The MD2 Molecular Dynamics Simulation Package

Cameron F. Abrams
Department of Chemical Engineering
University of California, Berkeley
Berkeley, California 94720
cfa@raven.cchem.berkeley.edu

February 22, 2000

This document serves as both an appendix in my PhD thesis as well as a standalone document included in the source code package available for downloading. The purpose of this document is to publish the simulation code that implements the Tersoff/Brenner Si–C–F force routine for MD simulation of fluorocarbon ion-surface bombardment. The code is written in the C programming language, [1] following ANSI standards, and is distributed over many source code and header files. Included in this document are instructions on how to download the code from the Graves group web site, how to compile and use the code to generate results, and a brief section of documentation, explaining how important pieces of code are implemented.

Although I wrote all the simulation codes used in the various molecular dynamics studies in my thesis, I am only publishing the Si–C–F code, which I call `md2`. I am not publishing the code used for the studies of SiO_2 (implements the Garofalini Si–O potential [2]) nor Cu (implements the Embedded Atom Method potential [3]). I am willing to release those codes to individual researchers upon request.

1 IMPORTANT DISCLAIMER

This document is NOT complete documentation for the MD2 code or any of its affiliated programs. It only highlights some important features, gives a few limited tutorial examples, and discusses programmatic aspects of the code that the author thinks the reader might find interesting. As MD2 was developed as part of academic research for this thesis, and not for commercial purposes, no further attempt at publishing documentation has been, nor is anticipated to be, attempted. This document does NOT constitute any kind of guarantee that the MD2 code or any of its affiliated programs is free of defects of any kind. Furthermore, the author accepts no responsibility for any incorrect results obtained using this code. Finally, beyond the filing date of the PhD thesis that originally contained this document as an appendix, no plans to maintain or support the MD2 code exist. Therefore, the code is available on a strictly 'as is' basis, and any corrections or extensions become the responsibility of whoever wants to use it.

2 Introductory comments

MD2 is a molecular dynamics code specifically written to simulate energetic molecule bombardment of surfaces. It employs a Brenner-form C–F potential [4, 5, 6, 7] with the Si extensions of Beardmore and Smith. [8] It can be used to study energetic neutral bombardment of carbon and silicon surfaces with varying amounts of fluorine incorporation. The energetic neutrals model positive ions that are accelerated through sheath potential drops from a plasma to a bounding surface. Sometimes these bombarding neutrals will therefore be referred to as "ions".

Like all molecular dynamics codes, MD2 can be used to obtain a large number of physical quantities that characterize an atomic-scale surface and its interactions with gas-phase molecules. Evolution of the surface during growth (deposition) or removal (etching) can be examined, and can be characterized in terms of the atomic compositions and microstructure

(via correlation functions). Film stress can be computed as well. Sticking coefficients and sputter yields are also readily obtained. Distributions of sputtered products, their energies and angles of ejection, can also be computed.

As mentioned in the preamble, MD2 is written in the C programming language. This release of MD2 has been run on several different UNIX platforms, including SGI/Irix 6.2, RedHat/Alpha Linux 5.2 and 6.1, IBM/AIX 3.2.5 and 4.1, and HPUX 10.10. Though it conforms to the ANSI standard for the C language, it has been engineered to take advantage of the many utilities of the Unix operating system for postprocessing and data analysis. The following discussion, therefore, applies to compilation and execution in a Unix environment only.

3 Obtaining the code

If you have not already downloaded the MD2 package, (chances are you have if you are reading this document) it can be downloaded from my home page, which is located off of the Graves group home page. You can access this page by first accessing the UC Berkeley College of Chemistry home page (`www.cchem.berkeley.edu`) and following the links through "Chemical Engineering," "People," "Faculty," "David B. Graves (Web Site)," "Group Members," "Cameron Abrams". Or, you can access my home page directly via the URL:

`http://kestrel.cchem.berkeley.edu/Cam/Cam.html`.

Once there, select the "Download Simulation Code" link, and follow the instructions on the next page to download the simulation code package.

Once downloaded, move `md2.tar.gz` into its own subdirectory, `cd` into that subdirectory, and issue the following command to unpack it:

```
gzcat md2.tar.gz | tar vxf -
```

This package includes makefiles that allow for easy compilation of the following programs:

`md2`	main molecular dynamics code
`cfginfo`	computes compositional information from configuration files
`cfg2r3d`	translates configurational data into Raster3D input
`mrg`	merges parallel data files to produce an average
`cgmin`	a conjugate-gradient geometry optimizer

Each of these codes is explained further in the next section.

Below is a listing of all files included in the package.

```
args.c
atom.c
bicubic.c
cfg.c
chem.c
clust.c
```

```
cryst.c
dblmat.c
domd.c
ion.c
point.c
push.c
tbt_sicf.c
tricubic.c
main.c
args.h
atom.h
bicubic.h
cfg.h
chem.h
clust.h
cryst.h
dblmat.h
domd.h
ion.h
point.h
push.h
tbt_sicf.h
tricubic.h
units.h
sicf_params.h
tricof.h
genforce.h
cfg2r3d.c
colormap.c
r3d_utils.c
colormap.h
r3d_utils.h
README
manual.ps
aSi.cfg
makefile.sgi
makefile.linux
```

4 Using the code

4.1 Compilation

Compilation of `md2` is managed by the Unix `make` utility. [9] The first step in compilation is to select the appropriate `makefile` for your system. The two makefiles included in the distribu-

tion work for the SGI/Irix (`makefile.sgi`) and Alpha/Linux platforms (`makefile.linux`). The Alpha/Linux makefile, `makefile.linux`, is the most general, and should work for any Unix operating system with a C compiler, provided you know the command that invokes the compiler (usually `cc` or `gcc` – `gcc` is assumed in `makefile.linux`).

Once you have selected (and perhaps edited) the appropriate makefile, copy it to the file `makefile` (no extension). Then, issue the command

```
make md2
```

to invoke the compilation. This should result in the executable `md2` residing in the current directory.

The makefiles can be used to make the auxiliary programs as well, except for `mrg` which can be compiled by itself.

4.2 Atomic configuration data files

Atomic configurations, or collections of atomic positions and velocities, which are used for both input and output by the MD2 code, are managed as ASCII text files. The atomic configuration file contains the definition simulation cell size and the positions, velocities, and atomic numbers of all atoms in the cell. Each line in this file is one of three possibilities: (1) a comment, (2) a keyword/value pair, or (3) an atom.

Comment lines always begin with the '%' character. Keyword/value pairs always begin with the '#' character, followed by a space, followed by the keyword and value(s). A line corresponding to an atom always begins with an integer. This integer is strictly not used by the code, but serves as the atom's unique identification number in some post-processing routines.

Below is an example of a configuration file that describes a single perfluoronaphthalene molecule near the center of a 22×22×22 Å simulation cell.

```
% md series 2 atomic configuration file (c) 1999 cfa
% Created by md2::cfg v. 1.00 b. 1
% '%' or ';' in first column means line is a comment
% '#' in first column means line is a keyword/value pair
#CreationDate   14:32PM;Fri11Feb2000
#UnitSystem     APVK
#BoxSize.xyz    21.61480000000000 21.61480000000000 21.61480000000000
#nCell.xyz      4 4 4
#cfgTime        0.000
#Periodicities  1 1 0
% Number of Atoms in this cfg = 18
% By elements: F 8 C 10
% Number of Fixed Atoms = 0
% Number of Trashed Atoms (not included) = 0
0       C       -0.0309 0.1334  0.0000  0.0000  0.0000  0.0000  0
1       C       -1.2909 -0.5896 0.0000  0.0000  0.0000  0.0000  0
2       C       1.2221  -0.5997 0.0000  0.0000  0.0000  0.0000  0
```

```
3       C       -1.2964 -2.0423 0.0000  0.0000  0.0000  0.0000  0
4       C       1.2159  -2.0493 0.0000  0.0000  0.0000  0.0000  0
5       C       -0.0417 -2.7738 0.0000  0.0000  0.0000  0.0000  0
6       F       -0.0275 1.4383  0.0000  0.0000  0.0000  0.0000  0
7       F       -2.4195 0.0659  0.0000  0.0000  0.0000  0.0000  0
8       C       2.4813  0.1225  0.0000  0.0000  0.0000  0.0000  0
9       F       -2.4289 -2.6907 0.0000  0.0000  0.0000  0.0000  0
10      C       2.4676  -2.7841 0.0000  0.0000  0.0000  0.0000  0
11      F       -0.0455 -4.0787 0.0000  0.0000  0.0000  0.0000  0
12      F       2.4602  -4.0888 0.0000  0.0000  0.0000  0.0000  0
13      F       2.4884  1.4273  0.0000  0.0000  0.0000  0.0000  0
14      C       3.7352  -0.6108 0.0000  0.0000  0.0000  0.0000  0
15      C       3.7283  -2.0632 0.0000  0.0000  0.0000  0.0000  0
16      F       4.8551  -2.7212 0.0000  0.0000  0.0000  0.0000  0
17      F       4.8690  0.0355  0.0000  0.0000  0.0000  0.0000  0
```

First, the comment and keyword lines shown here are always automatically generated whenever a configuration is output. There are six keyword/value pairs in this configuration file. The first is the creation date of the configuration. It is just a character string. The second is a special keyword that states the unit system of the following configuration data. "APVK" is the default, and means that lengths are in Ångstroms and times in picoseconds. (Actually, this is the only unit system this version of the code understands; previous versions could run in both APVK and Stillinger-Weber units. You should never see the Unit System designated as anything other than "APVK".) The third is the box size, or size of the simulation box. Note that this keyword has 3 values associated with it, one for each of the x, y, and z dimensions of the box respectively. The fourth is the number of crystal unit cells in each of the three dimensions, and is not strictly used by the code. Instead, this is leftover from when a configuration is created by the crystal generator – I reused that header for this configuration. The fifth is the configuration time, and is an obsolete parameter that I haven't gotten rid of yet. The last is very important – it tells the code in which dimensions to enforce periodic boundary conditions. Here, the configuration is telling the code to enforce them in the x and y dimensions, but not in the z dimension. This is the required standard for performing simulations of surface processes.

A special note about keyword/value lines in a configuration file: the code will allow any keywords to be specified in the configuration file, not only the ones shown. A list of keyword/value pairs the code understands appears later in Table 1 Due to the order with which keywords are processed when running the code, those keywords in the configuration files are always the last keywords processed, so their values supercede any other values that may have been entered earlier. This is discussed further in the section on running the code.

The rest of this file is devoted to describing atomic positions and velocities. The format of each atom data line is as follows:

```
id# sym rx ry rz vx vy vz fix?
```

where `id#` is the integer ID number of the atom (which should be unique, but not necessary), `sym` is the atomic symbol of the atom (`F` is fluorine, `C` is carbon, and `Si` is silicon), `rx ry rz`

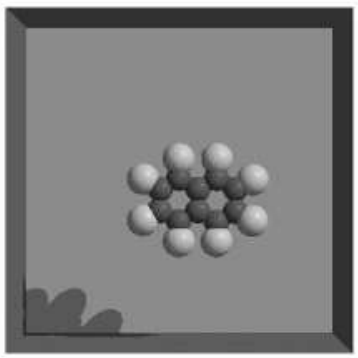

Figure 1: A rendering of the perfluoronapthalene molecule represented by the configuration file in the text.

are the x, y, and z coordinates of the atomic center, `vx vy vz` are the x, y, and z components of the atomic velocity, and `fix?` is 1 if the atom is static and 0 if the atom is dynamic. (Static, or "fixed", atoms are used to anchor the simulation cell at the bottom; most atoms are dynamic.) Note that since the unit system is APVK, that `rx ry rz` are in Å, and `vx vy vz` are in Å/ps.

Below is a rendering of the above configuration showing both the perfluoronaphthalene molecule and the simulation box.

A final note about configuration files: 99.9% of the time they are generated automatically by the code; that is, the user usually never has to type one in manually. The example configuration shown in this section was created by first manually entering approximate positions for atoms in a molecule of perfluoronaphthalene, and then using the `cgmin` code to perform a conjugate-gradient geometry optimization, leading to a relaxed perfluoronaphthalene molecule represented by this configuration of F and C atoms.

4.3 Running `md2`

4.3.1 Command-line arguments and keyword/value pairs

The MD2 code is controlled by several global variables that are set at runtime by the user via keyword/value (K/V) pairs. K/V pairs may be specified at runtime in three locations: (1) on the command line, (2) in an optional setup input file, and (3) in a configuration file. Typically, the K/V's in a configuration file are specific to that configuration, such as the box size and unit system.

An example command for running md2 is shown below.

```
% md2 -ic in.cfg -oc out.cfg -n 10000 -dt 1.e-3 +dtv
```

This command invokes `md2` with the following keyword/value pairs. The input configuration file is named `in.cfg`. This file must exist. The output configuration file that will be created is called `out.cfg`. The number of time steps is 10,000, and the time step value is $1.0\times10-3$ ps, for a total integration time of 10 ps. The `+dtv` is a flag that turns off the adaptive time step feature. A flag is technically a K/V pair, only the value is implicit in the existence of

Table 1: Keyword/value pairs for the MD2 code.

Command-line	Setup/Cfg-File	Global Variable	Description
+dtv	AdaptTimeStep	dtvar_	Flag: time step variation OFF
-Pset	BerendsenP	bmb_Pset_	setpoint pressure
-ptau	BerendsenPTau	bmb_tau_	P-bath rise time
-Tset	BerendsenT	bhb_Tset_	setpoint temperature
-tau	BerendsenTau	bhb_tau_	T-bath rise time
-bc	BondDefCtrlPar	bond_ctrl_	Bond definition param.
-Lx	BoxSize.x	Lr_.x	X-size of box
-Ly	BoxSize.y	Lr_.y	Y-size of box
-Lz	BoxSize.z	Lr_.z	Z-size of box
-L	BoxSize.xyz	Lr_	size of box as X Y Z
-ic	CfgInputFile	cfg_infile_0	input configuration file
-cnm	CfgNameDesc	cfgName_	configuration "name"
-oc	CfgOutputFile	cfg_outfile_0	output configuration file
-csn	CfgSnapshotFile	cfg_snapfile_0	output snapshot file
-ocl	ClustOutputFile	clust_outfile_0	output cluster data file
-ctm	ConfigTime	cfgTime_	configuration "time"
-ftol	ConjGradTol	ftol_	Tolerance of CG Minimizer
-cdt	CreationDate	cfgCreateDate_	Configuration "date"
-xlre	CrystRe	xl_Re_	Bond l. for crystal generation
-dc	DesorbCtrlPar	des_ctrl_	Desorption control param.
-echo	Echo	echo_	Flag: echo info ON
-fcctest	fcctest	FCC_TESTER	Flag: Test F_{cc} ON
-fccta	fcctestA	fcc_a_t	Test F_{cc} A-value
-i0	FirstImpactNum	impact0_num_	First impact number
-ff	FixFrac	ff_	Fraction of atoms fixed (gen.)
-hccta	hcctestA	hcc_a_t	Hcc test A-value
-hbn	HeatBathStartStep	bhb_start_	Time step to turn on T-ctrl
-hijtest	hijtest	HIJ_TESTER	Flag: Test H_{ij} ON

the flag, so no value actually accompanies it. This is an example of a "relaxation" run: no ion impact is performed. The atoms are merely "pushed around" for a given amount of time. This is the type of run in which one would compute time-averaged statistical quantities, such as radial distribution functions and velocity autocorrelation functions. Furthermore, it is an example of an NVE-integration, or an integration at constant number of atoms, constant volume, and constant total energy, sampling therefore the microcanonical ensemble of states. This is because no temperature or pressure control is specified. This run should conserve total energy (kinetic + potential).

In Tables 1, 2, and 3 are listed every keyword/value pair used by MD2, accompanied by a brief description of each. Although many keyword/value pairs exist, only relatively few are required to be set by the user in order to use the code. Many of them are for debugging purposes only, and should be used with a diagnostic compilation.

Table 2: Keyword/value pairs for the MD2 code (continued).

Command-line	Setup/Cfg-File	Global Variable	Description
-i	ImpactNum	impact_num_	Single-run impact number
-idi	IntDataInt	data_int_	Output interval for log data
-ion	Ion	ion_.spec	Ion species (H, CH3, Ar, etc)
-ionr	IonAtomPos	ion_.r0ex	Ion atom pos. as (# rx ry rz)
-ionv	IonAtomVel	ion_.v0ex	Ion atom vel. as (# vx vy vz)
-ionE	IonEnergy	ion_.E_i	Ion energy
-ionief	IonImpEFrac	ion_.imp_ef	Ion ke-frac @ impact
-oi	IonOutputFile	ion_.outfile_0	Ion output data file
-ioi	IonOutputInt	ion_.oint	Ion output data interval
-ionP	IonPhi	ion_.P_i	Ion azimuthal angle
-ipf	IonPoolFile	ionPool_filename_	Ion pool input file
-ionT	IonTheta	ion_.Th_i	Ion polar angle
-iui	IonUpdateInt	ion_.uint	Ion data update interval
-i1	LastImpactNum	impact1_num_	Final impact number
-t0	LowerIntegLim	tlim0_	Time value of t_0
-mxdt	MaxTimeStep	maxdt_	Max. time step value
-mndt	MinTimeStep	mindt_	Min. time step value
-mbn	MBathStartStep	bmb_start_	P-ctrl start step
-nc	nCell.xyz	nCell_	# of unit cells in x y z
-ncx	nCellx	nCell_.i	# of unit cells in x
-ncy	nCelly	nCell_.j	# of unit cells in y
-ncz	nCellz	nCell_.k	# of unit cells in z
-nocl	NoClusterReport	clust_report_	Flag: clustering OFF
-nocore	NoCore	USE_HIE_CORE	Flag: Moliere core OFF
+fcc	NoFcc	USE_FCC	Flag: Brenner F_{cc} OFF
+hij	NoHij	USE_HIJ	Flag: Brenner H_{ij} OFF
-n	NumOfTimeSteps	Ndt_	Number of Time Steps
-per	Periodicities	per_	PBCs: x[0/1] y[0/1] z[0/1]
-prt	PhiRateTol	phi_rate_tol_	Max. $\Delta\phi_{ij}$

Table 3: Keyword/value pairs for the MD2 code (concluded).

Command-line	Setup/Cfg-File	Global Variable	Description
-q	Quiet	quiet_	Flag: quiet mode ON
-rdf34	RDF34	RDF_34_	Flag: RDF 33,34,43,44 mode ON
-rdff	RDFFileName	RDF_filename_	RDF output file name
-rdflz	RDFLowZlimit	rdf_loZlim_	Lower z-limit in RDF comp.
-rdf	RDFOn	RDF_ctrl_	Flag: RDF computation ON
-rdfdr	RDFResolution	RDF_dR_	RDF discretization
-rdfn	RDFStartStep	rdf_start_	Time step to begin RDF comp.
-is	SetupInputFile	setup_infile_	Setup input file name
-os	SetupOutputFile	setup_outfile_	Setup output file name
-d2	Sicf_diag2	SIC_DIAG2_	Flag: 2-body diagnostics ON
-d3	Sicf_diag3	SIC_DIAG3_	Flag: 3-body diagnostics ON
-soi	SnapshotInt	cfg_outint_	Snapshot output step-interval
-tdA	ThmlDesThryA	td_A_	Preexp. factor, 1° Thml. Des. Thry.
-tdt	ThmlDesThryTau	td_tau_	Decay time, 1° Thml. Des. Thry.
-dt	TimeStep	dt_	Time step
-uc	UnitCell	unitCell_	Crystal unit cell designation
-t1	UpperIntegLim	tlim1_	Time value of t_1

The file name keywords are used in a special way. If the value entered for any filename contains a string of '#' or '?' characters, this string is replaced by the current value of the impact number. For example, if the user specifies that the output file for configurations is "`####.cfg`" (as I normally do), then the code will replace the `####` with "`0001`" for the first impact and write to "`0001.cfg`", "`0002`" for the second impact and write to "`0002.cfg`", etc. The only filename for which this is not the case is the input configuration filename, in which the '#'-string is replaced by the previous impact number. This is so the i impact trajectory uses as input the output configuration from the $i-1$ impact trajectory.

The "`-ion`" keyword is used to select the identity of the bombarding ion. The ion is selected using one of the following valid ion names:

```
He, C, F, Ne, Si, Ar, Kr, Xe,
CC, FF, SiSi,
CF, SiF, SiC,
CF2, SiF2, SiC2, Si2C,
CF3, SiF3, SiC3, Si3C,
CF4, SiF4, SiC4, Si4C
```

4.3.2 Example: Simulating continuous CF_3 bombardment of a:Si

In the following tutorial, I will demonstrate how to run the MD2 code to perform 10 serial impacts of 50 eV, normal incidence CF_3 onto an initially bare a:C surface. This is an example of the many sets of runs that were performed to compile the data that was analyzed and reported in the bulk of my PhD thesis. I will assume that you are running the code on a

typical Unix workstation. For each one of these impacts, we want the total integration time to be 0.25 ps. Furthermore, we would like to use Berendsen temperature control to keep the layer at 300 K, using a rise time of 0.01 ps.

The first step is to create a directory to hold the results of this run.

```
% mkdir cf3-1
% cd cf3-1
```

Now, it is advisable to create three subdirectories to hold data files. I usually call these subdirectories `ion`, `cfg`, and `clu`. `ion` will hold data files that record information about the bombarding ions. `cfg` will hold the configuration files, and `clu` will hold files on sputtered and desorbed clusters.

```
% mkdir ion cfg clu
```

Now, we need to obtain an initial configuration corresponding to an initially bare a:Si surface. Let's assume you have such a configuration stored in the file "`a:Si.cfg`" (which is indeed provided in this release of the MD2 package, for your convenience) in your home directory. We will copy it into the `cfg` subdirectory in the run directory.

```
% cp ~/a:Si.cfg ./cfg/0000.cfg
```

Notice that we have named it "`0000.cfg`". Because of the special way file names are handled, the first impact trajectory will read its input configuration from "`cfg/0000.cfg`".

Now we are ready to begin the simulation. We do so by issuing the following command:

```
% md2 -oc cfg/####.cfg -ion CF3 -ionE 50 -Tset 300 -tau 0.01 \
  -n 250 -dt 1.e-3 +dtv -i1 10 > log &
```

Here, we have taken advantage of the fact that the default value for the input configuration file name is "`cfg/####.cfg`", and that the default number for the first trajectory is 1. The data files for ion and cluster information are defaulted as "`ion/####.ion`" and "`clu/####.clu`" as well. We have told the code to output configurations as "`cfg/####.cfg`". This signifies to the code that we are running a recursive impact simulation, where the output of one impact trajectory is the input for the next. Furthermore, we have specified the ion as CF_3 with and energy of 50 eV. The setpoint temperature is 300 K, and the thermostat rise time is 0.01 ps. The number of time steps is set at 250, and the time step is fixed at 1.0×10^{-3} ps, giving a total of 0.25 ps of integration time per impact trajectory. The "`+dtv`" turns off the adaptive time step optimization feature. Finally, we have instructed the code to run recursive trajectories until the tenth one is completed. We are redirecting the terminal output of `md2` to a file called "`log`".

When the simulation has completed, here is what we see in the run directory:

```
% ls *
log

cfg:
0000.cfg  0002.cfg  0004.cfg  0006.cfg  0008.cfg  0010.cfg
```

```
0001.cfg  0003.cfg  0005.cfg  0007.cfg  0009.cfg

clu:
0001.clu  0003.clu  0005.clu  0007.clu  0009.clu
0002.clu  0004.clu  0006.clu  0008.clu  0010.clu

ion:
0001.ion  0003.ion  0005.ion  0007.ion  0009.ion
0002.ion  0004.ion  0006.ion  0008.ion  0010.ion
```

The file "log" contains a lot of information. Here are the first few lines:

```
# md series 2 argument handler, call 1
# CfgOutputFile cfg/####.cfg
# TimeStep 0.00100
# BerendsenT 300.00000
# BerendsenTau 0.01000
# IonEnergy 50.00000
# NumOfTimeSteps 250
# LastImpactNum 10
# AdaptTimeStep 0
# Ion CF3
# Welcome to md series 2 v 1.00 b 110200 (c) 1999 cfa
# md series 2 periodic table initializer (c) 1999 cfa
# md series 2 pef initializer (c) 1999 cfa
# md series 2 filenames initializer (c) 1999 cfa
# Data sizes:
#  atoms:       112     x       2000                    = 224000
# nNodes:       24      x       2000    x       100     = 4800000
# cNodes:       24      x       2000                    = 48000
# total: 5072000
# input cfg filename: cfg/####.cfg
# output cfg filename: cfg/####.cfg
# clust data filename: clu/####.clu
# ion data filename: ion/####.ion
#
```

Here, each module is reporting itself, and the total size of configuration data is reported (in bytes). Each of the important input/output file names is reported. The `log` continues with the beginning of trajectory 1:

```
# Trajectory #1 (1 out of 10) begins here
# md series 2 do_md (1) driver (c) 1999 cfa
# input cfg filename: cfg/0000.cfg
# output cfg filename: cfg/0001.cfg
# clust data filename: clu/0001.clu
```

```
# ion data filename: ion/0001.ion
# cfg/0000.cfg: 506 atoms: Si_506
# md series 2 argument handler, call 2
# CfgOutputFile cfg/####.cfg
# UnitCell DCSIA
# TimeStep 0.00100
# BoxSize.x 21.61480
# BoxSize.y 21.61480
# BoxSize.z 21.61480
# BerendsenT 300.00000
# BerendsenTau 0.01000
# IonEnergy 50.00000
# nCellx 4
# nCelly 4
# nCellz 4
# NumOfTimeSteps 250
# LastImpactNum 10
# AdaptTimeStep 0
# Ion CF3
# Eb_(300.00) = 5.35738e-01 eV
# md series 2 ion creator (c) 1999 cfa
# ion zbuf = 15.11143
# md series 2 cfg pusher (c) 1999 cfa
[data for trajectory #1 integration]
.
.
.
```

Here, the log is reporting the beginning of the first trajectory. You can see that the specific filenames have been substituted for the template filenames specified by the user. A second round of argument handling is done when the configuration file is read in, and this results in the reporting of a new set of parameters. Then the bombarding ion is created a certain height above the surface (with random center-of-mass positions in x and y, as is the default), and the integration begins. Each line in the section abbreviated as "`[data for trajectory #1 integration]`" has the following form:

```
# tm ke pe et ed %ed T nClst [0/1] Plat
```

where `#` is the time step number, `tm` is the current time value, `ke` is the instantaneous kinetic energy of the system, `pe` is the instantaneous potential energy of the system, `et` is the instantaneous total energy (kinetic + potential) of the system, `ed` is the amount of energy drift, or the current instantaneous total energy minus the total energy at time zero, `% ed` is the energy drift expressed as a percentage of the initial total energy, `T` is the instantaneous temperature of the system in Kelvins, `nClst` is the number of clusters detected in the system, `[0/1]` is a flag that is 1 if all the atoms in the system have been successfully assigned to a cluster (if this ever reads 0 the code has a bug), and `Plat` is the instantaneous lateral pressure of the system in GPa.

At the end of the data for trajectory 1, we see the following:

```
# md series 2 cfg pusher completed
# md series 2 cluster cleaner (c) 1999 cfa
# desorption control is 2: Thrml on
# Keep Product 54(Ok): Si_1 F_1 : be  -2.74647e+00 (ctrl=[PS+TDT])
# Keep Base 59(Base): Si_505 F_2 C_1 : be  0.00000e+00 (ctrl=[PS+TDT])
# md series 2 cluster cleaner complete
# Tr 1, N 510, tm 171 s = 2.850 m, <tm/tr> 171.000 s = 2.850 m, \
  <tm/st/a> 1.336e-03 s, Final El.#: Si 506 F 3 C 1 Total 510
#
# Trajectory #2 (2 out of 10) begins here
```

Here, the integration end is reported, and then the beginning of the cluster analysis begins. Desorption control (by default) is set to "Thermal", which means that any cluster with a finite binding energy absolutely less than the precomputed thermal binding energy reported at the beginning of the integration as "`Eb`" will be deleted from the configuration. In this case, a single SiF cluster is detected, but its binding energy is -2.75 eV, so it is not desorbed (i.e., not deleted). Also detected is the "base", which is the surface, and of course it is never deleted. The line that begins `# Tr 1` tells us that this 510-atom integration took 171 seconds of wall clock time. At this point, trajectory 1 is complete and trajectory 2 can begin.

At the very end of the `log`, we see the following line:

```
# Thank you for using md series 2.
```

Why does every line in the `log` begin with the '`#`' character? Only the lines that do not contain integration data start with '`#`'. That is so we can use `log` directly as a data file for `gnuplot`, or we can use any number of Unix utilities to parse it (e.g. `awk`, `sed`, `grep`, `perl`, etc), knowing that we can "grep out" any line that contains a '#' if we want to only consider integration data, or conversely, we can consider *only* lines that begin with '#' if we want to extract some reported time or parameter value. For example, because `gnuplot` treats any line beginning with `#` as a comment in a data file, I can very easily plot the temperature trace for an impact in the following way:

```
% gnuplot
gnuplot> set term post eps enh "Helvetica" 18
Terminal type set to 'postscript'
Options are 'eps enhanced monochrome dashed defaultplex "Helvetica" 18'
gnuplot> set out "t.eps"
gnuplot> set xlabel "t, ps"
gnuplot> set ylabel "T, K"
gnuplot> p "log" u 2:8 w l
gnuplot> q
```

Here, I am telling `gnuplot` to create a plot in the encapsulated PostScript file "`t.eps`", plotting column 8 (that's instantaneous temperature in K) versus column 2 (that's time in ps). Figure 2 depicts file "`t.eps`" (it's an encapsulated postscript file).

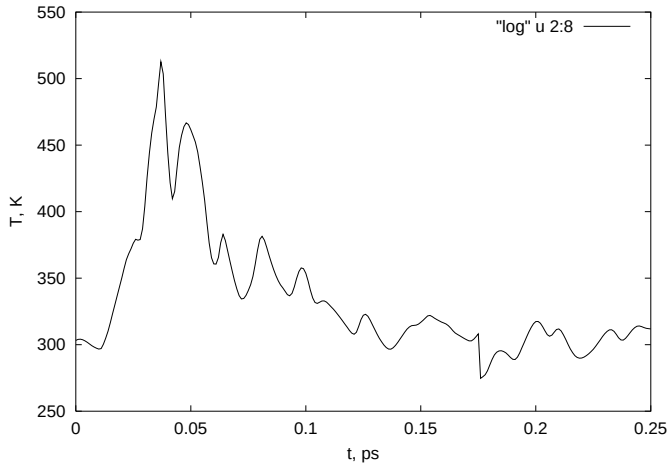

Figure 2: Temperature vs. time for a single CF_3^+ impact on a:Si: Example of how to plot data from a `log` file using `gnuplot`.

That is a small example of how the `log` file can be used. In the next section I will continue with this tutorial by describing the `ion` and `clu` data files.

4.4 Output of `md2`

Besides the integration log detailed in the tutorial, there are two other important output files generated when `md2` simulates an ion impact. These are (1) bombarding ion data files, and (2) sputtered cluster data files. These are described below, using the same tutorial example.

4.4.1 Bombarding ion data files

To explain the format and content of the `ion` data files, I will continue with the tutorial in which I explain continuous ion bombardment by CF_3^+. Here is the beginning of the file `ion/0001.ion`:

```
# md series 2 impacting ion data (c) 1999 cfa
# Constituent atoms: C#506 F#507 F#508 F#509
# Constituent atom initial configuration
# 0     C       0.0900  -6.0375 15.7361 0.0000  0.0000  -118.2442       0
# 1     F       1.2117  -5.3104 16.1295 -0.5505 -0.3568 -118.4373       0
# 2     F       -0.2479 -7.2127 16.4040 0.1658  0.5768  -118.5720       0
# 3     F       -0.6937 -5.5892 14.6749 0.3847  -0.2200 -117.7233       0
# Initial COM position: 0.09004 -6.03746 15.73612
# Initial energy: 50.00
# st, t, ke/E_i, ipe, epe, d.x, d.y, d.z, |d|, #b, imp?
```

Like the `log` file, the `ion` file is divided into header information (lines that begin with '`#`') and data. The header information tells us that the CF_3 ion contains carbon atom #506, and fluorines #507, #508, and #509. It then reports their positions and velocities in `cfg`-file

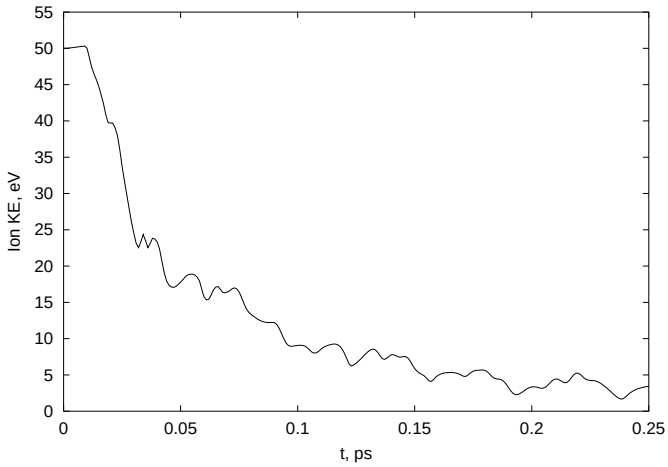

Figure 3: CF_3^+ ion kinetic energy vs. time

format. Then the center-of-mass position is reported, along with the initial energy in eV. Then the data begins. The format of a line of data in the `ion` file is as follows:

```
st, t, ke/E_i, ipe, epe, d.x, d.y, d.z, |d|, #b, imp?
```

where `st` is the time step, `t` is the integration time in ps, `ke/E_i` is the ion's kinetic energy expressed as a fraction of its initial energy, `ipe` is the ion's internal potential energy in eV, `epe` is the ion's external potential energy in eV, `d.x, d.y, d.z, |d|` are the center-of-mass displacements of the ion as x, y, z components and the absolute value, `#b` is the number of bonds in which atoms originally in the ion participate, and `imp?` is a flag that is 0 until the impact has occurred. The "impact" is defined as the moment when either (a) one of the ion bonds is broken (polyatomic ions only), or (b) the ion's kinetic energy drops below some specified percentage of its original kinetic energy. This specified percentage is set at 75% in the code by default. It can be changed using the "`-ionief`" keyword.

The ion's internal potential energy is just the sum of intra-ion bond energies. The external potential energy tells us how much potential energy exists due to bonds made between atoms that were originally part of the ion and atoms that were originally part of the surface. Initially, this value is 0.00 of course. When this value begins to deviate from 0.00, we know the ion is beginning to interact with the surface.

We can use the data in the `ion` data file to gain insight into what occurs in any given trajectory. For example, we can use `gnuplot` to make a plot of the kinetic energy of the ion as a function of time, as shown in Fig. 3.

4.4.2 Sputtered cluster data files

To explain the utility of the `clu` cluster data files, I will again continue the use of the tutorial.

When I ran this tutorial, it happened that all the atoms from the bombarding ions stuck to the surface except for one fluorine atom in trajectory number 7. Here is what the file `clu/0007.clu` contained:

```
#BoxSize.xyz    21.61480000000000 21.61480000000000 21.61480000000000
% Cluster 56(Trash): F_1  1.50193 0.00000
533 F 9.9427 3.7037 12.6112 19.0134 -23.5678 24.6687 0
% Cluster count 1
% End of cluster report. (c) 1999 cfa
```

The thing to notice first is that this is nothing more than a configuration file. It is a special configuration file in which the comments tell us something about what was sputtered or desorbed in this trajectory. This file is telling us that Cluster #56 (the number isn't important; it's just an index used by the cluster identification routine) was determined to be "Trash", or completely unbound from the surface. ONLY "trash" clusters are ever written to the cluster data file; other clusters detected in the integration remain part of the overall configuration.

This cluster contained 1 F atom, so its empirical formula is reported as `F_1`. The next number on that line is the kinetic energy of the cluster; here we see that the cluster as over 1.5 eV kinetic energy. The last number on the line is the binding energy of the cluster, or its interaction energy with the atoms of the surfaces. A value of 0 tells us that this cluster is not interacting with the surface. Following that line of cluster information is the configuration-style data for all the atoms in that cluster. This tells us exactly where the atoms of the cluster are at the end of the impact trajectory. Note that using the formula $KE = \frac{1}{2}mv^2$ and the configuration data given, we can compute the KE of the cluster, obtaining the same value as that displayed on the line displaying the empirical formula.

4.5 Using `cfginfo` and `mrg`

The utility code `cfginfo` has the primary function of computing depth profiles for various quantities, including species concentration and bond densities. `cfginfo` is written in C, and needs to be compiled.

`cfginfo` works by discretizing the configuration in the z direction. A bin width of 1 Å is assumed by default ($\delta z = 1$ Å), but can be changed by using the `-hdel <binsize>` keyword.

I will refer to the type of output created by `cfginfo` as an `info` file. To create an `info` file for a given `cfg` file, one issues a command of the following form:

```
% cfginfo 0007.cfg > 0007.info
```

Here, I have created the file `0007.info` by operating `cfginfo` on the configuration file `0007.cfg`. Now we consider `0007.info` to see what it tells us about `0007.cfg`.

The first few lines of `0007.info` are as follows:

```
# cfginfo v 2.0 (c) 1999 Cameron Abrams
# md series 2 periodic table initializer (c) 1999 cfa
# md series 2 pef initializer (c) 1999 cfa
# 0007.cfg: 533 atoms: Si_506 F_20 C_7
# md series 2 argument handler, call 1
# CreationDate 16:40PM;Fri11Feb2000
# BoxSize.xyz 21.614800 21.614800 21.614800
```

Table 4: Contents of an `info` file: Depth-resolved compositions.

Column	Quantity	description
1	z, Å	distance from bottom of cell
2	Si count	number of Si between z, $z + \delta z$
3	ρ_{Si}, Å^{-3}	number density of Si between z, $z + \delta z$
4	$\mathrm{x_{Si}}$	mole fraction of Si between z, $z + \delta z$
5	$\mathrm{Z_{Si}}$	avg. coordination of Si between z, $z + \delta z$
6	$\mathrm{dbd_{Si}}$	avg. number of dangling bonds per Si between z, $z + \delta z$
7	F count	number of F between z, $z + \delta z$
8	ρ_{F}, Å^{-3}	number density of F between z, $z + \delta z$
9	$\mathrm{x_F}$	mole fraction of F between z, $z + \delta z$
10	$\mathrm{Z_F}$	avg. coordination of F between z, $z + \delta z$
11	$\mathrm{dbd_F}$	avg. number of dangling bonds per F between z, $z + \delta z$
12	C count	number of C between z, $z + \delta z$
13	ρ_{C}, Å^{-3}	number density of Si between z, $z + \delta z$
14	$\mathrm{x_C}$	mole fraction of C between z, $z + \delta z$
15	$\mathrm{Z_C}$	avg. coordination of C between z, $z + \delta z$
16	$\mathrm{dbd_C}$	avg. number of dangling bonds per C between z, $z + \delta z$
17	$\mathrm{rho_{tot}}$, Å^{-3}	total number density between z, $z + \delta z$
18	$\sum x$	sum of all mole fractions (should always be 1.0)
19	Si-Si, Å^{-3}	bond density between z, $z + \delta z$

```
# nCell.xyz 4 4 4
# Periodicities 1 1 0
# UnitSystem APVK
# Eb_(0.00) = 0.00000e+00 eV
# 0007.cfg: Total 533 T=346.90 Si 506 F 20 C 7
# bin volume = 467.19958 cuA, = 4.67200e-22 cc
# 0007.cfg: V=1.64324e+04 cuA, Pxy(0)=1.78548e-01 GPa
```

The first bit of information we have about the `cfg` file is that it contains 533 atoms, 20 of which are fluorines, 7 of which are carbons, the remainder of which are silicons. Then we see the familiar K/V pairs. `cfginfo` then re-reports the atomic count along with the temperature. Then the bin volume is reported, both in Å^3 and cc, and then the configuration volume estimate is reported along with an instantaneous measure of the lateral pressure. That concludes the header information in the info file. The column-oriented, depth-resolved data contained in an `info` file is shown in Tables 4 and 5.

4.5.1 `mrg`: Creating an average data file from several inputs

In order to get meaningful statistics out of this type of data, one should average these quantities over several similar configurations. This is done using the `mrg` utility, which I wrote as part of the `md2` distribution. `mrg` simply averages many parallel column-oriented data files. For example, if one wanted to compute an average `info` file from the files `0000.info`,

Table 5: Contents of an `info` file: Depth-resolved compositions (concluded).

Column	Quantity	description
20	Si-F, Å^{-3}	bond density between z, $z+\delta z$
21	Si-C, Å^{-3}	bond density between z, $z+\delta z$
22	Si-C(sp^1), Å^{-3}	bond density between z, $z+\delta z$
23	Si-C(sp^2), Å^{-3}	bond density between z, $z+\delta z$
24	Si-C(sp^3), Å^{-3}	bond density between z, $z+\delta z$
25	F-C, Å^{-3}	bond density between z, $z+\delta z$
26	F-C(sp^1), Å^{-3}	bond density between z, $z+\delta z$
27	F-C(sp^2), Å^{-3}	bond density between z, $z+\delta z$
28	F-C(sp^3), Å^{-3}	bond density between z, $z+\delta z$
29	C-C, Å^{-3}	bond density between z, $z+\delta z$
30	C-C(sp^1), Å^{-3}	bond density between z, $z+\delta z$
31	C-C(sp^2), Å^{-3}	bond density between z, $z+\delta z$
32	C-C(sp^3), Å^{-3}	bond density between z, $z+\delta z$
33	C(sp^1)-C(sp^1), Å^{-3}	bond density between z, $z+\delta z$
34	C(sp^1)-C(sp^2), Å^{-3}	bond density between z, $z+\delta z$
35	C(sp^1)-C(sp^3), Å^{-3}	bond density between z, $z+\delta z$
36	C(sp^2)-C(sp^2), Å^{-3}	bond density between z, $z+\delta z$
37	C(sp^2)-C(sp^3), Å^{-3}	bond density between z, $z+\delta z$
38	C(sp^3)-C(sp^3), Å^{-3}	bond density between z, $z+\delta z$

`0001.info`, `0002.info`, `0003.info`, `0004.info`, and `0005.info`, one could use `mrg` to create the file `avg1.info`:

```
% mrg 000[0-5].info > avg1.info
```

`mrg` assumes that all files it operates on are *parallel*, that is, they have the same form, column-for-column and line-by-line. Only the values differ. `mrg` creates as output a single file of this same form in which each value is an average of all the identically located values from the input files. In the command above, I am taking advantage of the Unix C-shell's ability to expand the wildcard designation "`[0-5]`" to create the list of files.

4.6 Rendering images using `cfg2r3d` and Raster3D

The `Raster3D` [10] package, consisting of several programs, is a freeware geometry rendering program. I have decided to use it to make atoms-as-spheres renderings of some of my configurations. The command `render` from this package is required to do this. The package can be downloaded in source code form from

`http://www.bmsc.washington.edu/raster3d/raster3d.html`.

Any Unix system administrator should be able to help you build and install this package, if it is not already done on your computer system.

The code `cfg2r3d`, part of the `md2` distribution, translates `md2` configuration files into the input language needed by the `render` program to draw the atoms as colored spheres in three-dimensional space. `cfg2r3d.c`, and its companion source and header files, is included in the `md2` distribution. After compiling and installing it, issuing the `cfg2r3d` command with no arguments will give the following summary of the many command-line switches useful in rendering images:

```
Usage: cfg2r3d <cfgName> [control params] [> <r3dName>]
<cfgName>   = name of cfg data file
<r3dName>   = name of r3d data file
-df <directions output file>
Rotation Controls: Values are in degrees
 -xr <x-rotation> -yr <y-rotation> -zr <z-rotation>
Translation Controls: Values are in cfg-unit length
 -xtr <x-translation> -ytr <y-translation> -ztr <z-translation>
Shifting Controls: Values are in cfg-unit length
 -xs <x-shift> -ys <y-shift> -zs <z-shift>
(Shifting means that the actual atom coordinates are
translated *prior* to rendering the image.)
Zoom and Normalizing Length Controls:
 -zoom <scaleFactor> -norm <normMax>
Colormode:  Natural, KE-resolved, or Depth-resolved
 -colmode <nat|ke|dep|pe>
 -kemin <kemin,eV> -kemax <kemax,eV> -cooltrans <YES|NO> for ke
colormode only
 -pemin <pemin,eV> -pemax <pemax,eV> for pe colormode only
 -zmin <zmin> -zmax <zmax> for dep colormode only
Object controls:
 -noatoms does not render atoms
 -ionid <ionIndex> colors ion
 -ioncolor <r g b>
 -pbc renders periodic boundary planes
 -pbcHt <float> height extent of vertical pbc planes
 -pbcClr <float> clarity of pbc planes
 -pbcColor <rgb or color key> color of pbc planes
 -pbcCtwy cuts the -y pbc plane off at x for z > 0.0
 -velarrfor <id-list> atoms for which velocity arrows are drawn
 -velarrlf <float> length of velocity arrows
 -velarrshaftrad <float> radius of velocity arrow shafts
 -velarrheadw <float> width of velocity arrow heads
 -velarrheadl <float> length of velocity arrow heads
 -velarrheadl <float> length of velocity arrow heads
 -velarrheadnface <int> number of faces of pyramidal arrow heads
 -hlids <id-list> list of id's for atoms that are to
                 be highlighted.
```

```
 -hlclrs <color-namelist> list of colors for highlighted atoms
 -onlyelem <elem-str> only atoms of this element are rendered.
Raster3D render Header Macros:
 -xt <xTiles(32)> -yt <yTiles(32)>
 -xp <xPixelsPerTile(18)> -yp <yPixelsPerTile(18)>
 -scheme <scheme(3)>
 -bg (colorName or <r> <g> <b> (0 0 0))
 -noshadows -phong <phongPower(25)>
 -slc <secondaryLightContributionFraction(0.15)>
 -alc <ambientLightContributionFraction(0.05>
 -src <specularReflectionComponentFraction(0.25)>
 -eyepos <EYEPOS(4.0)>
 -mlpos <x> <y> <z> (1 1 1) (Main Light Position)
 -orbit or -oc <x> <y> <z> Orbit Center
Orbit means rotate the light source direction with any
object rotation to simulate the camera orbiting a stationary
object.  The (x, y, z) coordinates are the orbit center.
The -orbit keyword assumes the orbit center is 0, 0, 0.
The -oc keyword requires exactly 3 numbers to specify
the orbit center if the user wishes it to be something
other than (0, 0, 0).  Instead of a floating point value,
you may specify 'X' for +Box.x/2 and 'x' for -Box.x/2,
and likewise for <y> and <z> orbit center coords.
```

Continuing with the tutorial, I will now show how to use `cfg2r3d` and `render` to make images of some configurations. First, we can make top- and side-view images of the a:Si initial surface represented by `cfg/0000.cfg` using the following commands:

```
% cfg2r3d cfg/0000.cfg -bg white | render -tiff 0000-top.tif
% cfg2r3d cfg/0000.cfg -xr 90 -bg white | render -tiff 0000-side.tif
```

Each command is actually two commands "piped" together so the output of the first is the input of the second. First, `cfg2r3d` translates the configuration file `cfg/0000.cfg` into a form understandable by `render`, and then writes this translated output to the terminal. The `|` redirects this output into the `render` command, which is told to create the TIFF-format image `0000.tif`. The second command uses the `-xr 90` switch to rotate the configuration around the x-axis by 90°. Fig. 4 shows the images `0000-top.tif` and `0000-side.tif` (as encapsulated postscript files).

One is literally unrestricted in how one constructs the view of an object. One can specify any number of rotations, translations, lighting scenarios, colors, magnifications, image sizes, etc. Remember that rotation operations do not commute with each other: this means that the order in which one specifies rotations is important. `cfg2r3d` performs rotations on the object in the order specified on the command line.

One can also specify different color "schemes": kinetic energy, potential energy, or depth. This schemes will color atoms based on their individual metric of one of these quantities.

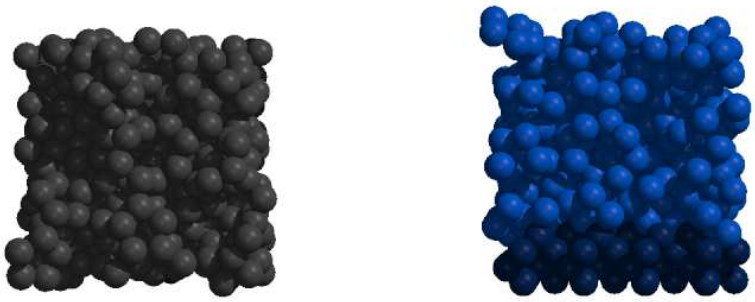

Figure 4: Renderings of a top-view and side-view of the a:Si surface used in the previous tutorial.

For the depth-resolved color scheme, for example, the deepest atoms are colored blue, and the highest atoms are red, and the atoms in between are colored such that the color changes along a spectrum from blue to red.

Finally, there is an easy way to change the color and sizes of the spheres according to their atomic identities. `cfg2r3d` looks for a file called `chem.inp` in whatever directory you issue the command. (If it doesn't exist, `cfg2r3d` used hard-coded default values for radii and colors.) One can create and edit this file to change how you would like the atoms to appear. The format of the `chem.inp` file is very simple. It should contain an arbitrary number of lines of the following form:

```
sym radius r g b
```

where `sym` is the atomic symbol (i.e. C, H, Si, F, etc.), `radius` is the radius (in σ's, where 1 σ is 2.1 Å), and `r g b` is the fractional (rgb) triple describing the color. [(111) is white, (000) is black, (100) is red, (010) is green, (110) is yellow, (0.5,0.5,0.5) is grey, etc.]

4.7 Making animations (SGI only)

It is relatively simple to construct animations of trajectories using the `cfg2r3d`, `render`, and SGI `dmedia`. The tasks required to make an animation are as follows.

1. Using `md2`, generate one configuration file for each frame of the animation. Animations typically run at 15 frames per second, so for a 10 second animation, you should create 150 frames.

2. For each configuration file, use `cfg2r3d` and `render` to make an image frame.

3. Using the SGI/`dmedia` command `makemovie`, compile the frames into an SGI-format animation (a `*.mv` file).

4. If you like, you can use `dmconvert` to translate the SGI movie into an MPEG.

I will explain each of these steps in greater detail using the following tutorial.

4.7.1 Making an animation: A tutorial

Say you wish to make an animation of a CF_3 radical impacting a surface. To do this, we will use `md2` to carry out the simulated impact, and we will tell it to output "snapshot" configurations at given intervals. These snapshot configurations will then be rendered into images that will become the frames of our animation.

The first step is to make a separate directory and subdirectories to hold the data for the movie:

```
% mkdir movie-1
% cd movie-1
% mkdir cfg tif
```

Here, `cfg` will hold the snapshot configurations, and `tif` will hold the image frames. Next, we obtain the initial configuration from somewhere. I will assume you have a configuration called `surf1.cfg` in your home directory.

```
% cp ~/surf1.cfg .
```

Now, we are ready to begin the simulation.

```
% md2 -ic surf1.cfg -oc surf1.2.cfg -ion CF3 -ionE 40 -Tset 300 -tau 0.01 \
      -csn cfg/####.cfg -soi 10 -dt 1.e-3 +dtv -n 1500 > log &
```

Here I have used two new switches. `-csn` names the snapshot configuration files, which we supply as a template such that the `####` substring is replaced with the time step number. `-soi` tells `md2` how often to output snapshots. Here, we have specified that it output a snapshot every 10 timesteps. `-n` and `-dt` are set so that the integration lasts 1.5 ps, and we'll be creating 150 snapshots.

When the simulation is over, the `cfg` directory will be contain 250 configuration files: `0000.cfg`, `0010.cfg`, `0020.cfg`, ..., `2490.cfg`, `2500.cfg`. Now we will tell `cfg2r3d` to create the TIFF images for each configuration. But first, we need to decide on the object orientation within the image, so we run `cfg2r3d` and `render` on one of the configurations several times using different magnifications, rotations, translations, lighting scenarios, etc, until we find one we like. Let's say that the following image rendering was the one we liked:

```
% cfg2r3d cfg/0000.cfg -xr 90 -yr -20 -xr -30 -pbc -bg white\
      -zoom 1.5 -ytr -12.5 | render -tiff test.tif
```

Now, if we were extremely patient, we could issue this command once for each configuration, being careful to change the output TIFF image file name to something appropriate, like `tif/0000.tif`. But a better way to do this is to write a shell script that does the repetition automatically. I like to write shell scripts in the `tcsh` shell so I can use the `foreach` command. (You might prefer to use `for` in either the Bourne shell or C shell.) Here is an example of a `tcsh` script called `mktif` that can be used in this situation:

```
#!/bin/tcsh
set c2rflags="-xr 90 -yr -20 -xr -30 -pbc -bg white -zoom 1.5 -ytr -12.5"
foreach cfg (cfg/????.cfg)
  set n=$cfg:t:r
  set tif=tif/${n}.tif
  cfg2r3d $cfg $c2rflags | render -tiff $tif
  echo Created $tif
end
echo Finished.
```

If you create this script using a text editor, then you must change its mode so that Unix knows it is executable, and then you can run it like any other command.

```
% chmod 744 mktif
% mktif > mklog &
```

After `mktif` finishes, then the `tif` directory will contain 250 TIFF image files: `0000.tif`, `0010.tif`, `0020.tif`, ..., `2490.tif`, `2500.tif`. Now we are ready to compile them into a SGI-format animation:

```
% makemovie -f sgi -c mvc1 -o 01.mv tif/????.tif
```

Check the man pages on `makemovie` for more information on this command. Briefly here, I have told it to create an SGI-format movie (the alternative is QuickTime, or `-f qt`), that the compression scheme is "MVC1" (a moderate compression), and the output file is to be named `01.mv`. The final arguments to `makemovie` are always the image files in the order in which you want them to appear in the movie. Because the shell expands `tif/????.tif` into a list of files in the order we created them (because of the way we named them using the time step value), we know this is the proper way to specify the input image files to `makemovie`.

After `makemovie` finishes, the file `01.mv` exists and can be played using the SGI movie player, `movieplayer`:

```
% movieplayer 01.mv
```

(Note that you can only watch the movie if you are on the native display. Otherwise the transfer rate is too slow.) The final task is, if we wish, to compress this movie file into MPEG form, which is the highly-compressed digital animation standard. We can do this on the SGI using the `dmconvert` command, if your SGI machine has the MPEG license. (Ask your system administrator to find out.) To do the conversion we issue the following command:

```
% dmconvert -f mpeg1v -p video,sx=288,sy=288 01.mv 01.mpg
```

Again, consult the man pages for `dmconvert` for more information on this powerful command. After this command finishes, the final product is a 288×288 pixel MPEG animation file, ready for viewing. You can make the pixel size of the MPEG anything $n \times n$ such that n is divisible by 16. 288×288 is an adequate size. Such an animation will consume perhaps 1–2 megabytes of disk space.

The MPEG format is compact enough to put these animations on web sites. The Graves group web site, `kestrel.cchem.berkeley.edu`, contains several such animations. Animations created in this way have been shown at numerous conferences by both myself and Prof. Graves. It is both of our opinion that animations like these truly enhance our understanding of the complicated atomic-scale nature of surface bombardment by reactive ions.

4.8 Running `cgmin`

`cgmin` is a molecular geometry optimization code that uses the Si–C–F interatomic potential. A different version, `cgmin-sich`, encodes Brenner's C–H potential [4] directly, and was used to reproduce all of Brenner's molecular atomization energies to check the accuracy of the Brenner force routine I developed. [11] Both versions encode the Fletcher-Reeves-Polak-Ribiere conjugate gradient function minimization algorithm, detailed on p. 423 in *Numerical Recipes in C, 2nd Ed.*, specifically to the function $E(\mathbf{r}_i)$, where $\mathbf{r}_i$ are the coordinates of atoms in a given molecule.

The primary role played by `cgmin` in my thesis work has been as a way to check that my Si–C–F force routine predicts the same atomization energies as the C–F force routine developed by Junichi Tanaka. [6] Below, I explain how to use `cgmin` to find a minimum energy of a perfluoronaphthalene molecule, pictured in Fig. 1.

First, one must create an input configuration with a "best-guess" as to the Cartesian positions of each atom in the molecule. I created the configuration below by "gluing" together two perfluorobenzene molecules, performing all of the translation calculations by hand. What this configuration represents is an initial guess of the atomic positions in a perfluoronaphthalene molecule in the xy plane near the middle of a 22×22×22 Å simulation cell.

```
#CreationDate   10:53AM;Fri12Nov1999
#UnitSystem     APVK
#BoxSize.xyz    21.61480000000000 21.61480000000000 21.61480000000000
#nCell.xyz      4 4 4
#cfgTime        0.000
#Periodicities  1 1 0
% Number of Atoms in this cfg = 12
% By elements: F 6 C 6
% Number of Fixed Atoms = 0
% Number of Trashed Atoms (not included) = 0
0       C       0.0049  0.1042  0.0000  0.0000  0.0000  0.0000  0
1       C       -1.2383 -0.6080 0.0000  0.0000  0.0000  0.0000  0
2       C       1.2432  -0.6166 0.0000  0.0000  0.0000  0.0000  0
3       C       -1.2433 -2.0408 0.0000  0.0000  0.0000  0.0000  0
```

```
4       C       1.2383  -2.0494 0.0000  0.0000  0.0000  0.0000  0
5       C       -0.0049 -2.7615 0.0000  0.0000  0.0000  0.0000  0
6       F       0.0083  1.1740  0.0000  0.0000  0.0000  0.0000  0
7       F       -2.1629 -0.0699 0.0000  0.0000  0.0000  0.0000  0
8       C       2.4717  0.1851  0.0000  0.0000  0.0000  0.0000  0
9       F       -2.1717 -2.5722 0.0000  0.0000  0.0000  0.0000  0
10      C       2.4629  -2.7874 0.0000  0.0000  0.0000  0.0000  0
11      F       -0.0083 -3.8312 0.0000  0.0000  0.0000  0.0000  0
12      F       2.5083  -3.8312 0.0000  0.0000  0.0000  0.0000  0
13      F       2.5083  1.2740  0.0000  0.0000  0.0000  0.0000  0
14      C       3.7432  -0.6166 0.0000  0.0000  0.0000  0.0000  0
15      C       3.7383  -2.0494 0.0000  0.0000  0.0000  0.0000  0
16      F       4.3717  -2.9722 0.0000  0.0000  0.0000  0.0000  0
17      F       4.4717  0.2000  0.0000  0.0000  0.0000  0.0000  0
```

To minimize this structure, one issues the following command, and the following output is generated on the terminal:

```
% cgmin -ic perfluoronaphthalene.cfg \
  -oc perfluoronaphthalene.cg3.cfg -ftol 1.e-7
# md series 2 argument handler, call 1
# CfgInputFile perfluoronaphthalene.cfg
# CfgOutputFile perfluoronaphthalene.cg3.cfg
# ConjGradTol 0.00000
# md series 2 periodic table initializer (c) 1999 cfa
# md series 2 pef initializer (c) 1999 cfa
# md series 2 filenames initializer (c) 1999 cfa
# perfluoronaphthalene.cfg: 18 atoms: F_8 C_10
# md series 2 argument handler, call 2
# CfgInputFile perfluoronaphthalene.cfg
# CfgOutputFile perfluoronaphthalene.cg3.cfg
# CreationDate 10:53AM;Fri12Nov1999
# ConjGradTol 0.0000001
# BoxSize.xyz 21.614800 21.614800 21.614800
# nCell.xyz 4 4 4
# Periodicities 1 1 0
# UnitSystem APVK
# Eb_(0.00) = 0.00000e+00 eV
#iter   PE      Tol     nCalls
#[#]    [eV]
0       -7.805676e+01   0.000000e+00    1
1       -8.995412e+01   2.379473e+01    26
2       -9.123127e+01   2.554297e+00    47
3       -9.219374e+01   1.924931e+00    71
4       -9.277012e+01   1.152767e+00    97
```

```
5       -9.294451e+01   3.487829e-01    113
6       -9.303106e+01   1.730977e-01    130
7       -9.312747e+01   1.928067e-01    146
8       -9.324537e+01   2.357995e-01    163
9       -9.328715e+01   8.356805e-02    186
10      -9.330946e+01   4.461878e-02    203
11      -9.331688e+01   1.485105e-02    221
12      -9.331816e+01   2.548251e-03    240
13      -9.331970e+01   3.085228e-03    259
14      -9.332049e+01   1.582013e-03    278
15      -9.332073e+01   4.734865e-04    296
16      -9.332091e+01   3.672378e-04    317
17      -9.332105e+01   2.717569e-04    335
18      -9.332114e+01   1.812497e-04    357
19      -9.332118e+01   8.538287e-05    378
20      -9.332122e+01   7.586948e-05    398
21      -9.332127e+01   1.039290e-04    419
22      -9.332129e+01   3.137100e-05    442
23      -9.332130e+01   2.587599e-05    461
24      -9.332131e+01   1.742612e-05    486
# Thank you for using md series 2.
```

Here, the input configuration file is named `perfluoronaphtalene.cfg`, and the output configuration to be created is `perfluoronaphthalene.cg3.cfg`. The tolerance for the minimization, defined as the difference in the objective function value (here, total energy of the molecule) for any two consecutive interations, is specified at 1×10^{-7} eV. The output is structured as follows. The header output is the familiar header information, just like with `md2`. The data shows the progression toward convergence of the minimzation. The first column is the iteration number; this particular run took 24 iterations. The second column is the objective function value at that iteration; here it is the total energy of the molecule in eV. The third column is the residual error in the objective function value; this is the quantity that the conjugate gradient minimization algorithm is trying to drive to zero. The fourth column is the cumulative number of function evaluations at each iteration; a total of 486 evaluations were required for this minimization. The final total energy of the minimized structure is 93.32131 eV. The minimized configuration is the one shown in Sec. 4.2.

5 About the code

In this section, I briefly describe the code, primarily in terms of its data types and algorithms. Although I am not including here any complete source code listings (these are available for downloading), I am including examples which serve to illustrate my development philosophy. Again, this section does *not* constitute full documentation of the `md2` code. In this description, I will use the terminology of the C programming language, [1] so familiarity with C is a prerequisite for understanding this section. Also, abstract data types such as linked lists,

stacks, and queues appear in this discussion. For in-depth descriptions of such data types, the reader is referred to Chapter 3 of Lewis and Denenberg. [12]

5.1 Important abstract data types (ADTs) and their implementations

5.1.1 `point`: 3-component vectors

The header file `point.h` defines the `pt` ("point"), `ipt` ("integer point") and `ptPtr` ("point pointer") data types, and declares several functions that operate on them. A point is a structure that contains three double precision floating point members, called `x`, `y`, and `z`.

```
typedef struct POINT * ptPtr;
typedef struct POINT
{
    double x;
    double y;
    double z;
} pt;
```

A `ptPtr` is a pointer to a point data type.

Points are used for atomic positions, velocities, accelerations, and forces, as well as any other quantities which are mathematically treated as three-component vectors. Integer points are defined like points, only with integer members `i`, `j`, and `k`. The simulation box size is measured in number of crystal unit cells in each dimension, so it is represented by an `ipt`.

`point.h` declares several functions which operate on `pt`'s, but always in the form of `ptPtr`'s. For example, the function header `double ptPtr_abs (ptPtr p);` declares the function that, given a pointer to a point "p", returns the absolute value of that point, which is defined as the square root of the sum of the squares of its members.

One important example: **the minimum image convention** [13] is implemented by the `point` module, using the function `ptPtr ptPtr_minimg (ptPtr res, ptPtr a);`, which computes the minimum image of vector pointed to by `a` and places the result in the vector pointed to by `res`. Below is the definition of `ptPtr_minimg`:

```
extern pt Lr_, half_Lr_; /* box size and 1/2-box size */
extern ipt per_; /* periodicities */
ptPtr ptPtr_minimg (ptPtr res, ptPtr a)
{
    int shift = 0;
    /* shift is one of -1, 0, 1 */
    /* the following three adjustments implement the minimum image
     * convention for a set box size (given by the global Lr_) */

    /* if res and a do not point to the same vector, copy a onto res */
    if (res!=a) ptPtr_copy(res, a);
```

```
    /* if x-dir has periodic boundaries, compute the x-shift */
    shift=per_.i*res->x/half_Lr_.x;
    if (shift==2)  shift=1;
    if (shift==-2) shift=-1;
    /* if the shift is nonzero, adjust the x-component of *res */
    if (shift) res->x-=Lr_.x * shift;

    /* if y-dir has periodic boundaries, compute the y-shift */
    shift=per_.j*res->y/half_Lr_.y;
    if (shift==2)  shift=1;
    if (shift==-2) shift=-1;
    /* if the shift is nonzero, adjust the y-component of *res */
    if (shift) res->y-=Lr_.y*shift;

    /* if z-dir has periodic boundaries, compute the z-shift */
    shift=per_.k*res->z/half_Lr_.z;
    if (shift==2)  shift=1;
    if (shift==-2) shift=-1;
    /* if the shift is nonzero, adjust the z-compenent of *res */
    if (shift) res->z-=Lr_.z * shift;

    /* done -- return res */
    return res;
}
```

5.1.2 `atom`: Atoms as ADTs

An `atom` is a structured data type that contains information about a simulation atom, including its index number, its element type, pointers to its position, velocity, acceleration, and force vectors, among other information.

The implementation of the `atom` data type requires the definition of two interconnected structured data types. The first is the `nNode`, or "neighbor list node". A "neighbor list" is a linked list of `nNodes` that belongs to an `atom`, where each node in the list contains an address, or `atomPtr`, of an atom that is a physical neighbor of the `atom` that owns the neighbor list. The `nNode` type definition, which is part of the `atom.h` header file, is shown below.

```
typedef struct theAtom *atomPtr;
typedef struct neighbor_list_node *nNodePtr;
/* nNode:  a Neighbor List entry */
typedef struct neighbor_list_node
{
    int         index;  /* index of this neighbor node */
    atomPtr     addr;   /* address of atom to which this node corresponds */
    nNodePtr    next;   /* pointer to next nNode in this nList */
} nNode;
```

The second data type is the actual `atom`, also part of `atom.h`.

```
typedef struct theAtom
{
    int         id;     /* unique index # of atom               */
    sym_type    sym;    /* atomic symbol (a spec_table index)   */
    ptPtr       pos;    /* pointer to pos_[id]                  */
    ptPtr       vel;    /* pointer to vel_[id]                  */
    ptPtr       hac;    /* pointer to hac_[id]                  */
    ptPtr       frc;    /* pointer to frc_[id]                  */
    double      mv2;    /* mass*velocity^2                      */
    double      q;      /* charge on/electron density at atom   */
    int         nNbrs;  /* number of neighbors on nList         */
    nNodePtr    nList;  /* pointer to list of neighbors         */
    nNodePtr    bCards; /* supply of business cards to "hand out"
                         * to neighbors for their nLists        */
    atomPtr     next;   /* ptr to the next atom                 */
    int         state;  /* atom state                           */
    void        *c;     /* cluster designation  (a cNodePtr)    */
    struct              /* bit-flags                            */
    {
        unsigned int is_RDF_a   :   1;  /* atom is RDF-active   */
        unsigned int is_BAD_a   :   1;  /* atom is BAD-active   */
        unsigned int is_PEF_a   :   1;  /* atom is PEF-dimer-active     */
    } flags;
    short pbc;          /* 4-bit periodic boundary code.  This field
                         * is only used for sputtered/scattered atoms. */
} atom;
```

I'll only explain the important members of `atom` here. First, `id` is the unique integer identification number assigned to each atom by the `cfg` module, described in the next section. `id` indexes the atom's vectors in the parallel arrays managed by `cfg`. `sym` is the atomic symbol of the atom, and the `sym_type` data type is defined by the `chem` module, also explained in a later section. `pos`, `vel`, `hac`, and `frc` each point to elements of arrays of `pt`'s declared by `cfg`, and correspond to the atom's position, velocity, 1/2-acceleration, and force, respectively. `nNbrs` is the number of neighbors currently occupying the atom's neighbor list, which is headed `nList` (recall that it is a linked list). Another linked list of `nNodes`, called `bCards` contains a supply of duplicate `nNodes` containing information about the owner of the list. This leads to a discussion of how **neighbor lists** are implemented.

Neighbor lists are implemented using the paradigm of exchanging business cards. Each atom begins with a "supply" of its own business cards, each of which contains the address of the atom. This supply is a linked list headed by `bCards`. When a pair of `atom`'s is considerd by the code, the code may determine that the two atoms are physically close enough to become designated as "neighbors". At that instant, the two atoms exchange business cards: an `nNode` is "popped" from the `bCard` stack of one atom and "pushed" onto the `nList` stack of the other atom, and vice versa. When a card is "pushed" onto a stack, it is "stamped"

with a unique sequential ID number. Since this only involves changing the appropriate `next` pointers (the realm of linked-list pointer arithmetic), very little data is actually moved in memory; consequently this is a fast way to implement neighbor lists. After all unique pairs of atoms have been considered in this way, each atom owns a stack of business cards in its `nList` which contains information about all of its neighbors. One need only traverse the linked list `nList` to visit all of the list-owner's neighbors.

The remaining important atom fields are now discussed. Since the configuration is represented to all other modules as a linked-list of `atom`'s, each `atom` has a `next` pointer, which contains either the address of the next `atom` in the list, or `NULL` if that atom is the last atom in the list. `state` is an integer designation of the state of the atom, where an atom is considered to be at all times in one of the following states:

```
enum {IS_UNDETERMINED, IS_FIXED, IS_SURF, IS_SPUTTERED, IS_UNCLEARED,
      IS_PUSHTHRU, IS_BULK, IS_UNBOUND, IS_SCATTERED, IS_PROJECTILE,
      ATOM_NSTATES};
```

Definitions of each of these states appears in the in-code documentation of `atom.c`, so for brevity these are not discussed further here. Finally, the void pointer `c` points the the `clustNode` that designates the cluster to which this atom belongs. Clusters are also explained in a later section.

5.1.3 `cfg`: Atomic configurations

The `cfg` module is responsible for declaring and managing the memory used to hold configuration data. Configuration data is primarily four parallel arrays of `pt`'s, an array of `atom`'s, the address of the first atom in the array, and the global supply of "business cards":

```
static pt pos_[MAXNUMATOMS];            /* positions */
static pt vel_[MAXNUMATOMS];            /* velocities */
static pt hac_[MAXNUMATOMS];            /* half-accelerations */
static pt frc_[MAXNUMATOMS];            /* forces */
/* Configuration interface */
static atom atomarray_[MAXNUMATOMS];    /* array of atoms */
atomPtr L_=&(atomarray_[0]);            /* address of head of array of atoms,
                                         * as atoms are accessed in a linked-list
                                         * fashion by some routines */
static nNode bcardarray_[MAXNUMATOMS][MAXNUMNEIGHBORS];
```

The first four arrays are static so that no other module can directly access them. They are only accessible through an `atom`'s `pos`, `vel`, `hac`, and `frc` `ptPtr`'s. `MAXNUMATOMS` is defined as a macro in `cfg.h`, and can be set to something other than the default of 2000 by a "-D" flag on the compilation. I *should* have implemented `cfg` to atomatically declare exactly the number of array elements necessary based on an input configuration size, but I didn't. The `atomPtr L_` is the head of the linked list of atoms by which the rest of the code refers to the configuration. The two-dimensional static `nNode` array, `bcardarray_` allocates all of the `nNode`'s necessary for the neighbor lists. Neither do the `atom`'s declared in `atomarray_` nor the `nNode`'s declared in `bcardarray_` ever move in memory; only their respective `next`

pointers change to determine the "order" of the lists. This makes sorting by any key and shuffling of "cards" very fast compared to sorting data in parallel arrays.

The main routine in `cfg` is called `cfg_establish`, which either reads in the configuration from a specified input file, or generates a crystal configuration based on specified keyword values. The routine `cfg_report` is responsible for outputting the resulting configuration to an output file, along with supplementary information.

5.1.4 `chem`: The 'periodic table'

The `chem` module defines an manages the `element` data type:

```
typedef struct ELEMENT
{   /* record of data on an element */
    char    name[15];       /* name of element */
    double  mass;           /* atomic mass */
    int     z;              /* atomic number */
    int     v;              /* # valence electrons */
    char    sym[3];         /* atomic symbol */
    double  radius;         /* radius (for rendering software) */
    struct
    {
        double r, g, b;
    } color;                /* color (for rendering software) */
} element;
```

The array of `element`'s, called `per_table[]` serves as an abbreviated periodic table of elements for the `md2` family of codes. This array is indexed by variables of the `sym_type` data type, which are just integers with names corresponding to atomic symbols:

```
#define Si              1
#define F               2
#define H               3
#define C               4
#define O               7
#define He              10
#define Ne              11
#define Kr              12
#define Ar              13
#define Xe              14
#define Br              15
#define Cu              16
#define Cl              22
```

The actual values of the integers do NOT correspond to atomic numbers. Declaring these as macros is a little troublesome – in every code that `#includes`'s `chem.h`, the string `H` is replaced by the literal integer 3, for example. That just means I can't use `H` (or any other atomic symbol) as a variable name.

The main routine in `chem.c` is `Chem_InitializePeriodicTable`, which sets up the periodic table array by assigned proper atomic numbers, masses, radii, "colors", etc. A call to `Chem_InitializePeriodicTable` *must* appear as one of the first things in any code that uses this module.

5.2 Routines

In this section, I discuss several important routines that are not discussed in the previous section on ADTs.

5.2.1 `args`: Argument handler

The `args` module handles all of the command line arguments and assigns the proper global variable values based on these argments. It functions by matching each keyword encountered in the `argv[]` array to a database of keyword/variable pairs. This database is an array of `arg_type` data types:

```
#define MAXARG 100
typedef struct ARGFUNC
{
    int i;                      /* Unique id number of this keyword,
                                 * assigned at runtime, never really used */
    short s;                    /* Flag that is set if the runtime
                                 * user sets this keyword */
    int l;                      /* Number of elements in value */
    char * key1;                /* Keyword 1; command-line */
    char * key2;                /* Keyword 2; input file */
    void * var;                 /* Value; points to a vector with
         * "l" members */
    void (*asnf)(void*, char*); /* Assignment function */
    void (*prnf)(FILE*,char*,void*); /* Output function */
} arg_type;
```

Each user-specifiable global variable has an `arg_type` instance in the database. Each `arg_type` instance represents a keyword/variable "pair". The two members `key1` and `key2` are strings that contain the name of the keywords that correspond to this variable. For example the keyword strings "-dt" and "TimeStep" both correspond to the variable "`dt_`", which is the integration time step. (Keyword/value pairs were explained previously in Sec. 4.3.)

The integer `l` states how many individual quantities constitute the "value" for this variable. The variable might be a vector, which has 3 quantities, or a single real number, which is only one. The void pointer `var` is typecast during the database declaration to point to an `l`-member array of whatever primitive quantity data type makes up the variable. That array is the global variable that is being assigned. The void function pointer `asnf` points to one of several functions that typecasts and assign the appropriate global variable based on the `var` array. For example, the time step variable, `dt_`, is assigned using the function `dblasn`:

```
double * dblp;
void dblasn (void *v, char *s){dblp=(double*)v;*dblp=(double)atof(s);}
```

The argument database is declared an initialized at the same time. Below is the initialization line for the `dt_` variable:

```
{0,0,1,"-dt","TimeStep",&dt_,dblasn,dblprn},
```

This initialization says that the `var` member of this database array element enclosed by {} is the address of the variable `dt_`, which has the keywords "-dt" and "TimeStep", has a single value, and uses the function `dblasn` to assign it from the string found next to the keyword in the argument list. The function `dblprn` is the appropriate output function for this variable.

This implementation of an argument handling database is flexible, in that we can easily introduce keyword/value pairs of almost any type into the same database array. For example, the global variable `Ndt_` is the number of integration time steps. Its initialization line the the declaration of the database is as follows:

```
{0,0,1,"-n","NumOfTimeSteps",&Ndt_,intasn,intprn},
```

All variables listed in Tables 1, 2, and 3 are set using the `arg_handler` routine, which uses the generality of the `arg_type` data type to compactly perform all assignments:

```
static char buf0[MAXCHAR]="";
void arg_handler (int argc, char * argv[], short bwc)
{
    int i=0,a=0,l=0;
    char * p;
    short found=0;
    void help (char *);

    if (!arg_initialized) arg_initialize();

    /* Visit each argument in the argument list */
    for (a=0;a<argc;a++)
    {
        i=0;
        found=0;
        /* search for the current keyword in the database */
        while (keys[i].var&&!found)
        {
            if (!strcmp(argv[a], keys[i].key1)||
                !strcmp(argv[a], keys[i].key2))
            {
                /* build the string buffer of values from the
                 * argument list */
                buf0[0]='\0';
                p=buf0;
```

```
                for (l=1;l<=keys[i].l;l++)
                {
                    sprintf(p, "%s%s", (l>1?" ":""), argv[++a]);
                    p+=strlen(argv[a])+(l>1?1:0);
                }
                /* Assign the variable value from the values in the buffer */
                keys[i].asnf(keys[i].var, (keys[i].l?buf0:NULL));
                keys[i].s=1;
                found=1;
            }
            i++;
        }
        /* if the current keyword is not found, help and exit */
        if (a&&!found&&!keys[i].var&&bwc==TRAP_BADWORDS) help(argv[a]);
    }
}
```

5.2.2 `push`: Velocity-Verlet integrator

The `push` module implements the velocity-Verlet [14] (vV) integrator. Below is the pared-down code fragment that shows this implementation. The actual routine in `push` contains a few more lines of code that handle (1) tabulation and output of integration data, (2) data updates for the bombarding ion, and (3) configuration snapshot outputs.

```
void cfg_push (void)
{
    FILE * efp=stdout;
    if (!dt2_) dt2_=dt_*dt_;

    if (!quiet_)
    {
        printf("# md series 2 cfg pusher (c) 1999 cfa\n");
        fflush(stdout);
    }

    t_=t0_;
    rt_=tlim0_;
    if (!tlim1_) tlim1_=tlim0_+Ndt_;
    cfg_setforce();         /* Initial force routine call */
    cfg_push2();            /* Initial velocity half-update */
    E_=Eo_=KE_+PE_;         /* Initial total energy calculation */
    header_out(efp);
    while ((!Ndt_&&tlim1_&&rt_<=tlim1_)||(Ndt_&&(t_-t0_)<Ndt_))
    {
        t_++;
```

```
        rt_+=dt_;
        cfg_push1();        /* Step 1 of Velocity Verlet */
        cfg_setforce();     /* Force routine */
        cfg_push2();        /* Step 2 of Velocity Verlet */
        E_=KE_+PE_;
    }
}
```

Here, `t_` is the integer time step, not to exceed `Ndt_` if set, and `rt_` is the integration run time in picoseconds, not to exceed `tlim1_`.

The function `cfg_push1()` implements the first half of the vV integration, in which atomic positions are updated and velocities are "half" updated using the old value of acceleration. Then, the function `cfg_setforce()` computes new forces on all atoms given the updated positions, and also computes new accelerations from these forces. `cfg_setforce()` also computes total potential energy, which is stored in the global `PE_`. Then, the function `cfg_push2()` implements the second half of the vV integration, in which atomic velocities are "half" updated using the new value of acceleration. This function also computes the new kinetic energy, which is stored in the global `KE_`. `cf_setforce()` is defined in the force routine module, and cfg_push1() and `cfg_push2()` appear in the `push` module.

5.2.3 `ion`: Bombarding ion controller

The bombarding ion is represented by a single instance of the `ion_type` data type:

```
typedef struct IONINFO
{
    /* Parameters set by/computed from users specifications */
    int spec;           /* species index */
    short pool;         /* flag used to specify whether ion is
                         * selected randomly from pool (1), or
                         * is set explicitly by the user (0) */
    double E_i;         /* incident energy */
    double Th_i;        /* incident polar angle */
    double P_i;         /* incident azimuthal angle */
    short pRand;        /* flag: randomize P_i */
    nNodePtr mem;       /* member list built from bCards
                         * donated by atom members */
    double mass;        /* ion mass */
    pt r0;              /* initial COM (center of mass) position */
    short rExplic;      /* flag: ion atom positions input explicitly by user */
    pt v0;              /* initial COM (center of mass) velocity */
    short vExplic;      /* flag: ion velocity is input explicitly by user */
    double a1, a2, a3;  /* Euler orientation angles (for polyatomic ions) */
    short aRand;        /* flag: randomize angles */

    double imp_ef;      /* prescribed kinetic energy fraction at "impact";
```

```
                        * default is 0.75.  This is used for
                        * monoatomic ions and polyatomic ions which aren't
                        * energetic enough to dissociate.
                        * "Impact" is also defined as the point at which a
                        * polyatomic ion loses its first internal bond. */

    /* Dynamic variables */
    double k;           /* Current kinetic energy of all atoms in ion */
    double t;           /* Current (internal) potential energy */
    double tex;         /* Current potential energy via interaction with
                         * atoms external to the ion */
    pt rx;              /* COM position at impact time */
    pt vx;              /* COM velocity at impact time */
    pt r;               /* Current COM position */
    pt v;               /* Current COM velocity */
    pt Dr;              /* Total vector displacement from initial position */
    double impt;        /* impact time */
    int nBonds0;        /* initial number of intact bonds */
    int nBonds;         /* current number of intact bonds */
    short imp_;         /* impact flag:
                         * imp_ == 0: ion has not yet impacted
                         * imp_ == 1: ion has impacted */
    /* Explicit atomic positions and velocities that may be user-specified */
    pt r0ex[MAXATOMSPERION];
    pt v0ex[MAXATOMSPERION];

    /* output info */
    int uint;           /* time step interval of updates */
    int oint;           /* time step interval of outputs */
    short ofex;         /* flag: output file existence */
    char outfile[MAXCHAR];
    char outfile_0[MAXCHAR];
} ion_type;
```

Most of these members are explained in the comments. The first two bear a brief explanation. `spec` is the species index of the ion. This is just an integer from the following enumerated list:

```
enum {I_He, I_C, I_F, I_Ne, I_Si, I_Ar, I_Kr, I_Xe, /* atoms */
      I_CC, I_FF, I_SiSi,                           /* homodimers AA */
      I_CF, I_SiF, I_SiC,                           /* heterodimers AX */
      I_CF2, I_SiF2, I_SiC2, I_Si2C,                /* AX2 */
      I_CF3, I_SiF3, I_SiC3, I_Si3C,                /* AX3 */
      I_CF4, I_SiF4, I_SiC4, I_Si4C,                /* AX4 */
      NULL_ION};
```

The user specifies which ion is to bombard the target using the "-ion" keyword and selecting from among the possible ion strings, explained in Sec. 4.3.

The second member, `pool`, denotes whether or not the ion is entered as a single ion on the command line, or is to be selected randomly from a pool of ions given in an ion pool input data file. This feature is not explained in running the code because I never used it for any of the work done for my thesis. Briefly, if the user specifies the name of a file using the "-ipf" flag, then that signifies that the code is to randomly select an ion for each impact from the pool of ions specified in the pool data file. The pool data file is line-oriented, where each line contains the following information:

```
ion-name  wt-factor
```

where `ion-name` is the character string recognizable by the code as the name of a bombarding ion (see above), and `wt-factor` is the weighting factor for that ion in the pool. For example, if one wanted to bombard the surface with a combination of Ar and CF_3 ions with an Argon-to-CF_3 ratio of 9:1, the following ion pool data file would be used:

```
Ar  0.9
CF3 0.1
```

The ion pool data file is read in once at the very beginning of the code execution.

The most important routine in the `ion` module is the function `ion_newion()`, which introduces the ion into the simulation volume. To do this, it first "adds" the appropriate number of blank atoms to the configuration, and assigns their identities accordingly. To assign the positions of each atom, it uses a hard-coded database of internal molecular coordinates for each of the allowable ion choices. This database is called `IonInternalCoords` and is an array of `iic_type`, where

```
typedef struct sym_dvec
{
    sym_type sym;
    pt r;
} sym_pt_type;
typedef struct ion_int_coords
{
    sym_type base;
    sym_pt_type x[MAXATOMSPERION-1];
} iic_type;
```

A single `iic_type` instance contains internal coordinates for one molecule, given that one atom of the molecule (called the "base") resides at (0,0,0). These internal coordinates are copied onto those of the new atoms, and then the resulting molecule is rotated by three random rotations on the x, y, and z axes. Then the molecule is translated by a random amount in x and y. Then the molecule is translated in the $+z$ direction until the lowest atom of the molecule is beyond the cutoff of the nearest surface atom. This places the ion in it initial position and orientation. Finally, the velocities of each atom in the molecule are assigned such that the total kinetic energy of the molecule is the specified value and the ion is traveling in the specified direction. Now the integration of the impact trajectory can begin.

5.2.4 `clust`: Cluster detector and handler

The `clust` module implements the routines for identifying and handling clusters. A cluster is represented by a single instance of the `cNode` data type.

```
typedef struct cluster_list_node *cNodePtr;
typedef struct cluster_list_node
{
    int id;
    short state;
    nNodePtr mem;
    cNodePtr next;
} cNode;
```

Here, `id` is a unique identification number for the cluster (i.e., it's index in an array), `state` is one of six enumerated constants that determine the state of the cluster (described below), `mem` heads a linked list of `nNode`'s that correspond to the members of the cluster, and `next` points to the next `cNode` in the linked list of `cNode`'s that makes up the list of clusters.

A cluster's state is determined using various information about the cluster. The state can be one of the following enumerated shorts:

```
enum {CS_BASE, CS_ION, CS_OK, CS_TRASH, CS_UNUSED, CS_NULLSTATE};
```

`CS_BASE` denotes the "base" cluster, which is the cluster corresponding to the surface. The base cluster by definition must contain the static layer of atoms. `CS_ION` denotes a cluster that is the bombarding ion. `CS_OK` denotes a cluster that is detected as unbound from the surface. `CS_TRASH` is an OK cluster that has been determined to have low enough binding energy with any surface atoms, and can therefore be considered as a sputtered cluster. `CS_UNUSED` denotes an unused `cNode` instance.

Initially, the `clust` module declares a fixed array of blank `cNode`'s. Cluster detection is done at each time step as a part of the force routine execution. Four functions accomplish cluster detection. The first is `clust_clear()` which re-initializes all clusters.

The next is `clust_enclust()` whose arguments are two `atomPtr`'s which point to two atoms that the force routine has designated as "bound" to one another. This function is called in the force routine each time it encounters a unique pair of atoms which are close enough to be defined as "bound" to one another. Note that two "neighbors" are not necessarily "bound" to one another. The function `clust_enclust()` assures that both atoms belong to the same cluster. The pair of atoms, A and B, evaluates to one of five possibilities:

1. Neither A nor B belongs to a cluster.
2. A belongs to a cluster, but B does not.
3. B belongs to a cluster, but A does not.
4. A and B already belong to the same cluster.
5. A and B belong to different clusters.

Each of these possibilities results in a distinct action by `clust_enclust()`:

1. Create a new cluster and put A and B into it.
2. Put B into A's cluster.
3. Put A into B's cluster.
4. Do nothing.
5. Merge A's cluster and B's cluster into a single cluster.

The act of putting an atom **A** into a cluster **c** requires two steps.

1. Pop **A**'s `bCard` stack and put the resulting `nNode` on the **c**'s `mem` stack.
2. Let the `c` member of **A** point to **c**.

The act of merging two clusters, **c** and **d**, into a single cluster **c** requires two steps as well.

1. While there are members on **d**'s `mem` stack, pop an `nNode` from the stack, set the `c` member of the atom addressed by the `addr` member of the `nNode` from **d** to **c**, and then push that `nNode` onto the `mem` stack of **c**.
2. Clear the now empty cluster **d**.

Once the force routine has visited every unique pair of atoms in the configuration, the two final cluster detection functions are called. One is `clust_sweepsingles()`, which has the important job of detecting any atoms that do not belong to clusters. These are atoms that the force routine determined are not bound to any other atoms, and therefore should each form a single-atom cluster. `clust_sweepsingles()` therefore creates a new cluster for each single unbound atom encountered in the configuration. The final function is `clust_fixlist()`, which "cleans up" the array of `cNodes` to produce a single linked list that, when traversed, visits only those `cNode`'s that correspond to actual clusters. This is required because, as clusters are detected and merged, a lot of once-used clusters become empty.

So, we see that at the end of each time step, we have a fully-determined list of all the clusters in the configuration. At the end of the integration, the final list of clusters is examined to find clusters that satisfy the specified desorption criteria. Any such clusters are designated as "Trash", and output as sputtered clusters by the `clust_clean()` function, which is called after the last time step executes by the `cfg_report()` function.

5.3 Drivers

5.3.1 `domd`: Driver for performing a single MD integration

The following code is the `domd` driver.

```
void do_md (int runId)
{
    /* Insert the run number into any "????" or "####" strings in the
     * file name templates. */
    filenames_apply();

    /* Establish the configuration data:
     * Read-in from input file, maybe introduce ion. */
    cfg_establish();

    /* Perform the integration */
    cfg_push();

    /* Output the configuration data, including info on clusters */
    cfg_report();

    /* Clear the configuration and cluster information for the next run */
    cfg_clear();
    clust_clear();
}
```

Here, `runId` is a unique integer assigned to this MD run. This is referred to as the "impact number" if this run is one of a sequence of impact simulations.

5.3.2 `main`: Driver for stringing MD integrations together

The following code is the `main` driver for stringin single MD integrations together into a sequence of integrations where the output configuration of one integration is the input configuration for the following integration.

```
int main (int argc, char * argv[])
{
    /* Handle all command line arguments */
    arg_handler(argc, argv, TRAP_BADWORDS);

    /* Say hello */
    if (!quiet_)
        printf("# Welcome to md series 2 v %.2lf b %i (c) 1999 cfa\n",
            version_, build_);

     /* Perform initializations */
    Chem_InitializePeriodicTable();
    pef_initialize();
    cfg_initialize();
    filenames_initialize();
    if (!quiet_) {cfg_showsizes();filenames_echo();}
```

```
    sdt=adt=0;

    /* Begin the trajectory loop. */
    for ((*i)=(*i0);(*i)<=(*i1);(*i)++)
    {
        if (!quiet_)
        {
            printf("#\n# Trajectory #%i (%i out of %i) begins here\n",
                    *i, (*i)-(*i0)+1, (*i1)-(*i0)+1);
            fflush(stdout);
        }
        t0=time(NULL);

        /* Perform the i'th trajectory */
        do_md(*i);

        /* Output some run-time/results info */
        time_info();

        /* Update the integration limits */
        tlim1_=rt_+(tlim1_-tlim0_);
        tlim0_=rt_;
    }

    /* Say goodbye */
    if (!quiet_) printf("# Thank you for using md series 2.\n");
    return 0;
}
```

5.4 The Tersoff/Brenner Si–C–F force routine module

In this section, I present the main coding algorithms used in the Tersoff/Brenner Si–C–F force routine module. In this discussion, I use the notation of the potential presented in the paper of Abrams and Graves. [7] The Si–C–F force routine module in `md2` consists of the following files.

1. `units.h`: Some useful unit conversion declarations;
2. `sicf_params.h`: Potential parameters as defined macros;
3. `genforce.h` and `tbt_sicf.h`: Headers included by other modules;
4. `tbt_sicf.c`: Main force routine;
5. `bicubic.h` and `bicubic.c`: Bicubic interpolation routines (Written by Junichi Tanaka);

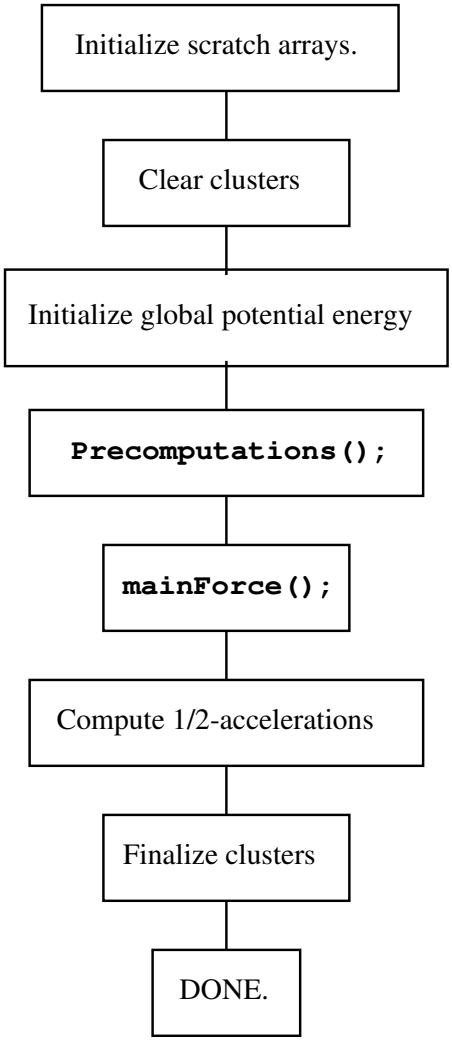

Figure 5: Flowsheet for the `cfg_setforce()` function in the `tbt_sicf.c` force routine module.

6. `tricubic.h` and `tricubic.c`: Tricubic interpolation routines (Written by Junichi Tanaka).

The only one of these that I will discuss in this documentation is the main force routine, `tbt_sicf.c`, as it implements the Si–C–F potential and its gradient in calculating forces on interacting atoms.

The flagship function of `tbt_sicf.c` is `cfg_setforce()`, which, as you may remember, is called by the velocity-Verlet algorithm function, `cfg_push()`. Fig. 5 depicts the flowsheet for this function.

The primary functions of `Precomputations()` is to compute interatomic separations and assign neighbors. This function traverses the configuration list `L_`, and for each atom in `L_`, it visits all *other* atoms whose ID number is greater than the current atom. In this way all unique pairs of atoms in the system are considered. For each pair of atoms, the minimum image separation vector and (squared) distance are computed. If this (squared) distance is

less than the (squared) cutoff distance for this *type* of pair, then the two atoms are made neighbors, as discussed previously. Other two-body data is computed and stored in the scratch arrays at this point to be used later by `mainForce()`.

The function `mainForce()` implements the bulk of the potential and force computation for each atom. It traverses the configuration list `L_`, and foreach atom **A** in `L_`, it visits each element of that atom's neighbor list *for which* the index of the atom **B** in the neighbor list of **A** is *greater* than that of **A**. In this way, `mainForce()` visits each unique pair of *neighbors*. For each pair of neighbors **A**,**B**, `mainForce()` performs the following tasks:

1. Computes the bond order $\overline{b_{AB}}$ and the bond energy ϕ_{AB};
2. Computes the force on atom **A** due to the bond **AB**, and the force on atom **B** due to the bond **AB**;
3. For each neighbor **C** of atom **A**, computes the force on **C** due to the bond **AB**;
4. For each neighbor **D** of atom **B**, computes the force on **D** due to the bond **AB**.

The total potential energy is calculated within `mainForce()` by incrementing the global variable `PE_` by each ϕ_{AB}.

Below I present the algorithm used by `mainForce()`.

for each atom i
  for each atom j neighbor of i, $j > i$
    Retrieve two-body data computed in `Precomputations()`:
      $\mathbf{r}_{ij}$, r_{ij}, f_{ij}, f'_{ij}, $N_{ij}^{(x)}$, $N_{ij}^{(t)}$, $N_{ji}^{(t)}$, V_A, V_R, V'_A, V'_R
    Compute H_{ij}, $H_{ij}^{(1)}$, $H_{ij}^{(2)}$ and H_{ji}, $H_{ji}^{(1)}$, $H_{ji}^{(2)}$
    for each atom k neighbor of i, $k \neq j$
      tally ζ_{ij} and $N_{ij}^{(conj)}$
      Compute and store $\cos\theta_{ijk}$, g_{ijk}, g'_{ijk}, ω_{ijk}, ω'_{ijk}, etc.
    Compute b_{ij} and b'_{ij}
    for each atom k neighbor of j, $k \neq i$
      tally ζ_{ji} and $N_{ji}^{(conj)}$
      Compute and store $\cos\theta_{jik}$, g_{jik}, g'_{jik}, ω_{jik}, ω'_{jik}, etc.
    Compute b_{ji} and b'_{ji}
    Compute $\overline{b_{ij}}$, ϕ_{ij}
    Compute **two-body forces**: $\mathbf{f}_{i\leftarrow j}^{(2)} = -\mathbf{f}_{j\leftarrow i}^{(2)}$
    $\mathbf{f}_i += \mathbf{f}_{i\leftarrow j}^{(2)}$ and $\mathbf{f}_j += \mathbf{f}_{j\leftarrow i}^{(2)}$
    for each atom k neighbor of i, $k \neq j$
      fetch stored data
      Compute $\mathbf{f}_{k\leftarrow ij}$
      $\mathbf{f}_k += \mathbf{f}_{k\leftarrow ij}$ and $\mathbf{f}_i -= \mathbf{f}_{k\leftarrow ij}$
    for each atom k neighbor of j, $k \neq i$
      fetch stored data
      Compute $\mathbf{f}_{k\leftarrow ji}$
      $\mathbf{f}_k += \mathbf{f}_{k\leftarrow ji}$ and $\mathbf{f}_j -= \mathbf{f}_{k\leftarrow ji}$

This ends my presentation of the algorithms used in my Si–C–F force routine. More detailed information is embedded in the code itself in the form of program documentation. The analytical gradients used in the code are published as Appendix A in my thesis.

6 Concluding remarks

With this document, and the accompanying release of my simulation code, I have sought a convenient and effective way to disseminate the work of my PhD thesis to the general scientific community. In earning my PhD, and in the process being priveleged enough to publish several papers, I have received all of the compensation for writing this code that I expected.

References

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language, 2nd Edition* (Prentice Hall, Englewood Cliffs, New Jersey, 1988).

[2] B. P. Feuston and S. H. Garofalini, J. Chem. Phys. **89**, 5818 (1988).

[3] K. Gärtner, D. Stock, B. Weber, G. Betz, M. Hautala, G. Hobler, M. Hou, S. Sarite, W. Eckstein, J. J. Jiménez-Rodríguez, A. M. C. Pérez-Martín, E. P. Andribet, V. Konoplev, A. Gras-Marti, M. Posselt, M. H. Shapiro, T. A. Tombrello, H. M. Urbassek, H. Hensel, Y. Yamamura, and W. Takeuchi, Nucl. Instr. Meth. Phys. Res. B **102**, 183 (1995).

[4] D. W. Brenner, Phys. Rev. B **42**, 9458 (1990).

[5] D. W. Brenner, Phys. Rev. B **46**, 1948 (1992).

[6] J. Tanaka, C. F. Abrams, and D. B. Graves, J. Vac. Sci. Technol. A **18**, (2000 (in press)).

[7] C. F. Abrams and D. B. Graves, J. Appl. Phys. **86**, 5938 (1999).

[8] K. Beardmore and R. Smith, Phil. Mag. A **74**, 1439 (1996).

[9] A. Oram and S. Talbot, *Managing Projects with Make* (O'Reilly, Cambridge, Massachussetts, 1991).

[10] E. A. Merritt and D. J. Bacon, Methods in Enzymology **277**, 505 (1997).

[11] V. Serikov, private communication, 1999.

[12] H. R. Lewis and L. Denenberg, *Data Structures & Their Algorithms* (Harper Collins, New York, NY, 1991).

[13] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Oxford University Press, New York, 1987).

[14] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson, J. Chem. Phys. **76**, 637 (1982).