

# MD++ User's Manual

Wei Cai

Lawrence Livermore National Laboratory  
7000 East Avenue, Livermore, CA 94550,USA  
caiwei@llnl.gov, (925) 424-5443

October 30, 2001

## Contents

<b>1</b>	<b>Overview</b>	<b>2</b>
<b>2</b>	<b>How to Use: An Example</b>	<b>3</b>
2.1	To Build . . . . .	3
2.2	To Use . . . . .	3
2.3	A Closer Look at the Script File . . . . .	4
<b>3</b>	<b>Code Structure</b>	<b>6</b>
<b>4</b>	<b>Control Variables and Functions</b>	<b>8</b>
4.1	List of Control Variables and Functions . . . . .	8
4.2	Log File and Control . . . . .	9
4.3	Make Perfect Lattice . . . . .	9
4.4	File I/O . . . . .	9
4.5	X-window Display . . . . .	10
4.6	Structure Manipulation . . . . .	10
4.7	Conjugate Gradient Relaxation . . . . .	10
4.8	External Fortran Calls . . . . .	10
4.9	Molecular Dynamics Simulation . . . . .	10
4.10	Parallel Execution . . . . .	10
<b>5</b>	<b>More Examples</b>	<b>10</b>

# 1 Overview

MD++ is a Molecular Dynamics (MD) simulation package written in C++. I started developing it at around 1999 or so, when I was a graduate student at MIT, Department of Nuclear Engineering. The computational part of this package was based on Fortran 77 codes by my mentor and colleague Vasily Bulatov, then at MIT Department of Mechanical Engineering. The Fortran codes were later modified by my colleague Jinpeng (Elton) Chang, and are included in this package as well. Several fundamental C++ objects in MD++ were written by or adapted from the codes of my colleague Dongyi Liao.

My intention to develop MD++ was to make atomistic simulations easier to set up, visualize, and control, which means, more fun. The simulations I had in mind were largely for defect in solids. But there is no reason that we can't use it to study liquids or gas. MD++ uses empirical potential models and is best applicable to systems with a moderate number of atoms, i.e. from a few thousand to a few hundred thousand atoms. It was recently parallelized by using shared memory, but it is not intended to be a massively parallel code. MD++ is most useful for setting up moderate size simulations (a few thousand atoms), and running them interactively on a workstation. The idea is to gain as much intuition as possible by running these simulation interactively and watching the dances of the atoms and defects on the screen, before we go ahead to huge simulations which require super-computing.

I had my first enthusiastic client when I joined the Lawrence Livermore National Lab (LLNL) and worked with Vasily (again). Apparently Vasily was having much fun with MD++, creating dislocation junctions and driving them around, and probably missed a few meals when doing that. Vasily recommend it to my new colleagues so that by now I already have a handful of clients, which made me confident enough to start writing up this manual.

MD++ uses an old (but very robust) relaxation subroutine to do static calculations, and uses standard integration scheme and temperature-stress controls for dynamics simulations. There is not much to boast about in those regards. What's special about MD++ lies in its treatment of the simulations' input and output, i.e. the interface between a dumb computer and an impatient physicist.

- With MD++ we can open an *X-window* and view the dynamics of atoms interactively during a simulation.
- The input files – the *scripts* – for MD++ are in a free format. The script files not only set values for internal variables, they also tell MD++ what to do. In fact the only thing that MD++ executables do is to parse the given script file, which will then ask MD++ to create and manipulate atomistic configurations, relax or run Molecular Dynamics with them, or simply quit. What MD++ provides us is an environment for atomistic simulations, something like Matlab for general technical computing, although this analogy is rather disproportionate.

Since the source codes only tell MD++ *how* to do things, but not *what* to do, MD++ is an object that has a natural tendency to grow, and it does. Whenever a new capability is developed in MD++, such as creating a new defect structure, or using a new boundary condition, we can use it by calling it from the script file, and can turn it off by simply commenting it out. Old scripts which does not involve the new capability still runs the same as before. Therefore if you find MD++ is helpful to you, you can also contribute to its growth, by either suggesting some capabilities that you want MD++ to have, or help implement them.

There are 3 ways to use MD++.

- The first one is to *use it as it is*. You can compose your own scripts to manipulate atoms with the functions that MD++ provides. If you find that MD++ does save you time in helping you finding the answer you want, or simply gives you some more fun in atomistic simulations, and that the time you spend in porting and learning MD++ is worthwhile, please drop me a note — I would be happy to hear that.
- You can also *build on top of MD++*. If you find that MD++ provides a useful environment for you but you also need some more functions that it doesn't have, you can declare a new object that *inherits* the objects implemented in MD++, such as `MDFrame`. In this way, you inherit all the existing functions and you are free to add whatever new stuff you like without worrying about disturbing MD++ itself.

- You are also encouraged to take pieces of MD++, such as the *script parser* and the *X-window* display, and apply them to your own code developments if you see fit, provided that you give appropriate acknowledgments. These two functions are written as C++ objects and should be rather easy to port if you are using C++ or C.

This MD++ manual mainly describes how to *use it as it is*. I will briefly mention the structure of the code (in Section 3), just to help explain the meaning of the functions and control variables in the scripts (in Section 4).

## 2 How to Use: An Example

### 2.1 To Build

Let's get started. Supposed you obtain an MD++ package, say `md++-10-19-2001.tar.gz`. Let's take a quick look at how to use it, through a short example.

First, let's unpack it in the `~/Codes` directory. If you don't have `Codes` directory under your home directory, you can create one. This is preferable because some example script files assumes it is in the `~/Codes/MD++` directory. So, let's say that you unpack it by typing the following.

```
cd
mkdir Codes
cd Codes
gzip -cd ~/md++-10-19-2001.tar.gz | tar xvf -
```

This will create a sub-directory MD++ in your `~/Codes` directory. Within MD++, there should be the source files such as `md.h`, `md.cpp` etc, and some sub-directories. In this section, we will run MD++ with some example script files from the `Examples` sub-directory. Now we need to build the executables from the source. Simply type,

```
cd MD++
gmake
```

This will use `gcc` to compile the C++ codes and `f77` to compile the companion Fortran codes in sub-directories such as `F90FS`, `F90SW`, etc. The compilation process can be horrifying if you hate to see tons of warning messages — but you can simply ignore them. If by the end of this we get new files such as `moscrew_gcc`, `siscrew_gcc`, `eam_gcc`, then we have been successful so far. `Gcc` is a very generic compiler and we have successfully built MD++ on several machine architectures such as Linux PC, SGI and Dec/Alpha. Please let me know if you have problem building MD++ on your machine.

We should also get executables such as `Relax` in Fortran sub-directories such as `F90FS`. If your Fortran compiler is not `f77`, such as in Linux alpha which uses `fort`, you can simply go into the Fortran sub-directories and modify the `Makefile` within them. These executables can be called within MD++ to provide a faster energy minimization than the build-in relaxation functions — Fortran codes are generally faster than C++ codes — but they are not essential.

### 2.2 To Use

Let's try to run MD++ from an example script file. Type

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

This will run the given script file using the MD++ executable compiled with Finnis-Sinclair potential for BCC Mo.

[ Note: MD++ does not run properly on **zamora** (one of our Linux Alpha machines), due to its inadequate compiler configurations. We can still run this example on **zamora** though, by calling the Fortran code to do relaxation (see below), instead of using the native relaxation subroutines of MD++. However, for the

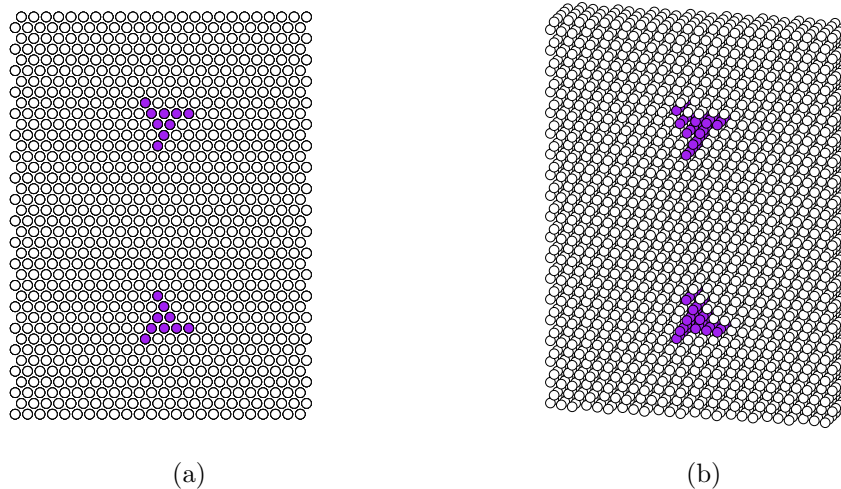


Figure 1: Snapshots from two viewing angles of the relaxed configuration after running `Examples/example01-mo.script`.

Fortran code to behave properly, we need to use `fort` instead of `f77` to compile. This can be done by changing the line `F77 = f77` into `F77=fort` in `F90FS/Makefile`, `F90SW/Makefile` and `F90EAM/Makefile`. ]

After you initiate the command above, a window will pop up, showing a BCC lattice of Mo atoms, containing a screw dislocation dipole. The high energy atoms are highlighted. MD++ then starts to relax this configuration, with intermediate energy information dumped on your terminal. As the relaxation proceeds, you will notice the pattern of high energy atoms also changes. At any point of the simulation, you can use your mouse to rotate the atomistic configuration in the X-window using your mouse (see Section 4.5). When the relaxation completes, the high energy atoms around the two screw dislocation cores exhibit a 3-fold dissociation pattern. The total energy of this configuration can be found on your terminal,

```
EPOT=-3.10591268138509e+04
```

in unit of electron-volts (eV). Relevant data files for this simulation is written into a sub-directory `runs/tmp`, as specified at the beginning of the script file. Fig. 2.2 shows two snapshots of the relaxed configuration from two viewing angles. (To take gif and postscript snapshots from X-window display, see Section 4.5.)

## 2.3 A Closer Look at the Script File

The script file that we just run is reproduced below. In this section we briefly discuss its meaning and how it does the job that we just see. Note that in the script everything between `#` sign and the end of the line is comment and has no effect at all.

`Examples/example01-mo.script` :

```
setnolog
setoverwrite
dirname = runs/tmp
shmsize = 31457280 shmallocate #10MB
#-----
#Read in potential file
#
potfile = ~/Codes/MD++/mo_pot readpot
#-----
#Create Perfect Lattice Configuration
```

```

#
latticestructure = body-centered-cubic latticeconst = 3.1472 #(A) for Mo
makecnspec = [ 1 1 -2 8 #for screw [111]
               1 -1 0 19
               0.5 0.5 0.5 5 ]
makecn finalcnfile = perf.cn writecn
#-----
#Create Dislocation Dipole
#(screw) [111]
mkdipole = [ 3 2 #z(dislocation line), y(dipole direction)
             0 0 0.2 #(burgers vector relative to box)
             0 -0.2632 0.2368 #Y=18 d=0.5 (b)
             0.305 #\nu
             -10 10 -10 10 1 #number of images
             0 0 0 0 ]
makedipole finalcnfile = makedp.cn writecn
#-----
#Plot Configuration
atomradius = 1.0 bondradius = 0.3 bondlength = 0
atomcolor = cyan highlightcolor = purple backgroundcolor = gray
bondcolor = red fixatomcolor = yellow
#hideatomenergy = [ 1 -6.725 -6 ]
highlightenergy = [ 1 -6.72 ]
#energycolorbar = [ 1 -6.8 -6.55 ] highlightcolor = red
plot_select = 5 plot_highlight = [ 0 0 1 2 3 4 5 6 7 8 9 ]
plotfreq = 10
rotateangles = [ 0 0 0 1.5 ]
win_width = 600 win_height = 600
openwin alloccolors rotate saverot refreshnlist eval plot
#sleep quit
#-----
#Conjugate-Gradient relaxation
conj_ftol = 1e-7 conj_fevalmax = 1000
conj_fixbox = 1
relax finalcnfile = relaxed.cn writecn
sleep quit
#-----
#Fortran Conjugate-Gradient relaxation
#
fortranpath = ~/Codes/MD++/F90FS potfile = Mo.pot fortranexe = Relax
conj_fixbox = 1 #stress is meaningful only if allow box to relax
#stress = [ 0 100 0
#          100 0 0
#          0 0 0 ] #(stress in MPa)
fortranrelax eval plot
sleep quit

```

The first three lines in the script file are mandatory, they describe to MD++ whether and where to put the log files that will be generated by the simulation (see Section 4.2 for more details). The first two lines can be commented out (by putting a # at the line beginning), but the line `dirname = ...` cannot. It specifies the name of the sub-directory where MD++ will store its log files for this simulation. Line `shmsize = ...` has no effect if the code is compiled in serial mode (`GEN+= -DMPLIB_SINGLE` is defined in `Makefile`). This line initiates the shared memory block if the code is compile to run in parallel.

As MD++ continues to parse through the script, it sets the variable `potfile` (a character string) to the

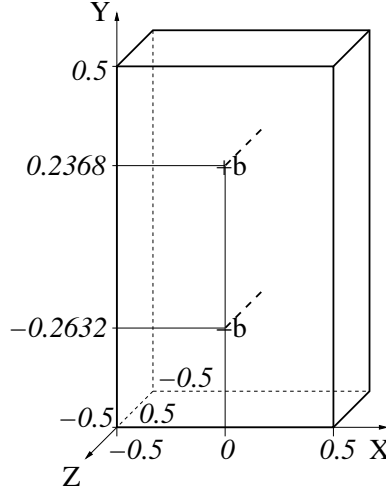


Figure 2: The geometry of the dipole created by `Examples/example01-mo.script`. The coordinates within the simulation cell goes from  $-0.5$  to  $+0.5$ .

supplied file name. Command `readpot` then asks MD++ to read the inter-atomic potential from this file.

The next command that MD++ executes is `makecn`, which will create a perfect lattice, under the specification that is given by the variables before it. Here, we are making a BCC lattice with a lattice constant  $3.1472\text{\AA}$ . The three dimensions of the simulation cell are  $X = [11\bar{2}] \times 8$ ,  $Y = [1\bar{1}0] \times 19$ ,  $Z = \frac{1}{2}[111] \times 5$ , respectively. This configuration is saved by calling `writencn`, into a file named `perf.cn` (in sub-directory `runs/tmp`), as specified by the variable `finalcnfile`.

The script then proceeds to create a screw dislocation dipole in the cell, by calling `makedipole`. Here I want to defer the detailed explanation of the parameter `mkdipole` to Section ??, and simply point out that with the current setting, the two dislocations will be along  $z$  direction, with Burgers vector  $\pm\frac{1}{2}[111]$ , and located at (in reduced coordinates)  $(0, -0.2632)$  and  $(0, 0.2368)$  respectively, as illustrated in Fig. 2.3. The configuration file which contains the newly created dislocation dipole is then saved into file `makedp.cn`.

The line `openwin alloccolors ... plot` opens an X-window and plot the current configurations. The lines above it specifies the style of plotting. More details will be given in Section 4.5. Here I just want to point out that, with line `highlightenergy = [ 1 -6.72 ]`, atoms with local energy above  $-6.72\text{eV}$  will be highlighted. As a reference, the energy per atom in the perfect BCC lattice for this potential is  $-6.82\text{eV}$ .

Line `relax ... writencn` then performs energy minimization on the structure it just created using a conjugate gradient algorithm. The relaxation will end when the residual gradient becomes smaller than  $1e-7$  (`conj_ftol`) or the maximum number of force evaluations becomes larger than 1000 (`conj_fevalmax`). The simulation box is not allowed to respond (`conj_fixbox = 1`) during the relaxation. The plot of atomistic configuration will be refreshed every 10 relaxation steps, as specified by `plotfreq = 10`.

After relaxation, the configuration is saved in `relaxed.cn` and the code sleeps (for 100 seconds) before it quits, so that you can look at the relaxed structure. Press `ctrl-c` in the terminal if you want to quit earlier.

We can modify this script to let it do other things. For example, we can comment out the two lines `relax ... writencn` and `sleep quit`. Then the code will proceed to line `fortranrelax ...`, which calls the Fortran code to perform the relaxation. This will be faster than the C++ relaxation subroutine, but you won't be able to see the change of atomic configurations during the relaxation. The atomic configuration will be loaded when the Fortran relaxation completes, and `eval plot` will then update the local atomic energy and refresh the X-window display.

### 3 Code Structure

MD++ code structure is illustrated in Fig. 3. The strings in the boxes represent C++ objects (classes) and the arrows indicate their inter-dependence. The basic object `MDFrame` inherits `SCPParser` for parsing

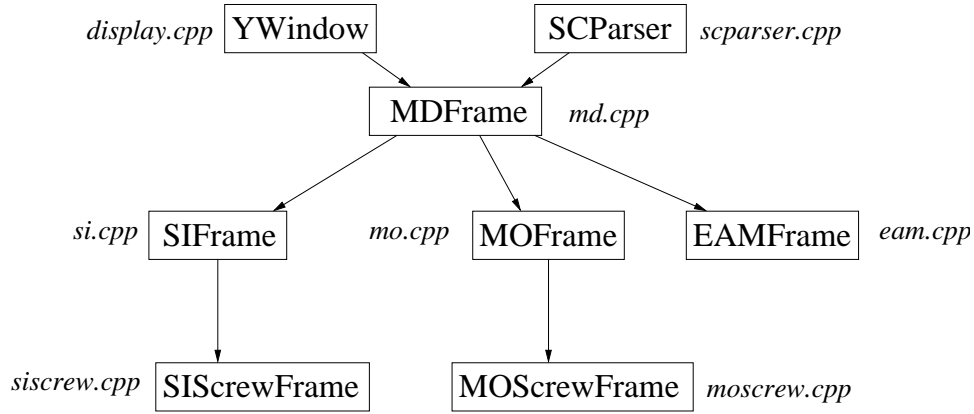


Figure 3: MD++ code structure.

script files, and contains a **YWindow** object as a member to plot atom configurations. **MDFrame** contains the essential variables (such as atom positions) and functions (such as conjugate gradient relaxation) to perform static and dynamic simulations, but it has an empty inter-atomic potential. **SIframe**, **MOFrame**, **EAMFrame** all inherit **MDFrame** and implement their own potentials — Stillinger-Weber for Si, Finnis-Sinclair for Mo, and EAM potential for FCC metals. **SIScrewFrame** and **MOScrewFrame** builds on top of **SIframe** and **MOFrame** respectively, and add more functions to facilitate simulation of screw dislocations.

For each inheritance, the child (e.g. **SIframe**) executable recognizes all the variables and functions that the parent **MDFrame** implemented. Because most of the functions are implemented in **MDFrame**, they are available to all of its children executables.

## 4 Control Variables and Functions

### 4.1 List of Control Variables and Functions

Table 1: List of all control variables in **MDFrame**, in alphabetical order. For detailed explanations, go to the corresponding section.

allocmultiple	Sec.4.3	fortranexe	Sec.4.8	plot_zmax	Sec.4.5
appyshear	Sec.4.6	fortranpath	Sec.4.8	plot_zmin	Sec.4.5
atomcolor	Sec.4.5	hideatomenergy	Sec.4.5	potavgfreq	Sec.4.5
atommass	Sec.4.9	hidetopology	Sec.4.5	potfile	Sec.4.4
atomradius	Sec.4.5	highlightcolor	Sec.4.5	removerc	Sec.4.6
atomTcpl	Sec.4.9	highlightenergy	Sec.4.5	rotateangles	Sec.4.5
autowritegiffreq	Sec.4.5	hrescale	Sec.4.6	shmsize	Sec.4.10
backgroundcolor	Sec.4.5	incnfile	Sec.4.4	savecnfreq	Sec.4.4
basisatoms	Sec.4.3	intercnfile	Sec.4.4	savecn	Sec.4.4
bondcolor	Sec.4.5	latticeconst	Sec.4.6	savepropfreq	Sec.4.4
bondlength	Sec.4.5	latticestructure	Sec.4.6	saveprop	Sec.4.4
bondradius	Sec.4.5	makecnspec	Sec.4.6	scalevelocity	Sec.4.9
boxTcpl	Sec.4.9	MDCASKpot	Sec.4.8	shearrate	Sec.4.9
calphonondispspec	Sec.4.9	mkcutspec	Sec.4.6	shiftspec	Sec.4.6
command	Sec.4.8	mkdipole	Sec.4.6	showavgpot	Sec.4.5
conj_dfpred	Sec.4.7	mkgbspec	Sec.4.6	shrinkboxspec	Sec.4.6
conj_fevalmax	Sec.4.7	moveatomspec	Sec.4.6	sleepseconds	Sec.4.2
conj_fixbox	Sec.4.7	ncpu	Sec.4.10	splicecnspec	Sec.4.6
conj_fixshape	Sec.4.7	outpropfile	Sec.4.4	strainspec	Sec.4.6
conj_ftol	Sec.4.7	perfcnfile	Sec.4.4	stress	Sec.4.7
conj_itmax	Sec.4.7	phonondispfile	Sec.4.4	strrotspec	Sec.4.7
cutpastecnspec	Sec.4.6	pickfixedatomspec	Sec.4.6	timestep	Sec.4.9
dirname	Sec.4.2	plotfreq	Sec.4.5	T_OBJ	Sec.4.9
DOUBLE_T	Sec.4.9	plot_highlight	Sec.4.5	totalsteps	Sec.4.9
energycolorbar	Sec.4.5	plot_restrict	Sec.4.5	usenosehoover	Sec.4.9
equilsteps	Sec.4.9	plot_select	Sec.4.5	vt2	Sec.4.9
extendboxspec	Sec.4.6	plot_xmax	Sec.4.5	win_height	Sec.4.5
finalcnfile	Sec.4.4	plot_xmin	Sec.4.5	win_width	Sec.4.5
fixatomcolor	Sec.4.5	plot_ymax	Sec.4.5	writeall	Sec.4.4
fixedatomenergypartition	Sec.4.5	plot_ymin	Sec.4.5	writevelocity	Sec.4.4



Table 2: List of all functions in **MDFrame**, in alphabetical order. For detailed explanations, go to the corresponding section.

alloccolors	Sec.4.5	refreshnmlist	Sec.4.5
alloccolorsX	Sec.4.5	relax	Sec.4.7
applystrain	Sec.4.6	removeextraatoms	Sec.4.6
calphonondisp	Sec.4.9	removepickedatoms	Sec.4.6
cutpastecn	Sec.4.6	replacenonfixatom	Sec.4.6
cutslice	Sec.4.6	restoreH	Sec.4.6
eval	Sec.4.5	reversergb	Sec.4.5
extendbox	Sec.4.6	rotate	Sec.4.5
fortranrelax	Sec.4.8	runcommand	Sec.4.8
initvelocity	Sec.4.9	run	Sec.4.9
makecn	Sec.4.3	saverot	Sec.4.5
makecut	Sec.4.6	scaleH	Sec.4.6
makedipole	Sec.4.6	setnolog	Sec.4.2
makegb	Sec.4.6	setoverwrite	Sec.4.2
moveatom	Sec.4.6	shiftbox	Sec.4.6
multieval	Sec.4.7	shiftpbc	Sec.4.6
openintercnfile	Sec.4.4	shrinkbox	Sec.4.6
openpropfile	Sec.4.4	sleep	Sec.4.2
openwin	Sec.4.5	splice	Sec.4.6
pickfixedatoms	Sec.4.6	step	Sec.4.9
plot	Sec.4.5	testcolor	Sec.4.5
quit	Sec.4.2	wintogglepause	Sec.4.5
readcn	Sec.4.4	writecn	Sec.4.4
readMDCASK	Sec.4.4	writeintercn	Sec.4.4
readPOSCAR	Sec.4.4	writeMDCASK	Sec.4.4
redefinepbc	Sec.4.4	writePOSCAR	Sec.4.4

## 4.2 Log File and Control

dirname

sleepseconds

setoverwrite setnolog quit sleep

## 4.3 Make Perfect Lattice

allocmultiple basisatoms

makecn

## 4.4 File I/O

finalcnfile incnfile intercnfile

outpropfile perfcnfile phonondispfile

savecnfreq savecn savepropfreq saveprop

potfile

writeall writevelocity

openintercnfile openpropfile readcn readMDCASK readPOSCAR redefinepbc writecn writeintercn writeMDCASK writePOSCAR

## 4.5 X-window Display

atomcolor bondcolor atomradius  
autowritegiffreq backgroundcolor bondlength bondradius  
energycolorbar  
fixatomcolor fixedatomenergypartition  
hideatomenergy hidetopology highlightcolor highlightenergy  
plotfreq plot\_highlight plot\_restrict plot\_select plot\_xmax plot\_xmin plot\_ymax plot\_ymin plot\_zmax plot\_zmin  
potavgfreq rotateangles showavgpot  
win\_height win\_width  
alloccolors alloccolorsX openwin plot reversergb rotate saverot testcolor wintogglepause  
refreshmnlst eval

## 4.6 Structure Manipulation

applyshear cutpastecnspec  
hrescale latticeconst latticestructure makecnspec  
mkcutspec mkdipole mkgbspec moveatomspec  
pickfixedatomspec  
removerc shiftspec shrinkboxspec splicecnspec strainspec extendboxspec  
mkdipole  
applystrain cutpastecn cutslice extendbox makecut makedipole makegb moveatom pickfixedatoms re-  
moveextraatoms removepickedatoms replacenonfixatom restoreH scaleH shrinkbox shiftbox shiftpbcs spliceen

## 4.7 Conjugate Gradient Relaxation

conj\_dfpred conj\_fevalmax conj\_fixbox conj\_fixshape conj\_ftol conj\_itmax  
stress strrotspec  
multieval relax

## 4.8 External Fortran Calls

command fortranexe fortranpath  
MDCASKpot  
fortranrelax runcommand

## 4.9 Molecular Dynamics Simulation

atommass atomTcpl boxTcpl  
calphonondispspec  
equilsteps  
DOUBLE.T  
scalevelocity shearrate  
timestep T\_OBJ totalsteps usenosehoover vt2  
calphonondisp initvelocity run step

## 4.10 Parallel Execution

ncpu shmsize  
spawn collect shmallocate

# 5 More Examples