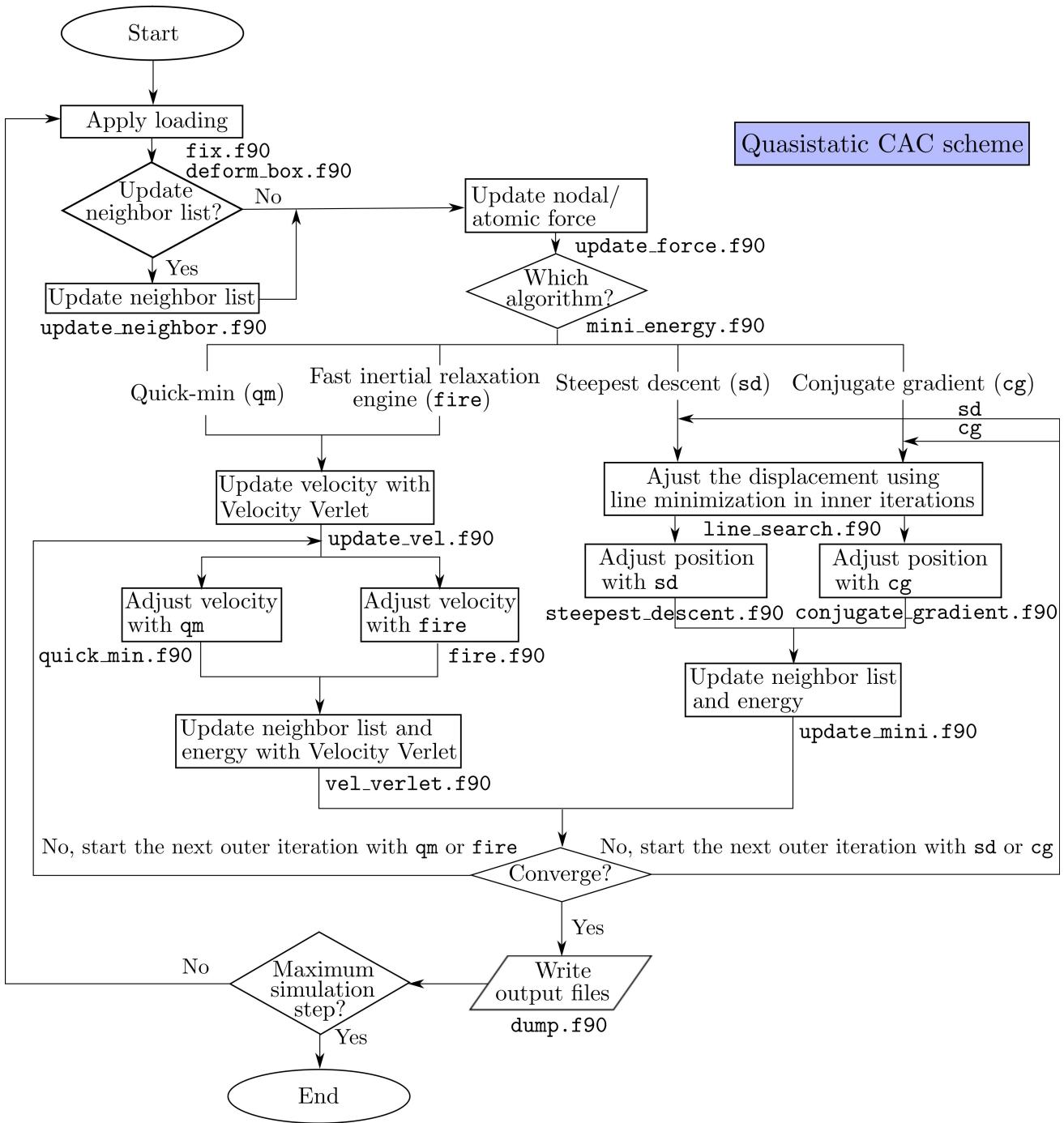
In CAC simulations, the elements/nodes/atoms information can either be created from scratch (`model_setup.f90`) or read from the `cac_in.restart` file (`read_restart.f90`), depending on the parameters in the `restart` command.

The dynamic CAC scheme is

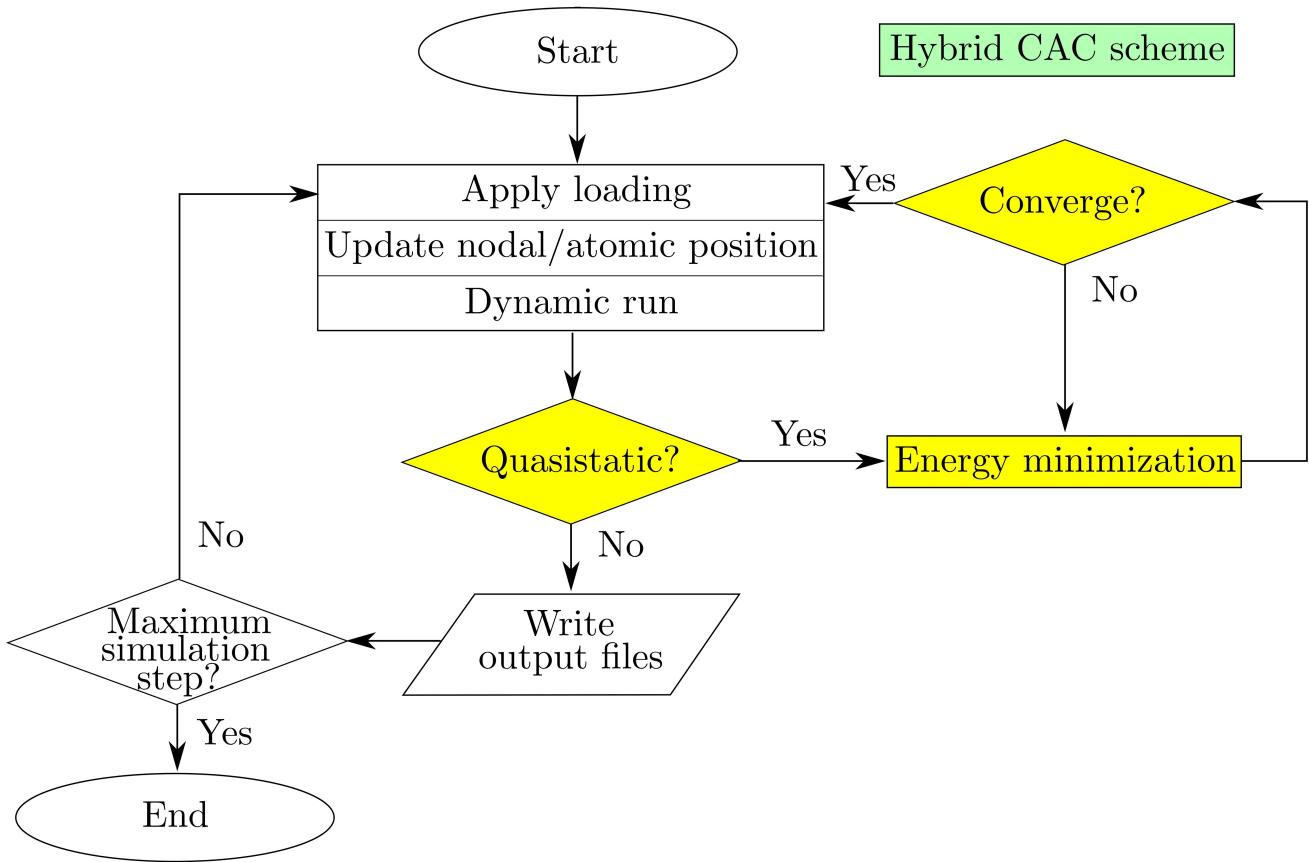
Scheme



The quasistatic CAC scheme is



The hybrid CAC scheme is

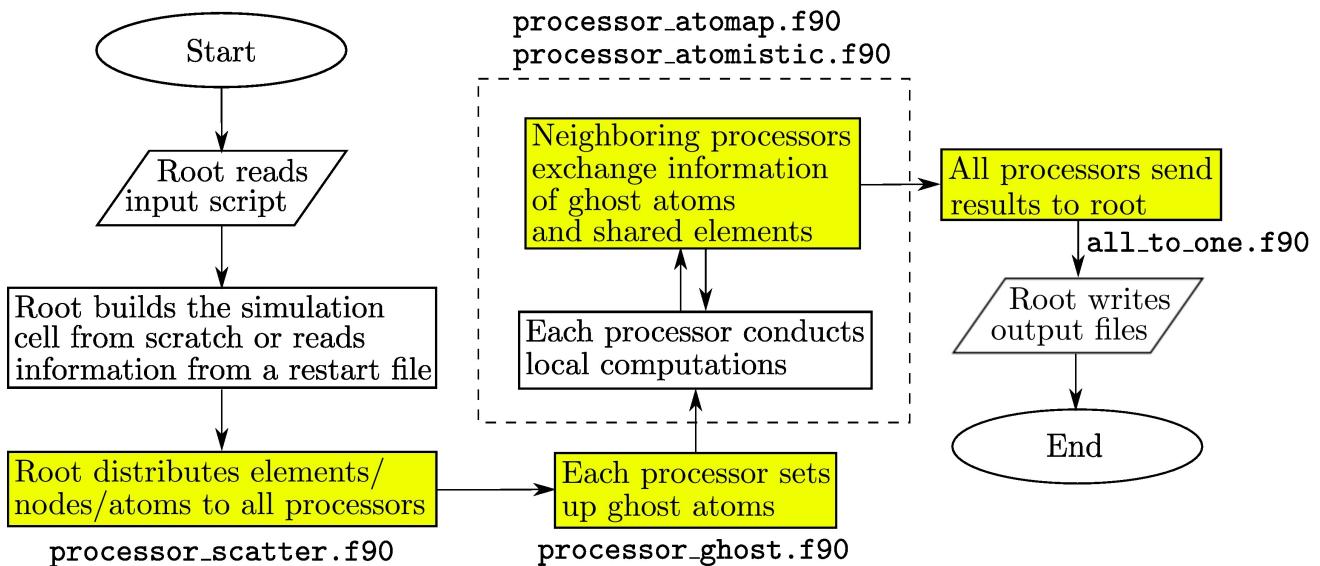


All these four figures are respectively adapted from Fig. 1, Fig. S1, Fig. S2, and Fig. 2 of Xu et al. More information of the dynamic and quasistatic CAC can be found in the [dynamics](#) and [minimize](#) commands, respectively.

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 is also encountered by other concurrent multiscale modeling methods.

The PyCAC code employs the SD algorithm in which the load balance is optimized, as shown in the figure below which is adapted from Fig. 5 of Xu et al.



Parallel CAC simulation scheme. Procedures that do not exist in the [serial CAC simulation scheme](scheme.md) are highlighted in yellow. Note that (i) in the serial scheme, the root processor does everything and (ii) the two procedures in the dashed box are conducted back and forth until the output begins.

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 real, the users can opt for 128-bit real by modifying `wp`.

The default size used for an integer is `KIND = 4`, meaning that any integer may have a signed value ranging from -2,147,483,648 to 2,147,483,647. In PyCAC, the maximum integer is usually the number of atoms (both the real atoms in the atomistic domain and the interpolated atoms in the coarse-grained domain). In the case that each element contains 2197 atoms, this limit suggests that there cannot be more than 977,461 elements in a fully coarse-grained simulation cell. If the user wants to study larger simulation cells, he/she needs to modify the source code.

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 is $1.602176565 \times 10^{-19}$ Joules (i.e., eV)
- The unit of force is $1.602176565 \times 10^{-9}$ Newtons (i.e., eV/Angstrom)
- The unit of pressure is 10^9 Pascales (i.e., GPa)

Input

To run a CAC simulation, one may 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 formulations for potential energy E and the force on atom k , \mathbf{f}_k , are

$$E = \frac{1}{2} \sum_i \sum_{j \neq i} V_{ij}(r_{ij}) + \sum_i F(\bar{\rho}_i)$$

$$\mathbf{f}_k = \sum_{j \neq k} \left[\frac{\partial V_{kj}(r_{kj})}{\partial r_{kj}} + \left(\frac{\partial F(\bar{\rho}_k)}{\partial \bar{\rho}_k} + \frac{\partial F(\bar{\rho}_j)}{\partial \bar{\rho}_j} \right) \frac{\partial \rho_{kj}(r_{kj})}{\partial r_{kj}} \right] \frac{\mathbf{r}_{kj}}{r_{kj}}$$

where

$$\bar{\rho}_i = \sum_{j \neq i} \rho_{ij}(r_{ij})$$

Note that the [force formulation](#) above only holds for monatomic pure materials.

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 eV.
- In `pair.tab`, the first column is the interatomic distance r , in Angstrom; the second

column is the pair potential V , in eV.

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

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 CAC simulations, an approximation is introduced to calculate the host electron density $\bar{\rho}$ of the integration points in the coarse-grained domain. For more information, read chapter 3 of [Shuzhi Xu's Ph.D. dissertation](#).

The readers may find EAM potential files in these database:

- [NIST](#)
- [University of Edinburgh](#)
- [Other resources](#)

Note that most of these files do not have the format that suits the CAC simulation.

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 continuously to zero at the cut-off distance r_c , i.e., if $r < r_c$, $f = f(r) - f(r_c)$; otherwise, $f = 0$.

In `lj.para`, a blank line or a line with the "#" character in column one (a comment line) is ignored; three positive real numbers (ϵ , σ , and r_c) and one non-negative real number (r_0) are given 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 smaller `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 files created on-the-fly

The main output of a CAC simulation are `cac_cg_#.vtk` and `cac_atom_#.vtk` files that contain elemental/nodal information and atomic information in the coarse-grained and the atomistic domains, respectively, where `#`, a non-negative integer, is the simulation step at which the file is created. These files, created by `vtk_legacy.f90` with a frequency of `output_freq`, can be read by [ParaView](#). Note that besides the nodal/atomic positions, the energy scalar, the force vector, and the stress tensor of each node/atom are also recorded in these `*.vtk` files.

One-time vtk and dump files

Besides the files that are created on-the-fly, 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 standard [LAMMPS](#) dump file `dump.lammps` which, in addition to the positions of the real/interpolated atoms, also contain the velocities of the real/interpolated atoms 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` for the coarse-grained and the atomistic domains, respectively. 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 one-time `*.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 certain [simulation step](#).

A series of `cac_out_#.restart` files, where `#` is a positive integer, are created with a frequency of `restart_freq`. One of these files can then be renamed to `cac_in.restart` to restart a prior simulation when `boolean_restart = t`.

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 positive integer. These files can then be renamed to `group_in_#.id` for [restart group](#) and [refinement](#) purposes.

Python scripting interface

The Python scripting interface, written by [Thomas G. Payne](#), is a Python wrapper module for CAC's underlying Fortran 2008 code, allowing handling of the program's input and output as well as links to some external visualization software. Written in Python 3 and contained in PyCAC.py, the Python module provides a robust user interface to facilitate parametric studies via CAC simulations without interacting with the Fortran code and to improve handling of input, output, and visualization options. The module works on local computers and serves as an interface with high performance computing clusters. In particular, the Python module consists of three main components: [input.py\(\)](#), [output.py\(\)](#), and [visualization.py\(\)](#).

input.py()

input.py() contains a graphical user interface that can be used to generate the [CAC input script](#), and to submit jobs via job schedulers on high performance computing clusters, e.g., those on [NSF XSEDE](#).

More information will be added soon.

output.py()

output.py() downloads the CAC simulation output data from the high performance computing clusters before processing them locally. As mentioned [earlier](#), a CAC simulation outputs vtk files containing elemental, nodal and atomic information. It follows that the output.py() component reads the vtk files and interpolates all atoms inside the elements in the coarse-grained domain. These interpolated atoms, together with the real atoms in the atomistic domain (also read from the vtk files), are used to generate standard [LAMMPS dump](#) files that can be visualized by atomistic model viewers and/or [read by LAMMPS](#) directly to carry out equivalent fully-resolved atomistic simulations.

More information will be added soon.

visualization.py()

visualization.py() works locally for integrating some visualization software. The dump files and vtk files are visualized by [OVITO](#) and [ParaView](#), respectively.

More information will be added soon.

Command

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

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 column one (a comment line) is discarded, and (ii) each command should contain no more than 200 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 integer and a real number, e.g., a real number must be written as 2. or 2.0, instead of 2.

The sequence of the commands in `cac.in` does not matter, except for the `modify`, `group`, `fix`, and `cal` commands, in which case extra commands that (i) appear later and (ii) exceed the numbers in `modify_number`, `new_group_number`, `fix_number`, and `cal_number`, 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, the user may get a self-explanatory error message, followed by termination of the program by:

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

if something is potentially wrong or a warning message.

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 34 CAC 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*

`cal, constrain, dump_num, dynamics, element, group_num, group, limit, minimize, neighbor, simulator, temperature`

- *Actions*

`deform, fix, refine, restart, run`

- *Miscellanies*

`convert, debug`

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 faces of the cell.

`p` sets periodic boundary conditions (PBCs). The nodes/atoms interact across the boundary and can exit one end of the cell and re-enter the other end. For more information of the PBCs in the coarse-grained domain, read chapter 3 of [Shuzhi 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 cell to the other. The positions of both faces are set so as to encompass the nodes/atoms in that dimension, no matter how far they move.

Under neither boundary condition will any nodes/atoms be lost during a CAC simulation.

Related commands

boundary

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 PBCs.

This command becomes irrelevant when `boolean_restart` = t , in which case the boundary conditions are read from the `cac_in.restart` file.

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

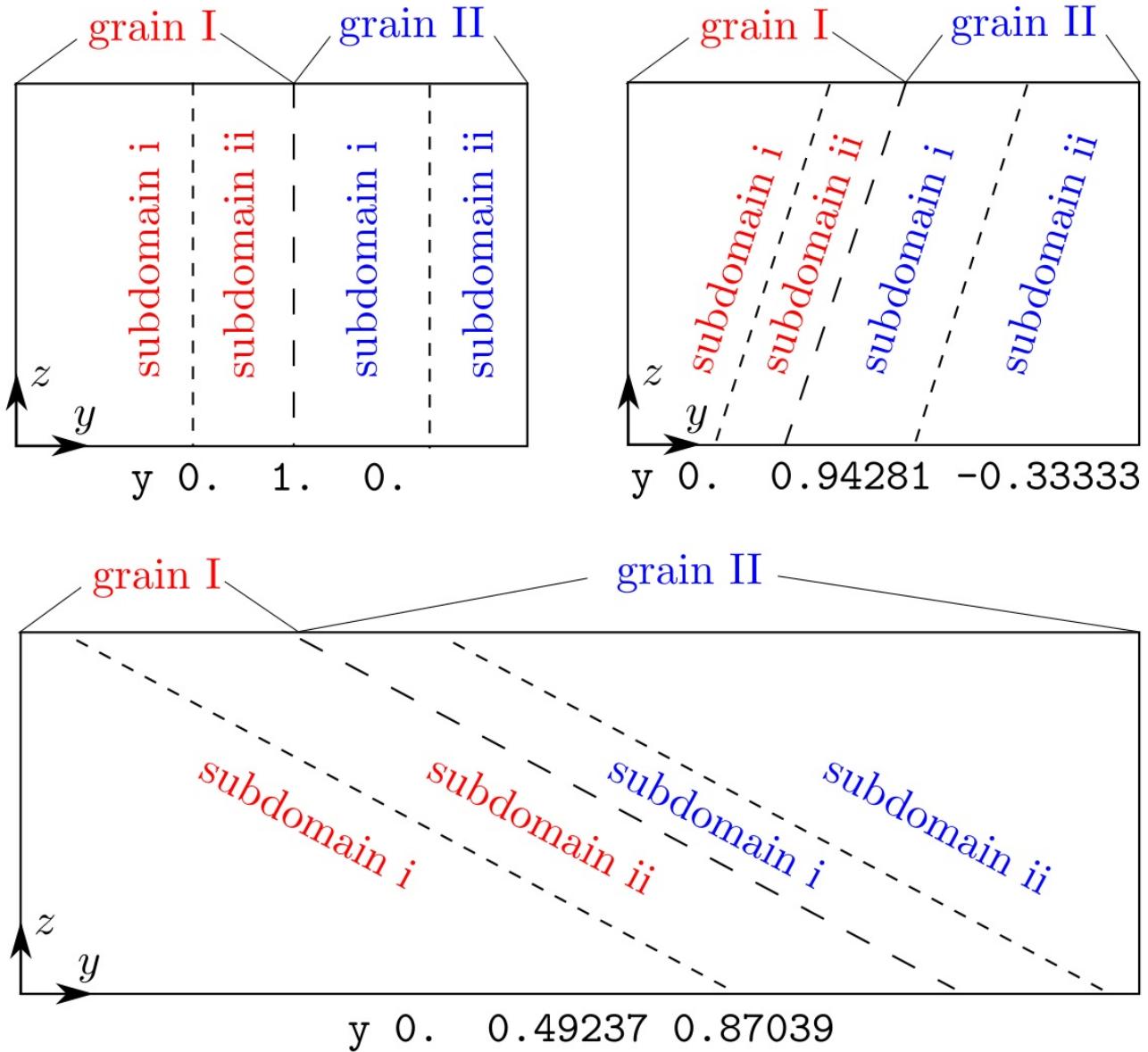
Examples

```
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.  
box_dir x 1. 0. 0. y 0. 0.49237 0.87039 z 0. 0. 1.
```

Description

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

Assume that `direction` = 2, i.e., the grains are stacked along the `y` direction, the first example results in a GB plane normal to the `y` axis; the second example 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 crystallographic orientation to the simulation cell-based orientation.

`box_dir`

This command becomes irrelevant when `boolean_restart` = t , in which case there is no need for the subdomain information.

Related files

`model_init.f90` , among many

Default

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

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 command calculates certain quantities associated with [new groups](#) and/or restart groups. The `group_name` must match that of one of these groups.

energy is the total potential energy in a group divided by the total number of nodes and 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_#` with a frequency of `reduce_freq`, 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`, which should not be larger than `new_group_number + restart_group_number`. When there are more `cal` commands in `cac.in` than `cal_number`, those appearing later will be ignored.

Related files

`calculation.f90` and `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 constraint is added to the system. When `boolean` is *t*, the equivalent nodal/atomic force vector is projected onto the [`ijk`] direction such that they can only move along that direction, either in dynamic or quasistatic CAC simulations. The only exception is that the external force applied by the [fix](#) command and the random force $\Theta(t)$ in [Langevin dynamics](#) can be along any direction.

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

constrain

```
update_force.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 crystallographic 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 the positive integer `#` is the grain ID.

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

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

Related commands

This command is useful when the user has a set of crystallographic 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` = *t*; the unit number is 13. The user can then write whatever he/she wants to the `debug` file using unit number 13, i.e.,

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

When `boolean_mpi` = *t*, all processors have access to the `debug` file, otherwise only the [root](#) does.

Related commands

None.

Related files

debug

```
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 *zx* 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 strain-controlled or stress-controlled homogeneous deformation of the simulation cell. 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` = 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 applied at each step, in units of `time_step`.

`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 GPa. In CAC simulations, all stress components are usually small at the beginning. Subject to the strain, most stress tensor components increase in magnitude 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 but each `ij` remains unchanged. 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 `simulation step` equals `time_start` and stops when it exceeds `time_end`.

When (i) the `simulation step` is larger than `time_always_flip` and (ii) the `simulation step` does not exceed `time_end` and (iii) the strain rate tensor did not change 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 dynamic or quasistatic 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` command may be homogeneously deformed along with the simulation cell, depending on the value of `boolean_def` set in the `fix` command.

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 purpose and for further analysis.

`reduce_freq` sets the frequency with which certain quantities are written to `group_cal_#` (when `cal_number > 0`) and `cac.log` 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 and the screen to monitor the simulation progress.

Related commands

None.

dump

Related files

`dump_init.f90` and `dump.f90`

Default

```
dump 1000 1000 5000 50
```

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 the dynamic run in CAC simulations.

When `dyn_style` = *ld*, the [Langevin dynamics](#) is performed, i.e.,

$$m\ddot{\mathbf{R}} = \mathbf{F} - \gamma m\dot{\mathbf{R}} + \Theta(t)$$

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, γ is the `damping_coefficient` in ps^{-1} , and *t* is the time in ps. The Velocity Verlet form is employed to solve the equations of motion, as given in Eqs. 1-3 in [Xu et al., 2016](#). The velocity $\dot{\mathbf{R}}$ is updated in `langevin_vel.f90`.

The *ld* style is used to keep a constant temperature in CAC simulations by adding to the force \mathbf{F} a time-dependent Gaussian random variable $\Theta(t)$ with zero mean and variance of

$\sqrt{2m\gamma k_B T/\Delta t}$, where m is the atomic mass, k_B is the Boltzmann constant (8.6173324×10^{-5} eV/K), T is the temperature in K, and Δt is the `time_step` in ps. The random variable is calculated and added to the force in `langevin_force.f90`. Note that when $T = 0$, the equation above reduces to

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

which is the equation of motion in damped molecular dynamics.

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.,

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.,

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`, a dynamic simulation following

$$m\ddot{\mathbf{R}} = \mathbf{F}$$

is performed using the Velocity Verlet scheme.

Note that the *vv* style cannot be used to keep a constant `temperature` and the *qd* style cannot be used to keep a finite `temperature`. When `boolean = t`, if the *vv* style is chosen and if, for a finite `temperature`, the *qd* style is chosen, the user will get a warning message.

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

Related commands

dynamics

[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.
```

element

Syntax

```
element mass_matrix intpo_depth
```

- `mass_matrix` = *lumped* or *consistent*
- `intpo_depth` = 1 or 2

Examples

```
element lumped 2
element consistent 1
```

Description

This command sets the element type used in the finite element calculation in the coarse-grained domain.

For `mass_matrix`, the *lumped* type approximates the mass of each element and distributes it to the nodes; the *consistent* type distributes the exact mass over the entire element.

`intpo_depth` decides whether the first nearest neighbor (1NN) or the second nearest neighbor (2NN) elements are employed in the coarse-grained domain; their differences are illustrated in Fig. B26 of [Xu et al., 2015](#).

Related commands

The atomic mass is provided in the [mass](#) command.

Related files

`mass_matrix.f90`, `integration_point.f90`, and `update_equiv.f90`

Default

```
element lumped 2
```

fix

Syntax

```
fix group_name boolean_release boolean_def
    assign_style assign_x assign_y assign_z disp_lim
    time time_start time_end
    boolean_grad
    grad_ref_axis grad_assign_axis
    grad_ref_l grad_ref_u
```

- `group_name` = a string (length <= 30)
- `boolean_release` , `boolean_def` , `boolean_grad` = *t* or *f*
 - t* is true
 - f* is false
- `assign_style` = *disp* or *force*
- `assign_x` , `assign_y` , `assign_z` = real number or *null*
- `disp_lim` = non-negative real number
- `time_start` , `time_end` = non-negative integer
- `grad_ref_axis` , `grad_assign_axis` = 1 or 2 or 3
- `grad_ref_l` , `grad_ref_u` = real number or *inf*

Examples

```
fix group_1 t t disp 0. null 0. 5. time 0 2500 f
fix group_2 t t disp 5. 0. 0. 10. time 0 2500 t 2 1 50. 60.
fix group_3 t t force 0. 1. 0. 3. time 0 2500 t 3 2 100. 10.
```

Description

This command applies displacements and/or forces to new groups and restart groups, the numbers of which are provided in the `group_num` command. The number of `fix` commands is `fix_number`. The new groups are created by first providing the elements/nodes/atoms information in the `group` command, while the same information for the restart groups, which are introduced when `restart_group_number > 0`, is read from `group_in_#.id`, where `#` is a positive integer starting from `new_group_number + 1`. A `group_in_#.id` file can be renamed from a `group_out_#.id` file that was created automatically in previous CAC simulations of which the total number of groups > 0 .

When the groups are at the simulation cell boundaries, this command is useful in applying displacement, traction, or mixed boundary conditions.

`group_name` must match one of the `new groups` or `restart groups`. All in this command take effect only when `time_start < simulation step < time_end`, unless stated otherwise.

when `simulation step > time_end`, the group is no longer assigned a displacement/force if `boolean_release = t`; the group is assigned a displacement/force vector [`assign_x`, `assign_y`, `assign_z`] whose non-*null* components are zeroed.

When `boolean_def = t`, the group is deformed [along with the simulation box](#). The deformation-induced displacement is added on top of the assigned displacement/force.

`assign_style = disp` or `force`, meaning that a displacement or force vector [`assign_x`, `assign_y`, `assign_z`], in Angstrom/ `time_step` or eV/Angstrom, is applied to all nodes/atoms in the group at each `simulation step`, after their `interatomic potential`-based displacement/force is discarded. If any component of the displacement/force vector is *null*, no displacement/force is assigned to this component. In the first example, `group_1` is fixed along the *x* and *z* directions but not along the *y* direction.

`disp_lim` is the upper bound of the magnitude of the total group displacement, in units of `lattice_constant`. If the total displacement magnitude (in Angstrom instead of in Angstrom/ `time_step`) is larger than `disp_lim`, the displacement vector is zeroed, regardless of whether `time_end` is reached or what value `boolean_release` is. `disp_lim` is irrelevant when `assign_style = force`. However, it needs to be provided regardless.

When `boolean_grad = f`, the same displacement/force vector [`assign_x`, `assign_y`, `assign_z`] is assigned to all nodes/atoms of the group; the following options, including `grad_ref_axis`, `grad_assign_axis`, `grad_ref_1`, and `grad_ref_u`, become irrelevant and do not need to be provided.

When `boolean_grad = t`, the displacement/force is assigned to the group gradiently, i.e., different elements/nodes/atoms in the group may have a different [`assign_x` , `assign_y` , `assign_z`] vector. The `grad_assign_axis` component of the displacement/force 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 bounds of the graded displacement/force, in units of the component of the [lattice periodicity length vector \$l'_0\$](#) along the `grad_ref_axis` direction, with *inf* referring to the lower (`grad_ref_l`) and upper (`grad_ref_u`) simulation cell boundaries.

If `grad_ref_l < grad_ref_u`, the elements/nodes/atoms located at or below `grad_ref_l` are assigned a zero displacement/force vector, i.e., fixed; those located at or above `grad_ref_u` are assigned [`assign_x` , `assign_y` , `assign_z`]. If `grad_ref_l > grad_ref_u`, the elements/nodes/atoms located at or above `grad_ref_l` are assigned a zero displacement/force vector, i.e., fixed; those located at or below `grad_ref_u` are assigned [`assign_x` , `assign_y` , `assign_z`]. In any case, the elements/nodes/atoms located between `grad_ref_l` and `grad_ref_u` are assigned a vector whose `grad_assign_axis` component is linearly graded while the other two components remain the same with respect to [`assign_x` , `assign_y` , `assign_z`].

In the second example, the elements/nodes/atoms which are located below $50.0 \cdot l'_0[2]$ along the `y` axis (because `grad_ref_axis = 2`) are assigned a zero displacement vector; those located above $60.0 \cdot l'_0[2]$ along the `y` axis are assigned [`assign_x` , `assign_y` , `assign_z`]; those in between are assigned a linearly graded displacement vector whose `x` component (because `grad_assign_axis = 1`) is varied between zero and `assign_x` while its `y` and `z` components are `assign_y` and `assign_z`, respectively.

In the third example, the elements/nodes/atoms which are located below $10.0 \cdot l'_0[3]$ along the `z` axis (because `grad_ref_axis = 3`) are assigned [`assign_x` , `assign_y` , `assign_z`]; those located above $100.0 \cdot l'_0[3]$ along the `z` axis are assigned a zero force vector; those in between are assigned a linearly graded force vector whose `y` component (because `grad_assign_axis = 2`) is varied between zero and `assign_y` while its `x` and `z` components are `assign_x` and `assign_z`, respectively.

Note that for each group concerned in this command, the displacement and force vectors are added to relevant nodes/atoms after their interatomic potential-based displacement/force vectored are zeroed. In particular, the force, stress, and energy are zeroed if `assign_style = disp`; the force, stress, and energy are replaced with those induced by this command if `assign_style = force`. In both cases, the velocity vectors are also zeroed in [dynamic and hybrid CAC](#).

In this sense, if the same atoms/nodes are included in multiple [groups](#) that are also concerned in this command, those appearing in the later fix commands will prevail. For example, if a node is assigned a displacement vector in the first fix command, a force vector in the second fix command, and another force vector in the third fix command, the force vector in the last fix command will be imposed. As another example, if an atom is assigned a force vector in the fourth fix command, and a displacement vector in the fifth fix command, the force/stress/energy vector of that atom will be zeroed. To avoid unintended effects, users are advised to carefully check if the same nodes/atoms are involved in different fix commands.

Related commands

There cannot be fewer `fix` commands than `fix_number`, which should not be larger than `new_group_number + restart_group_number`. When there are more `fix` commands in `cac.in` than `fix_number`, those appearing later will be ignored.

Note that all groups do not necessarily have corresponding `fix` command. The purpose of having a group that does not have a corresponding `fix` command is to [calculate](#) certain mechanical properties, e.g., energy, force, and stress, of the nodes/atoms it contains.

Related files

`fix.f90`, `fix_displacement.f90`, and `fix_force.f90`

Default

None.

fix

grain_dir

Syntax

```
grain_dir direction overlap
```

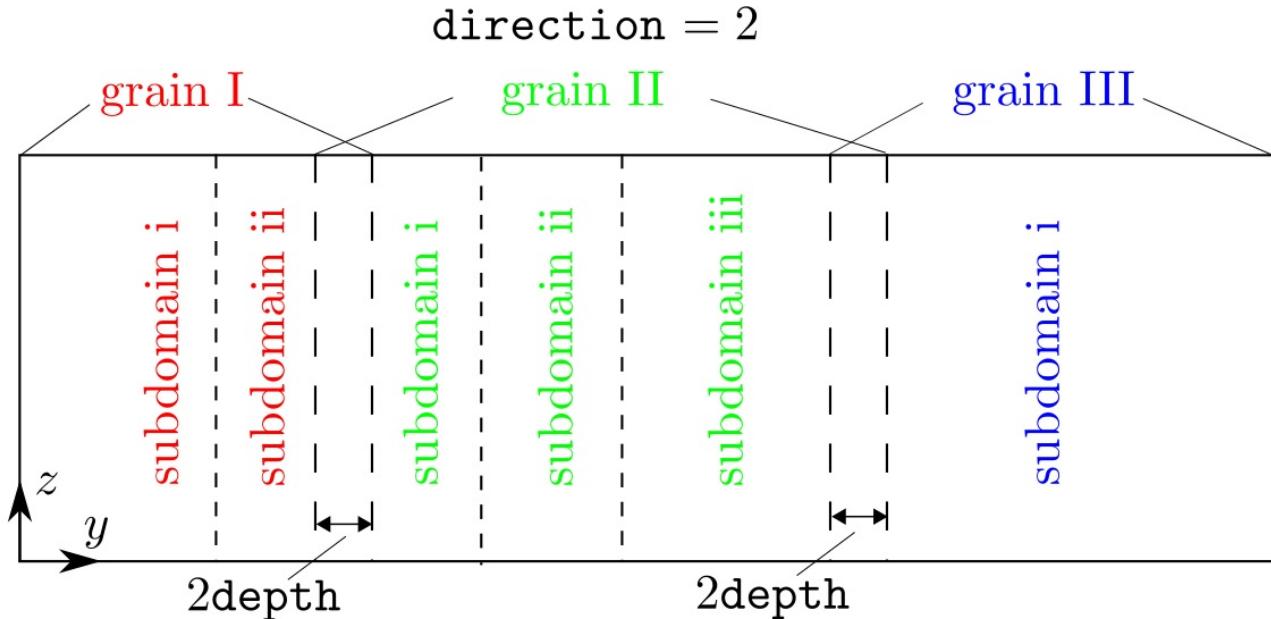
- `direction` = 1 or 2 or 3
- `overlap` = real number

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 be 1, 2, or 3, corresponding to the x, y, or z directions, respectively.

`overlap`, in units of the component of the lattice periodicity length vector l'_0 along the `direction`, sets the overlap distance between adjacent grains along the `direction`, as shown in the figure above. 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.

Note that the `direction` is also the `subdomain` stack direction if `subdomain_number > 1` even when there is only one grain, i.e., `grain_number = 1`. Since there is no overlap between adjacent subdomains within the same grain, `overlap` becomes irrelevant when `grain_number = 1`.

Related commands

This command is relevant when `grain_number > 1` or `subdomain_number > 1`.

This command becomes irrelevant when `boolean_restart = t`, in which case there is no need for the grain information.

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 crystallographic 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.,

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

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

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

The user will get an error message followed by the termination of the program if any of these requirements is not satisfied.

grain_mat

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*, in which case there is no need for the crystallographic orientations information.

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, respectively. When `move_x`, `move_y`, and `move_z` are all 0.0, the next grain's lower boundary is the current grain's upper boundary along the [grain stack direction](#). 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 [grain 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`, in which case there is no need for the grain information.

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 [grain stack direction](#). Each grain has its own [crystallographic 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`, in which case there is no need for the grain information.

Related files

`box_init.f90` and `grain.f90`

Default

```
grain_num 1
```

grain_num

group_num

Syntax

```
group_num new_group_number restart_group_number fix_number cal_number
```

- `new_group_number`, `restart_group_number`, `fix_number`, `cal_number` = non-negative integer (≤ 40)

Examples

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

Description

This command sets the numbers of [new groups](#), restart groups, [fix](#), and [calculations](#). In CAC, a group is a collection of elements/nodes/atoms. There are two purposes of having groups: (i) to [apply a displacement/force](#) to certain elements/nodes/atoms, (ii) to [calculate](#) some mechanical quantities, e.g., energy, force, and stress, of certain elements/nodes/atoms.

The new groups are defined in the [group](#) command. The elements/nodes/atoms contained in the restart groups, named `group_#`, are read from the `group_in_#.id` files, where `#` is a positive integer starting from `new_group_number + 1`, yet their displacement/force information is set in the [fix](#) command.

The total number of groups, i.e., `new_group_number + restart_group_number`, cannot be larger than 40. For each group, CAC outputs a `group_out_#.id` file containing relevant 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. On the one hand, there cannot be fewer `group_in_#.id` files than `restart_group_number`; on the

group_num

other hand, any `group_in_#.id` file with `# > new_group_number + restart_group_number` will be ignored. When `boolean_restart = f`, `restart_group_number` becomes 0, regardless of its value set in this command.

`fix_number` should not be larger than `new_group_number + restart_group_number`; neither should `cal_number`. Also, `fix_number + cal_number` should not be smaller than `new_group_number + restart_group_number`.

Related commands

The new groups are defined in the `group` command. The displacement/force and calculation information of each group is set in the `fix` and `cal` commands, respectively.

Related files

`group.f90` , `fix_displacement.f90` , `fix_force.f90` , `group_cal.f90`

Default

```
group_num 0 0 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
```

- `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` = *t* or *f*
 - `t` is true
 - `f` is false
- `group_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

Examples

```
group group_1 null atom block x inf inf 1. 0. 0. 0. y inf inf 0. 1. 0. z 14.4 inf 0. 0. 1
. t 3 20. 5. 0. 10. 10.
```

```
group
```

```
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.  
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.  
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.
```

Description

This command sets new groups, the number of which is provided in the `group_num` command. The elements/nodes/atoms in a group, either a new group or a restart group, can be `moved` at each `simulation step, deformed with the simulation cell` (when `boolean_def` in both `fix` and `deform` commands = `t`), or not moved/deformed. The syntax is similar to the first of that of the `modify` command.

Multiple groups cannot have the same `group_name`.

`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 discussed [here](#). `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`.

`lower_b` and `upper_b` are the lower and upper boundaries of the `group_shape`, respectively, in units of the component of the `lattice periodicity length vector l'_0` along the corresponding direction. When `lower_b` or `upper_b` = `inf`, the corresponding lower or upper simulation cell boundaries are taken as the `group_shape` boundaries, respectively. Note that when `group_shape` = `cylinder` or `cone` or `tube`, `lower_b` and `upper_b` are the lower and upper plane boundaries normal to the central axis `group_axis` direction, respectively.

`i`, `j`, and `k` decide the `group_shape` (\neq `sphere`) boundary plane orientations with respect to the simulation cell, similar to those in the `box_dir` command.

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`. Also, `group_axis` is irrelevant when `group_shape` = `block` or `sphere`. However, they need to be provided regardless.

group

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 units of the component of the lattice periodicity length vector l'_0 and with respect to the lower boundaries of the simulation cell along the corresponding direction, 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` direction 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`. Both `group_radius_large` and `group_radius_small` are in units of the maximum lattice periodicity length l'_{\max} .

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.

Related commands

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

This command becomes irrelevant when `new_group_number` = 0.

Related files

`group.f90`, `fix_displacement.f90`, `fix_force.f90`, and `group_cal.f90`

Default

None

group

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 Angstrom.

Note [that](#) (i) the current PyCAC 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.

Related commands

The atomic mass is provided in the [mass](#) command.

`lattice_structure` becomes irrelevant when `boolean_restart` = *t*, in which case there is no need for the lattice information.

Related files

lattice

```
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/integration point (`atomic_neighbor_number`). The numbers are used to allocate initial arrays for atoms in cells and neighbors of atoms/integration points. If, during a simulation, arrays larger than those initially allocated become necessary, the two numbers set in this command will increase by 20 to enlarge the arrays, until even larger arrays are needed, in which case these two numbers increase by 20 again, and so on.

Related commands

The initial values of these two numbers depend on the [cutoff distance \$r_c\$](#) 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
```

limit

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 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 in the coarse-grained domain is specified in the [element](#) command.

Related files

`crystal.f90` and `mass_matrix.f90`

Default

None.

mass

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](#) and [hybrid CAC](#).

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

Both *cg* and *sd* use the negative gradient of internal 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. Once the direction is set, the inner iterations begin in which a [line search](#) is conducted to determine the length by 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 [Shuzhi Xu's Ph.D. dissertation](#).

minimize

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](#) during the energy minimization until the internal energy converges at that step. For the `fire` and `qm` styles, the inner iteration is irrelevant.

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

Related commands

This command is relevant only when `simulation_style` = *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` , and `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*, in which case there is no need for the modification information.

Related files

```
model_modify.f90
```

Default

```
modify_num 0
```

modify_num

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 line_axis plane_axis
    modify_centroid_x modify_centroid_y modify_centroid_z
    dis_angle poisson_ratio

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` , `line_axis` , `plane_axis` = 1 or 2 or 3
- `modify_centroid_x` , `modify_centroid_y` , `modify_centroid_z` , `dis_angle` ,
`poisson_ratio` = 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 dislocation 1 3 1. 20. 3.2 60. 0.36
modify modify_4 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 is similar to that of the [group](#) command.

There are currently four `modify_style : delete`, `cg2at`, `dislocation`, and `cutoff`. When `modify_style = delete` or `cg2at`, the first syntax is used; when `modify_style = dislocation`, the second syntax is used; otherwise, the third syntax is used.

First syntax (`modify_style = delete or cg2at`)

The first syntax removes some elements/atoms (`delete`) or refines some elements into atomic scale (`cg2at`), based on the simulation cell built from scratch.

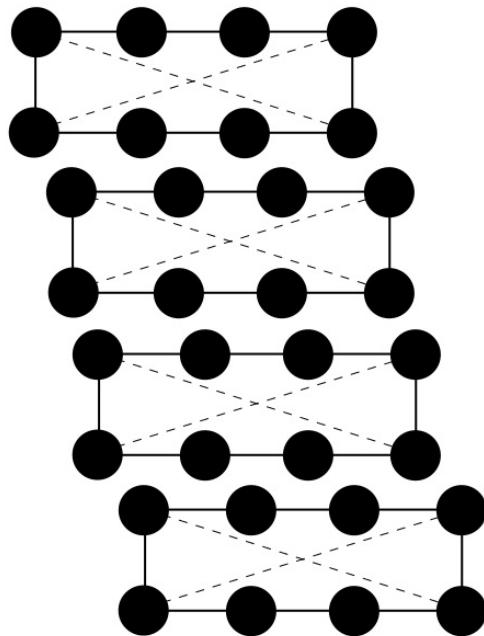
There are five `modify_shape : block`, `cylinder`, `cone`, `tube`, and `sphere`.

`lower_b` and `upper_b` are the lower and upper boundaries of the `modify_shape`, respectively, in units of the component of the [lattice periodicity length vector](#) ℓ'_0 along the corresponding direction. When `lower_b` or `upper_b = inf`, the corresponding lower or upper simulation cell boundaries are taken as the `modify_shape` boundaries, respectively. Note that when `modify_shape = cylinder` or `cone` or `tube`, `lower_b` and `upper_b` are the lower and upper plane boundaries normal to the central axis `modify_axis` direction, respectively.

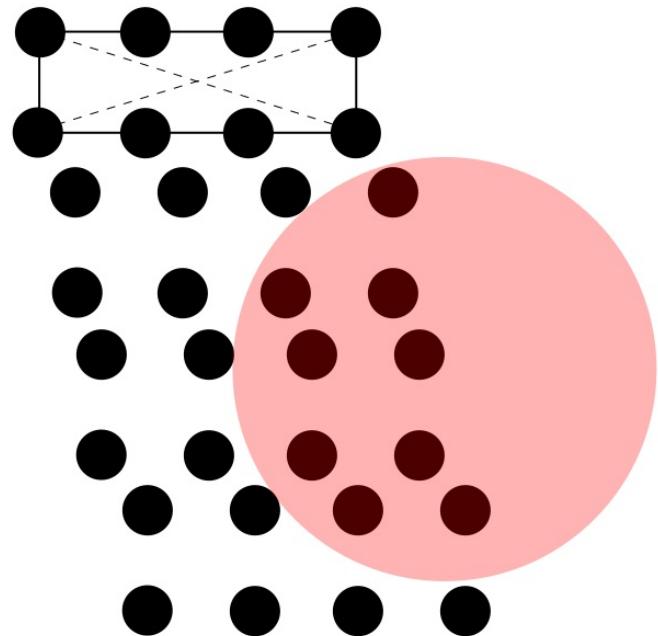
`i`, `j`, and `k` decide the `modify_shape` ($\neq \text{sphere}$) boundary plane orientations with respect to the simulation cell, similar to those in the [box_dir](#) command.

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 `modify_axis`. Also, `modify_axis` is irrelevant when `modify_shape = block` or `sphere`. However, they need to be provided regardless.

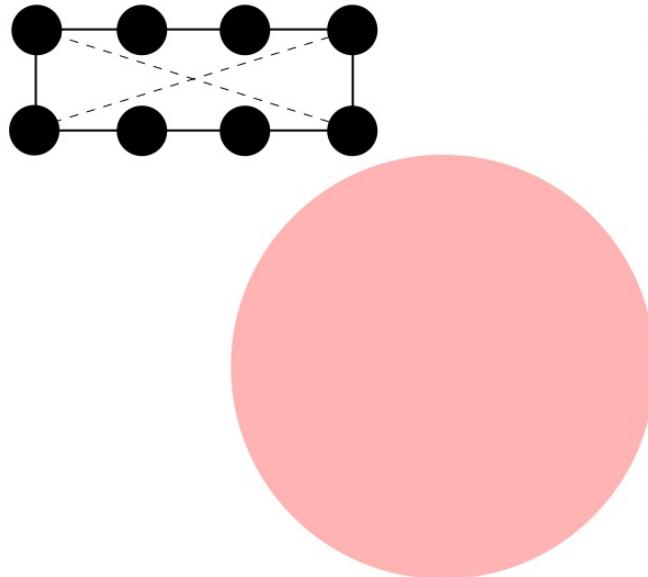
When `boolean_in = t`, elements with any of their parts (in the coarse-grained domain) and atoms (in the atomistic domain) inside the `modify_shape` are deleted (`delete`) or refined to atomic scale (`cg2at`); otherwise, those outside are. In the coarse-grained domain, an element might have some part of it 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`.



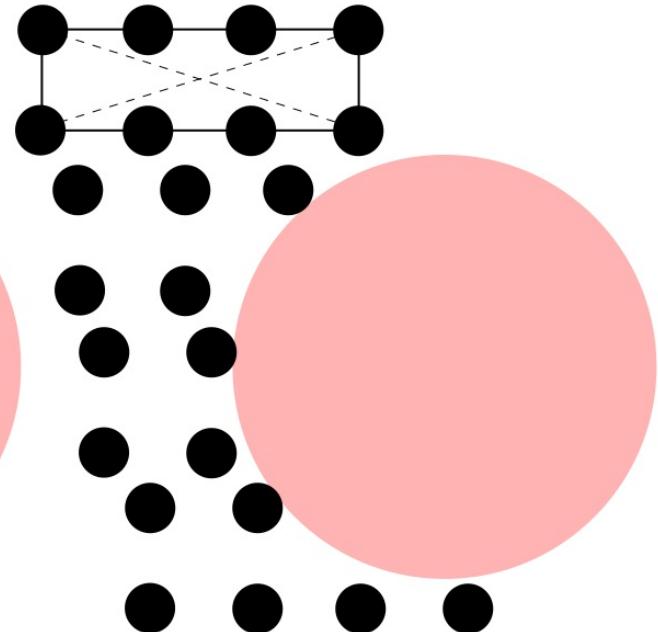
prior to modifications



modify_style = cg2at



modify_style = delete
boolean_delete_filled = f



modify_style = delete
boolean_delete_filled = t

Also note that while *delete* applies to both atomistic and coarse-grained domains, *cg2at* applied to the coarse-grained domain only. Different from the [group](#) command in which the user should pay attention to the difference between *element* and *node*, a modification

follows one simple rule in the coarse-grained domain: an element and all its nodes are selected if any interpolated atom of this element is inside (if `boolean_in = t`) or outside (if `boolean_in = f`) `modify_shape`.

`modify_centroid_x`, `modify_centroid_y`, and `modify_centroid_z`, in units of the component of the lattice periodicity length vector l'_0 and with respect to the lower boundaries of the simulation cell along the corresponding direction, 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 `modify_shape`. Both `modify_radius_large` and `modify_radius_small` are in units of the maximum lattice periodicity length l'_{\max} .

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.

Second syntax (`modify_style = dislocation`)

The second syntax builds a full dislocation into the simulation cell, with nodes/atoms displaced following the isotropic displacement field. In [FCC and BCC lattices](#), a full dislocation has a Burgers vector magnitude of $(\sqrt{2}/2)a_0$ and $(\sqrt{3}/2)a_0$, respectively, where a_0 is the `lattice_constant`. Multiple `modify` commands with `modify_style = dislocation` can be employed to introduce multiple dislocations.

`line_axis` and `plane_axis` are the dislocation line axis and the plane normal axis, respectively. They cannot be the same.

`modify_centroid_x`, `modify_centroid_y`, and `modify_centroid_z`, in units of the component of the lattice periodicity length vector l'_0 and with respect to the lower boundaries of the simulation cell along the corresponding direction, are the coordinates of the origin with respect to which the displacement field is built. For example, if one wants to build a

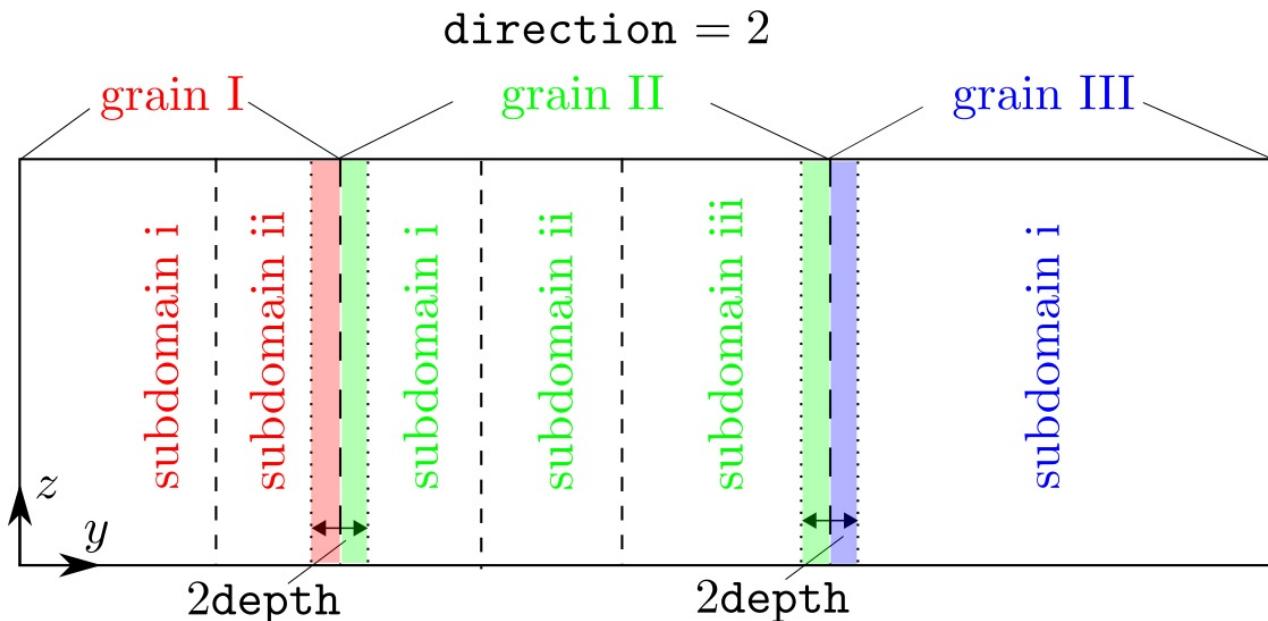
dislocation passing through the centroid of the simulation cell, these three quantities should be at the centroid. Note that in the coarse-grained and atomistic domains, the slip plane, which contains the dislocation along `line_axis` and has a normal direction `plane_axis`, should be located between two adjacent elements and two atomic layers, respectively.

`dis_angle` and `poisson_ratio` are the dislocation character angle (in degrees) and the isotropic Poisson's ratio of the material, respectively.

Third syntax (`modify_style = cutoff`)

The third syntax deletes one atom from a pair of atoms (either real atoms in the atomistic domain or interpolated atoms in the coarse-grained domain) when they are too close, at the grain boundary. The situation that some atoms are too close to each other is usually a result of the `overlap` or `grain origin displacements`. Among all `modify` commands, there should be no more than one with `modify_style = cutoff`.

`depth` and `tolerance`, in units of the component of the [lattice periodicity length vector \$l'_0\$](#) along the [grain stack direction](#), specify the size of the target region and the cutoff distance, respectively, as shown in the figure below. In most cases, `tolerance` should not be larger than or equal to the first nearest neighbor distance in a perfect lattice.



At each grain boundary, a check is first conducted, within the region set by `depth` along the [grain stack direction](#), on both the real atoms in the atomistic domain or the interpolated atoms in the coarse-grained domain. In the figure above, (i) all atoms in the red shaded

region (grain I) will be run against those in the left green shaded region (grain II), (ii) all atoms in the right green shaded region (grain II) will be run against those in the blue shaded region (grain III). 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, the user will get an error message because it is impossible to delete a single interpolated atom from an element, which would violate the hyperelastic body assumption of 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`, in which case there is no need for the modification information.

Related files

`model_modify.f90` , `model_modify_interpo.f90` , `model_cutoff.f90` , `model_cutoff_bd.f90` , `model_dislocation.f90` , `model_cg2at.f90` , `model_delete.f90` , and `model_rearrange.f90` .

Default

None.

Acknowledgements

[Ji Rigelesaiyin](#) and [Jaber R. Mianroodi](#) are acknowledged for helpful discussions in implementing the second syntax.

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 CAC simulations, 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 maintain neighbor lists because their force/energy etc. are not calculated.

`bin_size`, in Angstrom, sets the length of the bin, which adds to the cutoff distance r_c of the [interatomic potential](#). All atoms within $r_c + \text{bin_size}$ from an atom/integration point are the neighbors of this atom. Note that each edge length of the [processor domain](#) cannot be smaller than $2 \cdot (r_c + \text{bin_size})$.

`neighbor_freq` is the frequency with which a check of whether the neighbor list should be updated is conducted. The neighbor lists of all atoms/integration points are 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`.

Related commands

The initial number of neighboring atoms per atom/integration point is set in the [limit](#) command.

Related files

`neighbor_init.f90` and `update_neighbor.f90`

Default

```
neighbor 1. 200
```

potential

Syntax

```
potential potential_type
```

- `potential_type = lj or eam`

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

Examples

```
potential lj
potential eam
```

Description

This command sets the interatomic potentials. Currently, a CAC simulation accepts 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.

Related commands

None.

Related files

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

Default

potential

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*, which refines all or some elements into atomic scale, respectively.

When `refine_style` = *all*, all elements in the coarse-grained domain are refined into atomic scale. This is used when, e.g., the user wants to perform an equivalent full atomistic simulation using the PyCAC code. Currently, this option is correctly triggered 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

$$(6 + 1)^3 = 343 \text{ atoms.}$$

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 number of groups to be refined is `refine_group_number`. As a result, the number of

`group_in_#.id` files, which were renamed from the `group_out_#.id` files that were created automatically in previous CAC simulations when the total number of groups > 0, should be larger than or equal to `refine_group_number`.

Note that `refine_group_number` is irrelevant when `refine_style = all`, and `unitype` is irrelevant when `refine_style = group`.

Related commands

This command becomes irrelevant when `boolean_restart_refine = f`, in which case there is no need for the refinement information.

Related files

`refine_init.f90`

Default

None.

restart

Syntax

```
restart boolean_restart boolean_restart_refine
```

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

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

Examples

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

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 `boolean_restart_refine` becomes *f* regardless of its value set in this command.

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

Related commands

When `boolean_restart_refine` = *f*, the `refine` command becomes irrelevant, in which case there is no need for the refinement information.

Related files

`read_restart.f90` and `write_restart.f90`

Default

```
restart 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 simulation step of dynamic/hybrid CAC simulations or the total loading increment of quasistatic CAC simulations.

`time_step`, in ps, is the time step in [dynamic CAC](#) simulations, dynamic part in hybrid CAC simulations, and some quasistatic simulations when `mini_style` = *fire* or *qm*. It is also used in the [fix](#) command when `assign_style` = *disp*.

Related commands

`time_step` becomes irrelevant when `simulation_style` = *statics* with `mini_style` = *cg* or *sd*.

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

run

```
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 CAC 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 a CAC simulation is 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` = positive integer

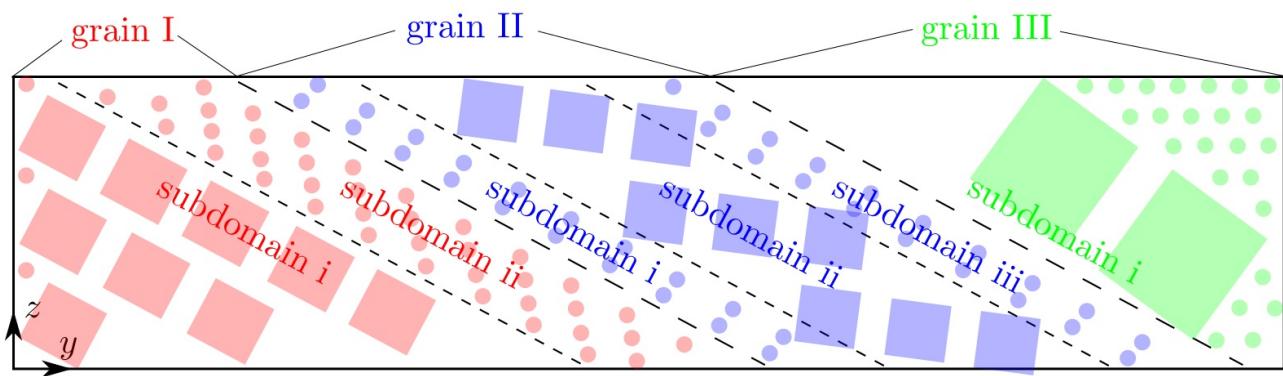
Examples

```
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. 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` .

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:



Note that in this figure, the atoms in subdomain i/grain I and subdomain i/grain III are employed to fill in the otherwise jagged interstices, because either `boolean_y` = *f* or `y` = *p*.

The size of each subdomain and the unit type in each subdomain in each grain is specified in the `unit_num` and `unit_type` commands, respectively. The grains and subdomains are stacked along a prescribed `direction`. 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*, in which case there is no need for the subdomain information.

Related files

`box_init.f90`

Default

```
subdomain 1 1
```

temperature

Syntax

```
temperature boolean temp
```

- `boolean = t` or `f`
 - `t` is true
 - `f` is false
- `temp =` non-negative real number

Examples

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

Description

This command sets whether the temperature is kept a constant in the system (`boolean`) for the [dynamic and hybrid](#) CAC simulations; and if yes, what is the desired temperature in K (`temp`).

A constant zero temperature is maintained in the system only when `dyn_style = ld` or `qd`, i.e., Langevin dynamics or quenched dynamics. Note that in this case, [the equation of motion for the Langevin dynamics reduces to that for the damped dynamics](#).

A constant finite temperature is maintained in the system only when `dyn_style = ld`, i.e., Langevin dynamics. The user will get a warning message if `temp` is finite and if `dyn_style = qd`.

In [quasi-static](#) simulations, `boolean` must be `f` and the temperature, which is effectively 0 K, is irrelevant.

temperature

Related commands

If `boolean = t` and `dyn_style = vv`, the user will get a warning message and the temperature `temp` becomes irrelevant, because the Velocity Verlet option cannot maintain a constant temperature.

Related files

```
thermostat.f90 , langevin_dynamics.f90 , and langevin_vel.f90
```

Default

```
temperature t 10.
```

unit_num

Syntax

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

- grain_id , subdomain_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]}
```

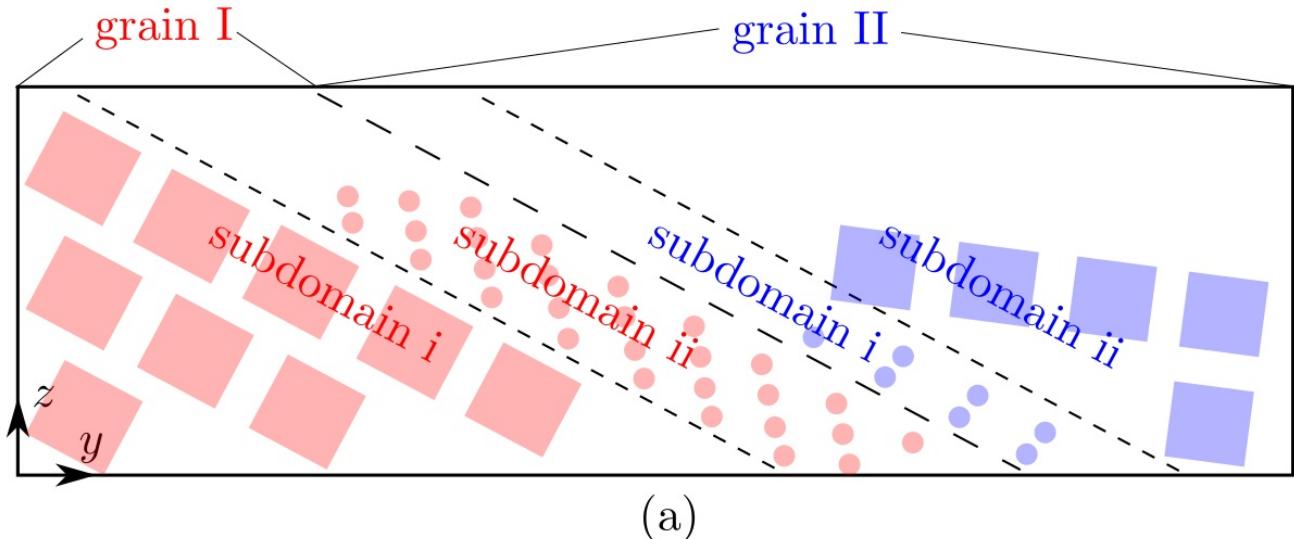
Description

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 units 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.

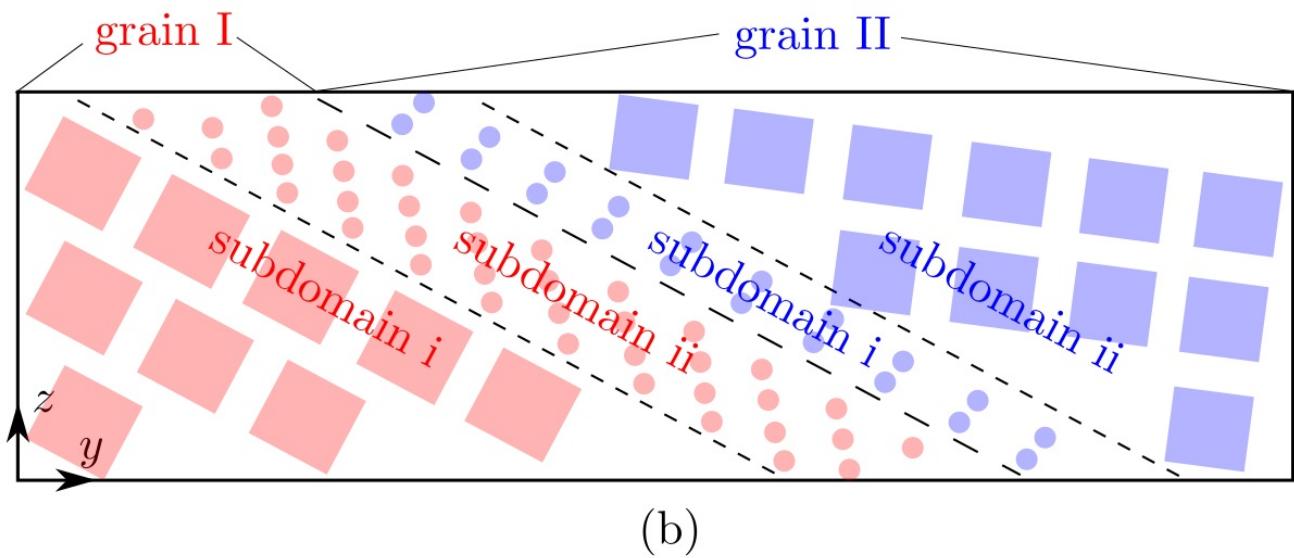
Similar to the unit_type command, this command consists of two loops. The outer loop, illustrated by {} , is based on grain; the inner loop, illustrated by [] , is based on subdomain. Note that the curly brackets { and } as well as the square brackets [and] in the syntax/examples are to separate different grains and subdomains, the number of which are grain_number and subdomain_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, in which the subdomain i/grain I has a

larger z length than the other subdomains. Since the `grain_stack_direction` is y, the size of all other subdomains along the x and z directions will be increased to match that of the subdomain i/grain I, respectively, as shown in Fig. (b) below.



(a)



(b)

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

Related commands

`unit_num`

This command becomes irrelevant when `boolean_restart` = t , in which case there is no need for the subdomain information.

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 [unit type](#) 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 grains and subdomains, the number of which are `grain_number` and `subdomain_number`, respectively; all brackets should not be included in preparing `cac.in`.

The number of atoms per unit is $(\text{unitype} + 1)^3$, 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 of the first order Gaussian quadrature employed to solve the [governing equations](#), (ii) it must be ≥ 4 because of the second nearest neighbor (2NN) element with 125 integration points (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. Within each grain, the maximum `subdomain_id` must equal the corresponding `subdomain_number`.

Related commands

This command becomes irrelevant when `boolean_restart = t`, in which case there is no need for the subdomain information.

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 corresponding jagged interstices, resulting in flat boundaries normal to the corresponding direction, unless the boundaries were already flat with rhombohedral elements, e.g., parallel to a {111} plane in an FCC lattice or to a {110} plane in a BCC lattice. Examples of the filled atoms include Fig. C27(b) of [Xu et al., 2015](#) and the figure for the `subdomain` command in which the atoms are filled in at the leftmost and rightmost simulation cell boundaries. 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 CAC simulations.

This command becomes irrelevant when `boolean_restart` = *t*, in which case there is no need for the boundary shape information.

Related files

`model_init.f90`

Default

```
zigzag t t t
```

Post-processing

A CAC simulation [outputs](#) a lot of files, most of which are `dump.*` and `*.vtk` 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 compatible 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

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 by the [output component of the Python scripting interface](#), 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 some sample problems:

- Stationary dislocations
- Dislocation migration
- Screw dislocation cross-slip
- Dislocation multiplication
- Dislocation/obstacle interactions
- Dislocation/stacking fault interactions
- Dislocation/coherent twin boundary interactions

Stationary dislocations

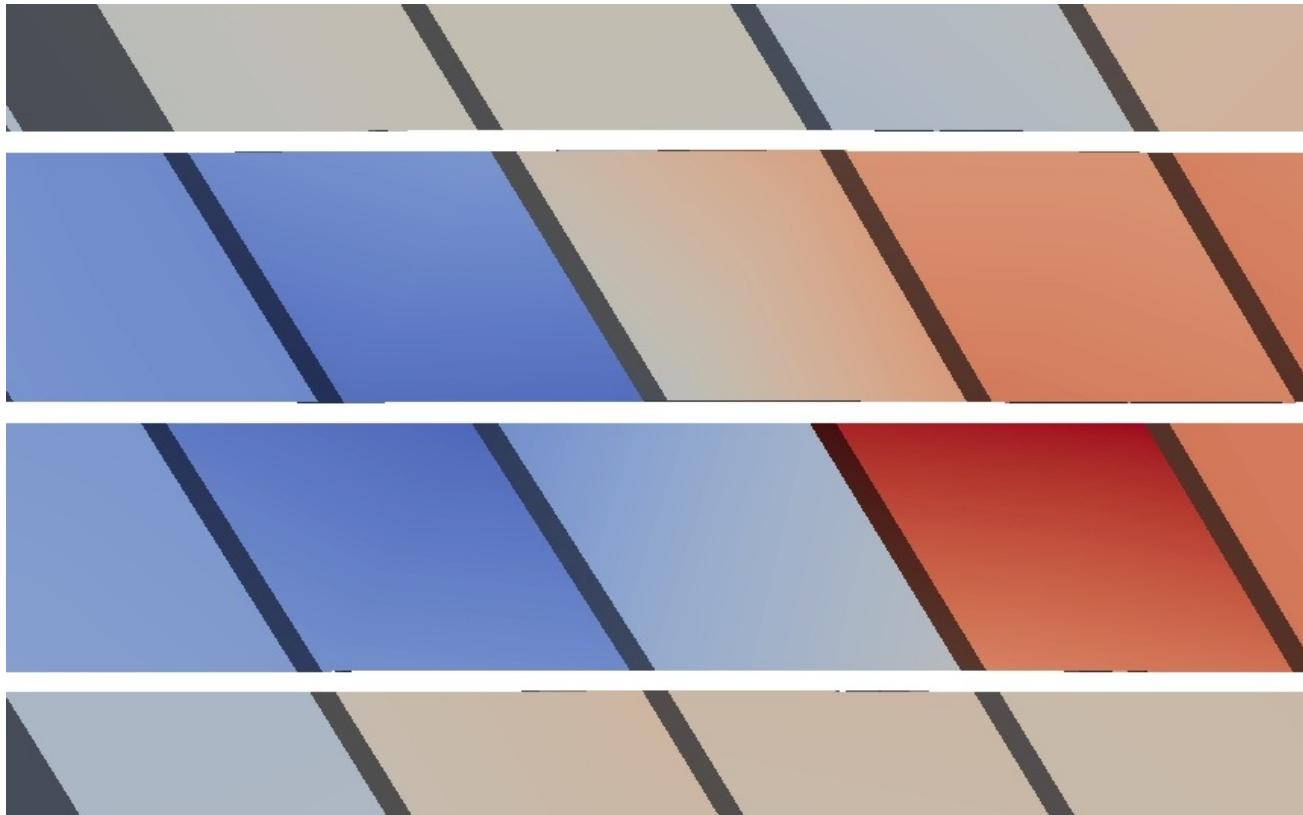
FCC Al, [Mishin EAM potential](#), 2197 atoms per element in the coarse-grained domain.

Results using larger models and/or in other metals are published in [Xu et al., 2015](#) and [Xu et al., 2016](#).

In the figures below, the atoms that fill in the jagged interstices are not shown for a better visualization of the elements; the nodes are colored by the in-plane shear stress. Langevin dynamic simulation at 10 K is carried out, with periodic energy minimization using the conjugate gradient method.

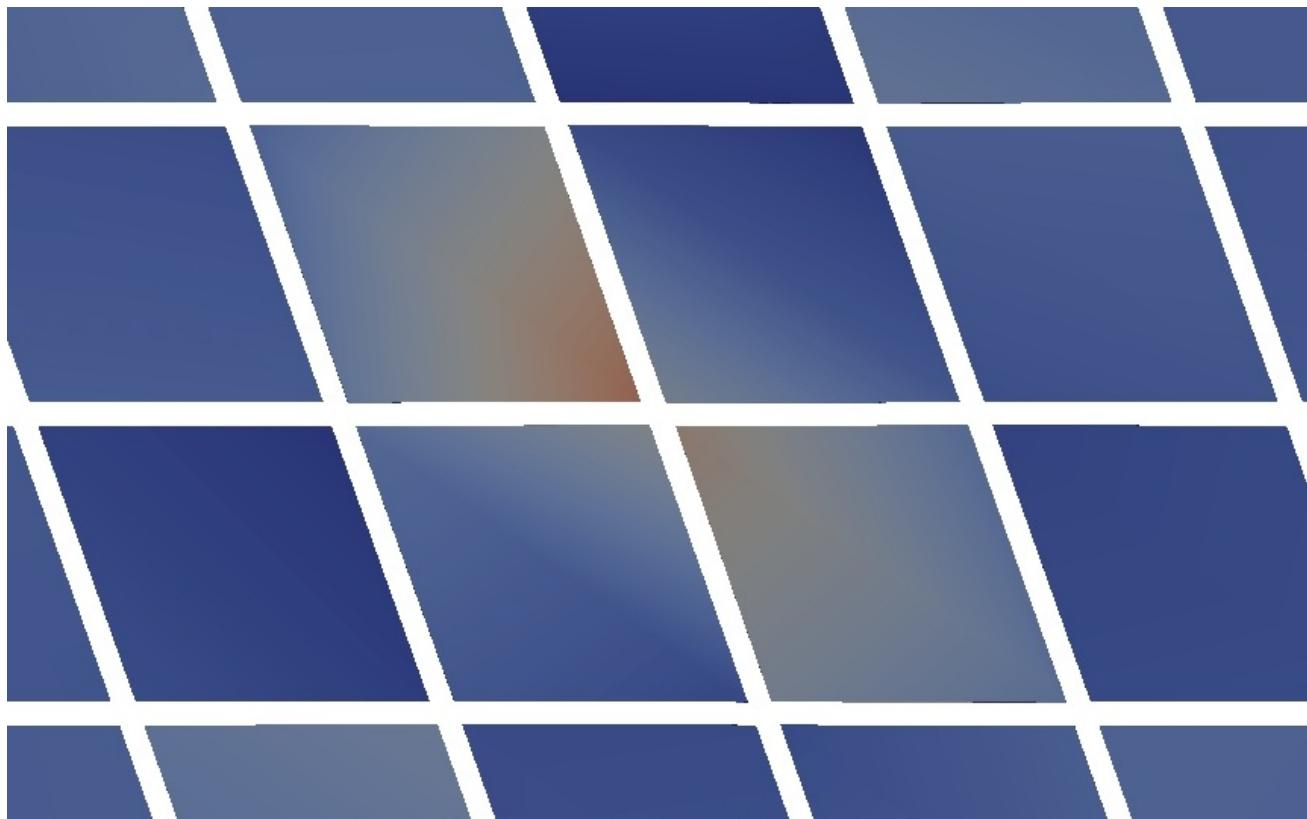
Edge dislocation

The figure below and the [log file](#) are produced using the [input file](#) and rendered by [ParaView](#).



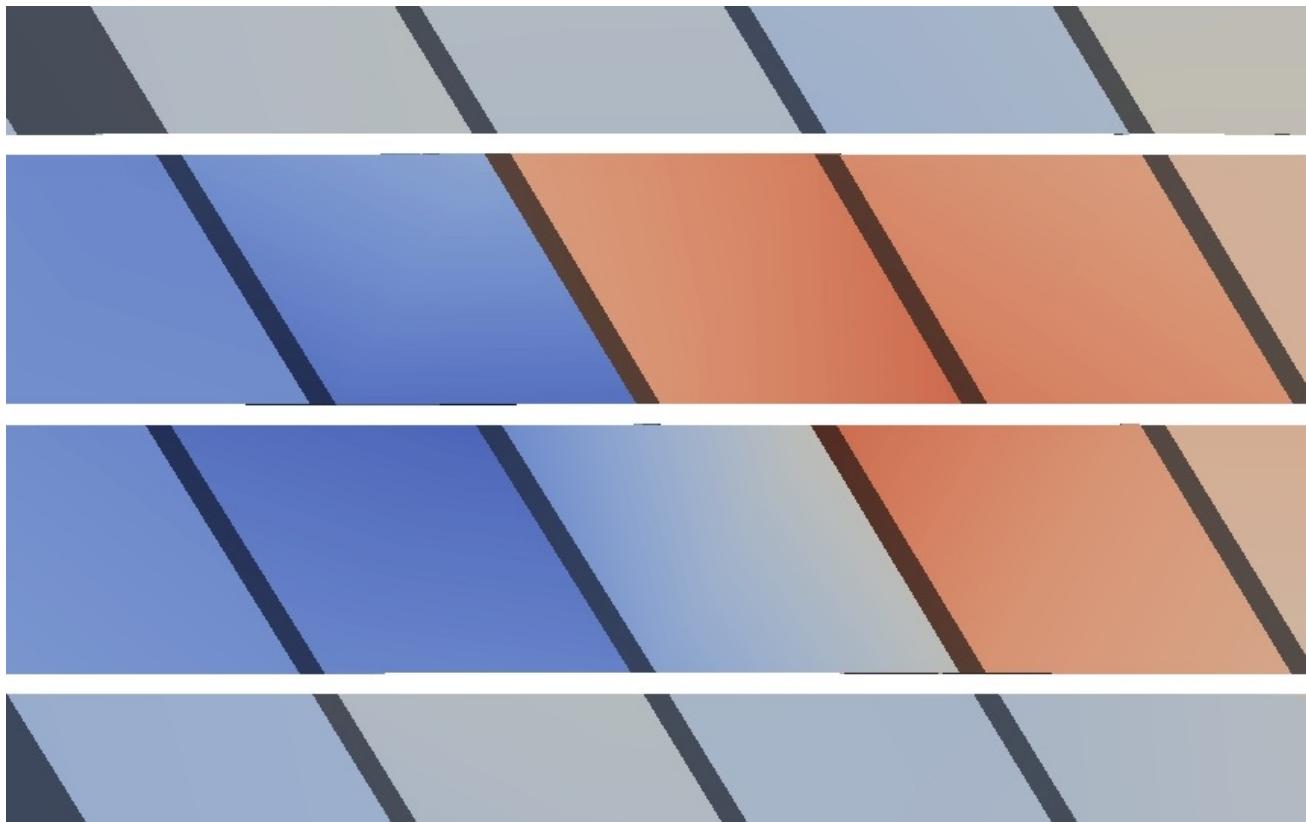
Screw dislocation

The figure below and the [log file](#) are produced using the [input file](#) and rendered by [ParaView](#).



30° mixed type dislocation

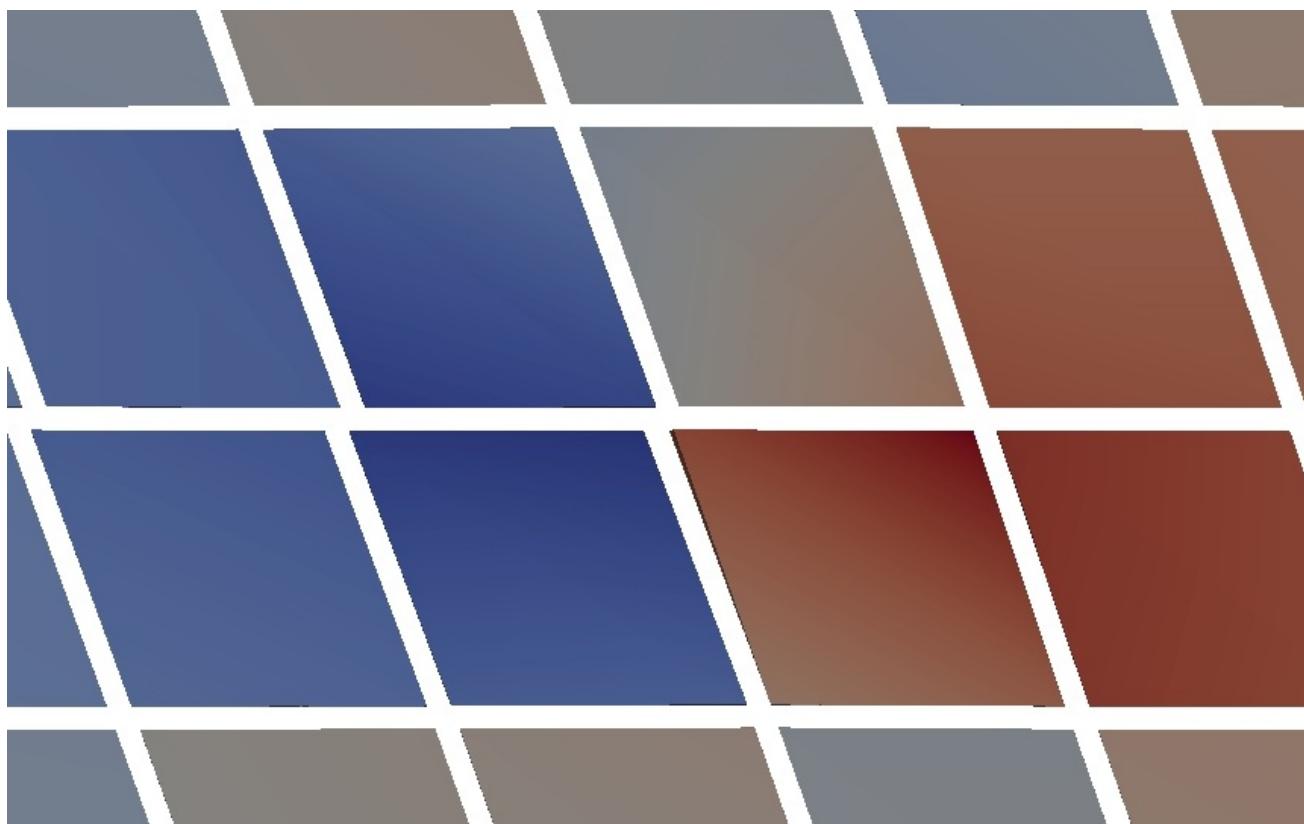
The figure below and the [log file](#) are produced using the [input file](#) and rendered by [ParaView](#).



60° mixed type dislocation

The figure below and the [log file](#) are produced using the [input file](#) and rendered by [ParaView](#), similar to Fig. 8(c) of [Xu et al., 2015](#).

Stationary dislocations

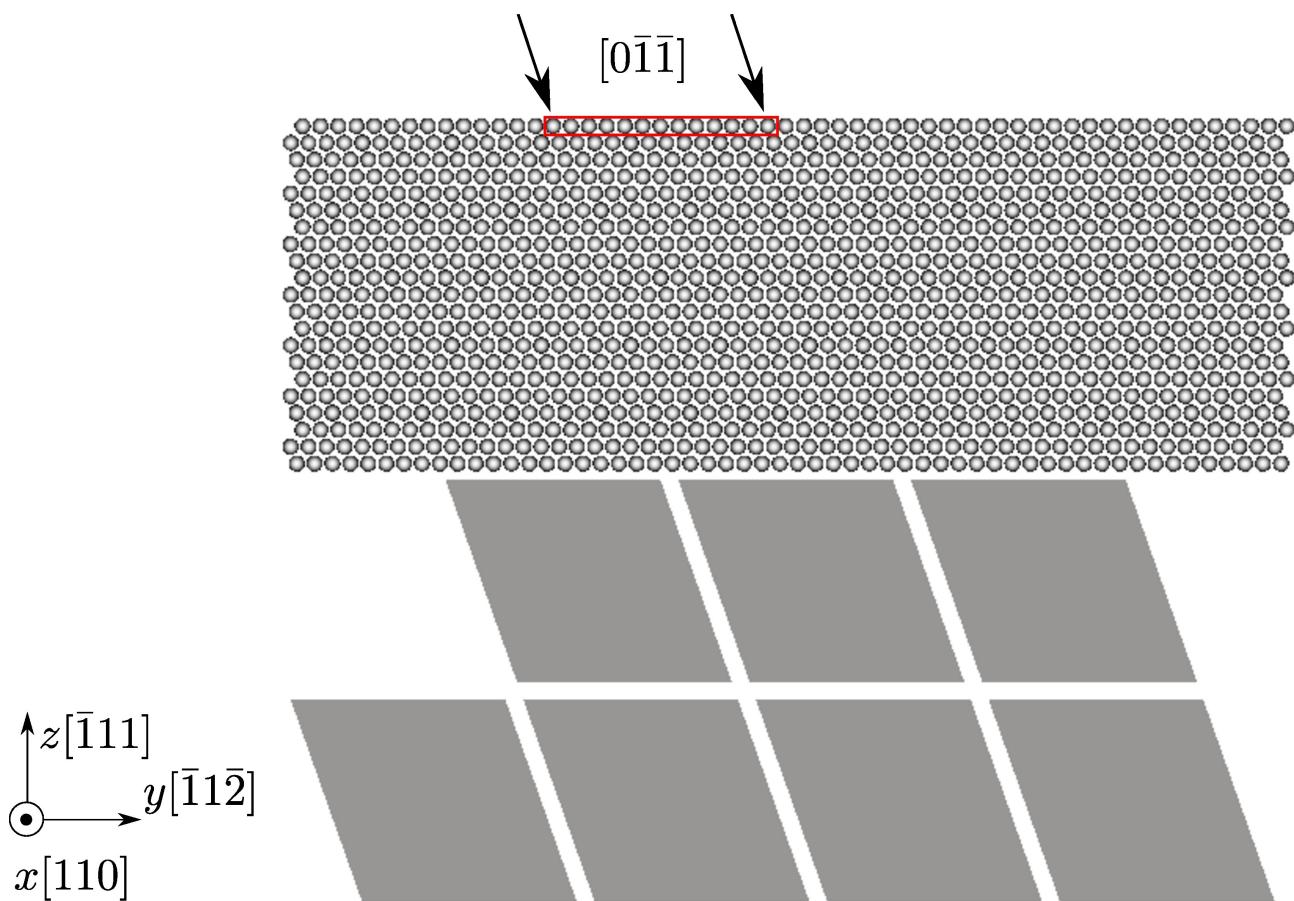


Dislocation migration across the atomistic/coarse-grained domain interface

FCC Cu, [Mishin EAM potential](#), 2197 atoms per element in the coarse-grained domain. Results using larger models are published in Sec. 5.4 of [Xu et al., 2015](#).

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

In the figure below, an indenter (red box) is displaced continuously along the $[0\bar{1}\bar{1}]$ direction to nucleate dislocations from the free surface in the atomistic domain. Note that the atoms that fill in the jagged interstices are not shown for a better visualization of the elements, similar to Fig. 14(b) of [Xu et al., 2015](#). The dislocations then migrate into the coarse-grained domain. Energy minimization using the conjugate gradient method is conducted at every simulation step.

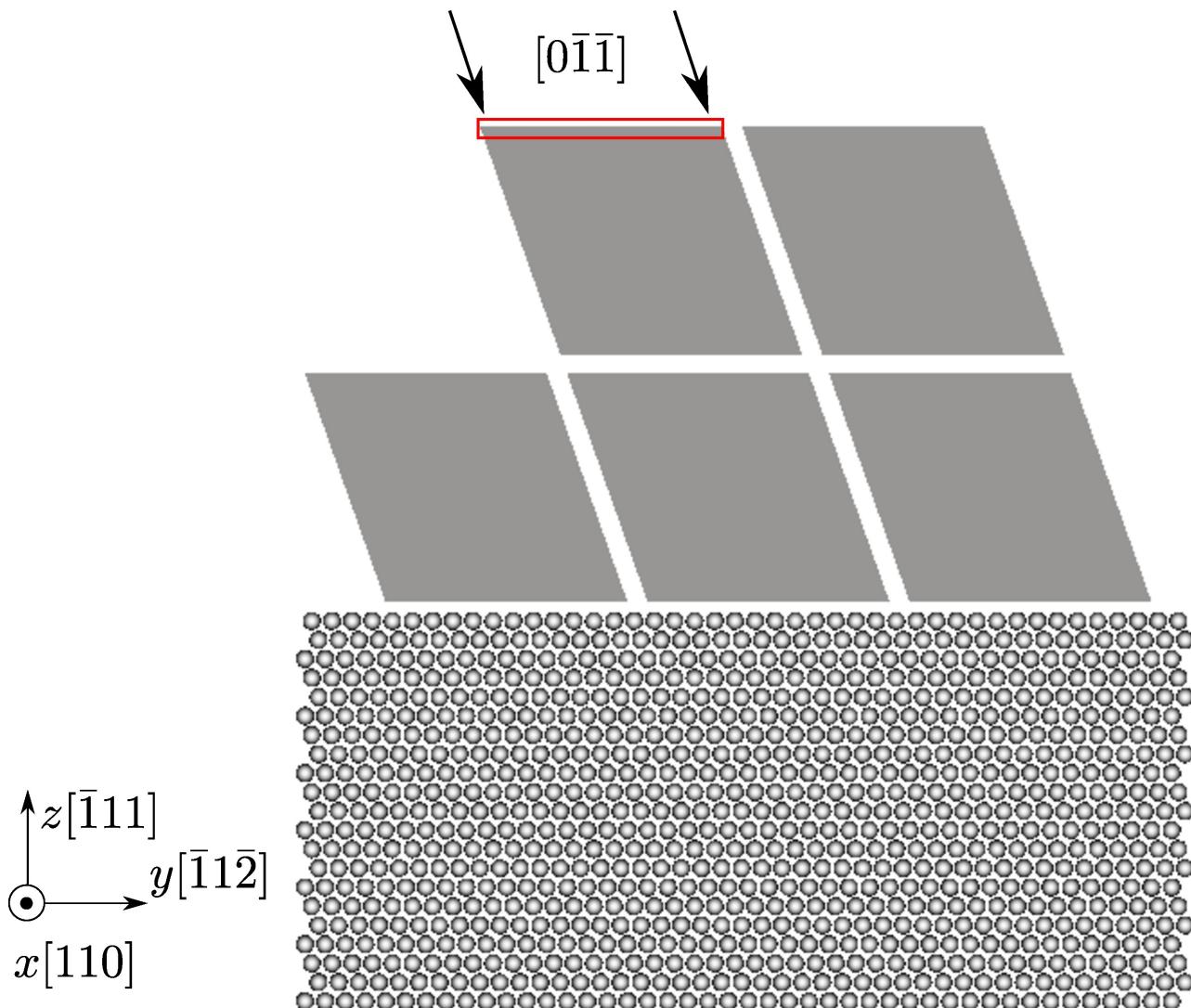


The movie below and the [log file](#) are produced using the [input file](#) and rendered by **OVITO**:



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

In the figure below, an indenter (red box) is displaced continuously along the $[0\bar{1}\bar{1}]$ direction to nucleate dislocations from the free surface in the coarse-grained domain. Note that the atoms that fill in the jagged interstices are not shown for better visualization of the elements, similar to Fig. 14(c) of [Xu et al., 2015](#). The dislocations then migrate into the atomistic domain. Energy minimization using the conjugate gradient method is conducted at every simulation step.



The movie below and the [log file](#) are produced using the [input file](#) and rendered by [OVITO](#).

Dislocation migration

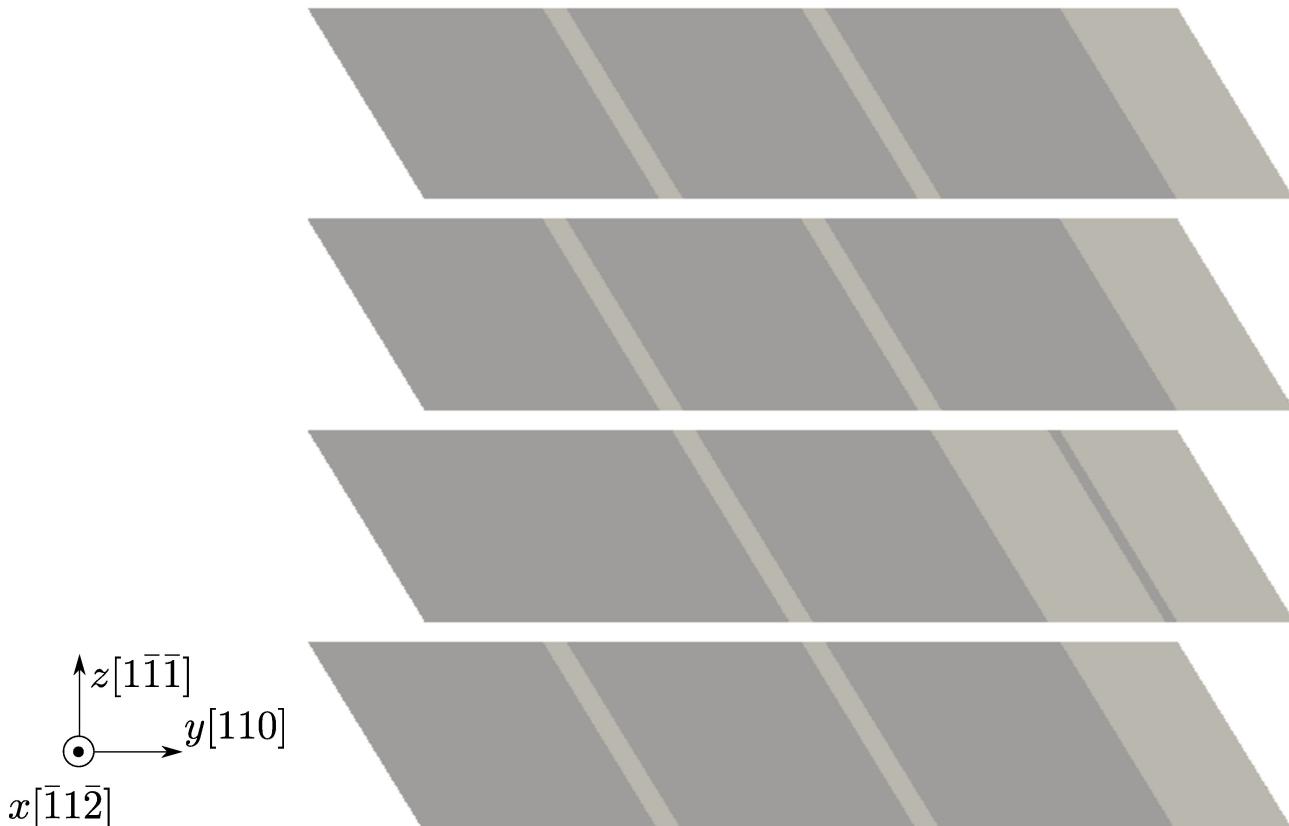


Screw dislocation cross-slip

FCC Cu, [Mishin EAM potential](#), 1331 atoms per element in the coarse-grained domain.

Results using larger models are published in [Xu et al., 2017](#).

In the figure below, the atoms that fill in the jagged interstices are not shown for a better visualization of the elements. In the Langevin dynamic simulation, a screw dislocation on the $(\bar{1}\bar{1}\bar{1})$ plane is first created; then subject to a γ_{zy} simple shear strain, it crosses slip onto the $(1\bar{1}\bar{1})$.



The movie below and the [log file](#) are produced using the [input file](#) and rendered by [OVITO](#):

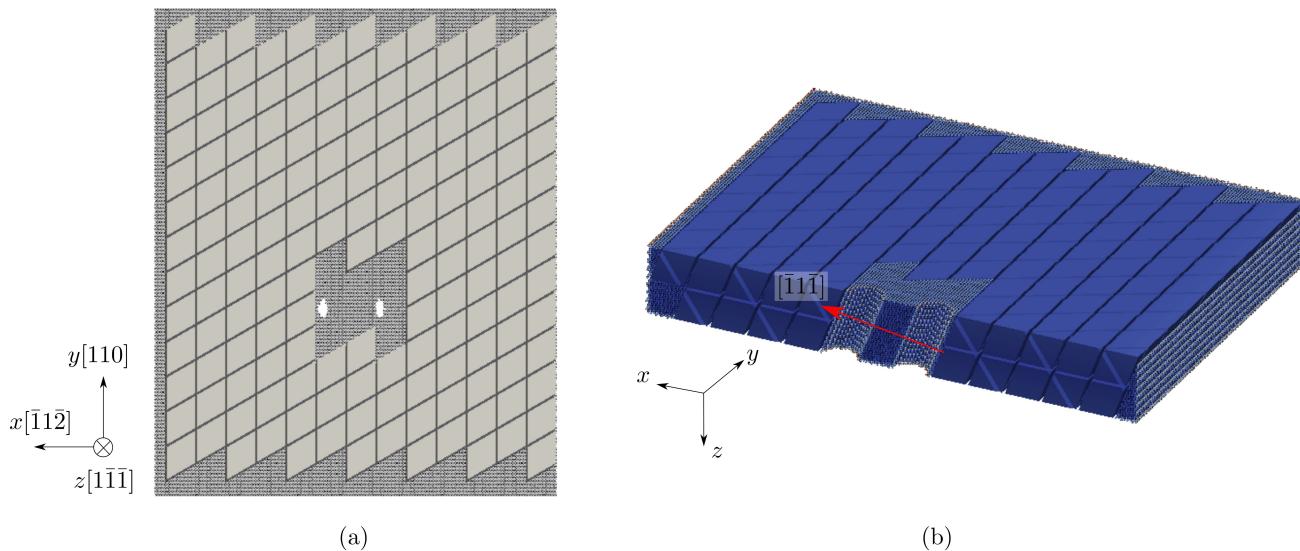
Screw dislocation cross-slip



Dislocation multiplication from a Frank-Read source

FCC Al, [Mishin EAM potential](#), 2197 atoms per element in the coarse-grained domain. Results using larger models are published in [Xu et al., 2016](#) and [Xu et al., 2016](#).

In the figure below, two cylindrical holes are carved out to serve as the Frank-Read source. The atoms and elements in figure (b) are colored by the atomic and nodal energy, respectively, and are sliced on the xz plane to highlight the holes. In the hybrid simulation, an edge dislocation is first created between the two holes; then subject to a γ_{zy} simple shear strain, it bows out and form a dislocation loop, leaving behind another edge dislocation segment between the two holes.



The movie below and the [log file](#) are produced using the [input file](#) and rendered by [OVITO](#):

Dislocation multiplication

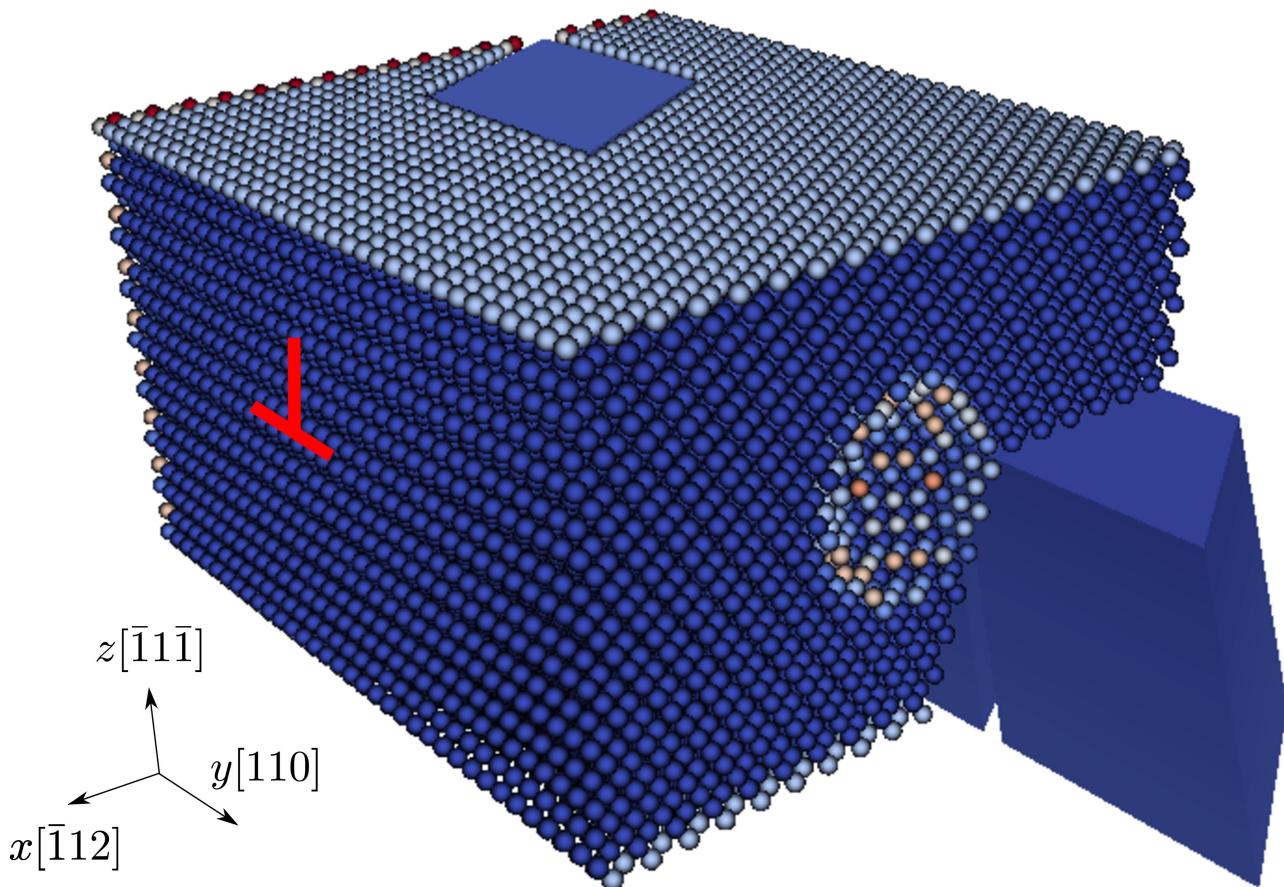


Dislocation/obstacle interactions

FCC Ni, [Mishin EAM potential](#), 2197 atoms per element in the coarse-grained domain. The spherical obstacle, with a radius of about 1 nm, is either a void or a precipitate. Results using larger models were presented at the [2017 MRS Spring Meeting](#).

Dislocation/void interactions

In the figure below, the atomistic domain is sliced on the xz plane for a better visualization of the void (atoms are colored by the atomic energy in the initial configuration). In the Langevin dynamic simulation, an edge dislocation on the $(\bar{1}\bar{1}\bar{1})$ plane is first created; then subject to a γ_{zy} simple shear strain, it migrates toward the void and bypasses it following the shearing mechanism.

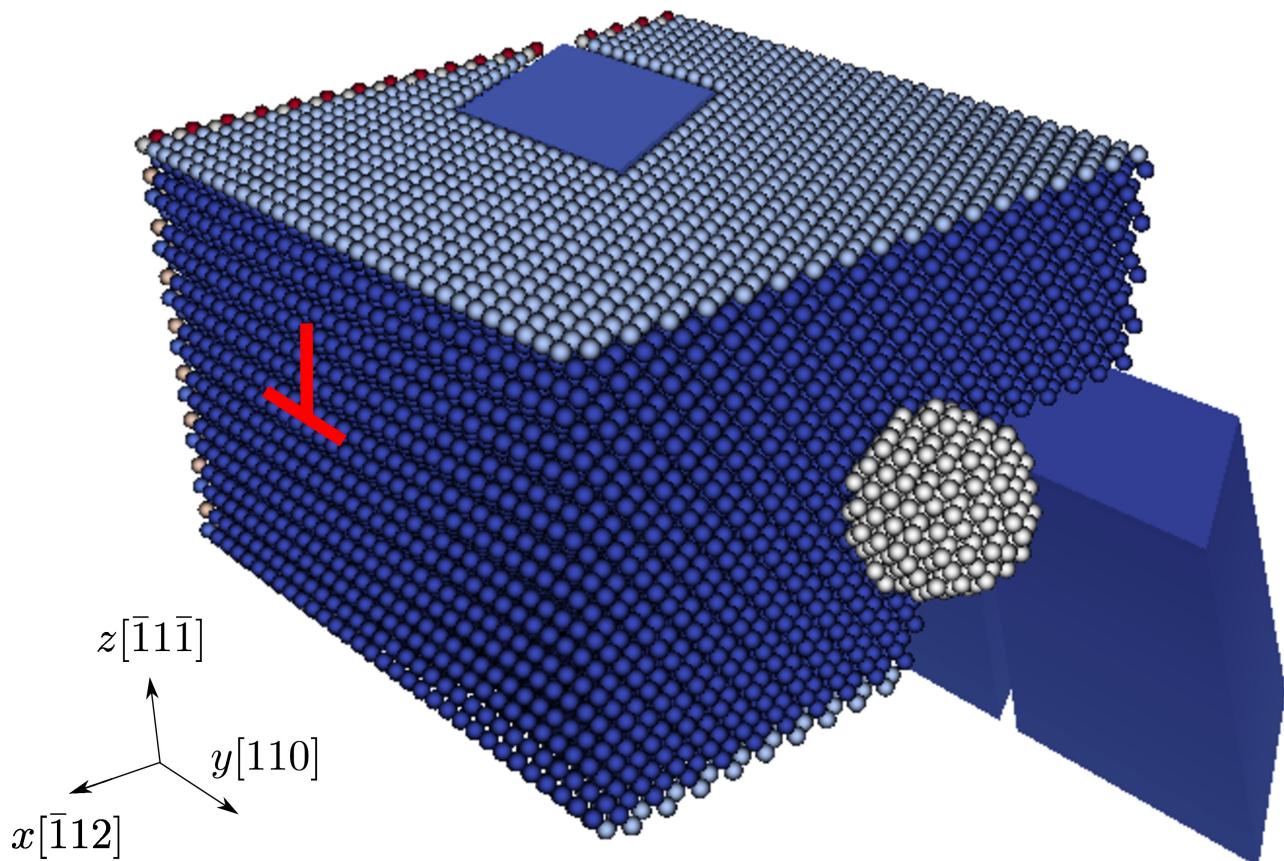


The movie below and the [log file](#) are produced using the [input file](#) and rendered by [OVITO](#):



Dislocation/precipitate interactions

In the figure below, the atomistic domain is sliced on the xz plane for a better visualization of the precipitate (atoms colored in white). In the Langevin dynamic simulation, an edge dislocation on the $(\bar{1}\bar{1}\bar{1})$ plane is first created; then subject to a γ_{zy} simple shear strain, it migrates toward the precipitate and bypasses it following the Orowan looping mechanism.



The movie below and the [log file](#) are produced using the [input file](#) and rendered by [OVITO](#):



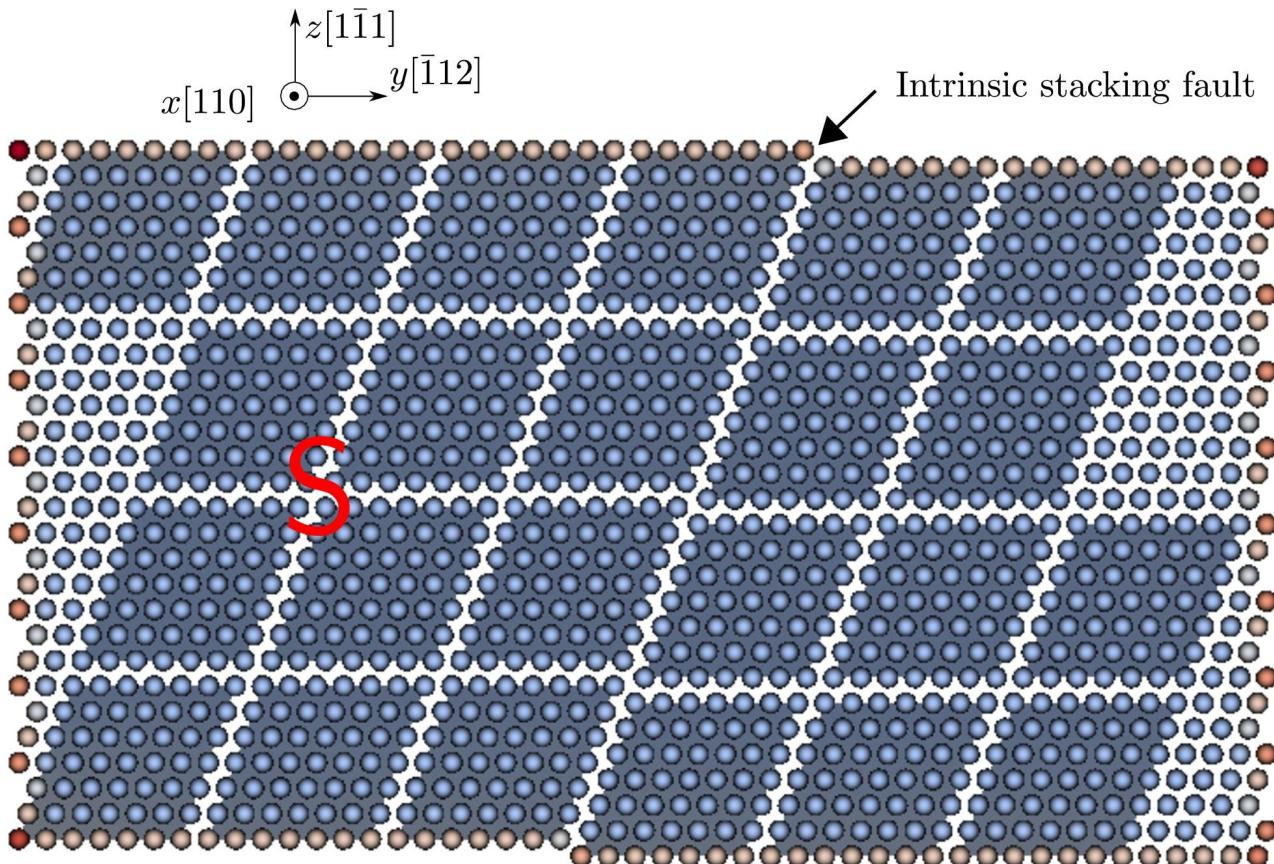
Note that the screw components of the Orowan loop begin to cross slip at about 24 s. The precipitate is not shown here.

Dislocation/stacking fault interactions

FCC Ag, [Williams EAM potential](#), 343 atoms per element in the coarse-grained domain.

Results using larger models are published in [Xu et al., 2017](#).

In the figure below, the atoms that fill in the jagged interstices are not shown for a better visualization of the elements. In the Langevin dynamic simulation, a screw dislocation on the $(1\bar{1}1)$ plane and an intrinsic stacking fault on the $(\bar{1}11)$ plane are first created; then subject to a γ_{zx} simple shear strain, the dislocation moves toward and is then transmitted across the stacking fault directly.



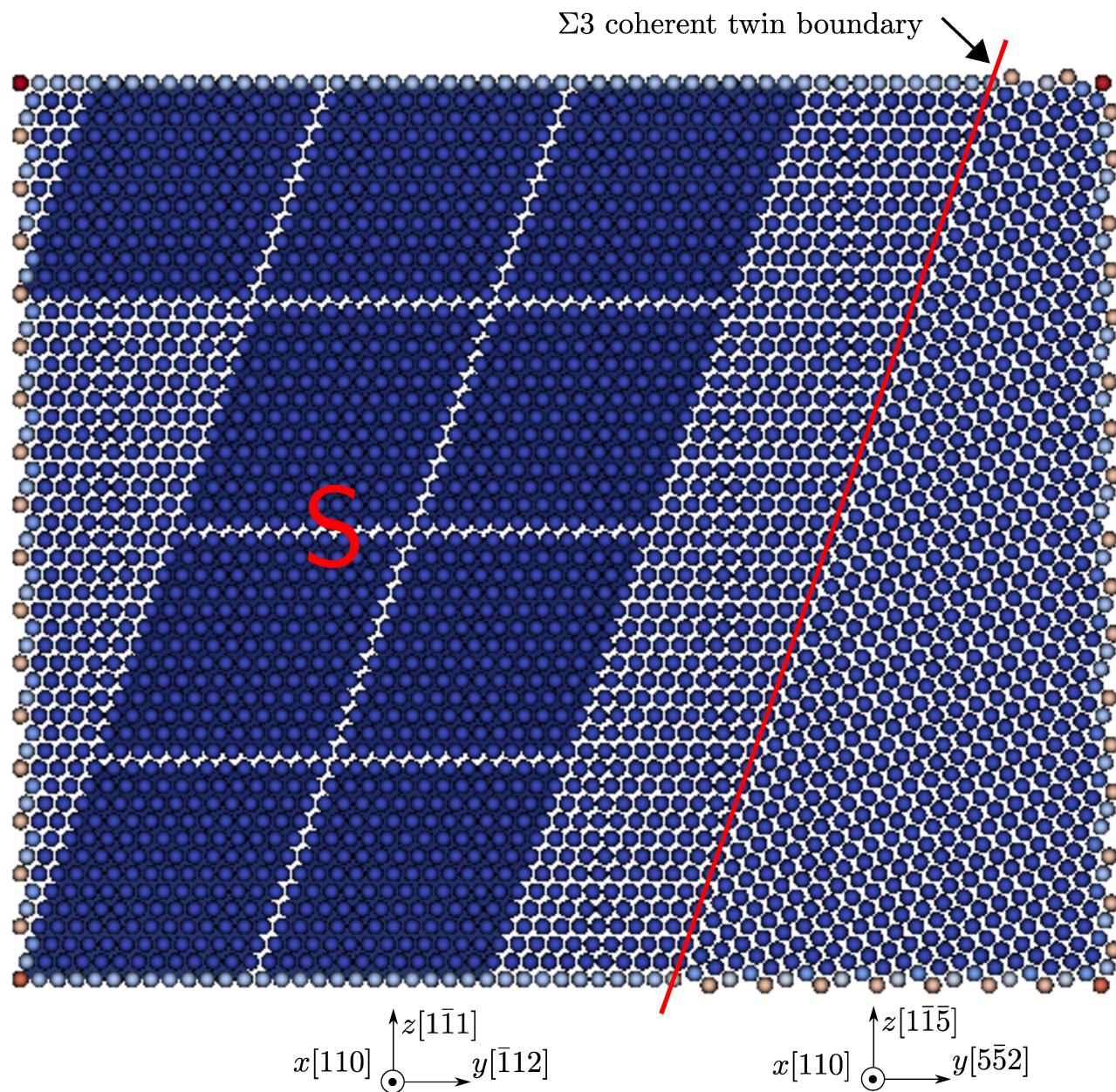
The movie below and the [log file](#) are produced using the [input file](#) and rendered by [OVITO](#):



Dislocation/coherent twin boundary interactions

FCC Ni, [Mishin EAM potential](#), 2197 atoms per element in the coarse-grained domain. Results using larger models and/or in other metals are published in [Xu et al., 2016](#) and [Xu et al., 2017](#).

In the figure below, the atoms that fill in the jagged interstices are not shown for a better visualization of the elements. In the Langevin dynamic simulation, a screw dislocation on the $(1\bar{1}1)$ plane is first created; then subject to a γ_{zx} simple shear strain, the dislocation moves toward and is then transmitted across the $\Sigma3\{111\}$ coherent twin boundary.



The movie below and the [log file](#) are produced using the [input file](#) and rendered by [OVITO](#):



Miscellanies

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

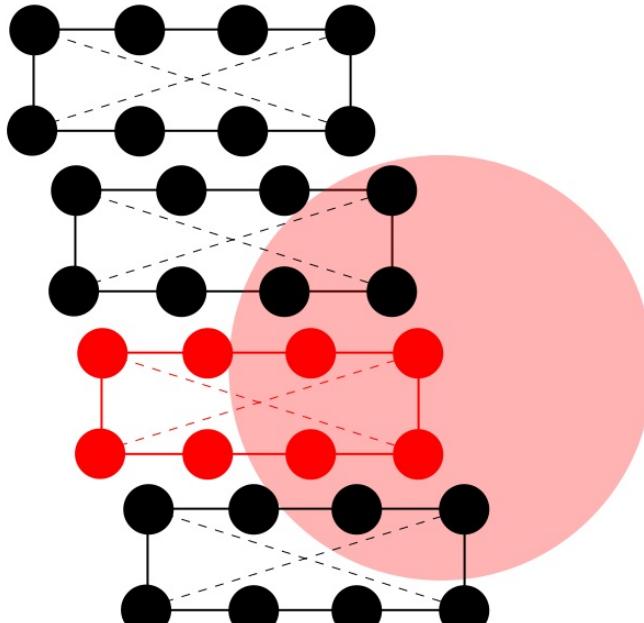
element vs node

In the `group` command, `style_cg` can be either *element*, *node*, or *null*.

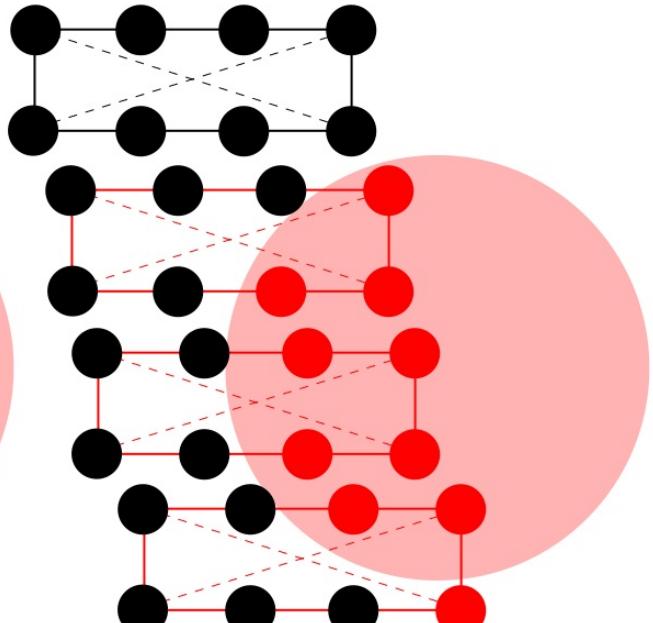
For *element*, if the centroid of an element is inside or outside (depending on `boolean_in`) `group_shape`, this element and all its nodes belong to the group.

For *node*, if some nodes of an element is inside or outside (depending on `boolean_in`) `group_shape`, this element and these nodes belong to the group.

The difference between *element* and *node* is explained in the figure below, where red elements (solid lines) and nodes (small spheres) belong to the group (large sphere) with `group_shape = sphere`.



`style_cg = element`



`style_cg = node`

lattice periodicity length

The length of periodicity of the lattice is the minimum distance at which the lattice repeats itself. For example, the lattice constant a_0 in cubic crystal systems is the lattice periodicity length along the $\langle 100 \rangle$ directions.

Once the crystallographic orientations are set, e.g., the x axis in the first grain has an orientation of $[abc]$, the lattice will repeat itself at every $\sqrt{a^2 + b^2 + c^2}a_0$ distant along the x direction. However, this distant may not be the smallest lattice periodicity length. For example, when $[abc] = [112]$, $\sqrt{a^2 + b^2 + c^2}a_0 = \sqrt{6}a_0$, yet the smallest lattice periodicity length $l_0 = (\sqrt{6}/2)a_0$.

So how is l_0 calculated for any given $[abc]$? First, one calculates $l = a^2 + b^2 + c^2$. Second, one divides l by 2, then by 2 again, and so on, until the result is not divisible by 2. For example, if $l = 24$, one gets $24/2 = 12$, then $12/2 = 6$, then $6/2 = 3$, then 3 is not divisible by 2. During this process, l and its quotients are divided by 2 for 3 times, then one get an integer $\Delta = 3$. Finally, $l_0 = (\sqrt{l}/\Delta)a_0$. Repeating this process for the remaining orientations results in the lattice periodicity length vector l_0 .

Since each grain has its own crystallographic orientations, each grain has its own l_0 . The length vector along each direction that is the largest in magnitude among all grains is the lattice periodicity length for the simulation cell, l'_0 . The largest component in the l'_0 vector is the maximum lattice periodicity length for the simulation cell, l'_{\max} .

l'_0 and l'_{\max} are the distant units in four commands: [fix](#), [grain_dir](#), [group](#), and [modify](#).

processor rank

In MPI, rank is a logical way of numbering processors. The processor 1 has rank 0, the processor 2 has rank 1, and so on. In the PyCAC code, the integer `root` is set to 0 in `processor_para_module.f90`. The processor 1, i.e., `root`, does heavy lifting in reading, writing, and collecting data from other processors.

shared elements

One issue that does not exist in parallel atomistic simulations but requires special attention in parallel finite element implementations is that in the latter some elements may be shared between neighboring processors. In CAC, this issue originates from the difference in shape between the parallelepipedal processor domain and the rhombohedral finite elements with arbitrary crystallographic orientations, the latter of which also results in the [jagged simulation cell boundaries](#). Instead of having all relevant processors calculate the same quantities (e.g., force, energy, and virial) within a shared element, in the PyCAC code, each relevant processor only calculates quantities of the integration points its domain contains. Then these quantities are summed in the `processor_equiv.f90` subroutine, after which all relevant processors have the same nodal quantities. This simple summation is feasible because of the trilinear shape function employed in the finite elements.

To facilitate the shared element-related calculations, a public array `tag_shared_ele` and a public variable `ele_shared_num` are introduced. For example, processor 3 has 6 local elements, with the 2nd, 4th, and 5th elements shared with other processors, then

```
tag_ele_shared(1) = 0
tag_ele_shared(2) = 1
tag_ele_shared(3) = 0
tag_ele_shared(4) = 2
tag_ele_shared(5) = 3
tag_ele_shared(6) = 0
```

and

```
ele_shared_num = 3
```

The array and the variable, defined in `processor_scatter_cg.f90` and updated in `update_neighbor_cg.f90`, are used in these three subroutines:

`processor_edenshost_intpo.f90`, `processor_equiv.f90`, and `processor_langevin_cg.f90`.

Note that in current PyCAC code, the "shared element communication" process mentioned above does NOT involve a host processor as described in page 123 of [Xu et al.](#), which was for a previous version. The host processor, which has the highest rank among all

processors that share the same element, is indeed used in the code, but only for the purposes of (i) sending the element/node information to the [root processor](#) for output, e.g., in `all_to_one_cg.f90` and `all_to_one_group_cg.f90`, and (ii) calculating certain variables based on the global arrays, e.g., the global force norm `force_norm` calculated in `conjugate_gradient.f90`, `fire.f90`, `langevin_vel.f90`, `quenched_vel.f90`, `quick_mini.f90`, `steepest_descent.f90`, and `update_vel.f90`. For example, when processor 2 and processor 3 share the same element (and thus have the same relevant nodal information), only one of them needs to send the information to root. The host processor is set in the `processor_scatter_cg.f90` subroutine, in which the array `who_has_ele(ie) = .true.` for the host processor and `.false.` for non-host processors, where `ie` is the local element id.

EAM potential

As mentioned [earlier](#), the EAM formulation for the potential energy is

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

where V is the pair potential, F is the embedding potential, and $\bar{\rho}$ is the host electron density, i.e.,

$$\bar{\rho}_i = \sum_{j \neq i}^j \rho_{ij}(r_{ij})$$

where ρ_{ij} is the local electron density contributed by atom j at site i .

Let \mathbf{r}_{ji} be the vector from atom j to atom i with norm r_{ji} ($= r_{ij}$), i.e.,

$$\mathbf{r}_{ji} = \mathbf{r}_i - \mathbf{r}_j$$

$$r_{ji} = \sqrt{(r_i^x - r_j^x)^2 + (r_i^y - r_j^y)^2 + (r_i^z - r_j^z)^2}$$

where

$$\mathbf{r}_j = r_j^x \mathbf{e}^x + r_j^y \mathbf{e}^y + r_j^z \mathbf{e}^z$$

Now, let's prove an important identity,

$$\frac{\partial r_{ji}}{\partial \mathbf{r}_j} = \frac{\partial r_{ji}}{\partial r_j^x} \mathbf{e}^x + \frac{\partial r_{ji}}{\partial r_j^y} \mathbf{e}^y + \frac{\partial r_{ji}}{\partial r_j^z} \mathbf{e}^z = -\frac{r_{ji}^x}{r_{ji}} \mathbf{e}^x - \frac{r_{ji}^y}{r_{ji}} \mathbf{e}^y - \frac{r_{ji}^z}{r_{ji}} \mathbf{e}^z = -\frac{\mathbf{r}_{ji}}{r_{ji}}$$

which will be used in the force formulation derivation later.

The force on atom k is

$$\mathbf{f}_k = -\frac{\partial E}{\partial \mathbf{r}_k} = -\frac{1}{2} \frac{\partial \sum_i \sum_{j \neq i}^j V_{ij}(r_{ij})}{\partial \mathbf{r}_k} - \frac{\partial \sum_i F(\bar{\rho}_i)}{\partial \mathbf{r}_k}$$

The first term in the force formulation is non-zero only when k is either i or j , thus it becomes

$$-\frac{1}{2} \left[\frac{\partial \sum_{j \neq k}^j V_{kj}(r_{kj})}{\partial \mathbf{r}_k} + \frac{\partial \sum_{k \neq i}^i V_{ik}(r_{ik})}{\partial \mathbf{r}_k} \right] = -\frac{1}{2} \left[\frac{\partial \sum_{j \neq k}^j V_{kj}(r_{kj})}{\partial r_{kj}} \frac{\partial r_{kj}}{\partial \mathbf{r}_k} - \frac{\partial \sum_{k \neq i}^i V_{ik}(r_{ik})}{\partial r_{ik}} \frac{\partial r_{ik}}{\partial \mathbf{r}_k} \right]$$

With the help of the identity, the term becomes

$$\frac{1}{2} \left[\frac{\partial \sum_{j \neq k}^j V_{kj}(r_{kj})}{\partial r_{kj}} \frac{\mathbf{r}_{kj}}{r_{kj}} - \frac{\partial \sum_{k \neq i}^i V_{ik}(r_{ik})}{\partial r_{ik}} \frac{\mathbf{r}_{ik}}{r_{ik}} \right]$$

where V_{kj} and V_{ik} are the pair potentials for the atomic pairs kj and ik , respectively, while $V_{kj} = V_{jk}$ and $V_{ik} = V_{ki}$. Since V is atom type-specific, V_{kj} and V_{ik} are likely not the same unless atom i and j are of the same type. Thus, if there are two types of atoms in the system, there will be three V , between type 1 and type 1, between type 2 and type 2, and between type 1 and type 2.

The second term in the force formulation can be written as

$$-\sum_i \frac{\partial F(\bar{\rho}_i)}{\partial \mathbf{r}_k} = -\sum_i \frac{\partial F(\bar{\rho}_i)}{\partial \bar{\rho}_i} \frac{\partial \bar{\rho}_i}{\partial \mathbf{r}_k} = -\sum_i \frac{\partial F(\bar{\rho}_i)}{\partial \bar{\rho}_i} \sum_{j \neq i}^j \frac{\partial \rho_{ij}(r_{ij})}{\partial \mathbf{r}_k} = -\sum_i \frac{\partial F(\bar{\rho}_i)}{\partial \bar{\rho}_i} \sum_{j \neq i}^j \frac{\partial \rho_{ij}(r_{ij})}{\partial r_{ij}} \frac{\partial r_{ij}}{\partial \mathbf{r}_k}$$

which is non-zero when k is either i or j , i.e., the term becomes

$$-\frac{\partial F(\bar{\rho}_k)}{\partial \bar{\rho}_k} \sum_{j \neq k}^j \frac{\partial \rho_{kj}(r_{kj})}{\partial r_{kj}} \frac{\partial r_{kj}}{\partial \mathbf{r}_k} - \sum_{i \neq k}^i \frac{\partial F(\bar{\rho}_i)}{\partial \bar{\rho}_i} \frac{\partial \rho_{ik}(r_{ik})}{\partial r_{ik}} \frac{\partial r_{ik}}{\partial \mathbf{r}_k}$$

Again, with the help of the identify, the term becomes

$$\frac{\partial F(\bar{\rho}_k)}{\partial \rho_k} \sum_{j \neq k}^j \frac{\partial \rho_{kj}(r_{kj})}{\partial r_{kj}} \frac{\mathbf{r}_{kj}}{r_{kj}} - \sum_{i \neq k}^i \frac{\partial F(\bar{\rho}_i)}{\partial \bar{\rho}_i} \frac{\partial \rho_{ik}(r_{ik})}{\partial r_{ik}} \frac{\mathbf{r}_{ik}}{r_{ik}}$$

Note that ρ_{kj} is the local electron density contributed by atom j at site k . In general,

$\rho_{kj} \neq \rho_{jk}$. This is different from the pair potential V , for which generally $V_{kj} = V_{jk}$. Also, generally $\rho_{kj} \neq \rho_{ij}$ unless atom k and atom i are of the same type.

In [classical EAM](#), $\rho_{kj} = \rho_{ij}$ even when atom k and atom i are of different type. If there are two types of atoms in the system, there are only two ρ , for the contribution from type 1 atom and for that from type 2 atom, regardless of which type of atomic site it contributes to. This is different from the pair potential V , which would have three expressions in this case. Extensions of ρ to distinguish contributions at different types of atomic sites have been proposed, e.g., in the [Finnis-Sinclair potential](#).

Adding the two terms in the force formulation together yields

$$\partial \sum_i V(r_i) \mathbf{r}_i + \partial \sum_i V(r_i) \mathbf{r}_i + \partial F(\bar{\rho}_k) \frac{\partial \rho_k}{\partial \mathbf{r}_k} + \partial \rho_k(r_k) \mathbf{r}_k + \partial F(\bar{\rho}_k) \frac{\partial \rho_k}{\partial \mathbf{r}_k}$$

$$\mathbf{f}_k = \frac{1}{2} \left[\frac{\partial \sum_{j \neq k}^j V_{kj}(r_{kj})}{\partial r_{kj}} \frac{\mathbf{r}_{kj}}{r_{kj}} - \frac{\partial \sum_{k \neq i}^i V_{ik}(r_{ik})}{\partial r_{ik}} \frac{\mathbf{r}_{ik}}{r_{ik}} \right] + \frac{\partial F(\bar{\rho}_k)}{\partial \bar{\rho}_k} \sum_{j \neq k}^j \frac{\partial \rho_{kj}(r_{kj})}{\partial r_{kj}} \frac{\mathbf{r}_{kj}}{r_{kj}} - \sum_{i \neq k}^i \frac{\partial F(\bar{\rho}_i)}{\partial \bar{\rho}_i} \frac{\partial \rho_{ik}(r_{ik})}{\partial r_{ik}}$$

Since i and j are just dummy indices, it is safe to replace all i with j . After that, with

$\mathbf{r}_{jk} = -\mathbf{r}_{kj}$, $r_{jk} = r_{kj}$, $V_{jk} = V_{kj}$, and $\rho_{jk} \neq \rho_{kj}$, the force on atom k becomes

$$\mathbf{f}_k = \sum_{j \neq k}^j \left[\frac{\partial V_{kj}(r_{kj})}{\partial r_{kj}} + \frac{\partial F(\bar{\rho}_k)}{\partial \bar{\rho}_k} \frac{\partial \rho_{kj}(r_{kj})}{\partial r_{kj}} + \frac{\partial F(\bar{\rho}_j)}{\partial \bar{\rho}_j} \frac{\partial \rho_{jk}(r_{kj})}{\partial r_{kj}} \right] \frac{\mathbf{r}_{kj}}{r_{kj}}$$

If there is only type of atoms in the system, $\rho_{jk} = \rho_{kj}$, and the force formulation is simplified to

$$\mathbf{f}_k = \sum_{j \neq k}^j \left[\frac{\partial V_{kj}(r_{kj})}{\partial r_{kj}} + \left(\frac{\partial F(\bar{\rho}_k)}{\partial \bar{\rho}_k} + \frac{\partial F(\bar{\rho}_j)}{\partial \bar{\rho}_j} \right) \frac{\partial \rho_{kj}(r_{kj})}{\partial r_{kj}} \right] \frac{\mathbf{r}_{kj}}{r_{kj}}$$

which is Equation 15 of [Xu et al.](#) Note that the last two equations hold for both [classical EAM](#) and [Finnis-Sinclair](#) potentials, because the relation between ρ_{kj} and ρ_{ij} is not used during the derivation.

Code developer guide

This chapter describes some algorithms and data structures of the PyCAC code. More information may be added in response to requests from developers and users.

atom and atomap

atom

There are some arrays with `atom` in their names, e.g., `r_atom`, `vel_atom`, `force_atom`, which respectively are the positions, velocities, and forces of real atoms in the atomistic domain. On the other hand, defined in `atom_para_module.f90`, there are `atom_num_1`, which is the number of local atoms in each processor domain, `atom_num_lg`, which is the number of local AND ghost atoms in each processor domain, and `atom_num_lr`, which is the actual size of the second dimension of some `atom`-related arrays. The differences between `atom_num_1`, `atom_num_lg`, and `atom_num_lr`, as well as their relation with some `atom`-related arrays are explained below.

Say that the system contains 100 atoms using three processors, then either `read_restart.f90` (if one restarts a previous simulation) or `model_assemble.f90` (if one builds the model from scratch), set by `boolean_restart`, will first calculate `atom_num_1` as

```
atom_num_1 = nint((real(atom_num, wp) / pro_num) * 1.2_wp)
```

In our case, `atom_num_1` = 40, following which most `atom`-related arrays are allocated.

Next, in `processor_scatter_atomistic.f90`, one first lets `atom_num_lr` equal `atom_num_1`, then the root processor distributes all atoms to all processors (including root itself); if the actual number of local atoms one processor should have is larger than `atom_num_lr`, `atom_num_lr` is increased by `seg_num` which is `1024` as set in `module/cac_para_module.f90`. In the meantime, many `atom`-related arrays also increase their size. Assume that the root processor should contain 50 local atoms while the other two processors 25 each, `atom_num_lr` becomes $40 + 1024 = 1064$ for root but still 40 for the other two processors. At the end of this subroutine, `atom`-related arrays, e.g., `r_atom` has a size of 3 by 1064 for root while 3 by 40 for the other two processors; `atom_num_1`, which is expected to be the number of local atoms for each processor, is re-calculated to be 50 for root and 25 for the other two processors.

Next, in `processor_ghost_atomistic.f90`, the ghost atoms are added to the end of the `atom`-related arrays. Again, if the total number of atoms (local+ghost) is larger than `atom_num_lr`, `atom_num_lr` is further increased by `seg_num`, along with the expansion in size of some `atom`-related arrays. Assume that the numbers of ghost atoms are 30, 10, and 20 for the three processors, `atom_num_lr` would be 1064 for root (because $50 + 30 < 1064$), 40 for processor 1 (because $25 + 10 < 40$), and 1064 for processor 2 (because $25 + 20 > 40$). In other words, `atom_num_lr` increases for the last processor while remaining the same for the other two processors. At the end of this subroutine, `atom_num_lg` is assigned as the actual number of local + ghost atoms, i.e., 80, 35, and 45 for the three processors, respectively.

In sum, `atom_num_l1` \leq `atom_num_lg` \leq `atom_num_lr`. Also note that `atom_num` is the number of real atoms in the system, i.e., 100, regardless of how many processors are involved and how many ghost atoms are needed.

atomap

There are also some arrays with `atomap` in their names, e.g., `r_atomap`, which is the positions of interpolated atoms in the coarse-grained domain. On the other hand, defined in `interpo_para_module.f90`, there are `atomap_num_l1`, which is the number of interpolated atoms in each processor domain, `atomap_num_lg`, which is the number of local AND ghost interpolated atoms in each processor domain, and `atomap_num_lr`, which is the actual size of the second dimension of some `atomap`-related arrays.

The differences between `atomap_num_l1`, `atomap_num_lg`, and `atomap_num_lr`, as well as their relation with some `atomap`-related arrays are similar to those of `atom`-related variables and arrays, except that two other subroutines, `processor_scatter_cg.f90` and `processor_ghost_cg.f90`, are involved.

Remark

In PyCAC, arrays for the atomistic domain, e.g., `atom`-related arrays, and those for the coarse-grained domain, e.g., `atomap`-related, `node`-related, and `e1e`-related arrays, are completely separated. Take the position vector as an example, each processor may have simultaneously have a `r_atom` array and a `r_atomap` array, yet if it only contains real/interpolated atoms, it only has a `r_atom / r_atomap` array. If one wants to add an additional array to the atomistic domain, e.g., to distinguish different types of real atoms,

one almost always has to also add a similar array to the coarse-grained domain to represent different types to distinguish different types of interpolated atoms, nodes, and elements.