

Appendix A

MD++ User's Guide

(10 pages)

MD++ is an atomistic simulation package on UNIX platforms, written in C++ (plus some Fortran). The intention to develop MD++ is to make atomistic simulations easy to set up, visualize and control. In particular, MD++ takes input from a free-format script file, and is capable of displaying real-time 3D atomic structures during the simulation. It hence provides an environment for interactive simulations, or a “virtual lab” to manipulate atomic structures. We think MD++ can be a useful tool accompanying this book for learning atomistic simulations of dislocations in solids. Specifically, usage or modification of source codes of MD++ is required to solve many problems in Chapter 2 and 3. We also want to point out that, MD++ is not a “toy” model just for learning purposes. It is actually widely used by the authors and their colleagues in their research projects.

Most of the codes for MD++ were developed during the graduate studies of one of us in Nuclear Engineering Department, MIT. It was subsequently used by many of our colleagues at Lawrence Livermore National Lab. The code hence benefited a lot from their suggestions.

A.1 Installation

The MD++ package, e.g. `md++-03-27-2002.tar.gz` can be downloaded freely from the web at <http://www.oup.edu>. We recommend you to unpack it in the `Codes` directory under your home directory, because some example scripts will be referring to this path. For example, you can use

```
cd
mkdir Codes
cd Codes
gzip -cd ~/md++-03-27-2002.tar.gz | tar xvf -
```

To compile, type

```
cd MD++
gmake
```

By default, this will use the GNU C++ compiler `gcc` and build executables such as `fs_gcc`, `eam_gcc`, corresponding to different inter-atomic potentials (see Chapter 2). We find that using `gcc`, MD++ can be compiled on a wide range of computer platforms, including Linux PC, DEC/Alpha, SGI, IBM/SP, etc. Therefore using `gcc` is probably the easiest way to build MD++. On the other hand, you will need to modify the `Makefile` if you cannot get `gcc` installed on your machine. MD++ will also use `f77` to compile the Fortran codes in subdirectories such as `F90FS` etc. If your Fortran compiler has a different name, e.g. `fort`, you can use it by typing `gmake F77=fort`. More installation details can be found in the `README` file in the MD++ folder.

A.2 Running Scripts

If you are successful so far, you are ready to make MD++ run some examples, such as,

```
./sw_gcc Examples/example00-siview.script
```

In this example, MD++ simply creates a perfect lattice of 512 silicon atoms and plots them, see Fig. A.1. We list the content of this script file and provide a brief explanation below. A detailed account of control variables and commands in the script files is given in the following section.

```
# example00-siview.script
# MD code of Stillinger-Weber Si
setnolog
setoverwrite
dirname = runs/tmp
```

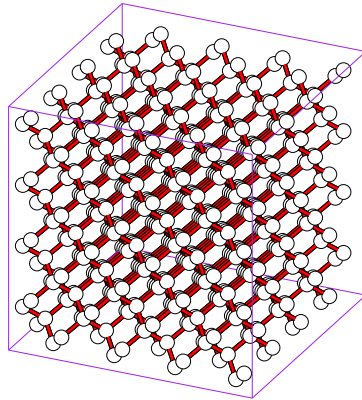


Figure A.1: Atomic configuration created by running `./sw_gcc Examples/example00-siview.script`.

```
#-----
#Create Perfect Lattice Configuration
latticestructure = diamond-cubic
latticeconst = 5.4309529817532409 #(A) for Si
makecnspec = [ 1  0  0  4  #(x)
               0  1  0  4  #(y)
               0  0  1  4  ] #(z)
makecn finalcnfile = perf.cn writecn
#-----
#Plot Configuration
atomradius = 0.6 bondradius = 0.3 bondlength = 2.8285 #(A)
atomcolor = orange bondcolor = red backgroundcolor = gray
openwin allocolors plot
sleep quit
```

In a script file, anything between a `#` sign and the end of the line is a comment and has no effect. A script file contains two types of sentences, separated by white spaces. They are variable assignments, in the form of *variable = value*, such as `latticeconst = 5.43095`, and commands in a single word, such as `makecn`. MD++ executes the variable assignments and commands in the same order as they are written.

Every script file *must* assign a name of directory to **dirname**. MD++ will change its directory at the beginning of the run into the specified directory, in which all output files will be stored. The only sentences that can precede **dirname** are **setnolog** and **setoverwrite**. By default, MD++ redirect all of its output into a log file **A.log** in the specified subdirectory. If **setnolog** appears before **dirname**, no log file will be created, and the content will be printed to the terminal. By default, MD++ will abort if the specified directory already exists, to avoid overwritten existing data files. If **setoverwrite** is given, MD++ will ignore this situation and will overwrite whatever file in that directory.

In the above example, a perfect lattice of silicon atoms is created by command **makecn**. Three variable assignments before **makecn** specifies how the lattice should be created. Variable **latticestructure** can be **simple-cubic**, **body-centered-cubic**, **face-centered-cubic**, or **diamond-cubic**, and **latticeconst** specifies lattice constant in Angstroms. **makecnspec** specifies the geometry of the simulation cell by an array of length 12. In this case, the three cell vectors are set to 4[100], 4[001], 4[001], respectively. After the configuration is created, it is saved by **writecn** into a file with name specified by **finalcnfile**.

The script file proceeds to open an window and plot the configuration. Variables **atomradius** and **bondradius** specifies the size in which atoms and bonds appear on the screen, in arbitrary units. A bond is drawn between any pair of atoms whose separation is less than **bondlength**. By default **bondlength** = 0 so that no bond will be drawn. After **plot**, MD++ is put to **sleep** before it terminates, so that the display window stays long enough to be examined. The 3D atomic structure can be rotated by clicking and dragging the mouse in the display window.

The following command will give a more complex example.

```
./fs_gcc Examples/example01-mo.script
```

Here a dislocation dipole is created in a periodic super-cell of Molybdenum atoms. Conjugate gradient relaxation is then performed using the Finnis-Sinclair potential. High energy atoms in dislocation cores are highlighted with a different color. The change of the shape of dislocation cores during the relaxation is easily noticeable.

A.3 List of Commands and Variables

A.3.1 Log File and Control

syntax	function
setnolog	outputs will be printed to the terminal default log file A.log will not be created
setoverwrite	ignore the potential problem of overwriting existing files if the directory specified by dirname already exists
dirname = <i>string</i>	directory name in which all output files are stored
sleepseconds = <i>int</i> sleep	put MD++ to sleep for <i>int</i> seconds
command = <i>string</i> runcommand	execute shell command specified in <i>string</i>

A.3.2 Atomic Structure Manipulation

syntax	function
latticestructure = <i>string</i>	lattice type structure, <i>string</i> can be simple-cubic , body-centered-cubic , face-centered-cubic , diamond-cubic
latticeconst = <i>double</i>	lattice constant in Angstrom
makecnspec = [<i>a_x a_y a_z n_a</i> <i>b_x b_y b_z n_b</i> <i>c_x c_y c_z n_c</i>]	simulation cell geometry $\vec{a} = n_a[a_x, a_y, a_z]$ $\vec{b} = n_b[b_x, b_y, b_z]$ $\vec{c} = n_c[c_x, c_y, c_z]$
makecn	create perfect lattice as specified above
splicecnspec = [<i>int double</i>] incnfile = <i>string</i> splicecn	read structure from file <i>string</i> and put it beside the current one along <i>x</i> , <i>y</i> or <i>z</i> directions if <i>int</i> = 1, 2, or 3, with additional displacement <i>double</i>
mkdipole = [···] makedipole	create dislocation dipole in the simulation cell (see Chap. 2)
pickfixedatomspec = [<i>n i₁ i₂ ··· i_n</i>] pickfixedatoms	set <i>n</i> atoms fixed whose indices are <i>i₁, i₂, ···, i_n</i>

syntax	function (continued)
extendboxspec = $[dir\ num]$ extendbox	extend the current simulation box along x , y or z direction if $dir=1, 2$ or 3 by repeating it num times
cutslice	reverse of extendbox
shiftspec = $[i\ j\ p]$ shiftbox redefinepbc	changing column vectors of box matrix $H = [H_1 H_2 H_3]$ (see Chap. 2) by $H_i = H_i + H_j * p$ with relative coordinates s of atoms fixed $H_i = H_i + H_j * p$ with real coordinates r of atoms fixed
shiftspec = $[x\ y\ z]$ shiftpbc	shift relative coordinate s of every atom by (x, y, z)
hrescale = r scaleH restoreH	scale every component of matrix H by r the original matrix is stored in H_0 $H = H_0$

A.3.3 File Input Output

syntax	function
incnfile = <i>string</i> readcn readPOSCAR	read configuration from the file named <i>string</i> read in VASP's POSCAR format atomic configuration from the file <i>string</i>
writeall = <i>int</i>	if <i>int</i> $\neq 0$, writencn will write <i>all</i> information about every atom, including position, velocity, energy, atom type. By default, only positional information is written.
finalcnfile = <i>string</i> writencn writePOSCAR	write configuration into a file named <i>string</i> write atomic configuration in VASP's POSCAR format into a file named <i>string</i>
intercnfile = <i>string</i> .cn openintercnfile	must be called before writeintercn
writeintercn	write current configuration into file <i>stringxxxx.cn</i> with <i>xxxx</i> incremented by one every time this function is called

syntax	function (continued)
<code>savecn = int</code>	if <i>int</i> = 1 then <code>writeintercn</code> will be called
<code>savecnfreq = n</code>	every <i>n</i> iterations in <code>relax</code> or <code>run</code>
<code>potfile = string</code> <code>readpot</code>	read potential parameters from file

A.3.4 X-window Display

syntax	function
<code>atomcolor = string</code>	color of atoms
<code>fixatomcolor = string</code>	color of fixed atoms
<code>highlightcolor = string</code>	color of highlighted atoms
<code>bondcolor = string</code>	color of bonds
<code>backgroundcolor = string</code>	color of X window background
<code>atomradius = double</code>	size of atoms
<code>bondradius = double</code>	thickness of bonds
<code>hideatomenergy = [int E_{min} E_{max}]</code>	if <i>int</i> =1, atoms with energy outside of range $[E_{min}E_{max}]$ will not be shown. By default all atoms will be shown.
<code>highlightenergy = [int E]</code>	if <i>int</i> =1, atoms with energy higher than E will be shown with <code>highlightcolor</code> .
<code>plot_highlight =</code> <code>[n i_1 i_2 \cdots i_n]</code>	if $n \geq 1$, then atoms with indices i_1, i_2, \cdots, i_n will be plotted with <code>highlightcolor</code> .
<code>plot_restrict = int</code>	if <i>int</i> = 1, then only atoms with reduced
<code>plot_xmin = x_{min} plot_xmax = x_{max}</code>	coordinate satisfying $x_{min} < x < x_{max}$,
<code>plot_ymin = y_{min} plot_ymax = y_{max}</code>	$y_{min} < y < y_{max}$, $z_{min} < z < z_{max}$ will be plotted.
<code>plot_zmin = z_{min} plot_zmax = z_{max}</code>	By default all atoms will be plotted.
<code>rotateangles = [α β γ]</code>	the default viewing direction will be rotated by the three Euler angles
<code>win_height = int win_width = int</code>	dimension of X window
<code>eval</code>	compute energies of atoms from potential so that they can be shown in different colors
<code>plot</code>	plot the atomic structure in an X window with the above specifications
<code>plotfreq = int</code>	<code>plot</code> will be called every <i>int</i> steps in <code>relax</code> or <code>run</code>

A.3.5 Conjugate Gradient Relaxation

syntax	function
<code>conj_dfpred = double</code>	estimate of energy drop in the first conjugate-gradient (CG) step
<code>conj_fevalmax = int</code> <code>conj_fixbox = int</code>	maximum energy evaluation in CG relaxation if <i>int</i> = 1 (default) box is fixed, otherwise box is allowed to relax
<code>conj_ftol = double</code>	residual gradient tolerance
<code>stress =</code> [σ_{xx} σ_{xy} σ_{xz} σ_{xy} σ_{yy} σ_{yz} σ_{xz} σ_{yz} σ_{zz}]	stress (MPa) in Parinello-Rahman boundary condition, only effective if <code>conj_fixbox</code> = 0
<code>relax</code>	CG relaxation
<code>shmsize = int</code> <code>shmallocate</code>	allocate <i>int</i> byte of shared memory for parallel computation (only if <code>-DPARALLEL</code> is in Makefile)
<code>ncpu = int spawn</code> <code>collect</code>	make <i>int</i> number of processes for parallel computing (call before <code>relax</code> or <code>run</code>) return to single process (call after <code>relax</code> or <code>run</code>)
<code>fortranpath = string</code> <code>fortranexe = string</code> <code>potfile = string</code> <code>fortranrelax</code>	directory holding Fortran codes name of the Fortran executable name of Fortran code potential file call Fortran program to do CG relax

A.3.6 Molecular Dynamics

syntax	function
<code>atommass = double</code>	mass of atoms (kg/mol)
<code>T_OBJ = double</code>	desired temperature in MD simulation
<code>scalevelocity = int</code>	if <i>int</i> = 1, atom velocities will be scaled to control temperature
<code>atomTcpl = double</code>	velocity scaling coefficient for atoms
<code>boxTcpl = double</code>	velocity scaling coefficient for box
<code>usenosehoover = int</code>	if <i>int</i> = 1, use Nose-Hoover thermostat to control temperature
<code>vt2 = double</code>	coefficient of Nose-Hoover thermostat

syntax	function (continued)
<code>equilsteps = int</code>	number of equilibration steps
<code>totalsteps = int</code>	total number of MD steps
<code>timestep = double</code>	MD time step in picoseconds
<code>initvelocity</code>	initialize atom velocity according to T.OBJ
<code>run</code>	MD simulation

A.4 Code Structure

The structure of the MD++ source code is illustrated in Fig. A.4. The boxes represent C++ classes and the arrows indicate their mutual dependence. The basic object `MDFrame` inherits `SCParser` for parsing script files, and contains a `YWindow` object as a member for plotting atomic configurations. Most control variables and commands are implemented in `MDFrame`, but it lacks a real inter-atomic potential. `SWFrame`, `FSFrame`, `EAMFrame` all inherit `MDFrame` and implement their own potentials, i.e. Stillinger-Weber (SW), Finnis-Sinclair (FS), and EAM respectively. A good way to add new functionalities or to try out new ideas on MD++ is to declare a new class, inheriting `MDFrame` or its child classes. In this way all existing variables and commands will be available and the new code will not disturb existing codes.

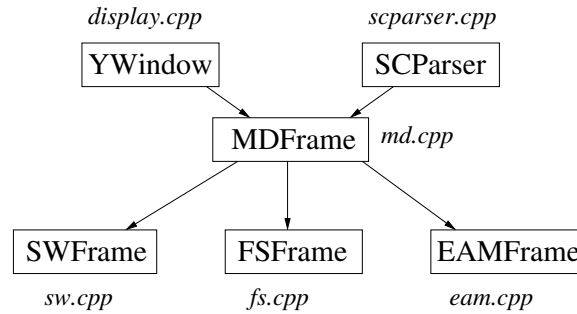


Figure A.2: MD++ code structure.

A.4.1 Variables

The internal units of MD++ is electron volts (eV) and Angstrom (\AA) for energy and length respectively. Thus the internal stress unit is $\text{eV}/\text{\AA}^3 \approx$

160GPa, but it is converted into MPa in both inputs and outputs.

The most relevant internal variables for an atomistic simulation are defined within the `SimFrame` class in `md.h`, and are listed below.

name	type	meaning
<code>_NP</code>	int	number of atoms
<code>_R</code>	Vector3 *	real coordinates (x, y, z) of atoms
<code>_R0</code>	Vector3 *	auxiliary space for storing old values of <code>_R</code>
<code>_H</code>	Matrix33	3×3 matrix whose columns are box vectors of simulation cell
<code>_SR</code>	Vector3 *	scaled coordinates of atoms $_R[i] = _H * _SR[i]$
<code>_VR</code>	Vector3 *	real velocity $_VR = d_R/dt$
<code>_VSR</code>	Vector3 *	scaled velocity $_VSR = d_SR/dt$
fixed	int *	flag of whether (1) or not (0) atom is fixed
<code>_F</code>	Vector3 *	forces on atoms
<code>_EPOT</code>	double	total energy
<code>_VIRIAL</code>	Matrix33	Virial stress tensor
<code>_TOTALSTRESS</code>	Matrix33	total (Virial + kinetic) stress
<code>_EXTSTRESS</code>	Matrix33	external applied stress
<code>_EXTSTRESSMUL</code>	double	a pre-factor of applied stress
<code>_EPOT_IND</code>	double *	local energy of every atom
<code>_KATOM</code>	double	kinetic energy of every atom
<code>_ATOMMASS</code>	double	atom mass
<code>_WALLMASS</code>	double	fictitious mass of simulation box in Parinello-Rahman MD
<code>_TIMESTEP</code>	double	MD time step
<code>_TDES</code>	double	desired temperature (in K)
<code>_T</code>	double	current kinetic temperature (in K)
<code>_RLIST</code>	double	cut-off distance of neighbor list
<code>_SKIN</code>	double	skin depth of neighbor list
<code>nn</code>	int *	number of neighbors for every atom
<code>n</code>	int **	$n[i][j]$ is the index of j'th neighbor of atom i, $0 < j < nn[i]$

A.4.2 Key Functions

Class `MDFrame` inherits `SimFrame` and allows user to set control variables and call functions from script file. The “binding” between a string in the script

file and a MD++ control variable or function in the code is accomplished by functions `bindvar` and `bindcommand`, which are called by `initparser` and `exec` respectively.

For those who are interested in modifying the source codes, in the remaining of this section we briefly describe several important functions in class `MDFrame` for MD simulations and static relaxations.

<i>Function</i> <code>potential</code>
Declared but intentionally left empty in <code>MDFrame</code> . It is the responsibility of child classes, such as <code>FSFrame</code> (in <code>fs.cpp</code>) to implement a real potential. Function <code>potential</code> should compute the total energy (<code>_EPOT</code>), force (<code>_F</code>) and local energy (<code>_EPOT_IND</code>) of every atom, and Virial stress (<code>_VIRIAL</code>).
<i>Function</i> <code>calcprop</code>
Called every time after <code>potential</code> . Compute all other properties of interest based on quantities computed by <code>potential</code> .
<i>Class</i> <code>Gear6Int</code>
Gear's 6th order predictor-corrector method for integrating equations of motion in MD simulations.
<i>Function</i> <code>step</code>
One step of MD simulation, by calling <code>potential</code> and Gear's predictor-corrector once.
<i>Class</i> <code>Verlist</code>
Neighbor list constructor using Verlet and bin-list algorithm.
<i>Function</i> <code>CGRelax</code>
Conjugate gradient relaxation. It requires the potential wrapper function <code>potential_wrapper</code> as one of its arguments.