Moldy User's Manual

Release Revision: 2.11

Keith Refson
Department of Earth Sciences
Parks Road
Oxford OX1 3PR
keith@earth.ox.ac.uk

November 12, 1996

Acknowledgements

The initial implementation of the Nosé-Hoover and Gaussian thermostats was by Vladimir Murashov at Dalhousie University. Rafael R. Pappalardo contributed the modifications to "mdshak" to write XYZ format molecular graphics files.

Thanks are due to Craig Fisher and José Manuel Martinez Fernandez for diligent proofreading of several versions of the manual and many helpful suggestions on how to improve it. Any mistakes that remain are my own.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Moldy is a computer program for performing molecular dynamics simulations of condensed matter. It can handle any assembly of rigid polyatomic molecules, atoms or ions and any mixture thereof. It uses the 'link cell' method to calculate short-range forces and the Ewald sum technique to handle long-range electrostatic forces. Simulations may be performed either in the usual NVE ensemble or in NVT, $N\sigma H$ or $N\sigma T$ ensembles using Nosé-Hoover thermostat and Parrinello and Rahman constant-stress methods. As the MD cell need not be cubic, the program is equally suitable for simulations of solids and liquids.

Most existing MD programs are limited in their capabilities, for example to one kind of potential function, or molecular symmetry, or to some restricted number of molecules. *Moldy* is (as far as possible) free from such arbitrary constraints. The system is specified at the beginning of each run and its size is only limited by the amount of memory available to the program: if a system is too large to handle, the solution is to buy some more memory. The system may contain a mixture of an arbitrary number of molecular species, each with an arbitrary number of atoms and an arbitrary number of molecules of each. Molecules or ions may be monatomic or polyatomic, linear or three dimensional in any combination. The potential functions may be of the Lennard-Jones, Buckingham (including Born-Mayer) or MCY types, and other potential types may be easily added. Such flexibility is possible because *Moldy* is written in the 'C' language which permits dynamic memory allocation.

Moldy is written to be highly portable and has been tested on a wide range of computers and operating systems, including VAX/VMS, MS-DOS and Unix[1] (both BSD and system V varieties). It should be straightforward to move it to any other machine with a good 'C' compiler.

To be of real use a simulation program must run efficiently on modern high-speed computers, which are increasingly of vector or parallel architectures. *Moldy* is written so as to be highly vectorizable and has been tested on a range of vector machines from manufacturers including Cray, Convex, Stardent and Alliant. On the cray XMP-48 its performance can exceed 100 MFlop/sec (on a suitably large system). *Moldy* is also able to run on a parallel computer of either shared or distributed memory architectures, and has been tested on multiprocessors from Stardent, Convex, Cray Research, SGI and IBM SP1 and Cray Research T3D massively parallel machines.

1.1 Terms and Conditions

Permission to compile and run *Moldy* is granted to any individual or organization without restriction. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Moldy is free software; you may redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version. The terms of the GNU General Public License are described in the file COPYING which is included with the source distribution, or from the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.

[1] Goodness knows who will own the Unix registered trademark by the time you read this.

Chapter 2

Algorithms and Equations

This chapter describes the implementation of the molecular dynamics technique as used in *Moldy*. It is not intended as an introduction to MD simulations, and does assume some familiarity with the basic concepts of microscopic models and simulations thereof. The book by Allen and Tildesley[2] is a very good introductory text, covering both the theory and the practicalities and is highly recommended. It also contains comprehensive references to the scientific literature of microscopic computer simulation.

Moldy is designed to simulate a common class of models of atomic or molecular systems. The assumptions are: that the system is an assembly of *rigid molecules*, atoms or ions; that the forces of interaction are derived from *continuous potential functions* acting between (usually atomic) *sites* on each molecule; that the dynamics are governed by the *classical* Newton-Euler equations of motion. A major aim of *Moldy* has been to allow the most general of models within that class and to impose as few restrictions as possible. In particular arbitrary mixtures of different molecules are allowed and several popular forms of potential functions are catered for.

2.1 The Equations of Motion

If we denote the force exerted by atom α of molecule i on atom β of molecule j by $\mathbf{f}_{i\alpha j\beta}$[1] then the total force acting on molecule i is

$$\mathbf{F}_i = \sum_j \sum_\beta \sum_\alpha \mathbf{f}_{i\alpha j\beta} \tag{2.1}$$

and the torque is given by

$$\mathbf{N}_i = \sum_\alpha (\mathbf{r}_{i\alpha} - \mathbf{R}_i) \times \mathbf{f}_{i\alpha} \tag{2.2}$$

where $\mathbf{R}_i = 1/M_i \sum_\alpha m_{i\alpha}\mathbf{r}_{i\alpha}$ is the centre of mass of molecule i.

The motion is governed by the Newton-Euler equations

$$M_i\ddot{\mathbf{R}}_i = \mathbf{F}_i \tag{2.3}$$

$$\mathbf{I}_i \cdot \dot{\omega}_i - \omega_i \times \mathbf{I}_i \cdot \omega_i = \mathbf{N}_i \tag{2.4}$$

where ω_i is the angular velocity of the molecule, $\mathbf{I}_i = \sum_\alpha m_{i\alpha}(p_{i\alpha}^2\mathbf{1} - \mathbf{p}_{i\alpha}\mathbf{p}_{i\alpha})$ is the inertia tensor and $\mathbf{p}_{i\alpha} = \mathbf{r}_{i\alpha} - \mathbf{R}_i$ is the atomic site co-ordinate relative to the molecular centre of mass.

The orientations of the molecules are represented by *quaternions* as has now become common practice. They are preferred over Euler angles for two reasons. Firstly they lead to equations of motion which are free of singularities[10] which means that no special-case code is required. This leads to much improved numerical stability of the simulation[11]. Secondly, molecular symmetry operations and combinations of rotations are elegantly expressed in terms of a simple quaternion algebra[11, 38].

[1] A comment on notation is appropriate here. In this chapter, site quantities are denoted by *lowercase* letters, molecular quantities by *uppercase*, sites are indexed by *greek* letters and molecules by *roman*. A missing index denotes a sum over the corresponding sites or molecules so that, for example $\mathbf{r}_{i\alpha j\beta}$ is a site-site vector and $\mathbf{F}_{ij}$ the molecule-molecule force.

A quaternion is an ordered number quartet which obeys the algebra given by Pawley[36]. The multiplication rule in that reference may be restated as a matrix product treating each quaternion as a 4-vector. If $\mathbf{p} \equiv (p_0, p_1, p_2, p_3)$ and $\mathbf{q} \equiv (q_0, q_1, q_2, q_3)$ are quaternions then

$$\mathbf{pq} = \begin{pmatrix} p_0 & -p_1 & -p_2 & -p_3 \\ p_1 & p_0 & -p_3 & p_2 \\ p_2 & p_3 & p_0 & -p_1 \\ p_3 & -p_2 & p_1 & p_0 \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{pmatrix} \tag{2.5}$$

The quaternion $\tilde{\mathbf{q}}$ conjugate to $\mathbf{q}$ is defined as $\tilde{\mathbf{q}} = (q_0, -q_1, -q_2, -q_3)$ so that

$$\mathbf{q}\tilde{\mathbf{q}} = (q_0^2 + q_1^2 + q_2^2 + q_3^2, 0, 0, 0). \tag{2.6}$$

The *norm* is defined as $|\mathbf{q}| \equiv \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$ and $\mathbf{q}$ is called a *unit* quaternion if $|\mathbf{q}| = 1$. Any possible rotation can be represented by a unit quaternion. Du Val shows[50] that if $\mathbf{q} = (\cos\frac{\alpha}{2}, \mathbf{l}\sin\frac{\alpha}{2})$ (where we have combined the last three components to form a 3-vector) and $\mathbf{p} = (0, \mathbf{r})$ then the operation

$$\mathbf{p}' \equiv (0, \mathbf{r}') = \mathbf{qp}\tilde{\mathbf{q}} \tag{2.7}$$

corresponds to a rotation of the vector $\mathbf{r}$ by an angle of α about the axis $\mathbf{l}$. The components may also be expressed in terms of the Euler angles as[2]

$$\begin{aligned} q_0 &= \cos\frac{\phi+\psi}{2}\cos\frac{\theta}{2} \\ q_1 &= \sin\frac{\phi-\psi}{2}\sin\frac{\theta}{2} \\ q_2 &= \cos\frac{\phi-\psi}{2}\sin\frac{\theta}{2} \\ q_3 &= \sin\frac{\phi+\psi}{2}\cos\frac{\theta}{2}. \end{aligned} \tag{2.8}$$

The relationship between the time derivative of a quaternion and the principal frame angular velocity was given by Evans[10, Equation 27] and rewritten using quaternion algebra by Refson[44] as

$$2\dot{\mathbf{q}} = \mathbf{q}(0, \omega^p) \tag{2.9}$$

The second derivative is given by

$$\begin{aligned} 2\ddot{\mathbf{q}} &= \mathbf{q}(-1/2(\omega^p)^2, \dot{\omega}^p) \\ &= \mathbf{q}(-2|\dot{\mathbf{q}}|^2, \dot{\omega}^p) \end{aligned} \tag{2.10}$$

Equations 2.10 and 2.4 allow the simulation to be implemented using quaternions and their derivatives as the dynamic variables for rotational motion, and this is the method employed in *Moldy*. This second order formulation was first used by Powles *et al.*[39] and Sonnenschein showed[49] that it gives substantially better stability than if angular velocities and accelerations are used as dynamic variables.

Using equations 2.10 to describe the dynamics means that they are integrated as if all four components were independent. Therefore the normalization condition $\mathbf{q}\tilde{\mathbf{q}} = \mathbf{1}$ may not be exactly satisfied after performing an integration step. *Moldy* adopts the usual practice of scaling all components of the quaternion after each timestep to satisfy the normalization condition[11].

It is less widely realized that the second order equations (2.10) introduce a *second* unconstrained variable into the procedure. Differentiating equation 2.6 gives a constraint on the quaternion derivatives

$$q_0\dot{q}_0 + q_1\dot{q}_1 + q_2\dot{q}_2 + q_3\dot{q}_3 = 0 \tag{2.11}$$

[2] The definition of quaternions used here differs from that used in Evans' paper[10, equation 21] in the sign of q_2 or ξ. This error has been compounded by subsequent authors[49, 48, 28] who also managed to permute the components which means that the parameters do not form an ordered number quartet which obeys quaternion algebra. Like Allen and Tildesley[2, page 88] we follow the definition of Goldstein[16, pages 143 and 155].

which is just the q_0 component of equation 2.9. Just as with the normalization condition, the integration algorithm will not preserve this condition exactly unless explicit measures are taken. After each timestep the constraint may be re-established by subtracting the discrepancy from the quaternion derivatives. If $\delta = q_0\dot{q}_0 + q_1\dot{q}_1 + q_2\dot{q}_2 + q_3\dot{q}_3$ then the corrected quaternion derivatives are given by

$$\dot{\mathbf{q}}' = \dot{\mathbf{q}} - \delta\mathbf{q}. \tag{2.12}$$

Experiments conducted while developing *Moldy* show that enforcing this constraint significantly decreases the fluctuations in the total energy.

Linear molecules are a slightly special case as the moment of inertia about the molecular axis is zero. Though there are unique methods to represent this situation[2, page 90] *Moldy* uses a minor modification of the quaternion algorithm. All that is necessary is a little special-case code to avoid dividing by the zero component of inertia in the solution of equation 2.4 and to hold the components of angular velocity and acceleration about the molecular axis to zero. This has the considerable advantage of uniform treatment of all kinds of molecules which is convenient when dealing with heterogeneous mixtures.

2.2 Integration Algorithms

The dynamical equations 2.3 and 2.4 are integrated using this author's modification[43] of the Beeman algorithm [3]. For atomic systems the accuracy is of the same order as the commonly used Verlet algorithm[51].

The accuracy of common integration algorithms was discussed by Rodger[45] who showed that the Beeman algorithm is the most accurate of the supposedly "Verlet-equivalent" algorithms. The Beeman algorithm is the only one accurate to $O(\delta t^4)$ in the co-ordinates and $O(\delta t^3)$ in the velocities compared to the velocity Verlet algorithm which is only $O(\delta t)$ in the velocities. This is insufficient for an accurate determination of the kinetic energy, pressure and other dynamic quantities. More seriously, it fails badly in the case of polyatomic molecules and for constant-pressure and constant-temperature methods where the (generalized) velocities enter the dynamical equations themselves.

Velocity-dependent forces occur in equations 2.4 and 2.10, in the Parrinello-Rahman constant-pressure equations (section 2.7) and in the Nosé-Hoover heat bath algorithms (section 2.6.2). These usually present a problem to non "predictor-corrector" algorithms which are based on the assumption that the forces depend only on the co-ordinates. Fincham has devised a scheme to allow integration of the rotational equations using Verlet-like algorithms[12], which is widely used despite the potential problems caused by the low accuracy of the velocities being propagated into the dynamics.

These cases are easily and accurately handled by the modification to Beeman's equations proposed by the author[43]. These may be summarized using the symbol x to represent any dynamic variable (centre-of-mass co-ordinate, quaternion or MD cell edge), $\dot{x}^{(p)}$ and $\dot{x}^{(c)}$ to represent "predicted" and "corrected" velocities respectively.

$$\left.\begin{array}{lrcl}
\text{i} & x(t+\delta t) & = & x(t) + \delta t\,\dot{x}(t) + \frac{\delta t^2}{6}\left[4\ddot{x}(t) - \ddot{x}(t-\delta t)\right] \\
\text{ii} & \dot{x}^{(p)}(t+\delta t) & = & \dot{x}(t) + \frac{\delta t}{2}\left[3\ddot{x}(t) - \ddot{x}(t-\delta t)\right] \\
\text{iii} & \ddot{x}(t+\delta t) & = & F(\{x_i(t+\delta t), \dot{x}_i^{(p)}(t+\delta t)\}, i = 1\ldots n) \\
\text{iv} & \dot{x}^{(c)}(t+\delta t) & = & \dot{x}(t) + \frac{\delta t}{6}\left[2\ddot{x}(t+\delta t) + 5\ddot{x}(t) - \ddot{x}(t-\delta t)\right] \\
\text{v} & \multicolumn{3}{l}{\text{Replace } \dot{x}^{(p)} \text{ with } \dot{x}^{(c)} \text{ and goto } iii\text{. Iterate to convergence}}
\end{array}\right\} \tag{2.13}$$

The predictor-corrector cycle of steps *iii* to *v* are iterated until the predicted and corrected velocities have converged to a relative precision of better than 1 part in 10^{-7}, which in practice takes 2 or 3 cycles. This iteration is not as inefficient as it might at first appear as it does *not* include the expensive part of the calculation — the recalculation of the site forces. Only the angular accelerations and quaternion second derivatives must be evaluated at each iteration using equations 2.4 and 2.10, and this operation is relatively cheap.

2.3 Potentials and Short-Range Forces

The forces determining the dynamics of the system are derived from the potential function denoted by $\phi_{i\alpha j\beta}(\mathbf{r}_{i\alpha j\beta})$. The indices i and j run over all molecules in the system and α and β over sites on the respective molecule. The total potential energy of the system is

$$U = \sum_i \sum_{j>i} \sum_\alpha \sum_\beta \phi_{i\alpha j\beta}(\mathbf{r}_{i\alpha j\beta}). \tag{2.14}$$

where $\mathbf{f}_{i\alpha j\beta} = -\nabla\phi_{i\alpha j\beta}(\mathbf{r}_{i\alpha j\beta})$ is the force acting on site β of molecule j from site α of molecule i.

The total force and torque acting on any particular molecule are calculated using equations 2.1 and 2.2. Since $\mathbf{f}_{i\alpha j\beta}$ and therefore $\mathbf{F}_{ij}$ are short-ranged forces (*i.e.*they decay faster than r^{-3}) one can define a *cutoff* radius, r_c. Interactions between sites whose separation r_{ij} is greater than r_c are assumed to be negligible and need not be evaluated. In the case of a polyatomic molecular system it is usual to apply the cutoff according to the *intermolecular* separation R_{ij}.

2.3.1 Pressure and Stress

The internal stress of a system of interacting molecules is given by Allen and Tildesley[2, pp 46-49] in terms of the molecular virial, but we may rewrite it more conveniently in terms of the atomic site virial as

$$\begin{aligned} V\pi^{sr} &= \left\langle \sum_i M_i \mathbf{V}_i \mathbf{V}_i + \sum_i \sum_{j>i} \mathbf{R}_{ij} \mathbf{F}_{ij} \right\rangle \qquad (2.15) \\ &= \left\langle \sum_i M_i \mathbf{V}_i \mathbf{V}_i + \sum_i \sum_{j>i} \sum_\alpha \sum_\beta \mathbf{r}_{i\alpha j\beta} \mathbf{f}_{i\alpha j\beta} - \sum_i \sum_\alpha \mathbf{p}_{i\alpha} \mathbf{f}_{i\alpha} \right\rangle \end{aligned}$$

The quantity $\mathbf{p}_{i\alpha} \equiv \mathbf{r}_{i\alpha} - \mathbf{R}_i$ is the co-ordinate of each site relative to the molecular centre-of-mass. The pressure is easily evaluated as one third of the trace of the stress tensor.

2.3.2 Long Range Corrections

The truncation of the interactions at the cut-off radius does introduce some errors into the calculated potential energy and stress. By neglecting density fluctuations on a scale longer than the cutoff radius we may approximate the errors and calculate correction terms[2, pp 64-65].

$$\begin{aligned} U_c &= \frac{2\pi}{V} \sum_\alpha \sum_\beta N_\alpha N_\beta \int_{r_c}^{\infty} r^2 \phi_{\alpha\beta}(r)\,\mathrm{d}r \qquad (2.16) \\ P_c V &= \frac{2\pi}{3V} \sum_\alpha \sum_\beta N_\alpha N_\beta \int_{r_c}^{\infty} r^3 \frac{\mathrm{d}\phi_{\alpha\beta}(r)}{\mathrm{d}r}\,\mathrm{d}r \\ &= U_c + \frac{2\pi}{3V} \sum_\alpha \sum_\beta N_\alpha N_\beta r_c^3 \phi_{\alpha\beta}(r_c) \qquad (2.17) \\ \pi_c &= P_c \delta \qquad (2.18) \end{aligned}$$

where the sums run over all the distinct kinds of sites, N_α is the total number of sites of type α in the system and δ is the unit matrix.

2.3.3 Potential Functions

Moldy supports most common forms of potential function. In particular,

Lennard-Jones $\phi(r) = \varepsilon((\frac{\sigma}{r})^{12}) - (\frac{\sigma}{r})^6)$

6-exp $\phi(r) = -\frac{A}{r^6} + B\exp(-Cr)$
This is the most general 6-exp form and includes potential of the *Born-Mayer* and *Buckingham* forms.

MCY $\phi(r) = A\exp(-Br) - C\exp(-Dr)$

generic $\phi(r) = A\exp(-Br) + C/r^{12} - D/r^4 - E/r^6 - F/r^8$
This is a composite which contains terms of several inverse powers which may be useful for ionic solution simulations.

In addition to the short-range potential electric charges may be specified for each atomic site which interact according to Coulomb's Law. See the following section for details.

2.4 The Ewald Sum

The long range Coulomb interactions are handled using the Ewald Sum technique in three dimensions[4, 2, p. 156]. The electrostatic potential of a system of charges is expressed as a sum of short-range and long-range contributions. Each term is written as a series, the first in real space and the second, obtained by Fourier transformation using the periodicity of the MD cell, in reciprocal space. The expression for the Coulomb energy U is

$$
\begin{aligned}
U \quad = \quad & \underbrace{\frac{1}{4\pi\varepsilon_0}\sum_{\mathbf{n}}{}^{\dagger}\sum_{i=1}^{N}\sum_{j=i+1}^{N} q_i q_j \frac{\operatorname{erfc}(\alpha|\mathbf{r}_{ij}+\mathbf{n}|)}{|\mathbf{r}_{ij}+\mathbf{n}|}}_{\text{Real-space term}} \\
+ \quad & \underbrace{\frac{1}{\varepsilon_0 V}\sum_{\mathbf{k}>0}\frac{1}{k^2}e^{-\frac{k^2}{4\alpha^2}}\left\{\left|\sum_{i=1}^{N} q_i\cos(\mathbf{k}\cdot\mathbf{r}_i)\right|^2 + \left|\sum_{i=1}^{N} q_i\sin(\mathbf{k}\cdot\mathbf{r}_i)\right|^2\right\}}_{\text{Reciprocal-space term}} \\
- \quad & \underbrace{\frac{\alpha}{4\pi^{\frac{3}{2}}\varepsilon_0}\sum_{i=1}^{N} q_i^2}_{\text{Point self-energy}} - \underbrace{\frac{1}{4\pi\varepsilon_0}\sum_{n=1}^{M}\sum_{\kappa=1}^{N_m}\sum_{\lambda=\kappa+1}^{N_m} q_{n\kappa}q_{n\lambda}\frac{\operatorname{erf}(\alpha|\mathbf{r}_{\kappa\lambda}|)}{|\mathbf{r}_{\kappa\lambda}|}}_{\text{Intra-molecular self energy}} \\
- \quad & \underbrace{\frac{1}{8\varepsilon_0 V\alpha^2}\left|\sum_{i=1}^{N} q_i\right|^2}_{\text{Charged system term}} + \underbrace{\left[\frac{1}{6\varepsilon_0 V}\left|\sum_{i=1}^{N} q_i\mathbf{r}_i\right|^2\right]}_{\text{Surface dipole term}}
\end{aligned}
\tag{2.19}
$$

where the "daggered" summation indicates omission of site pairs i, j belonging to the same molecule if $\mathbf{n} = \mathbf{0}$. The meaning of the symbols is

$\mathbf{n}$	lattice vector of periodic array of MD cell images
$\mathbf{k}$	reciprocal lattice vector of periodic array of MD cell images
k	modulus of $\mathbf{k}$
i, j	absolute indices of all charged sites
n	index of molecules
κ, λ	indices of sites within a single molecule
N	total number of charged sites
M	total number of molecules
N_m	number of sites on molecule m
$\mathbf{p}_i$	co-ord of site i relative to molecular centre-of-mass, $\mathbf{r}_i - \mathbf{R}_i$
q_i	charge on absolute site i
$q_{m\kappa}$	charge on site κ of molecule m
$\mathbf{r}_i$	Cartesian co-ordinate of site i
$\mathbf{r}_{ij}$	$\mathbf{r}_j - \mathbf{r}_i$
α	real/reciprocal space partition parameter
π_{lm}	instantaneous stress tensor
δ_{lm}	Kronecker delta symbol
l, m	*xyz* tensor indices
V	volume of MD cell

and the force on charge i is given by

$$\begin{aligned}
\mathbf{f}_i &= -\nabla_{\mathbf{r}_i} U \\
&= \underbrace{\frac{q_i}{4\pi\varepsilon_0} \sum_{\mathbf{n}}^{\dagger} \sum_{\substack{j=1 \\ j\neq i}}^{N} q_j \left\{ \frac{\operatorname{erfc}(\alpha|\mathbf{r}_{ij}+\mathbf{n}|)}{|\mathbf{r}_{ij}+\mathbf{n}|} + \frac{2\alpha}{\sqrt{\pi}} e^{-\alpha^2|\mathbf{r}_{ij}+\mathbf{n}|^2} \right\} \frac{\mathbf{r}_{ij}+\mathbf{n}}{|\mathbf{r}_{ij}+\mathbf{n}|^2}}_{\text{Real-space term}} \\
&+ \underbrace{\frac{2}{\varepsilon_0 V} \sum_{\mathbf{k}>0} q_i \frac{\mathbf{k}}{k^2} e^{-\frac{k^2}{4\alpha^2}} \left\{ \sin(\mathbf{k}\cdot\mathbf{r}_i) \sum_{j=1}^{N} q_j \cos(\mathbf{k}\cdot\mathbf{r}_j) - \cos(\mathbf{k}\cdot\mathbf{r}_i) \sum_{j=1}^{N} q_j \sin(\mathbf{k}\cdot\mathbf{r}_j) \right\}}_{\text{Reciprocal-space term}} \\
&+ \underbrace{\left[\frac{q_i}{6\varepsilon_0 V} \left(\sum_{j=1}^{N} q_j \mathbf{r}_j \right) \right]}_{\text{Surface dipole term}}
\end{aligned} \tag{2.20}$$

The molecular forces and torques $\mathbf{F}$ and $\mathbf{N}$ are evaluated from the site forces $\mathbf{f}_i$ using equations 2.1 and 2.2.

Notice that the equation 2.19 for the energy contains a correction for the intra-molecular self-energy, whose derivative is absent from the equation for the forces (equation 2.20). This term corrects for interactions between charges on the *same* molecule which are implicitly included in the reciprocal space sum, but are not required in the rigid-molecule model. Though the site forces $\mathbf{f}_i$ do therefore include unwanted terms these sum to zero in the evaluation of the molecular centre-of-mass forces and torques (equations 2.1 and 2.2) (by the conservation laws for linear and angular momentum).

2.4.1 Parameter Values

Both the real- and reciprocal-space series (the sums over $\mathbf{n}$ and $\mathbf{k}$) converge fairly rapidly so that only a few terms need be evaluated. We define the *cut-off* distances r_c and k_c so that only terms with $|\mathbf{r}_{ij}+\mathbf{n}| < r_c$ and $|\mathbf{k}| < k_c$ are included. The parameter α determines how rapidly the terms decrease and the values of r_c and k_c needed to achieve a given accuracy.

For a fixed α and accuracy the number of terms in the real-space sum is proportional to the total number of sites, N but the cost of the reciprocal-space sum increases as N^2. An overall scaling of $N^{\frac{3}{2}}$ may be achieved if α varies with N. This is discussed in detail in an excellent article by David Fincham[13]. The optimal value of α is

$$\alpha = \sqrt{\pi} \left(\frac{t_R}{t_F} \frac{N}{V^2} \right)^{\frac{1}{6}} \tag{2.21}$$

where t_R and t_F are the execution times needed to evaluate a single term in the real- and reciprocal-space sums respectively. If we require that the sums converge to an accuracy of $\varepsilon = \exp(-p)$ the cutoffs are then given by

$$r_c = \frac{\sqrt{p}}{\alpha} \tag{2.22}$$

$$k_c = 2\alpha\sqrt{p} \tag{2.23}$$

A representative value of t_R/t_F specific to *Moldy* has been established as 5.5. Though this will vary on different processors and for different potentials its value is not critical since it enters the equations as a sixth root.

It must be emphasized that the r_c is used as a cutoff for the short-ranged potentials as well as for the electrostatic part. The value chosen above *does not* take the nature of the non-electrostatic part of the potential into account. It is therefore the responsibility of the user to ensure that r_c is adequate for this part too.

2.4.2 Uniform Sheet Correction

The 5th term in equation 2.19 is necessary only if the system has a non-zero net electric charge, and is useful in special cases such as framework systems.

In a periodic system the electrostatic energy is finite only if the total electric charge of the MD cell is zero. The reciprocal space sum in equation 2.19 for $\mathbf{k} = 0$ takes the form

$$\frac{1}{k^2} e^{-\frac{k^2}{4\alpha^2}} \left| \sum_{i=1}^{N} q_i \right|^2$$

which is zero in the case of electroneutrality but infinite otherwise. Its omission from the sum in equation 2.19 is physically equivalent to adding a uniform jelly of charge which exactly neutralizes the unbalanced point charges. But though the form of the reciprocal space sum is unaffected by the uniform charge jelly the real-space sum is not. The real-space part of the interaction of the jelly with each point charge as well as the self-energy of the jelly itself must be included giving the fifth term in equation 2.19.

2.4.3 Surface Dipole Term

The optional final term in equations 2.19 and 2.20 if used performs the calculations under different periodic boundary conditions. It was suggested by De Leeuw, Perram and Smith[29] in order to accurately model dipolar systems and is necessary in any calculation of a dielectric constant.

The distinction arises from considerations of how the imaginary set of infinite replicas is constructed from a single copy of the MD box[2, pp 156-159]. Consider a near-spherical cluster of MD cells. The "infinite" result for any property is the limit of its "cluster" value as the size of the cluster tends to infinity. However this value is non-unique and depends on the dielectric constant, ε_s of the physical medium surrounding the cluster. If this medium is conductive ($\varepsilon_s = \infty$) the dipole moment of the cluster is neutralized by image charges, whereas in a vacuum ($\varepsilon_s = 1$) it remains. It is trivial to show that in that case the dipole moment per unit volume (or per MD cell) does *not* decrease with the size of the cluster.

The final term in equation 2.19 is just the dipole energy, and ought to be used in any calculation of the dielectric constant of a dipolar molecular system. It is switched on by *Moldy*'s control parameter `surface-dipole`. Note that as it represents the dipole at the surface of the cluster the system is no longer truly periodic.

Conversely it *must not* be used if the simulated system contains mobile ions. Consider an ion crossing a periodic boundary and jumping from one side of the MD cell to another. In that case the dipole moment of the MD cell changes discontinuously. Because of the surface dipole term the calculation would model a discontinuous macroscopic change in the dipole moment of the whole system caused by an infinite number of ions jumping an infinite distance. This is manifested in practice by a large and discontinuous change in the energy of the system and on the force on each charge within it.

This situation is completely non-physical but is easily avoided. However the problem may also arise more subtly even when there are no mobile ions if a framework is being simulated (section 2.10). The framework is treated as a set of discrete, but fixed atoms rather than a molecular unit. If the shape of the unit cell is allowed to vary then ions constituting the framework may indeed cross MD cell boundaries causing the aforementioned problems.

2.4.4 Stress

The internal stress (and pressure) of an atomic system is given by the volume-derivative of the internal energy. The situation is slightly more complicated for rigid molecules since molecules do not scale with volume and only the inter-molecular distances vary. The resulting expression for the Coulombic part of the instantaneous stress π^e_{ik} is[34, Appendix A]

$$\begin{aligned} V\pi^e_{lm} &= \underbrace{\frac{1}{4\pi\varepsilon_0} \sum_{\mathbf{n}}{}^{\dagger} \sum_{i=1}^{N} \sum_{j=i+1}^{N} q_i q_j \left\{ \frac{\operatorname{erfc}(\alpha|\mathbf{r}_{ij}+\mathbf{n}|)}{|\mathbf{r}_{ij}+\mathbf{n}|} + \frac{2\alpha}{\sqrt{\pi}} e^{-\alpha^2|\mathbf{r}_{ij}+\mathbf{n}|^2} \right\} \frac{(\mathbf{r}_{ij}+\mathbf{n})_l(\mathbf{r}_{ij}+\mathbf{n})_m}{|\mathbf{r}_{ij}+\mathbf{n}|^2}}_{\text{Real-space term}} \\ &+ \underbrace{\frac{1}{\varepsilon_0 V} \sum_{\mathbf{k}>0} \frac{1}{k^2} e^{-\frac{k^2}{4\alpha^2}} \left(\delta_{lm} - 2\left[\frac{1}{k^2} + \frac{1}{4\alpha^2} \right] \mathbf{k}_l \mathbf{k}_m \right) \left\{ \left| \sum_{i=1}^{N} q_i \cos(\mathbf{k}\cdot\mathbf{r}_i) \right|^2 + \left| \sum_{i=1}^{N} q_i \sin(\mathbf{k}\cdot\mathbf{r}_i) \right|^2 \right\}}_{\text{Reciprocal-space term}} \end{aligned}$$

$$- \underbrace{\sum_{i=1}^{N} (\mathbf{F}_i)_l (\mathbf{p}_i)_m}_{\text{Molecular Virial Correction}} - \underbrace{\frac{\delta_{lm}}{8\varepsilon_0 V \alpha^2} \left| \sum_{i=1}^{N} q_i \right|^2}_{\text{Charged system term}} \tag{2.24}$$

The true internal stress is the ensemble average, $\pi_{lm} = \left\langle \pi^e_{lm} + \pi^{\text{s.r.}}_{lm} + \pi^K_{lm} \right\rangle$, where $\pi^{\text{s.r.}}_{lm}$ and π^K_{lm} are the short-range force and kinetic contributions respectively. π^e_{lm} enters into the Parrinello-Rahman equations of motion (see section 2.7).

The term marked *Molecular Virial Correction* in equation 2.24 is the difference between the site-site virial $\sum_i \mathbf{f}_i \cdot \mathbf{r}_i$ and the molecular virial $\sum_i \mathbf{F}_i \cdot \mathbf{R}_i$ and is subtracted after all of the site forces and the molecular forces have been calculated including the short-range potential components which are not included in the equations above. Though it is not apparent in reference[34, Appendix A] this term has exactly the same form for all parts of the stress — the short-range potential and the real- and reciprocal space Coulombic parts.

2.5 Periodic Boundaries — the Link Cell Method

The real space part of the Ewald (equation 2.19) and short-range potential energy is a sum of contributions over pairs of sites. In both cases the interaction decays rapidly with separation, which means that only site pairs closer then some *cutoff* distance r_c need be considered. Several methods are available to enumerate site pairs and choose those within the cutoff.

Most simple MD codes simply loop over all pairs of particles in the MD box and compute the separation r for each. If $r < r_c$ the interaction is computed. This method suffers from several disadvantages. Since for any given site, the interaction with any other site is considered only once, only the *nearest* periodic image of that site is included (the *minimum-image* convention). However this restricts the cutoff radius to less than half the MD cell dimension $r_c < 2L$. More serious is the way the computational time scales with the number of sites. If there are N sites, there are $O(N^2)$ separations to compute and the overall time therefore scales as $O(N^2)$.

The Verlet *Neighbour List* scheme[2, pp 146-149] makes use of the spatial locality of the interaction potential by maintaining a list for each site of all the "neighbouring" sites (*i.e.* all those within the cutoff distance). This can give considerable gains for moderate numbers of sites, but it ultimately requires $O(N^2)$ time (to build the lists) as well as $O(N)$ storage for the lists.

Moldy uses an implementation of the link cell method of Quentrec *et al.*[41] described in Allen and Tildesley's book[2, pp 149-152] which is a true $O(N)$ algorithm. The fundamental idea is that the MD cell is partitioned into a number of smaller cells, known as *subcells*. Every timestep a linked list of all the particles contained in each subcell is constructed. The selection of all pairs of *particles* within the cutoff is achieved by looping over all pairs of *subcells* within the cutoff and particles within the subcells. Because of their regular arrangement, the list of neighbouring subcells is fixed and may be precomputed. Its construction takes only $O(N)$ operations and only $O(N)$ pair interactions need be calculated.

When the system consists of polyatomic molecules it is important that all sites belonging to a particular molecule are assigned to the same cell. Otherwise it is impossible to calculate the stress and pressure (equation 2.24) correctly as MD cell vectors are added to some intra-molecular distances. Therefore all sites of any molecule are assigned to a cell if the molecular centre-of-mass lies inside it.

One drawback of the link cell method has been the difficulty of implementing it efficiently for vector processors. The linked list of "neighbour" particles is not stored in the regular-stride array which is required for vectorization. Heyes and Smith[19] pointed out that *gather* operations might be used to assemble a temporary array of neighbour particle co-ordinates from which the interaction potential and forces could be evaluated in vector mode. A *scatter* operation is then used to add the resulting forces to the total force array. This is the technique used in *Moldy*. Almost all modern vector machines have scatter/gather hardware which means these operations are fairly cheap.

2.5.1 No minimum-image convention

One notable feature of the implementation of the link cell method in *Moldy* is that it does *not* follow the *minimum image* convention used in most MD codes. Instead the list of neighbouring cells, and hence interactions considered, includes *all* periodic images of a particle which are within the cutoff. This means that it is quite safe to use a cutoff of more than half of the MD cell side in any direction.

2.5.2 Cell or Strict cutoff

There are two options for the way in which site-site interactions are selected for inclusion in the total energy and forces. These are *cell-based* cutoff and *strict* cutoff and are selected by the `strict-cutoff` control parameter.

In cell-based mode (`strict-cutoff=0`) the neighbour cell list is built to include only those cells whose centre is within the cutoff radius of the centre of the reference cell. All interactions between sites belonging to molecules in the neighbouring cell list are computed. This is a “quick and dirty” method as some interactions between sites closer than the cutoff will inevitably be excluded whereas some outside the cutoff range will be included.

In strict mode (`strict-cutoff=1`) all interactions between pairs of *sites* within the cutoff are included. The neighbouring cell list contains all pairs of cells with any parts closer than the cutoff plus twice the greatest molecular radius. This ensures that all appropriate interactions are included. Furthermore, all interactions between sites further apart than the cutoff are excluded (by the expedient of setting their separation to a large value in the potential calculation). This means that large and asymmetric molecules are handled correctly.

For a given cutoff radius the cell-based mode is rather quicker than the strict mode since the neighbouring cell list is much smaller and fewer interactions are computed. However to ensure that significant interactions are not omitted, the cutoff ought to be set to a greater value than strictly required. This tends to offset the potential speed gain. On the other hand, if strict isotropy is required in a liquid simulation for example then the strict cutoff option ought to be used.

2.6 Temperature Initialization and Control

From the equipartition theorem, every degree of freedom in the system, f has the same kinetic energy, given by $\langle \mathcal{K} \rangle_f = \frac{1}{2} k_B T$. The effective temperature of the system is therefore given by the ensemble average of its kinetic energy.

$$T = \langle \mathcal{T} \rangle = \frac{2}{g k_B} \left\langle \sum_{f=1}^{g} \mathcal{K}_f \right\rangle = \frac{1}{3 N k_B} \left\langle \sum_{i=1}^{N} m_i \mathbf{v}_i^2 + \omega_i \cdot \mathbf{I} \cdot \omega_i \right\rangle \tag{2.25}$$

Here $\mathcal{K}_f$ is the instantaneous kinetic energy of degree of freedom f, g is the number of degrees of freedom, N is the number of molecules, $\mathcal{T}$ is an instantaneous “temperature”.

It is almost always desirable that a simulation be conducted so that the temperature is the supplied parameter rather than the kinetic energy. This requires some mechanism to fix the *average* kinetic energy at thermal equilibrium. The *initial* kinetic energy may be set approximately by choosing random velocities which sample the Maxwell-Boltzmann distribution at the desired temperature, and this is indeed what *Moldy* does on starting a new run (see section 2.9.2). But because the initial configuration is usually far from equilibrium it will have too much potential energy. As the run progresses this will be converted into kinetic energy, raising the temperature above the desired value. It is therefore necessary to have some mechanism for removing excess kinetic energy as the run progresses.

Moldy offers several mechanisms to control the temperature. The common technique of *velocity scaling* is suitable for use during the equilibration period but does not generate meaningful particle trajectories. The Nosé-Hoover method couples the system to a heat bath using a fictional dynamical variable and the Gaussian thermostat replaces the Newton-Euler equations by variants of which the kinetic energy is a conserved quantity.

2.6.1 Rescaling

At periodic intervals linear and angular velocities are multiplied by a factor of

$$s = \sqrt{\frac{g k_B T}{2 \mathcal{K}}} \tag{2.26}$$

where T is the desired temperature. By repeatedly setting the “instantaneous” temperature to the correct value while the system approaches its equilibrium state, the kinetic energy is made to approach its desired value. *Scaling* may be performed every timestep, or every few depending on the simulation conditions.

An MD run with scaling does not generate a valid statistical ensemble, and it must therefore be switched off before any calculation of thermodynamic averages is performed.

Moldy incorporates two refinements to the basic scaling algorithm (which are selected by the control parameter `scale-options`). Linear and angular velocities can be scaled independently, either for the whole system or for each species individually. In this way, one does not rely on the interactions of these degrees of freedom for convergence to equilibrium. In many systems these degrees of freedom are loosely coupled and the exchange of energy between them is slow. In these cases individual scaling can speed up the approach to equilibrium considerably.

The other refinement addresses the problem of setting the temperature accurately. At equilibrium the system's kinetic energy fluctuates with mean-square amplitude[3] $\left\langle \delta\mathcal{K}^2 \right\rangle = \frac{1}{2} g \left(k_B T\right)^2$, which corresponds to a rms fluctuation in the instantaneous "temperature" $\sqrt{\langle \delta\mathcal{T}^2 \rangle} = \sqrt{2/g}T$. The difficulty with applying equation 2.26 is the instantaneous kinetic energy $\mathcal{K}$ in the denominator. Strictly, scaling ought to use the *average* kinetic energy $\langle \mathcal{K} \rangle$ as in equation 2.25, but this quantity is not known until after the run is completed. Because of this equation 2.26 can only set the temperature to an accuracy of $\sqrt{1/g}$. This is often inadequate for purposes of comparison with experiment.

In order to allow the temperature to be set with greater accuracy, *Moldy* allows the use of a partial average in the denominator,

$$s = \sqrt{\frac{g k_B T}{2 \langle \mathcal{K} \rangle'}} \tag{2.27}$$

where $\langle \mathcal{K} \rangle'$ is the "rolling" average of $\mathcal{K}$ over some number of preceding timesteps. That number is determined by the control parameter `roll-interval`.

This option should be used cautiously. The change in $\langle \mathcal{K} \rangle$ upon scaling only has a significant effect on the average after many timesteps. If the subsequent scaling is performed before this change is reflected in the value of $\langle \mathcal{K} \rangle'$ it will use an out-of-date value of the average kinetic energy. It is therefore recommended that the number of timesteps between scalings be greater than or equal to the number used to compute the rolling average. Otherwise it is possible to produce wild overshoots and oscillations in the temperature.

Finally, there is a method for tackling really difficult cases when even individual scaling is unable to keep the temperature under control. This might be a far-from-equilibrium configuration where the potentials are so strong that velocities rapidly become very large, or when a single molecule acquires a very large velocity. In that case the velocities can all be re-initialized randomly from the Maxwell-Boltzmann distribution periodically. This provides a kind of pseudo Monte-Carlo equilibration for difficult cases.

2.6.2 Nosé-Hoover Thermostat

A more sophisticated approach than rescaling is to couple the system to a heat bath. Nosé[32] proposed an extended-system Hamiltonian to represent the degrees of freedom of the thermal reservoir: *Moldy* uses the simpler but equivalent formulation by Hoover[20, 2]. The equations of motion (equations 2.3 and 2.4) are modified thus

$$\begin{aligned} \ddot{\mathbf{R}}_i &= \frac{\mathbf{F}_i}{M_i} - \zeta \dot{\mathbf{R}}_i \\ \mathbf{I}_i \cdot \dot{\omega}_i - \omega_i \times \mathbf{I}_i \cdot \omega_i &= \mathbf{N}_i - \zeta \mathbf{I}_i \cdot \omega_i \\ \dot{\zeta} &= \frac{g}{Q}\left(k_B \mathcal{T} - k_B T\right) \end{aligned} \tag{2.28}$$

where most of the symbols have their previous meanings, g is the number of degrees of freedom in the system. ζ is a new "heat bath" dynamic variable and Q is the associated fictitious "mass" parameter. With a suitable choice of Q these equations generate trajectories which sample the canonical ensemble[32]. In other words both averages and fluctuations calculated as time averages from a simulation run tend to their canonical ensemble limits.[4] This does not necessarily guarantee the correctness of *dynamical* quantities.

It is apparent from equations 2.28 that ζ has the dimensions of a time derivative, and is analogous to the unit cell velocities in equations 2.30. It is therefore treated as if it was a velocity and updated using only steps *ii–v* of the Beeman integration procedure (equations 2.2). Note that the equation for the centre-of-mass acceleration

[3] This formula actually applies to the Canonical rather than the microcanonical ensemble, but it serves for the purpose of this argument.

[4] There has been some discussion in the literature of the validity of representing a heat bath by a single dynamical variable, which the more diligent reader may wish to explore[7, 33].

depends *indirectly* on the velocities through $\mathcal{T}$ as well as explicitly. The iteration on steps *iii–v* of equations 2.2 is therefore essential to integrate these equations consistently.

There are two criteria to be considered when choosing the value of Q. The coupling to the heat bath introduces non-physical oscillations of period $t_0 = 2\pi\sqrt{Q/2gk_BT}$ which may be easily detected in the total energy[33]. Firstly it must be arranged that there are sufficiently many oscillations during a simulation run that the computed thermodynamic values represent averages over many periods t_0. This ensures that the configurations used to calculate the averages sample the whole of the phase space generated by the fluctuations in ζ to represent the canonical ensemble. Secondly Q should be chosen so that t_0 is large compared to the characteristic decay time of dynamical correlation functions. This is to ensure that the fictitious dynamics of the heat bath are decoupled from the real molecular dynamics, and is particularly important in a simulation of dynamic properties[6] such as velocity correlation functions. Since the first criteria favours a small Q and the second a large one, it may be necessary to increase the total simulation time in order to satisfy them both.

2.6.3 Gaussian Thermostat

An alternative approach is known as "constrained dynamics" whereby the equations of motion are modified to generate trajectories which exactly satisfy $\mathcal{T} = T$ at all times. One such is the Gaussian constraint (see Allen and Tildesley[2] pp 230-231). The equations of motion are

$$
\begin{aligned}
\ddot{\mathbf{R}}_i &= \frac{\mathbf{F}_i}{M_i} - \zeta_T \dot{\mathbf{R}}_i \\
\zeta_T &= \frac{\sum_j \mathbf{v}_j \cdot \mathbf{F}_j}{\sum_j M_j \mathbf{v}_j^2} \\
\mathbf{I}_i \cdot \dot{\omega}_i - \omega_i \times \mathbf{I}_i \cdot \omega_i &= \mathbf{N}_i - \zeta_R \mathbf{I}_i \cdot \omega_i \\
\zeta_R &= \frac{\sum_j \omega_j \cdot \mathbf{N}_j}{\sum_j \omega_j \cdot \mathbf{I}_j \cdot \omega_j}
\end{aligned} \tag{2.29}
$$

These bear a superficial resemblance to equations 2.28 but now ζ is not itself a dynamical variable but a function of the other dynamical variables. Note that T does not enter explicitly into equations 2.29 since $\mathcal{T}$ is now a conserved quantity equal to its initial value at all subsequent times. Perhaps surprisingly these equations of motion generate configurational *averages* from the canonical ensemble, but this does *not* hold true for the momenta and therefore dynamics nor for fluctuations.

2.7 Constant Stress

It is frequently desirable to conduct simulations under conditions of constant pressure or stress, rather than constant volume. For example, this allows the simulation of a solid-state phase transition with a change of symmetry or unit cell size. *Moldy* incorporates the constant pressure method of Parrinello and Rahman[35].

In a constant-stress simulation the MD cell changes in size and shape in response to the imbalance between the internal and externally applied pressure. For an exposition of the method the reader should refer to Parrinello and Rahman's paper[35] and to Nosé and Klein's extension to rigid molecular systems[34]. The equation of motion for the reduced centre-of-mass co-ordinates $\mathbf{S}_i = \mathbf{h}^{-1}\mathbf{R}_i$ is

$$
M_i \ddot{\mathbf{S}}_i = \mathbf{h}^{-1}\mathbf{F}_i - M_i \mathbf{G}^{-1}\dot{\mathbf{G}}\dot{\mathbf{S}}_i \tag{2.30}
$$

replacing the straightforward Newton equation 2.3. $\mathbf{h}$ denotes the 3×3 MD cell matrix whose columns are the MD cell vectors $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$, $\mathbf{F}_i$ is the centre-of-mass force and $\mathbf{G} = \mathbf{h}'\mathbf{h}$. Equation 2.4, the Euler equation, governs the angular motion exactly as in the constant-volume case.

The dynamics of the unit cell matrix $\mathbf{h}$ are governed by the equation

$$
W\ddot{\mathbf{h}} = (\pi - p)\,\sigma \tag{2.31}
$$

where W is a fictitious mass parameter, $\sigma = V\mathbf{h}'^{-1}$ and p is the external pressure. The instantaneous internal stress π is given by

$$\pi = \frac{1}{V}\sum_{i=1}^{N} m_i(\mathbf{h}_i\dot{\mathbf{s}}_i)^2 + \pi^{\mathrm{s.r.}} + \pi^e \tag{2.32}$$

with the short-ranged and electrostatic components given by equations 2.15 and 2.24 respectively.

The choice of fictitious mass, W is governed by similar considerations to those concerning the heat bath mass, Q discussed in section 2.6.2.

Nosé and Klein[34] describe and address the problem of the whole MD cell rotating during the course of the simulation. This is because angular momentum is not conserved in a periodic system, and because the **h** matrix has 9 degrees of freedom, three more than needed to specify the position and orientation of the MD cell. Their solution is to constrain the **h** matrix to be symmetric, and involves a modification of the Parrinello-Rahman equations.

Moldy incorporates a rather different constraint which is not only simpler to implement (as it does not require modification of the equations of motion) but which also has a more obvious physical interpretation. The lower three sub-diagonal elements of the **h** matrix are constrained to zero. In other words the MD cell vector **a** is constrained to lie along the x-axis and **b** is constrained to lie in the xy-plane. Physically this may be thought of as implementing an MD box lying on a horizontal surface under the influence of a weak gravitational field. The implementation is trivial; at each timestep the acceleration of those components, $\ddot{\mathbf{h}}_{ij}$, is set to zero which is equivalent to adding a fictitious opposing force.

This constraint technique is not restricted merely to eliminating redundant degrees of freedom, but can also be used for other purposes. For example it may be used to allow uniaxial expansion only. The most important use of **h** matrix constraints however is probably for simulations of liquids. Since a liquid can not support shear stress there is no restoring force to keep the simulation cell nearly cubic, and it will therefore drift to a parallelepiped shape. To counter this tendency **h** may be constrained so that only the diagonal elements are non-zero and allowed to change. This does not give a completely isotropic MD cell expansion, but the time average should tend towards a cubic cell.

MD cell constraints are selected using the control parameter `strain-mask` (see section 3.8). A value of 238 will freeze the off-diagonal components of **h**.

2.8 Radial Distribution Functions

The *radial distribution function* or RDF is one of the most important structural quantities characterizing a system, particularly for liquids. For a one-component system, the RDF is defined as[17, p445]

$$\begin{aligned} g(r) &= \frac{1}{\rho^2}\left\langle \sum_i \sum_{j\neq i} \delta(\mathbf{r}+\mathbf{r}_i-\mathbf{r}_j)\right\rangle \\ &\approx V\left\langle \delta(\mathbf{r}+\mathbf{r}_1-\mathbf{r}_2)\right\rangle \end{aligned} \tag{2.33}$$

where we use the angle brackets to denote a spherical average as well as the usual configurational average. Allen and Tildesley[2, pp184,185] describe how to evaluate $g(r)$ from a histogram of pair distances accumulated during a simulation run. With the notation that N is the total number of particles, b is the number of the histogram bins, δr is the bin width (so that $r = b\delta r$), $n_{\mathrm{his}}(b)$ is the accumulated number per bin, τ is the number of steps when binning was carried out

$$g(r+\delta r/2) = \frac{3n_{\mathrm{his}}(b)}{4\pi\rho N\tau[(r+\delta r)^3 - r^3]} \tag{2.34}$$

In the case of a molecular system, the partial RDF for atoms α and β is defined as[17, p 445]

$$g_{\alpha\beta}(r) = \frac{1}{\rho^2 V}\left\langle N(N-1)\delta(\mathbf{r}+\mathbf{r}_{1\alpha}-\mathbf{r}_{2\beta})\right\rangle \tag{2.35}$$

which may be rewritten more usefully for an arbitrary multi-component mixture by eliminating the molecular density ρ and number N as

$$g_{\alpha\beta}(r) = V\left\langle \delta(\mathbf{r}+\mathbf{r}_{1\alpha}-\mathbf{r}_{2\beta})\right\rangle \tag{2.36}$$

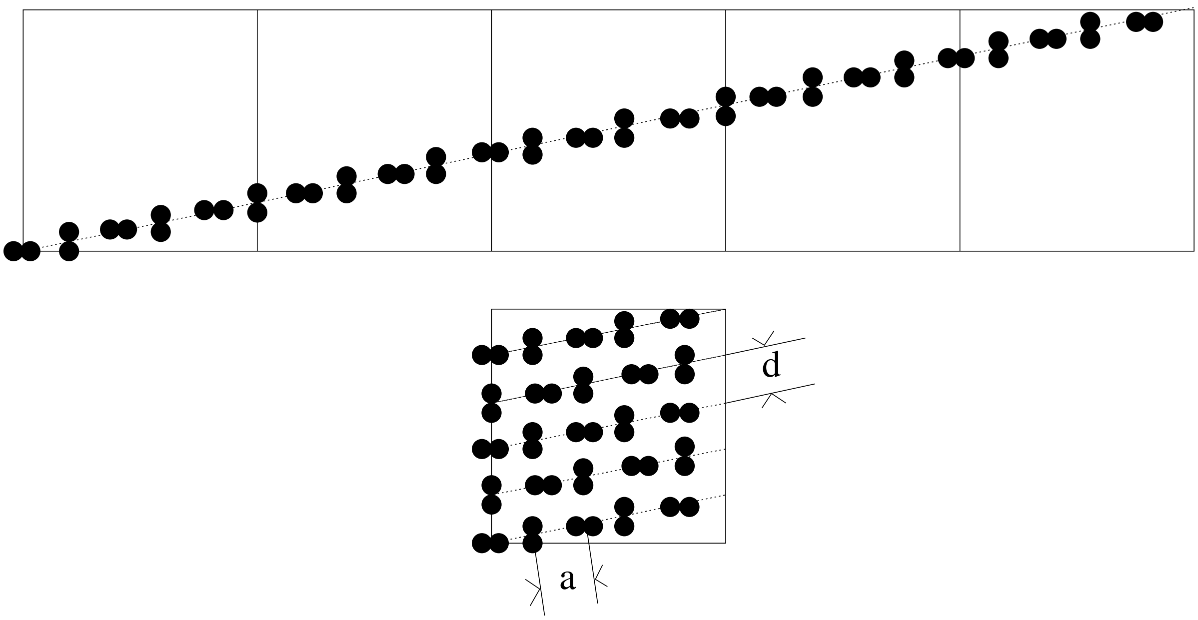


Figure 2.1: The Skew Start method. *N* molecules are placed at evenly-spaced intervals, *a* on a line joining a corner of the MD cell to its image *k* cells away (5 in this illustration). When mapped back into the original cell this guarantees a minimum separation of $min(d,a)$.

In the simulation this is evaluated by an expression very similar to equation 2.34

$$g_{\alpha\beta}(r+\delta r/2) = \frac{3Nn_{\mathrm{his}}(b)}{4\pi\rho N_\alpha N_\beta \tau[(r+\delta r)^3 - r^3]} \tag{2.37}$$

2.9 The Initial Configuration

One of the more trying aspects of initiating a molecular dynamics simulation is getting it started in the first place. It is not hard to see the reason why. An MD integration algorithm will only generate a good approximation to the correct equations of motion if the forces and velocities of the particles are less than some value. The timestep is chosen so that this criterion is satisfied for near-equilibrium configurations. But if the configuration is far from equilibrium certain forces may be extremely large (due to atoms approaching each other too closely). And worse, the breakdown of the integrator leads to breakdown of the conservation laws and the system evolves to a state even further from equilibrium.

One way around this difficulty is to start the system from a configuration known to be near-equilibrium, such as a known crystal structure. For a solid-state simulation this is the method of choice, and *Moldy* allows any initial structure to be specified and replicated to fill the MD cell. In the case of a liquid, a crystalline initial state is less desirable, and indeed, none may be known. Furthermore such a starting configuration restricts the allowed number of molecules to a multiple of the number in the unit cell and worse, may force the use of a non-cubic MD cell.

Moldy incorporates a novel method of generating an initial configuration which, in the main, overcomes these problems.

2.9.1 The Skew Start Method

The essence of the *Skew Start* method is to generate a configuration which, though not periodic in 3 dimensions, nevertheless is sufficiently regular to guarantee a minimum separation between molecular centres of mass. Figure 2.1 demonstrates the principle in 2 dimensions.

The *N* molecules are placed at some suitable interval *a* on a line drawn between one corner of the MD cell (of side *L*) and one of its periodic images. Clearly the index (h,k) of the image cell corner should be chosen so that the molecule spacing, a is close to the spacing of the line's images, d. For simplicity, choose $k=1$ which leads to the condition:

$$\begin{aligned} a = \tfrac{L\sqrt{h^2+1}}{N} &\approx \tfrac{L}{\sqrt{h^2+1}} = d \\ \Rightarrow h &\approx \sqrt{N-1} \end{aligned} \tag{2.38}$$

Therefore the optimum value of h is the nearest integer to $\sqrt{N-1}$.

The formalism may be extended to three dimensions, and yields the results for the molecular and "inter-line" spacings a, d_y and d_z respectively

$$\begin{aligned} a &= \frac{L}{N}\sqrt{h^2+k^2+l^2} \\ d_y &= L\frac{\sqrt{k^2+l^2}}{\sqrt{h^2+k^2+l^2}} \\ d_z &\approx L\frac{l}{k} \qquad \text{(assuming } h/k \text{ is an integer)} \end{aligned} \tag{2.39}$$

The "equal spacing" requirements are satisfied approximately by

$$\begin{aligned} h &\approx N^{2/3} \\ k &\approx N^{1/3} \\ l &= 1 \end{aligned} \tag{2.40}$$

which when substituted into equation 2.39 yields

$$a \approx d_y \approx d_z \approx LN^{-1/3} \tag{2.41}$$

Using this method an arbitrary number of molecules may be packed into a cubic cell with a guaranteed minimum centre-of-mass separation of approximately $LN^{-1/3}$. In contrast to other methods, such as random placement with excluded volume, it will always yield a configuration no matter how high the density. It is also very simple to implement.

It still remains to determine the initial orientations of the molecules in the case of polyatomics. In the current implementation these are assigned randomly, which works well for small or near-spherical molecules. Further refinements which may help avoid overlaps for elongated molecules are possible, such as a periodic sequence of orientations along the line, but no investigation of this possibility has yet been carried out.

In practice the method has proved to work well for water and aqueous systems and always yields a configuration which may be evolved towards equilibrium by the MD equations of motion. Any feedback on its performance in more difficult cases will be welcome.

2.9.2 Initial Velocities

It remains to choose the initial velocities of the molecules to complete the specification of the initial configuration. The recipe is the same irrespective of whether the molecules are started on a lattice or from a skew start. The initial centre-of-mass velocities are chosen from the Maxwell-Boltzmann distribution at the temperature specified for the simulation[2, pp 170]. That is, the velocities are chosen from a set of random numbers with a Gaussian distribution and normalized so that the probability density $p(v)$ of the xyz component of the velocity v_{ik} of molecule k is

$$p(v_{ik}) = \left(\frac{m_k}{2\pi k_B T}\right)^{1/2} \exp\left(-\frac{m_k v_{ik}^2}{2k_B T}\right) \tag{2.42}$$

This is easily accomplished given a random number generator which samples from a Gaussian distribution with unit variance. Given a random number R_{ik}, each component of velocity is set to

$$v_{ik} = \sqrt{\frac{k_B T}{m_k}} R_{ik} \tag{2.43}$$

Each component of the angular velocity (expressed in the principal frame) has a probability distribution

$$p(\omega^p_{ik}) = \left(\frac{I_{ik}}{2\pi k_B T}\right)^{1/2} \exp(-\frac{I_{ik}(\omega^p_{ik})^2}{2k_B T}) \tag{2.44}$$

which is ensured by assigning a random velocity

$$\omega^p_{ik} = \sqrt{\frac{k_B T}{I_{ik}}} R_{ik} \tag{2.45}$$

Since the dynamical variables used by *Moldy* for the angular co-ordinates are in fact the quaternions and their derivatives, we must set the quaternion derivatives and accelerations to the corresponding values. Using equations 2.9 and 2.10 we have

$$\dot{\mathbf{q}} = \mathbf{q}(0, \omega^p/2) \tag{2.46}$$

and

$$\ddot{\mathbf{q}} = -\frac{1}{4}(\omega^p)^2 \mathbf{q} \tag{2.47}$$

Finally, we note that if a molecule has less than three degrees of freedom, that is $I_{ik} = 0$ for some i the corresponding angular velocities *etc.* are simply set to zero.

2.10 Frameworks

In addition to the bulk properties of solids and liquids, much attention has been devoted in recent years to using MD simulation to model atoms or molecules interacting with surfaces or other structures such as zeolites. The distinctive feature of such a situation is that the 2- or 3-dimensional surface or structure is larger than the interacting molecules by many orders of magnitude. In fact the idealization of this system makes the structure infinite in extent. An atomistic model of this kind of a system necessitates choosing the periodic boundary conditions of the MD cell to be commensurate with the periodicity of the surface or structure. This manual will refer to an infinitely repeating crystalline surface or structure as a *framework*.

There are two possible formulations of a system of this kind for use in a MD simulation. Firstly the framework may be modelled as a set of independent atoms interacting via pair potentials. This merely requires specifying the correct initial structure and MD cell plus a suitable set of pair potentials. The latter model both the internal forces which determine the crystal structure of the framework and its interaction with the atoms or molecules of the fluid. Conceptually there is no distinction between this situation and a regular solid or liquid system, and the mechanics of initiating a simulation are handled in exactly the usual manner.

However there are situations in which this “atomic” approach is impractical. Because the system being modelled is essentially “two-phase” the atoms of the framework find themselves under the influences of two distinct kinds of force. There are the strong forces, usually covalent or ionic, which bind the atoms to form the framework itself and the weaker, non-bonded forces of the interaction with the fluid. Ideally these could all be modelled by a single consistent set of interatomic potentials which are sufficiently transferable to yield an accurate crystal structure for the framework as well as the molecular structure of the fluid and its interaction with the surface. Regrettably such potentials are hard to find.

Furthermore the characteristic vibrational frequencies of the solid framework will probably be much higher than those of the fluid. Consequently the timestep must be chosen sufficiently small to accurately model the crystalline vibrations. This will usually be far smaller than the value required for the fluid, necessitating very lengthy MD runs to model both regimes properly. This is, of course, exactly the argument used to justify rigid-molecule models.

Moldy implements a variation on the rigid-molecule model to simulate a rigid framework structure periodically extended throughout space. There are a few subtleties which must be correctly handled to achieve a consistent implementation, which are described hereafter.

2.10.1 Implementation

The framework is in many respects exactly like an ordinary molecule. It should be defined to exactly fill the MD cell so that the periodic repeat of the cell generates the periodicity of its crystal structure. The distinctive features of a framework molecule are:

- The framework is fixed in space and is not allowed to rotate. Any rotation would of course destroy the 3D structure. For most purposes it is convenient to regard the framework as being at rest, hence translational motion is forbidden also.
- No interactions between sites on a framework molecule and on itself or any of its periodic images are evaluated. That is, framework-framework interactions, both potentials and forces, are systematically excluded from the real-space and reciprocal-space parts of the Ewald sum, including the point and intra-molecular self terms of equation 2.19. (The exact modifications to equations 2.19,2.20 *etc.* are left as an exercise for the reader.)
- In the real-space force calculation, sites are treated as being independent atoms rather than belonging to a molecule. In particular the cutoff is applied on the basis of the (framework) site to (fluid) molecule distance. By virtue of the "all-image" convention, all the requisite molecule-framework interactions are correctly included. When assigning sites to sub-cells, each site is therefore placed in the sub-cell which contains its co-ordinate. (By contrast sites belonging to an ordinary molecule are placed in the cell which contains the molecular centre of mass.)

With these modifications, *Moldy* is able to successfully model a fluid in contact with a 3D rigid framework. 2-dimensional systems such as a fluid at a surface or between two surfaces may be represented as a 3D system by adding an artificial periodicity in the third dimension. To reduce the errors so introduced, the framework can be made "large with space inside" to fill a MD cell with a large c-axis.

In the case of a 3D framework it is clearly not sensible to allow the MD cell to change shape or size, since this would destroy the internal structure of the framework. However if the framework represents a 2D layer or surface, then the layer spacing may be allowed to vary using the constant-stress equations (section 2.7) and applying constraints to allow only the c-axis to fluctuate. In that case, bear in mind that the dynamics are governed by the Parrinello-Rahman equations of motion for the cell, rather than the Newtonian equations for the layer. In particular the mass is given by the parameter W rather than the mass of the framework molecule. (These may, of course be set equal if required.) Note also that no layer-layer interactions are calculated, and any force is the result of the externally applied pressure.[5]

Finally there are two subtle complications which arise from the framework concept.

2.10.2 Stress and Pressure Undefined

There is no unique definition of the internal stress or pressure of a framework system. The pressure of a system in a space with periodic boundary conditions is defined in terms of the molecular virial

$$\mathcal{W} = \frac{1}{3}\sum_{i=1}^{N}\sum_{j\neq i}^{N}\mathbf{R}_{ij}\cdot\mathbf{F}_{ij} \tag{2.48}$$

But the framework has no centre-of-mass, and so the quantity $\mathbf{R}_{ij}$ can not be defined. The site-virial formulation of equation 2.15 is of no assistance as the definition of the "internal" co-ordinate $\mathbf{p}_{i\alpha}$ involves the centre of mass co-ordinate $\mathbf{R}_i$. Neither can one simply choose a convenient reference $\mathbf{R}_i$. Since the force exerted by the fluid acting on the framework is in general non-zero, the term

$$\sum_i\sum_\alpha \mathbf{p}_{i\alpha}\mathbf{f}_{i\alpha} = \sum_i\sum_\alpha \mathbf{r}_{i\alpha}\mathbf{f}_{i\alpha} - \sum_i \mathbf{R}_i\mathbf{F}_i \tag{2.49}$$

clearly depends on $\mathbf{R}_i$. The situation may also be viewed physically. The definition of pressure of a system is the derivative of the free energy with respect to volume. But with an infinite rigid framework the volume derivative can not be defined.

[5]This is a restriction of the current implementation and may be lifted in future versions.

The useful quantity in this case is the partial pressure of the fluid. Though not currently implemented, this may be rectified in a future version of *Moldy*.

Finally we note that in the case of a 2D layer structure, which is *not* rigid in the third dimension the perpendicular component if the stress tensor *does* have a physical meaning and is correctly calculated.

2.10.3 Charged Frameworks

A minor complication arises when using a framework which has a non-zero net electric charge. Although the system as a whole may be electrically neutral, the omission of framework-framework interactions from the calculation also means that the $\mathbf{k} = 0$ term does not vanish. To see this examine equation 2.19. In the non-framework case the indices i and j in the terms

$$\left|\sum_{i=1}^{N} q_i \cos(\mathbf{k}\cdot\mathbf{r}_i)\right|^2 = \sum_{i=1}^{N}\sum_{j=1}^{N} q_i q_j \cos(\mathbf{k}\cdot\mathbf{r}_i)\cos(\mathbf{k}\cdot\mathbf{r}_j)$$

run over all site pairs. If $\mathbf{k} = 0$ the squared sum is

$$\left|\sum_{i=1}^{N} q_i\right|^2 = 0$$

If a framework is present the formulation becomes

$$\left|\sum_{i=1}^{N} q_i \cos(\mathbf{k}\cdot\mathbf{r}_i)\right|^2 - \left|\sum_{i=M+1}^{N} q_i \cos(\mathbf{k}\cdot\mathbf{r}_i)\right|^2$$

assuming sites $M+1\ldots N$ are the framework sites. On setting $\mathbf{k} = 0$ this reduces to

$$\left|\sum_{i=1}^{N} q_i\right|^2 - \left|\sum_{i=M+1}^{N} q_i\right|^2$$

It is therefore necessary to modify the charged-system term of equation 2.19 to

$$U_z = -\frac{1}{8\varepsilon_0 V \alpha^2}\left\{\left|\sum_{i=1}^{N} q_i\right|^2 - \left|\sum_{i=M+1}^{N} q_i\right|^2\right\} \tag{2.50}$$

Chapter 3

Running Moldy

The way *Moldy* is invoked depends to some extent on the operating system, but usually by issuing the command `moldy`.[1] For Unix(tm) and MS-DOS the executable file moldy or MOLDY.EXE should be placed in the shell search path (*e.g.* in the current directory). There are two optional arguments - the name of the control file (see section 3.1) and the output file (see section 3.5). If either is omitted, control input is read from the "standard input" which may be a terminal or a job command file depending on the operating system and circumstances, and the output is written to "standard output" which may be a terminal or batch job logfile.[2] Here are examples for VAX/VMS and Unix (tm), which assume that in each case the command has been set up to invoke *Moldy*. Under VMS the commands

```
$ moldy control.dat output.lis
$ moldy control.dat
```

will start *Moldy* which will read its input from control.dat. The output will be directed to the file output.lis in the first case and written to the terminal or batch log in the second. Under UNIX any of

```
% moldy < control > output.lis
% moldy control output.lis
% moldy control
```

will cause moldy to read from the file called control and in the first two examples to write its output to output.lis. The command-line interface for MS-DOS is very similar.

3.1 The Control File

The information needed to initiate and control a run of *Moldy* is specified in a file known as the *control file*. This contains the parameters governing the run *e.g.* the number of timesteps to be executed or the frequency of output, and the names of files to be used *e.g.* for reading a restart configuration from or for writing the output to. Parameters in the control file are specified by entries of the form `keyword = value` which appear one to a line, terminated by the special keyword `end`. Spaces and blank lines are ignored as are comments (*i.e.* the remainder of a line following a # symbol) and keywords may be entered in upper or lower case. For example

```
title= Moldy example      # This is a comment

# The above blank line is ignored
nsteps = 1000
step=0.0005
restart-file = RESTART.DAT
end                       # The control file ends here
```

[1]On VMS, `moldy` may be defined as a foreign command by `$ moldy :== $mydisk:[mydir]moldy`

[2]Some operating systems (Unix and MS-DOS) allow *file redirection* whereby the standard input is associated with some file. This may also be used to supply the control file, provided that no command line parameter is given.

sets the title of the simulation to "Moldy example", the number of steps to execute to 1000, the timestep to 0.0005ps and supplies the name of a restart file to read from.

It is not necessary to specify all of the parameters on each run. Unless it is explicitly assigned a value in the control file, each parameter has a default value. This is either the value listed in table 3.1 or, in the case where the simulation is continuing from a restart file, the value it had on the previous run (see section 3.3). Parameters are read in sequence from the control file and if one appears more than once only the final instance is used.

The two most important parameters are `step` which sets the size of the simulation timestep (in ps), and `nsteps` which specifies the number of steps to perform. Together these control the length of time to be simulated. It is also possible to specify that a run should be terminated after a certain amount of computer time has been used - given by parameter `cpu-limit`. This will be particularly useful in batch mode systems, where the run is killed after some specified CPU time has elapsed. Setting `cpu-limit` to the maximum time allowed will cause *Moldy* to terminate the run *before* the limit is reached and write out a backup file (see section 3.3.1).

There are several kinds of parameters:

character strings Apart from `title` these are just file names *e.g.* `sys-spec-file`. No checks are performed on the validity of the name (because *Moldy* has to work on many different computer systems), so if you make a mistake you are likely to get an error message to the effect that *Moldy* couldn't find the file you asked for. To remove a default value, just specify a null string *e.g.* `save-file = `.

booleans These are just switches which turn a feature off or on. '0' means off or false and '1' means on or true. The parameters `text-mode-save`, `new-sys-spec`, `surface-dipole` and `lattice-start` are booleans.

real parameters Several parameters are real numbers *e.g.* `step` which specifies the timestep. They may be entered in the usual floating point or scientific notation *e.g.* `step = 0.0005` or `step = .5e-3`, and are taken to be in the units given in table 3.1.

integer parameters Parameters such as `dump-level` take a numeric value, which should be an integer.

timestep-related parameters Several parameters govern when some calculation begins and ends and how frequently it is performed in between. These are known as "begin", "end" and "interval" parameters, but are really a special case of integer parameters. For example `begin-average`, `dump-interval` and `scale-end`. The calculation begins *on* the timestep specified on the `begin` parameter, occurs every `interval` timesteps thereafter and ends *after* the timestep specified by the `end` parameter. Setting the `interval` parameter to zero is the usual method of turning that calculation off.

The `begin` and `end` parameters behave in a special fashion when the simulation is continued from a restart file; they are interpreted *relative* to the current timestep. Notice especially that `nsteps`, the number of timesteps is treated in this way.

A complete list of the parameters, their meanings and default values appears in table 3.1.

3.2 Setting up the System

3.2.1 System Specification

The information which describes to *Moldy* the system to be simulated and the interaction potentials is contained in a file known as the *system specification file*. This may be presented to *Moldy* in either of two ways: If the control file parameter `sys-spec-file` is null or absent, it is assumed to be appended to the end of the control file. Otherwise it is read from the file whose name is the value of `sys-spec-file`.

This file is divided into two sections. First is the description of the molecules, atoms or ions, which is followed by the potential functions. As for the control file, the input is case independent and free format, but line structured. Blank lines, spacing and comments are ignored.

The physical description consists of a series of entries, one for each molecular species, terminated by the

Table 3.1: Control Parameters

name	type[a]	default	function
`title`	character	Test Simulation	A title to be printed on all output.
`nsteps`	integer	0	Number of MD steps to execute.
`cpu-limit`	real	1e20	Terminate run if excessive CPU time used.
`step`	real	0.005	Size of timestep
`sys-spec-file`	character	null	Name of system specification file. Appended to control file if null.
`lattice-start`	boolean	false	Switch for crystalline initial configuration.
`save-file`	character	null	File to save restart configuration in.
`restart-file`	character	null	File to read restart configuration from.
`new-sys-spec`	boolean	false	Read restart configuration with changed system specification.
`text-mode-save`	boolean	false	Write a portable "restart" file consisting of control, system specification and lattice start files.
`density`	real	1.0	Initial density in $\mathrm{g\,cm^{-3}}$. Used by *skew start* only to determine initial MD cell dimensions.
`scale-interval`	integer	10	Number of steps between velocity scalings.
`scale-end`	integer	1000000	When to stop scaling.
`const-temp`	integer	0	1 for Nosé-Hoover, 2 for Gaussian thermostat.
`ttmass`	real	100	Translational inertia parameter for Nosé-Hoover thermostat ($\mathrm{kJ\,mol^{-1}\,ps^2}$).
`rtmass`	real	100	Rotational inertia parameter for Nosé-Hoover thermostat ($\mathrm{kJ\,mol^{-1}\,ps^2}$).
`scale-options`	integer	0	Select variations on scaling or thermostat.
`temperature`	real	0	Temperature of initial configuration for scaling and thermostat (K).
`const-pressure`	boolean	false	Whether to use Parrinello and Rahman constant stress.
`w`	real	100.0	Value of P & R mass parameter (amu).
`pressure`	real	0	External applied pressure (MPa).
`strain-mask`	integer	200	Bitmask controlling $\mathbf{h}$ matrix constraint.
`alpha`	real	*auto*	α parameter for Ewald sum.
`k-cutoff`	real	*auto*	Reciprocal space cut off distance in $\mathrm{Å}^{-1}$.
`cutoff`	real	*auto*	Direct space cutoff distance in Å.
`strict-cutoff`	boolean	false	Flag to select rigorous or cheap but approximate cutoff algorithm.
`surface-dipole`	boolean	false	Include De Leeuw & Perram term in Ewald sum.
`roll-interval`	integer	10	Period over which to calculate rolling averages.
`print-interval`	integer	10	How frequently to print normal output.

[a] See section 3.1

Table 3.2: Control Parameters (continued)

name	**type**[a]	**default**	**function**
`begin-average`	integer	1001	When to start accumulating the thermodynamic averages.
`average-interval`	integer	5000	How frequently to calculate and print averages.
`reset-averages`	boolean	false	Discard accumulated averages in restart file.
`begin-rdf`	integer	1000000	When to start accumulating radial distribution function information.
`rdf-interval`	integer	20	How frequently binning calculation is performed.
`rdf-out`	integer	5000	How frequently to calculate and print RDFs.
`rdf-limit`	real	10	Calculate RDFs up to what distance? (Å)
`nbins`	integer	100	Number of binning intervals between 0 and rdf-limit.
`xdr`	boolean	true	Write restart, backup and dump files in portable binary format using Sun XDR.
`dump-file`	character	null	Template of file names used for data dumps.
`begin-dump`	integer	1	Timestep to begin dumping at.
`dump-interval`	integer	20	How frequently to perform dumps.
`dump-level`	integer	0	Amount of information to include in dump.
`ndumps`	integer	250	Number of dump records in each dump file.
`backup-interval`	integer	500	Frequency to write backup file.
`backup-file`	character	MDBACKUP	Name of backup file.
`temp-file`	character	MDTEMPX	Name of temporary file used for writing restart configurations.
`subcell`	real	0	Size of sub-cell (in Å) to divide MD cell into for link cell force calculation.
`seed`	integer	1234567	Seed for random number generator.
`page-width`	integer	132	Number of columns on output paper.
`page-length`	integer	44	Number of lines on a page of output.
`mass-unit`	real	1.6605402e-27	Unit of mass for system specification file.
`length-unit`	real	1e-10	Unit of length for system specification file.
`time-unit`	real	1e-13	Unit of time for system specification file.
`charge-unit`	real	1.60217733e-19	Unit of charge for system specification file.

[a] See section 3.1

keyword `end`. The entry for species i should have the form

$$\begin{array}{lllllll}
\textit{species-name}_i & N_i & & & & & \\
id_1 & x_1 & y_1 & z_1 & m_1 & q_1 & name_1 \\
id_2 & x_2 & y_2 & z_2 & m_2 & q_2 & name_2 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
id_{n_i} & x_{n_i} & y_{n_i} & z_{n_i} & m_{n_i} & q_{n_i} & name_{n_i}
\end{array}$$

where *species-name*$_i$ is the name of the molecule and N_i is the number of molecules of that type in the system. Each molecule has n_i atoms, one for each line in that group and each kind of atom is identified by a number id_i (the site id) which will be used to specify the appropriate potential parameters. Its co-ordinates are (x_i, y_i, z_i), its mass is m_i, its charge is q_i and its name is $name_i$. See Appendix A for some sample system specification files.

If there is more than one atom of any type (in the system - not just the same molecule) it is sufficient to identify it by its *id* (and the site co-ordinates!). If m_i, q_i or $name_i$ *are* given they must agree exactly with the previous values or an error will be signalled.

Site ids, masses and charges are all checked for 'reasonableness' and impossible values cause an error. The set of site ids does not have to start at 1 or be contiguous, but since this may indicate a mistake, a warning is issued.

Following the physical specification is the specification of the potential functions. This takes the form

$$\begin{array}{llllll}
\textit{potential-type} & & & & & \\
i & j & p_{ij}^1 & p_{ij}^2 & \cdots & p_{ij}^r \\
k & l & p_{kl}^1 & p_{kl}^2 & \cdots & p_{kl}^r \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
m & n & p_{mn}^1 & p_{mn}^2 & \cdots & p_{mn}^r \\
\texttt{end} & & & & &
\end{array}$$

where *potential-type* is one of the keywords `Lennard-Jones`, `Buckingham` or `MCY` or `generic` to identify the kind of potentials to be used, i, j, k, l, m, n are site ids and p_{ij}^{α} is the α^{th} potential parameter between sites i and j. There should be one line for each distinct pair of site ids. If any pair is omitted a warning is issued and the parameter values are set to zero.

The meaning of the parameters for the currently defined potentials is as follows:

`Lennard-Jones` The potential is

$$\phi(r_{ij}) = \varepsilon((\sigma/r_{ij})^{12}) - (\sigma/r_{ij})^6),$$

and has two parameters, $\varepsilon(\equiv p_{ij}^1)$ and $\sigma(\equiv p_{ij}^2)$, which occur on each line in that order. Note that the definition of ε *includes* the factor of 4 more usually separated out. The control parameter `time-unit` may be divided by four to read potentials specified in the standard form.

`Buckingham` This includes potentials of the Born-Mayer type and has formula

$$\phi(r_{ij}) = -A_{ij}/r_{ij}^6 + B_{ij}\exp(-C_{ij}r_{ij}).$$

The three parameters appear on each line in the order A, B, C.

`MCY` This type supports potentials of the same form as the water model of Matsuoka, Clementi and Yoshimine[30],

$$\phi(r_{ij}) = A_{ij}\exp(-B_{ij}r_{ij}) - C_{ij}\exp(-D_{ij}r_{ij}),$$

and the four parameters appear on the line in the order A, B, C, D.

`generic` This contains a number of inverse power terms and an exponential repulsion in support of ionic solution models. It takes the form

$$\phi(r_{ij}) = A_{ij}\exp(-B_{ij}r_{ij}) + C_{ij}/r_{ij}^{12} - D_{ij}/r_{ij}^4 - E_{ij}/r_{ij}^6 - F_{ij}/r_{ij}^8$$

with the six parameters $A_{ij} - F_{ij}$.

Other types of potential may be easily added: see section 4.3.1.

It is possible to specify the units in which these quantities are given by means of the control file parameters `mass-unit`, `length-unit`, `time-unit` and `charge-unit` (which are themselves specified in SI units). All quantities read from the system specification file (dimensions as well as potentials) are taken to be in those units. Their default values are amu, Å, 0.1ps and q_e, which means that the unit of energy is kJ mol^{-1}. So to read in Å, amu and kcal mol^{-1}, specify `time-unit=4.8888213e-14`.

Once the system specification has been read in, all quantities are converted to 'internal' units: a.m.u., Å, ps, and $\sqrt{(\text{a.m.u. } \text{Å}^3 \text{ps}^{-2}/(4\pi\varepsilon_0))}$. The prototype molecule for each species is then shifted so that its zero of coordinates lies on its centre of mass, and rotated into the principal frame (polyatomics only).

3.2.2 The Initial Configuration

Moldy provides two methods of setting up an initial configuration. By default the *skew start* method of section 2.9.1 is used to place the molecular centres of mass in a regular arrangement which ensures molecular separation. If there is more than one species present, molecules of each are chosen randomly for each site. Molecular orientations are chosen randomly from a uniform distribution. This method has been found to work well for reasonably small or fairly isotropic molecules and it is anticipated that it will be the usual method of starting a simulation of the liquid state. On the other hand, if the constituent molecules are sufficiently large and irregular, or if it is intended to simulate the solid state then the *lattice start* method will be more appropriate.

This method is activated by setting the control parameter `lattice-start` to 1, and creates the initial configuration by periodic replication of some crystalline unit cell. In that case *Moldy* expects to find, following the `end` which terminates the system specification, an initial configuration specification of the following form:

$$
\begin{array}{llllllllll}
a \;\; b \;\; c \;\; \alpha \;\; \beta \;\; \gamma & n_x & n_y & n_z & & & & \\
\textit{species-name}_1 & X_1 & Y_1 & Z_1 & q_{10} & q_{11} & q_{12} & q_{13} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\textit{species-name}_i & X_i & Y_i & Z_i & & & & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\textit{species-name}_n & X_n & Y_n & Z_n & q_{n0} & q_{n1} & q_{n2} & q_{n3} \\
\texttt{end.} & & & & & & &
\end{array}
$$

Here a, b, c and α, β, γ are the crystal unit cell parameters, and n_x, n_y, n_z are the number of unit cells in each direction which comprise the MD cell. The next n lines describe the n molecules of the basis which will be replicated to form the full configuration. Molecules may appear in any order, but of course the total number of each, multiplied by the number of unit cells $n_x n_y n_z$ must agree with that given in the system specification file.

Each molecule is identified by its name, as given in the system specification file. X, Y and Z are *fractional* co-ordinates, between 0 and 1 giving the location of the molecular centres of mass in the crystal unit cell. The orientation is given by the four quaternions q_0, q_1, q_2, q_3 which specify a rotation *relative to the orientation of the prototype molecule in the system specification file*. (Notice the slight inconsistency with the positions, which are of the centres of mass, *not* the zeroes of co-ordinates in the system specification file. This may be fixed in future releases.) Quaternions need only be included for polyatomic species, that is molecules 1 and n above, and omitted for the monatomic species i.

After the molecular positions and orientations have been set up, their velocities (and angular velocities if appropriate) are initialized. Their values are sampled at random from the Maxwell-Boltzmann distribution for the temperature T, as given by the control parameter `temperature`. This is done for both starting methods.

3.3 Restarting from a Previous Run

At the end of a simulation run, it is often desirable to store the configuration of the system in a file. This *restart file* may be used at a later date to continue the simulation from that point rather than from scratch. To instruct *Moldy* to write a restart file, simply set the control parameter `save-file` to a suitable filename; to start from a restart file, set `restart-file` to be the name of that file.

Each restart file is a binary file which contains enough information to reconstruct the exact state of the system and of the program. It includes a copy of all the control parameters in force, the current timestep number, a complete system specification, all the simulation dynamic variables and the intermediate data used in the calculation of averages and radial distribution functions. Thus a run continued from a restart file will proceed just as if there had been no interruption and will generate identical results (provided the control parameters are not changed).

When continuing a simulation, it is only necessary to explicitly specify control parameters which are to be changed. Their previous values are read from the restart file and are used as defaults when reading the control file. Consequently control files for restarting tend to be rather short. **Caution**: always include a new (possibly null) value for `save-file`. Otherwise when the new run terminates, the new restart file may overwrite the old one.[3]

Neither is it necessary to repeat the system specification since that too is stored in the restart file. However there are occasions when it is desirable to do just that, for example if the value of one of the potential parameters is to be modified. In that case, set the switch `new-sys-spec` to 1 (true) and provide a system specification as per a new simulation. This is checked for consistency with the existing one and if correct replaces it. The following checks are applied, which only verify that it is sensible to assign the old dynamic variables to the new system. *1*. The number of species must be the same. *2*. Each species must have the same number of rotational degrees of freedom as its predecessor. It is not possible to replace a polyatomic by a monatomic or linear molecule, for example. *3*. The number of molecules of each species must not change. This means that the order in the specification file must be identical too. It is however possible to change the number of sites on a molecule, subject to *2*.

3.3.1 Periodic Backup

Closely related to restarting is the backup mechanism. This is provided to guard against the complete loss of a simulation due to computer failure. Periodically during a run, *Moldy* writes its state to a *backup file* – which is in fact just a restart file. In the event of a crash, the simulation can be restarted from the point the last backup was written rather than from the beginning. The related control parameters are `backup-file` which specifies the file name and `backup-interval` which gives the frequency of backups. It should not normally be necessary to change the name, but the optimum interval will depend on the size of the simulated system and the speed of the computer. By default it is 500. At the successful end of a run the backup file is deleted so that only if there is an abnormal termination does one remain.[4]

The restart from a backup is entirely automatic. If a backup file exists when a run is started, it is read in and the run continues from it. In contrast to a normal restart all of the control parameters are taken from the backup file and the control file (and a restart file if one is specified) is ignored.[5] In consequence, if a run is aborted or stops abnormally for some reason, the backup file must be removed manually otherwise next time a run starts, the unwanted simulation will continue instead.

If a run terminates abnormally there may also be a *lock file* called MDBACKUP.lck which ought to be removed. *Moldy* attempts to prevent two runs from overwriting each other's backup files by creating a lock file whose name is formed from the backup name by appending .lck. A second run which attempts to use the same backup file will test for the presence of the lock file and abort the run if it finds it.

A restart or backup file is created by first writing the data to a temporary file which is then renamed to the final name. This ensures that there is no possibility of a file being left incomplete or corrupt if the computer crashes part-way through the write. If the file already exists either it is replaced (on systems which only keep one version of a file) or a new version is created (on systems such as VMS which retain multiple versions). In the unlikely event of it being necessary to change where the temporary file is kept,[6] it may be specified with the control parameter `temp-file`.

[3]Whether the old file is lost depends on the operating system. Under systems such as VMS which have version numbers a new version is created and the old one remains. Under Unix, the old file is renamed by the addition of a "%" character and thus is saved. On other systems it will be lost.

[4]A backup file is also written if the run is terminated for exceeding its cpu limit.

[5]This is not quite true. *Moldy* does read the control file and any restart file but only to determine the name of the backup file. Thus even if the backup has a non-standard name it can still be found.

[6]This may be necessary if the restart file is located on a different device or disk partition from the current directory. To rename the temporary file successfully, it must reside in the same partition or device as the restart file.

3.3.2 Portable Restart Files

Moldy is able[7] to read and write restart and dump files in a portable binary format which is transportable between computers of different architectures. So a restart file written on, for example, a Sun may be used to initiate a new run on a Cray, and the dump files generated on the Cray may be analyzed on the Sun. This feature will also be of considerable use in modern heterogeneous networks where diverse machines frequently share a common file space.

The format is based on Sun Microsystems XDR protocol[21]. The XDR routines are available on almost every modern Unix machine, and are simple enough to implement on any other system.[8] If the control parameter `xdr` is set to 1 then all files will be written using this format. *Moldy* automatically determines whether a restart file was written using XDR (by examining the file header) and reads it in the appropriate fashion irrespective of the value of `xdr`.

3.4 Setting the Temperature

Moldy implements three different methods to control the temperature of the simulated system. These are the velocity rescaling technique described in section 2.6.1, the Nosé-Hoover thermostat and constrained equations of motion (section 2.6.2). Scaling is selected by the parameters `scale-interval` *etc.* Every `scale-interval` timesteps until `scale-end`, the velocities are adjusted so that the kinetic energy corresponds exactly to the desired temperature (the value of control parameter `temperature`). The Nosé-Hoover and constrained thermostat are selected by setting `const-temp` equal to `1` or `2` respectively.

The control parameter `scale-options` selects refinements to the basic scaling or thermostat algorithms. This is an integer parameter interpreted as a set of bit flags with the following meanings.

bit 0 perform scaling or thermostatting for each molecular species individually.

bit 1 scale/thermostat the rotational and translational components of the kinetic energy separately.

bit 2 use the rolling averages of kinetic energy to calculate the scale factor rather than the instantaneous values.

bit 3 discard all existing velocities and accelerations and re-initialize from the Maxwell-Boltzmann distribution.

The bits may be set in any combination so, for example `scale-options=6` sets bits 1 and 2 ($6 = 2^1 + 2^2$) and scales separately for rotation/translation using the rolling averages. If bit 3 is set the others are ignored. Only bits 0 and 1 have any meaning in the case of a thermostat, and signify that each species, or the translational and rotational degrees of freedom are isolated from each other and coupled to their own, individual heat baths.

The options for scaling separately rotation and translation, and per species may be useful for achieving equilibration in "difficult" systems where mode-coupling is ineffective. In those situations it is otherwise possible for all the energy to be transferred into the rotational modes of a particular species, halting any progress to equilibrium for other degrees of freedom. These options ensure that all degrees of freedom have some thermal energy.

The option controlled by bit 3, to discard all existing information and start from a random set of velocities may be of use when starting from far-from-equilibrium situations. In such cases the forces are frequently so large that the velocities and accelerations exceed the limits of the integration algorithm and timestep, which results in *Moldy* stopping with a *quaternion normalization* or *quaternion constraint* error. Judicious use of this option every few timesteps (using `scale-interval`) ought to allow the system to relax to a state sufficiently close to equilibrium for normal scaling to take over.

Bit 2 is intended to deal with the problem of setting the temperature accurately using scaling. The *ensemble average* kinetic energy which characterizes the temperature of the system and the instantaneous value fluctuates about this value. However in the traditional implementation of scaling, velocities are multiplied by a factor of $\sqrt{desired\ KE / instantaneous\ KE}$. Thus the scaling factor is "wrong" by the ratio of the instantaneous to average KE's which means that the temperature can not be set more accurately than the relative size of the fluctuations in the KE. The option selected by bit 2 goes some way towards the ideal scaling factor by using the rolling average KE instead of the instantaneous value. The fluctuations in this short-term average should be much lower than in

[7] From version 2.1 onwards

[8] Because the XDR calls are not part of ANSI standard C, however, the XDR code is conditionally compiled into *Moldy* only if the `USE_XDR` preprocessor symbol is defined during compilation.

```
               Nov 17 15:13:06 1989          Water_test          Page 4
Trans KE     Rot KE     Pot Energy   Tot Energy    TTemp     RTemp      Temp    h(1,*)  h(2,*)  h(3,*)       Stress  Stress  Stress
========Timestep 10 Current values ==================================================
  243.88     453.88       -187.35       533.5      305.5     568.6     424.4     12.53    0.00    0.00          589    46.4     120
  22.053          0        1.0401                  221.0       0.0                0.00   12.53    0.00         46.4     373    90.1
                                                                                  0.00    0.00   12.53          120    90.1    -207
---- Rolling averages over last 10 timesteps ------------------------------------------------------------------
  240.27     319.31       -82.472      533.39      301.0     400.0     342.9     12.53    0.00    0.00      1.2e+03     296     127
  22.077          0        34.205                  221.3       0.0                0.00   12.53    0.00          296     589     133
                                                                                  0.00    0.00   12.53          127     133    -132
---- Standard deviations --------------------------------------------------------------------------------------
  1.8214     71.893        56.441      .19942        2.3      90.1      43.4      0.00    0.00    0.00     1.32e+03     750      51
   0.013          0        17.173                    0.1       0.0                0.00    0.00    0.00          750     119    55.2
                                                                                  0.00    0.00    0.00           51    55.2      49
```

Figure 3.1: Sample *Moldy* output from a simulation of a two component mixture. The first component is a polyatomic molecule and the second is atomic. There are three *frames*, for the instantaneous values, the rolling averages and their associated standard deviations. Within a frame, each row has the following meaning: for translational and rotational kinetic energies and temperatures it is the per-species value; for the potential energy it is the direct and reciprocal space components, and the MD cell matrix, `h` and the stress are laid out as 3×3 matrices.

the instantaneous value, allowing more accurate temperature control. However it will almost always be easier to use a true thermostat to achieve this goal.

3.5 Output

At the beginning of each run *Moldy* writes a *banner page* containing a summary of the system being simulated and details of the important control parameters. The bulk of the output file is the *periodic output* which contains the instantaneous values of various thermodynamic variables, their rolling averages and associated standard deviations. The *rolling average* is just the mean over the preceding *n* timesteps where *n* is set by the control parameter `roll-interval`. An annotated example is given in figure 3.1. The frequency of periodic output may be altered by setting the control parameter `print-interval` to the interval required. (This may be necessary to constrain the size of the output file which can grow to be very large indeed with the default interval of only 10.)

As well as the short term rolling averages, long term averages are calculated and printed out at regular but usually infrequent intervals. Accumulation starts on the timestep given by the control parameter `begin-average` and every `average-interval` timesteps thereafter, the means and standard deviations are calculated and printed. This output is interspersed with the periodic output and is formatted with one variable to a line in the form *mean +/- s.d.*. Where a variable has more than one component (such as multiple species for the translational temperature or Cartesian components for the mean square forces) the components are printed across the page.[9] In addition to those variables printed as part of the periodic output, the pressure, the virial, mean square forces, mean square torques and total dipole moments are calculated.

3.5.1 Output units

All of the various forms of output use the same units, though for brevity they are not explicitly mentioned on the periodic output. Lengths are measured in Å, energies are all expressed in kJ mol^{-1}, temperatures in Kelvin, pressure and stress in MPa, mean square forces and torques in N^2 mol^{-1} and Nm^2 mol^{-1}, charge in electron charges and dipole moments in Debye. Because energy is an extensive quantity the printed values refer to the *whole system*. (There is no practical way of expressing energies per mole of any particular constituent in a program capable of simulating arbitrary mixtures.)

If these units do not suit, they can be changed in the configuration file defs.h, where the conversion from internal to output units is parameterized.

[9] Remember that the standard deviation is a measure of the *fluctuations* about the mean, **not** the *uncertainty* in the mean. For that the standard error in the mean is required, which is more difficult to evaluate. Theoretically it is the *s.d.* divided by $\sqrt{N}$ where N is the number of independent observations. But successive timesteps are highly correlated and do not count as independent. See ref [2] section 6.4, page 191 onwards for a full discussion.

```
        Radial Distribution Functions   Bin width=0.1
        O-O RDF
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000481 0.035710 0.183334 0.442186 0.613992 1.024402
1.046396 0.964906 0.830174 0.660035 0.693341 0.615902 0.593192 0.510595 0.530697 0.532030 0.535959 0.524457 0.523221 0.466219
0.496028 0.438487 0.456500 0.410547 0.443861 0.457956 0.446822 0.452202 0.419768 0.439333 0.465509 0.486887 0.461970 0.475745
0.478883 0.480854 0.509090 0.533728 0.552747 0.552555 0.575402 0.547278 0.544836 0.493597 0.488168 0.520727 0.508073 0.479948
0.501159 0.484000 0.485378 0.489160 0.464448 0.466791 0.476508 0.446576 0.470948 0.474468 0.449340 0.462169 0.501220 0.519107
0.513338 0.510192 0.499766 0.525963 0.504663 0.517673 0.498359 0.512156 0.507061 0.466390 0.464342 0.445886 0.417555 0.407778
0.387220 0.374041
        O-H RDF
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 26.976688 0.000000 0.000000 0.000000 0.000000
0.000000 0.016214 0.061257 0.304082 0.647342 0.847404 0.757188 0.601222 0.478273 0.462682 0.449614 0.450424 0.518998 0.572242
0.689704 0.914269 1.184674 1.441772 1.570390 1.609068 1.600392 1.430457 1.322722 1.183606 1.103701 1.061788 0.980018 0.960570
0.924390 0.908883 0.877591 0.857668 0.890761 0.852463 0.815447 0.824963 0.841255 0.890416 0.929030 0.960589 0.984145 1.020650
1.028199 1.047496 1.064600 1.099812 1.095715 1.073793 1.078131 1.049212 1.052160 1.052001 1.020737 1.010782 0.979748 0.983158
0.988946 0.967620 0.955655 0.944384 0.952145 0.948509 0.946692 0.960097 0.959299 0.964074 0.969219 0.972704 0.998504 1.027791
1.041576 1.037637 1.039961 1.016804 1.004726 1.026805 1.030903 1.006268 0.972421 0.948140 0.908959 0.877089 0.849855 0.817964
0.776986 0.721485
```

Figure 3.2: Example output of radial distribution functions. After the header line consisting of underscores there is an indication of the bin width b (that is the distance between points at which the RDF is tabulated). Then for each site type pair $\alpha\beta$ there is a line listing which pair (*e.g.* `O-O RDF`) followed by `nbins` values of $g_{\alpha\beta}((i+1/2)b)$.

3.6 Radial Distribution Functions

Radial distribution functions are calculated by binning site pair distances periodically throughout the simulation (see section 2.8). As this process is expensive in computer time the binning subroutine is invoked only every few timesteps, as set by the control parameter `rdf-interval` (20 by default). Since the pair distances only change a little on each timestep, very little statistical information is lost. Collection of binning data may also be turned off during an equilibration period: specify when binning is to start by means of the parameter `begin-rdf`. The parameters `rdf-limit` and `nbins` control the details of binning, giving respectively the largest distance counted and the number of bins that interval is divided into. The calculation of the interatomic distances is done separately from that used in the evaluation of the forces, using the same link cell scheme. This ensures that all site pairs separated by less than `rdf-limit` are included. This parameter may be varied independently of the interaction cutoff, thereby allowing RDFs to be evaluated out to large distances without incurring the time penalty of increasing the cutoff.[10]

Every `rdf-out` timesteps (by default 5000) the RDFs are calculated from the binned distances and printed out, and the counters are reset to zero to begin accumulation again. Distances are binned and RDFs $g_{\alpha\beta}(r)$ calculated separately for each distinct type of atom-atom (or site-site) pair. An explanation of the output format is given in figure 3.2. Note that each number should be considered as the value at the *centre* of its bin, so that entry i in each list is the value of $g_{\alpha\beta}((i+1/2)b)$ where b is the bin width.

There are a couple of tricks which may be played with the system specification if the atomic pair RDFs do not give exactly the functions required. Firstly, it is possible to calculate RDFs about a particular site, distinguishing it from otherwise identical atoms by assigning it a new and unique site id in the system specification file. (This is the MD equivalent of the isotopic substitution method used in neutron diffraction). Secondly, if the *molecular* pair correlation is required, this is identical to the RDF of an atom located at the molecular centre-of-mass. A "ghost" site without charge, mass or potentials may be added if necessary.

3.7 Dumping

The dump facility is provided in order to allow the calculation of dynamic properties, such as time correlation functions and additional static averages not normally calculated by *Moldy*. During a run, dump files are produced which contain a record of the simulation dynamic variables (positions, quaternions *etc.*) at varying degrees of detail. Any property of interest, dynamic or static, may then be evaluated using the data in the dump.

A dump consists of a sequence of files since the amount of data generated in a run can be very large indeed and it is usually more convenient to manipulate a series of smaller files rather than one large and unwieldy one. *Moldy* takes considerable pains to ensure that a contiguous sequence of dump files is maintained and also ensures that dumps from different runs are not accidentally intermixed. There is no requirement that a dump file be produced

[10] In previous versions of *Moldy* the calculation of the interatomic distances was done on the basis of the "minimum image" convention. Consequently the calculated value of $g_{\alpha\beta}(r)$ tailed off for $r_c > L/2$. This restriction is now lifted

by a single run of *Moldy* , which extends an existing file or starts a new one as appropriate. A simulation may stop and be restarted many times without disturbing the dump file sequence. The sequence should (in most cases) even survive a system crash and a restart from a backup file (see section 3.3.1).

Each dump file in a sequence is a binary file consisting of a *dump header*, which contains information about the contents of the file followed by a number of *dump records* which contain the actual data.

Several control parameters govern dumping. It starts at the timestep specified by `begin-dump`, and a dump record is written every `dump-interval` timesteps thereafter. After `ndumps` dump records have been written to a file, it is closed and another is begun. Filenames are generated from a prototype (given by the parameter `dump-file`) by appending a number, so that if the prototype is MDDUMP then successive files will be named MDDUMP0, MDDUMP1, MDDUMP2 *etc.* If it is not convenient for the sequence number to appear at the end of the file, include the characters "%d" at an appropriate point.[11] For example under VMS, specifying `dump-file=mddump%d.dat` will name the files mddump0.dat, mddump1.dat *etc.*

Each dump record is a sequence of single-precision floating point binary numbers. These are written either in native (*i.e.* the machine's own) format or XDR format (see section 3.3.2) depending on the value of the control parameter `xdr`. The record's exact contents are determined by the control parameter `dump-level` which is a bit flag *i.e.* a value of 2^n means that bit n is set. Four bits are used and any combination may be specified but the cumulative values 1, 3, 7 and 15 are most useful. A value of 0 disables dumping. The data dumped for each bit are as follows:

bit 0 centre of mass co-ordinates, quaternions, unit cell matrix and potential energy.

bit 1 centre of mass velocities, quaternion and unit cell matrix derivatives.

bit 2 centre of mass accelerations, quaternion and unit cell matrix second derivatives.

bit 3 forces, torques and stress tensor.

Items selected are written in the order laid out above. Within each set of variables, values are ordered primarily by species in the order they appeared in the system specification. Within a species ordering is by molecule (or atom) and at the finest level by x, y or z component ($q_0, \ldots q_3$ for quaternions). Therefore if n is the total number of molecules and n_r is the number with rotational freedom the size of each record is

$$\begin{array}{lll} & 3n+4n_r+9+1 & \text{(if bit 0 is set)} \\ + & 3n+4n_r+9 & \text{(if bit 1 is set)} \\ + & 3n+4n_r+9 & \text{(if bit 2 is set)} \\ + & 3n+3n_r+9 & \text{(if bit 3 is set)} \end{array}$$

single precision floating point numbers.

The header is a copy of a `struct dump_t` (see appendix 5.1 for the format). It contains the simulation title and version number, the timestep at the beginning of the file, the control parameters `dump-interval` and `dump-level`, the maximum and actual number of dump records in the file, a unique marker (actually a timestamp), common to all the files in a dump run, and the timestamp[12] of any restart file used to start the run.

It is not possible to dump directly to magnetic tape. *Moldy* must rewind to the beginning of a file to keep the header up to date with the number of dumps in the file, as well as extend existing files. Neither operation is allowed on a tape drive. Large disk stores are now very cheap so this should not be a problem in practice. If disk store *is* limited then the simulation may be divided into multiple *Moldy* runs interspersed with copying of dump files to tape.

Notice that *Moldy* must sometimes read an existing but complete dump file to propagate the unique marker to all of the files in a sequence. Therefore when continuing a simulation and a dump run, at least the immediately preceding dump file must still be accessible. This should be borne in mind when copying dumps to tape!

Moldy is careful to ensure that existing files are not overwritten - especially necessary since dump records are added to the end of an existing dump file. Whenever *Moldy* prepares to start a new dump file it checks to see if one of that name is already present. If so, a new name is chosen by "mutating" the old one, and a warning message to that effect is written to the output file. On the other hand, if the *first* file of a new dump run (including one initiated

[11] This is actually the code `sprintf()`, the C library function uses to signify converting an integer to a decimal character string. This function is used to create the actual file name from the prototype and the integer dump sequence number. (See any C library manual for details.)

[12] A timestamp is simply the number of seconds elapsed since midnight on January 1, 1970.

because of some error in continuing an old one) already exists, the *prototype* file name is mutated as above and the whole dump run is written to files based on the mutated name.

When a run is restarted checks are made to ensure that the values of the dump control parameters have not been altered. If they have, it is not possible to continue an existing dump sequence and a new one will be started. (If existing dump files are present the new sequence will have mutated file names.) This also happens if an existing file does not appear to be a *Moldy* dump. Existing dump files are also tested to ensure that there is no corruption (due, for example to a system crash) and that they contain the correct number of records. If the dump sequence can not be continued in these circumstances, *Moldy* terminates with a fatal error rather than waste computer time.

Two utility programs included in the distribution are *dumpanal* which identifies dump files by printing out the headers and *dumpext* which extracts atomic or molecular trajectories. The latter should be useful as a prototype for writing programs to analyse dump data.

It is frequently convenient to perform analysis of dump data, and perhaps graphical output on a different computer to that which generated the data. In the past it has not usually been possible to sensibly transfer binary data between computers of different architectures. However *Moldy* is able to write dump files in a portable format using XDR (see section 3.3.2) which may be read by *dumpext* on any machine. The control parameter `xdr` enables XDR mode for dumps as well as restart files. As yet, XDR is not available on every machine. Therefore a program called *dumpconv* is provided which converts dump files to a portable text file format (which may be easily moved between machines) and back again. It is described in appendix B.3.

3.8 Constant Stress Simulation

Setting the control parameter `const-pressure` switches from a constant-volume simulation to a constant-stress simulation using the method of Parrinello and Rahman[35] (see section 2.7). The value of the MD cell mass parameter, W, is given by the control parameter `w` and the external pressure by `pressure`. At present it is not possible to specify an anisotropic external stress, though this capability may be added in future versions of the program.

The **h** matrix may be constrained so as to disable motion of any or all of its components using the parameter `strain-mask`. `Strain-mask` is a bitmap: each "bit" of the integer freezes one of the components of **h**; bit i freezes $\mathbf{h}_{kl}$ with $i = 3(k-1)+l-1$. The bitmask is the sum of 2^i over the i's to be set, so the `strain-mask` values

$$\begin{pmatrix} 1 & 2 & 4 \\ 8 & 16 & 32 \\ 64 & 128 & 256 \end{pmatrix} \text{ constrain the corresponding components of } h, \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix}.$$

Thus the default constraint of $\mathbf{h}_{?1} = \mathbf{a} = (a_x, 0, 0), \mathbf{h}_{?2} = \mathbf{b} = (b_x, b_y, 0)$ is given by `strain-mask=200` (8+64+128). Another useful value is 238 which freezes all the off-diagonal components. This is needed for a liquid simulation since there are no shear restoring forces acting on those components.

3.9 Cutoffs and Adjustable Parameters

There are four parameters related to the Ewald sum method of force evaluation (see section 2.4), α, r_c, k_c, and `subcell`. In addition the two options `strict-cutoff` and `surface-dipole` select how rigorously the real-space cutoff is applied and whether to include the De Leuuw surface dipole term.

By default α, r_c and k_c are chosen automatically, using equations 2.23 to give a default accuracy of $\varepsilon = \exp(-p) = 10^{-5}$ (*i.e.* $p = 11.5$) in the Coulombic potential energy evaluation. An empirically determined value of $t_R/t_F = 5.5$ is used. If a different accuracy is desired the cutoffs may be adjusted using equations 2.23. The α parameter is specified by `alpha` in units of Å^{-1} and the direct and reciprocal space cutoff distances r_c and k_c by `cutoff` and `k-cutoff` in units of Å and Å^{-1} respectively. The value of α should only be changed after careful timing tests if the system size is large. The power-law given in equation 2.21 gives a theoretical scaling of execution time with number of ions of $T \propto N^{1.5}$. In practise $T \propto N^{1.57}$ has been achieved over a range of N from 72 to 7800, which is very close to optimal.

Important note: The automatically determined value of r_c is chosen to converge the Coulombic part of the potential only. Due to the very general nature of the potentials it is not possible to choose r_c automatically so as to guarantee convergence of the non-electrostatic part. Although in many cases the automatic value will be adequate

it is the user's responsibility to ensure that it is large enough. If there are no charges in the system specification file then r_c is not set and an error message is issued.

As an example of manual determination of the parameters, for a simulation of 512 MCY water molecules the values $\alpha = 0.3\text{Å}^{-1}$, $r_c = 9\text{Å}$ and $k_c = 1.9\text{Å}^{-1}$ give potential energies correct to approximately 1 part in 10^5. For a simulation including ions - 1.1 Molal Magnesium Chloride solution - the same accuracy is attained with $\alpha = 0.45\text{Å}^{-1}$, $r_c = 9\text{Å}$ and $k_c = 3\text{Å}^{-1}$.

The other relevant parameter is the switch `surface-dipole` which includes the dipole surface energy term of De Leeuw, Perram and Smith[29]. See the note in section 2.4 for an explanation of why this term should *never* be used for an ionic (as opposed to dipolar) system.

The two adjustable parameters which control the link cell force calculation (see section 2.5) are `subcell` and `strict-cutoff`. The former specifies the length (in Å) of the side of a link cell and determines the number of cells the MD cell is divided into. In fact the MD cell is divided into a whole number of subcells whose side in each of the three directions is nearest to the value of `subcell`. (The default of zero though, is special and sets subcell to one fifth of the cutoff radius.) In general the smaller the link cell, the more accurately the cutoff radius is implemented, but too many of them reduces the efficiency of the program.

In the default cutoff mode `strict-cutoff=false` the list of neighbour cells is constructed to include all cells whose centre-centre distance is less than the cutoff. This means that some molecule pairs separated by more than the cutoff will be included and some by less will be omitted. Setting `strict-cutoff` to true generates a larger cell neighbour list which is guaranteed to include all appropriate molecule pairs. Furthermore, molecules separated by more than the cutoff are excluded from the force calculation by setting their separation to a large number, 100 times the cutoff, at which distance it is assumed the potential is very close to zero. This is therefore the mode of choice for liquid simulations where any artificial anisotropy is undesirable. See section 2.5 for a full explanation.

It is worth noting that it is unnecessary to recompile the program or change anything else when the cutoffs are modified. Unlike most MD codes, *Moldy* employs dynamic array allocation and automatically sets up arrays of the correct size (and no more!) for any given k_c.

3.10 Framework Simulations

There has recently been much interest in simulations of systems of molecules interacting with some rigid framework such as zeolites, clays and other surfaces. *Moldy* has the capability to include such a framework in a simulation by defining it as a special kind of molecule.

The system specification should contain an entry, similar to that for a normal molecule, which describes the atomic sites belonging to one MD cell's worth of the framework. Its periodic images should fill space to construct the required infinite framework. This is notified to the program by modifying the first line of the specification of that molecule to read

species-name$_i$ `1 Framework`

⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮

(compare with section 3.2.1). The effect of the special keyword `framework` is

1. to remove the rotational freedom of the molecule. This preserves the infinite structure over MD cell repeats by disallowing relative motion of its parts. (Linear motion does not destroy the structure and *is* allowed.)

2. to modify the effect of the periodic boundary conditions. Normally a molecule is assumed to be "small" and periodic relocations are applied to *all* of its atoms depending on its centre-of-mass co-ordinates relative to some interacting molecule. In contrast, the atoms of a framework are independently relocated. This ensures that each molecule "sees" all framework atoms from any unit cell which are within the cut-off distance.

In the present version of the program, only one framework molecule is allowed, though more may be permitted in future versions. Consequently the configuration given as a lattice start must fill the entire MD box. (A skew start is not sensible under these circumstances since the orientation of the framework must be explicitly specified to construct a good space-filling structure.)

3.11 Messages and Errors

Apart from the periodic output, there are occasional once-off messages which *Moldy* writes to the usual output file. Such messages begin with the characters `*I*`, `*W*`, `*E*` or `*F*` denoting the classes *information*, *warning*, *error* or *fatal* respectively. Their meanings are

`*I*` information. These are often produced by subroutines to give useful information on their particular calculations. For example when temperature scaling is turned off a message to that effect is recorded in the output file. Various routines which calculate internal quantities such as the Ewald sum self energies and distant potential corrections also record their values using an information message.

`*W*` warning. When the system specification is suspicious but not clearly wrong, or some untoward condition is detected such as two atoms approaching too closely, a warning message is issued.

`*E*` error. Error messages are issued when a mistake is detected reading any of the input files. To make correction easier, processing continues until the end of that file, so that all of the errors are found. The simulation is then stopped with a fatal error.

`*F*` fatal. The simulation is terminated immediately. Faulty input files generate fatal errors after they have been completely processed. There are many other conditions which also generate fatal errors, for example if the simulation algorithms violate some criterion such as quaternion normalization or constraints (see section 2.1), if the program runs out of memory or if a restart file can not be correctly opened or is of the wrong format.

Most of the messages are self-explanatory. However there are two fatal errors which occasionally arise and are somewhat cryptic:

```
*F* Quaternion n (x,x,x,x) - normalization error in beeman
```

and

```
*F* Quaternion n - constraint error (x)
```

Technically these refer to violations of the conditions that the quaternions representing the angular co-ordinates be normalized to 1 and that equation 2.11 be satisfied. Either may occur if the angular velocity of some molecule becomes too high for accurate integration of the equations of motion. This may have a number of causes. First the timestep may simply be too large. Second the system may be in a state where atoms are so close as to generate large torques which accelerate the molecule to a high angular velocity. This commonly arises if the starting configuration is very far from equilibrium, particularly in the case of molecules with small moments of inertia, such as methane. In most cases the simulation may be restarted using strong rescaling or a Gaussian thermostat to limit velocities during the approach to equilibrium. Occasionally a smaller timestep may help during equilibration. The third cause of normalization or constraint errors is an error in the potentials or the units which allows for a state with high or infinite negative binding energy.

Note that these messages only occur for polyatomic molecules. If the system is monatomic the errors mentioned above may still be present but will not be detected and the simulation may fail in some less predictable manner, for example particles may approach too closely and/or acquire high velocities and fly off to infinity. The message

```
*W* Sites n and m closer than 0.5A.
```

gives some warning of this condition.

Chapter 4

Compiling and Modifying Moldy

The *Moldy* distribution consists of numerous files of "C" source code for *Moldy* and the utility programs, command or job files to compile the source, LaTeX input for the manual and example control and system specification files. For ease of transport these are packed into one large archive file, whose format and method of unpacking depends on the operating system of the target machine. At present it is available for:

unix The archive is usually a tar archive called moldy.tar possibly compressed with the "compress" or "gzip" programs and named moldy.tar.Z or moldy.tar.gz. These files may be uncompressed using the "gunzip" or "uncompress" programs, *viz.* `gunzip moldy.tar.gz` or `uncompress moldy.tar.Z` whereupon the archive is unpacked by the command `tar xvf moldy.tar`.

An alternative form of archive for those unusual systems without a "tar" program is a shell archive — a Bourne shell script called moldy.shar. This is unpacked by `/bin/sh moldy.shar`.

VMS The archive is a DCL command file called moldy.com, and is unpacked by the command `@moldy`.

MS Windows 3/Windows 95 The shareware program "winzip" may be used to unpack the compressed tar archive moldy.tar.gz.

MS-DOS The files must be unpacked from the tar archive on another host and transferred to the PC by disk or ftp.

4.1 Compilation

The source code of *Moldy* consists of 23 files of C programs (which have suffix ".c") and 8 header files (suffix ".h"). To build *Moldy*, all of the .c files must be compiled and linked together to form an executable. The method of doing this depends on the operating system of the target computer.

unix The "make" program is used and the make file is supplied in the distribution. The Makefile supplied will attempt to compile using the command `cc -O -DUSE_XDR` which will usually be sufficient to build a working executable on most operating systems. However some systems may require extra libraries to be specified and it will usually be possible to build a faster-running executable by judicious use of optimization options. Options for many common compilers/operating systems are listed in commented-out form. To enable them, simply uncomment the appropriate lines in the makefile. Then just type `make` to compile and link *Moldy* and `make utilities` for the utility programs.

VMS Simply type `@compile` to execute the command file compile.com. This will build *Moldy* and the utilities.

DOS There is a makefile for Borland Turbo C called Makefile.mak in the standard distribution. This must be edited to select the appropriate compiler options before executing *make*.[1] Alternatively the programs may be built within the interactive environment. Consult the makefile to find out which source files to link to build *Moldy* and the utilities. Moldy has also been built using Watcom C.

[1] Be sure to delete or rename the unix make file Makefile since Turbo C *make* will attempt to execute this in preference to Makefile.mak

MS Windows 3/Windows 95 It is probable that the Borland C Makefile.mak will work under Borland C for Windows or Windows 95.[2]

4.1.1 XDR Dump and Restart

As described in section 3.3.2 binary restart and dump files may be written and read in a portable way using the Sun XDR calls. Since these are are not part of the ANSI C standard libraries, this code is conditionally included by defining the C preprocessor macro `USE_XDR`. Though non-standard, XDR is almost universally available on unix operating systems, so the makefile defines this by default, assuming the syntax of the compiler flag `-DUSE_XDR` to do so. If the XDR headers and libraries are not present on your system, then comment out the line reading `XDR=-DUSE_XDR` at the top of the makefile to deactivate this feature. On certain systems (*e.g.* older versions of SGI IRIX and Sun Microsystems Solaris 2.x) it may be necessary to include some additional library in the link, via the `LDFLAGS` make macro.

4.1.2 Parallel Version (Message Passing)

The parallel version of *Moldy* relies on an interface with a suitable message-passing library. This is the recommended version and supersedes the "shared-memory" parallel implementation described in section 4.1.3. The current release contains interfaces to the MPI library[14], the TCGMSG library and the Oxford BSP library. MPI is the most recommended interface since it is the new standard for message-passing libraries, and should become ubiquitous. If none of these are installed on your machine, some public-domain implementations are available for workstation clusters, multiprocessors and many distributed-memory parallel machines.

MPI The MPICH and CHIMP implementations can be fetched by anonymous ftp from the URLs
`ftp://info.mcs.anl.gov/pub/mpi/mpich` and `ftp://ftp.epcc.ad.ac.uk/pub/chimp/release`.

TCGMSG This may be obtained by anonymous ftp from
`ftp://ftp.tcg.anl.gov/pub/tcgmsg/tcgmsg.4.04.tar.Z`.

BSP The Oxford BSP Library is available through Oxford Parallel's WWW server
`http://www.comlab.ox.ac.uk/oucl/oxpara/bsplib1.htm` or by anonymous ftp
`ftp://ftp.comlab.ox.ac.uk/pub/Packages/BSP/bsp1.2.tar.Z`.

SHMEM The CRI native communications library for T3D and T3E systems.

Alternatively a port to another parallel interface should be quite straightforward, see section 4.3.2.

Once a suitable message-passing library is installed the procedure for building *Moldy* is quite simple. The C preprocessor macro `SPMD` must be defined as well as one of `MPI`, `TCGMSG`, `BSP` or `SHMEM`. This is usually done in the makefile by setting the Make macro `PARLIBC=-DSPMD -DMPI`, for example. This macro should also include a `-I` directive specifying the directory for the library's header files if these are not in the default path searched by the compiler. The similar make macro `PARLIBL` should contain the linker directives necessary to link to the library itself. Examples are provided at the top of the supplied Makefile.

This parallel implementation makes use of the *replicated data* approach[46] whereby every processor has a complete copy of all the arrays containing dynamical variables and every site on every molecule. The computation of the real-space potential and forces is distributed over processors on the basis of link cells. For the reciprocal-space part of the Ewald sum, the *k*-vectors are distributed among processors. This is an extremely efficient way of implementing parallelism since the forces *etc.* must be summed over processors only once per timestep, thus minimizing interprocessor communication costs. It is therefore possible to get considerable speedup for a small number of workstations coupled by a fast network such as an Ethernet.

The biggest disadvantage of the replicated data strategy is that every processor must maintain a copy of all of the data, and therefore that the memory requirement per processor increases with the size of the system. In many cases this is not a severe problem, as MD memory requirements are not large compared with memory sizes of modern computers. However the poor scaling will eventually limit the number of processors which may be used.

[2]The author's experience of Windows 3 and Windows 95 platforms is very limited. I would very much welcome any reports from users on how to build *Moldy* in these environments for inclusion in future versions of this manual.

On a shared-memory multiprocessor, the alternative parallel version in section 4.1.3 may provide a solution, if it can be ported to that machine.

The memory limitation will be most acute when the goal is to simulate an extremely large system on a massively parallel distributed-memory computer where it is desirable to scale the system size with the number of processors. In such architectures the available memory per processor is usually a constant independent of the number of processors. But the memory needed *per processor* increases with system size. Mostly the scaling is linear, but the reciprocal-space sum uses temporary arrays whose size scales with the product of the number of sites and k-vectors, and hence to the $\frac{3}{2}^{\text{th}}$ power of the system size.

An alternative version of ewald.c which implements the reciprocal-space term of the Ewald sum by distributing over *sites* rather than k-vectors is included in the distribution as ewald-RIL.c. It is based on the RIL algorithm of Smith [47] and *distributes* the temporary arrays containing the $\cos(\mathbf{k} \cdot \mathbf{r}_i)$ and $\sin(\mathbf{k} \cdot \mathbf{r}_i)$ over the nodes. In other words, each node only stores the terms involving the $\mathbf{r}_i$'s to be considered on that node. Since these arrays are by far the largest users of memory there is a substantial decrease in overall memory requirement. Moreover the size per node now scales *linearly* with the number of k-vectors and therefore (assuming α is optimized), to the two-thirds power of the number of sites. These arrays will not therefore dominate the memory requirement in the limit of large numbers of processors and system size. The disadvantage of the RIL scheme is that the partial sums of $\cos(\mathbf{k} \cdot \mathbf{r}_i)$ and $\sin(\mathbf{k} \cdot \mathbf{r}_i)$ must be summed over nodes separately for each k-vector. Though the amount of data transferred each time is small, the communication and inter-processor synchronization is far more frequent than for the RKL scheme and the parallelism becomes very fine-grained. The upshot is that only machines with very low communications latency can run this version effectively. Practical tests show that the communications overhead completely negate any parallel gain on systems of networked workstations and most multiprocessors. However a significant speedup is obtained on a Cray T3D, which is exactly the case where this version is needed.

4.1.3 Shared-Memory Parallel Version

An alternative parallel version is available for shared-memory multiprocessors with "parallelizing" compilers. This relies on the compiler handling the multi-threading, synchronization and allocation of local memory stacks for inner function calls. It requires compiler-specific directives to be inserted in the code and is therefore less portable than the distributed-memory version of the previous section. (Note that that version works on this class of machines too under the message-passing interface.) Nevertheless, it works and has been run on Stardent, Convex and Cray computers. It consists of replacements for files force.c and ewald.c called force_parallel.c and ewald_parallel.c. Then the program should be compiled with the preprocessor macro `PARALLEL` defined (not `SPMD`).

The distributed-memory parallel version (section 4.1.2) is generally recommended over this one. However because the parallel sections reference a single global copy of most of the arrays, the shared-memory version uses much less memory. This version may therefore be of use if memory limits the size of the system on a multiprocessor machine.

4.2 Portability

A major goal in writing *Moldy* was that it be as portable as possible between different computers and operating systems. It is written in the Kernighan and Ritchie[24] compatible subset of ANSI C and assumes the library calls and header files defined for a hosted implementation of the standard. It should therefore be possible to compile and run *Moldy* on any computer which has a good C compiler.

There are two possible sources of difficulty in moving *Moldy* to a new machine. Though hosted ANSI standard C environments are now commonly available it may still be necessary to compile *Moldy* using a pre-ANSI compiler. In that case some library functions and header files may not be present. Secondly, to make good use of vector or parallel architectures, compiler directives or calls to specialized library functions are usually required. Replacement ANSI library functions are supplied in ansi.c for the VMS and unix (both Berkeley and AT&T system V varieties) operating systems. Different versions of a function are selected by the C preprocessor and conditionally compiled according to the pre-defined preprocessor symbols (see the documentation for your compiler). For ease of portability *all other system-dependent functions are in the module* auxil.c and *all preprocessor conditionals are in the header file* defs.h.

If the target machine has ANSI conformant C libraries, all that must be done is to define the preprocessor symbol `ANSI_LIBS`, either in defs.h or by using a compiler option *e.g.* `-DANSI_LIBS`. This is done automatically in defs.h for several machines known to have conformant environments. If the target operating system is the system V variant of UNIX, the preprocessor symbol `USG` is defined automatically in defs.h. It is possible that some environments may defeat the selection making it necessary to define it by hand, either in defs.h or by setting the compiler option `-DUSG` in the makefile.

4.2.1 System Dependencies

In this section, details of system-dependent functions are described for the major operating systems.

Replacement ANSI header files The ANSI header files string.h, stdlib.h, stddef.h and time.h are missing from Berkeley unix, or incomplete. Replacements are included which may be dispensed with on an ANSI conformant system - If the symbol `ANSI_LIBS` is defined they simply include the system version.

Replacements for ANSI functions

- The ANSI function to delete a file, `remove()`, the signalling function `raise()` and the string functions `strstr()` and `strerror()` are missing from pre-ANSI libraries. Replacements are supplied in ansi.c.
- Replacements are provided in ansi.c for functions `memset()`, `memcpy()` and `strchr()` which are missing from Berkeley UNIX.
- The function `vprintf()` is often absent from older libraries. Replacements are provided which *a)* call the internal function `_doprnt()` or *b)* implements a portable `vprintf()`. Use the preprocessor macros `HAVE_VPRINTF` or `HAVE_DOPRNT` to select which.

Timing routines The supplied `clock()` function on 32-bit UNIX systems resets to zero after 36 minutes. Replacements, called `cpu()` for system V and Berkeley UNIXes and POSIX are supplied in auxil.c. The function `rt_clock()` is also defined and returns the elapsed time in seconds. For a non-unix system `cpu()` and `rt_clock()` simply call the ANSI functions `clock()` and `time()`.

File manipulation routines Auxil.c contains the functions `replace()` and `purge()`. `replace()` renames a file, making a backup of any existing file of that name. `purge()` removes the previous or backup version of a file. These functions make use of the file name syntax of the host operating system and are therefore system-dependent. Unix file systems do not have explicit version numbers but *Moldy* keeps a single previous version by appending a “%” character to the name. The pure ANSI versions just interface to `rename()` and do nothing respectively.

4.2.2 Optimization and Vectorization

Moldy has been designed to run fast on a wide range of computers, and in particular on those with vector, multiprocessor and parallel architectures. This is a difficult problem, since the constructs which run fast on different architectures may be quite distinct and occasionally in conflict. Nonetheless, it has been found that following a few basic rules gives extremely good performance on a wide range of computer architectures. In a rough order of importance these are:

1. Minimize the number of memory references to floating point data in critical loops. Memory access is the major bottleneck on almost every modern computer, scalar, vector or parallel.
2. Minimize the number of memory references to floating point data in critical loops. This cannot be emphasized enough.
3. Ensure that memory is accessed contiguously within critical loops. That is, arrays should be accessed with a stride of 1 and with the last index varying most rapidly.[3] This is absolutely critical on machines where memory is accessed via a cache, *i.e.* all workstations and many parallel systems, and frequently very important on machines with interleaved memory (*i.e.* most vector machines).

[3] C uses the opposite convention to FORTRAN in storage layout of multidimensional arrays

4. If the value of any array element is used more than once in a loop, write the loop using temporary scalars to store results and assign them to the arrays at the end of the loop. This allows the compiler to optimize memory references.[4]

5. Minimize the floating-point operation count in critical loops.

6. Minimize integer arithmetic in critical code. CRAY vector machines in particular have no integer multiplication hardware, and integer operations are slow as a result.

The performance of *Moldy* has been carefully studied using profiling tools, and all critical regions of code are written as efficiently vectorizable loops.

The most critical sections of code (*i.e.* those which use the majority of the computer time) are all to do with the site forces calculation. Thus it is the inner loops in force.c, ewald.c and kernel.c to which most attention should be paid. The pair-distance loop of `rdf_calc()` in rdf.c should vectorize for efficient radial distribution function evaluation. Others which are of minor importance are in beeman.c, matrix.c, quaterns.c and algorith.c. Auxil.c contains alternative versions of various sum, dot product, scatter and gather routines *etc.* which are interfaces to machine-specific libraries *e.g.* Cray scilib, Convex veclib (which usually have FORTRAN calling conventions). There are also default versions coded in C which do vectorize, for machines lacking specialist libraries as well as for scalar computers.

4.2.3 Optimization for Vector Architectures

The program should, of course, be compiled with options specifying vectorization. Since highly optimizing and vectorizing compilers frequently contain bugs, and since some options generate "unsafe" optimizations, it may be necessary to restrict the highest optimization level to those modules which contain critical code.

To allow the compiler to generate vector code, it must be instructed to ignore apparent vector recurrences. The reason is that the run-time dimensioned arrays necessary to implement such a flexible program must use pointers as their base. (See any C textbook, *e.g.* Kernighan and Ritchie[25, Chapter 5] for an explanation of C pointers and arrays.) Unfortunately this means that the compiler can not determine that each iteration of the loop is independent of the preceding iterations. In the jargon of vectorizing compilers, there may be a *vector dependency* or *recurrence*. The compiler can be notified that these are not genuine recurrences either globally by use of a command-line directive or on a per-loop basis using machine-specific compiler directives inserted into the source.

Most compilers also have an option which directs it to ignore recurrences throughout the whole program, *e.g.* `-va` on the Convex, `-va` and `-h ivdep` on the Cray compilers. It should normally be safe to use these options. Each manufacturer's compiler has its own peculiar set of inline directives. For example the CRAY compilers use a `#pragma ivdep` statement whereas the convex and Stellar compilers use a "significant comment" `/*$dir no_recurrence*/`.[5]

4.3 Modifying Moldy

4.3.1 Adding a New Potential

By default *Moldy* supports potential functions of the Lennard-Jones, six-exp and MCY forms. However it should be very easy to add further types. The program is written in a highly modular fashion so that *the only code which need be altered is in file* kernel.c (and occasionally in defs.h).

[4]Technically, the C standard treats arrays passed as formal function parameters as pointers which are permitted to refer to overlapping areas of memory. The compiler must therefore assume that if an array element is written in a loop then elements of any other arrays may also be changed. It must therefore reload from memory even though it already has a copy of the value in a register. But if all loads are completed before any stores then the compiler is at liberty to re-use the values and save memory accesses.

[5]A mechanism is provided to insert appropriate directives using the C preprocessor. The text `VECTORIZE` has been placed before each loop which ought to be vectorized, and the file defs.h contains machine-conditional `#define`s to replace it with the appropriate directive. Currently directives for the CRAY, Stellar and Convex compilers are included, and null text is substituted for other machines. Notice that in each case the substituted text is *not* the directive described in the manual, but rather that directive *after* it has been passed through the preprocessor. To determine what should be substituted on a new vector machine, create a small test containing the documented directive and use the C preprocessor on that file. The output will show the form that should be defined in defs.h.

Unfortunately this method had been made obsolete by the ANSI C standard, which makes it impossible to insert pragmas using the preprocessor.

The calculation of the potential and forces is performed entirely in the function `kernel()`. This function is called repeatedly with a vector of (squared) distances between some reference site and its neighbour sites. Vectors of potential parameters and charges are supplied which bear a one to one correspondence with the elements of the distance vector. It calculates the corresponding values of $\frac{dU(r_{ij})}{dr_{ij}}$ which it stores in `forceij[]`. There are several variants of the force-calculation loop, one for each kind of potential. The potential type in use is passed as a parameter to `kernel()` and is used in a `switch` statement to select the appropriate code.

To add a new potential the array of structs called `potspec[]` must be extended. The new array element should contain the name of the new potential (against which, the names given in system specification files will be matched) and the number of potential parameters for each site pair.[6] In parallel with `potspec[]`, the array `pot_dim[]` must also be updated with a new entry which describes the dimensions of each parameter for the new potential. It is used by the input routines to convert from the input units into program units. The entry consists of triplets containing the powers of mass, length and time, one for each parameter. Then define a new preprocessor symbol to the index of the new type in the array `potspec[]` (after the line `#define MCYPOT 2`). The value must correspond to the index of the new entry in `potspec[]` starting from 0 in accordance with the usual C convention. This constant should be used to define a new case in the `switch` statement of `kernel()`, and this is where the code to evaluate the potential goes.

The existing cases may be used as a model, especially for the evaluation of the electrostatic term $\mathrm{erfc}(\alpha r)/r$ which is evaluated by the polynomial expansion of Abramowitz and Stegun[1, section 7.1.26]. There are currently *two* versions of each loop, the second omitting this term for efficiency when all the electric charges are zero (which case is flagged by a negative value of α).

Finally, the distant potential correction for the new potential should be added as a new case to function `dist_pot()`. The code should evaluate

$$-\int_{r_c}^{\infty} r^2 U(r)\,\mathrm{d}r$$

for the potential $U(r)$.

4.3.2 Porting the Parallel Version

It should be relatively straightforward to port the distributed-memory parallel version to a new message-passing library. Section 5.4.1 describes the parallel implementation. All of the interface code is contained in parallel.c and it will only be necessary to modify this file. A new port should declare a new preprocessor macro along the lines of `MPI` *etc.* which should be used to conditionally compile its code only. Any header files may be included in the appropriate place in parallel.c. Then the interface functions should be written to call the underlying message passing library. These should again be conditionally compiled. It should be obvious where to place them in the file and the existing versions will provide a model. Their specifications are:

`par_sigintreset(void)` Moldy sets a handler for SIGINT. This function is called from the signal handler to restore the default.

`par_begin(int *argc, char ***argv, int *ithread, int *nthreads)` Initialize the library and return the number of processes and the ID of this process.

`par_finish(void)` Terminate the parallel run normally.

`par_abort(int code)` Terminate the run abnormally. Return code if possible.

`par_broadcast(void *buf, int n, size_mt size, int ifrom)` Broadcast the specified buffer from node `ifrom` to all nodes.

`par_{r,d,i}sum(void *buf, int n)` Perform a global parallel sum reduction on the buffer containing n reals,[7] doubles or ints.

[6] By default the arrays are sized for up to seven parameters. If this is not sufficient, the limit, set by the value of the constant `NPOTP` defined in defs.h may be increased.

[7] `real` is a typedef defined in defs.h which is set either to `float` or `double` (see section 5.1.1). The code for `par_rsum()` must handle either case, which may be tested using the `sizeof` operator. For example the preprocessor macro `#define M_REAL (sizeof(real)==sizeof(double)?MPI_DOUBLE:MPI_FLOAT)` is used to determine which constant to pass to the MPI sum function.

`par_imax(int *idat)` Perform a global maximum reduction on the single int argument.

The SPMD parallel strategy updates the dynamic variables independently on all processors (see section 5.4.1). To update the separate copies of the co-ordinates and other dynamic variables synchronously the results of the floating-point arithmetic must be identical. Therefore the result returned by `par_rsum` and `par_dsum` must be identical to the last bit on all processors: see the footnote on page 55. Another consequence is that execution on heterogeneous workstation networks is not supported - the identity of floating-point operations in not guaranteed even if all use IEEE arithmetic. *Moldy* periodically tests for divergence of trajectories and will exit with the message

```
*F* Trajectories on parallel threads are diverging.
```

if this condition is detected.

Chapter 5

Program Structure

The source of *Moldy* consists of 31 different C source files amounting to 167 functions and 9000 lines of code. A complete and detailed description would be a compendious volume of questionable value. Instead, much of the detailed documentation is contained in the source code in the form of comments. This chapter concentrates on describing the organization of the calculation and the data structures used throughout the code, describes some of the more complicated and less obvious algorithms and provides call graphs and brief function descriptions to act as a map of the program structure.

Moldy is written in modular fashion in adherence to the principles of *structured programming* to as great a degree as practical. This does not mean merely the avoidance `goto`s but, instead the organization of the program into modules and functions which are independent of each other and of global environmental variables. Functions are kept to a limited size and as far as practical and serve a single, well defined purpose.[1] This ensures that the internal workings of a function are unaffected by changes elsewhere in the program, and do not have any influence on any other part of it, except through the defined interface. In *Moldy* functions are grouped into different files according to a rough classification of their purpose.

The other primary consideration in the design of a large computer program is the provision and organization of storage for the data. Structured programming advocates that definitions of data objects be restricted as closely as possible to the code that uses them, and that the code be organized in a modular fashion to encourage data locality. This minimizes the risk of a programming error in one part of the code modifying a variable used in a completely different part and producing difficult-to-locate side effects. With two exceptions,[2] global data is avoided in *Moldy* and all arrays are passed as function arguments where necessary. Heavy use is made of C structures (or "structs") to further group related data so that it may be manipulated *in toto*.

5.1 Data Structures

5.1.1 Types and Typedefs

A number of derived types are defined in the header file defs.h for reasons of economy of expression, portability and ease of customization. Most of these derived types are named using a suffix `_mt` to mark them as such. These are

`real` This is the "standard precision" floating-point type used for all the dynamic variables and most other internal variables. It is set to `double` by default. Changing this to `float` will give a single-precision version of *Moldy* with a consequent memory saving. However additional work must be done to create a fully single-precision version, since the standard double-precision maths library will still be used. The C standard does not require a single-precision maths library, but most systems do make the functions available. However there is no standardization of the interface so this can not be done portably.

`boolean` Used for storing logical values. Typed to `int`.

[1]The term "function" in C corresponds to both functions and subroutines in FORTRAN.

[2]These are the struct `control` (section 5.1.3) and the integers `ithread` and `nthreads` holding the parallelization parameters (section 5.4.3).

`gptr` Generic pointer type, set to `void` in the case of ANSI C. Pre-ANSI compilers do not support `void` so `char` is used instead.

`time_mt` This is used to store integer format times and dates as returned by the `time()` function. It is declared as `unsigned long` and is used because pre-ANSI compiling systems may not define the `time_t` type for this purpose.

`size_mt` Used for storage of C object sizes as returned by `sizeof`. Like `time_mt` this would be unnecessary if we were guaranteed an ANSI C compilation system.

`vec_mt` Array of 3 `real`s for holding a vector type.

`quat_mt` Array of four reals for holding a quaternion type.

`mat_mt` 3×3 array of reals for storing a matrix.

5.1.2 Memory Management

The allocation of memory for storage of the multitude of data required in a molecular-dynamics simulation is one of the main design criteria of the code. The general nature of the systems to be accepted by *Moldy*, namely the arbitrary mixtures of molecules with different numbers and kinds of atoms requires a number of large multidimensional arrays with system-dependent bounds in more than one dimension. It is impractical to declare these statically with dimensions fixed at some suitably large value because total memory use would then be infeasibly large. The availability of standard, portable, dynamic memory allocation was one of the major reasons the author chose to write *Moldy* in C.[3] *Moldy* uses C's capability of array emulation by pointers and heap-based memory allocation[25] rather than true C arrays which, like FORTRAN's, are restricted to bounds fixed at compile-time.

Much of the dynamic memory used in *Moldy* is allocated on entry to a function and deallocated before exit to emulate local, variably-dimensioned arrays. The main exceptions are the arrays of dynamical variables which are allocated once during the startup phase and not freed until program exit. All dynamic memory is allocated using the function `talloc()` which is a wrapper around the standard library function `malloc()`. Function `talloc()` simply calls `malloc()`, tests the return value and calls an error exit function if it failed to allocate the memory requested. Its interface takes advantage of the C preprocessor macros `__LINE__` and `__FILE__` to print out the location of a failing call, and a wrapping macro `aalloc()` is provided in defs.h for this purpose. This also contains other *x*`alloc()` macros customized for various data types. The complementary function `tfree()` calls the library function `free()` but also allows tracing of allocation and freeing for debugging purposes. All of *Moldy*'s memory-management functions are in the source file alloc.c.

Like FORTRAN, C only permits the declaration of arrays of size fixed at compile time. Unlike FORTRAN, C lacks any method of declaring arrays with adjustable innermost dimensions as function formal parameters. However through the use of pointer — array mapping and dynamic memory allocation, variable-sized multidimensional arrays may be emulated[25, p107],[40, pp 20–23]. Multidimensional array emulation is done by the function `arralloc()` (see Figure 5.1). This takes the size of the atomic data-type, the number of dimensions and the lower and upper bounds for each dimension as arguments and returns a pointer to the allocated pseudo-array. This is an array of *pointers* (to an array of pointers to ...) to the data area. The C mapping of array notation onto pointer syntax allows this construct to be referenced exactly as if it was a true multidimensional array. For example

```
double **r;
r = arralloc(sizeof(double), 2, 0, m, 0, n);
```

defines a 2D array, $m \times n$ of `double` so that `r[i][j]` is a reference to the i, j th element in the usual manner. This pseudo-array may be passed directly to a function, *e.g.*

[3] At the time *Moldy* was being planned in 1988 C was the *only* widely available language offering dynamic memory allocation. Fortran 90, which also offers dynamically declared arrays was standardized in 1991 and compilers only became common in the mid-nineties.

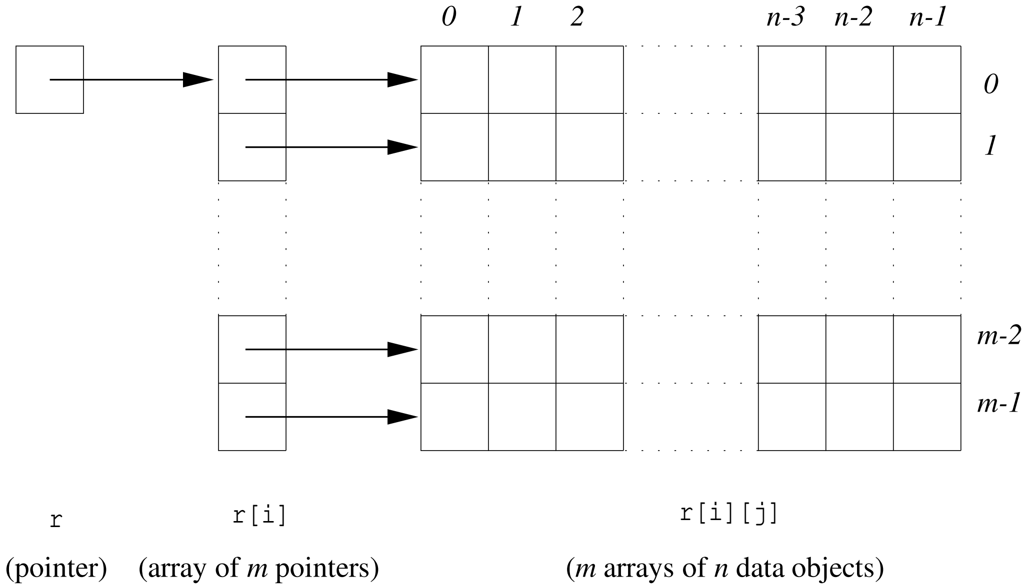


Figure 5.1: Storage layout of a 2-dimensional pointer-based pseudo-array. The base pointer `r` is of type "pointer to pointer to double" and declared `double **r`. This points at a 1-dimensional array with length *m* of pointers, which in turn point to the *m* rows of data arrays. This structure contains all the information needed to access element `r[i][j]` using pointer indirection rather than an indexing computation and therefore without any reference to the values of *m* or *n*. Higher dimensional arrays are set out on a similar fashion with more levels of pointer arrays, $n-1$ for an *n*-dimensional array

```
double doit(r, i, j)
double **r;
int i, j;
{
       r[i][j] = 0.0;
}
```

since the underlying pointer arrays contain all the necessary shape information to access it. Function `arralloc()` is implemented so as to lay out the individual pointer and data arrays as part of a single block of memory which is allocated by a single call to `talloc()`. It then sets up the values of the pointers to emulate the requested array. The memory can be freed with a single call to `tfree()`.

The `arralloc()` mechanism is unnecessary for 2D arrays where the innermost dimensions are fixed, such as for an array of position co-ordinates which has dimensions `[n][3]`. In such cases one of the *x*`alloc()` macros is used to allocate an array with fixed innermost dimensions and whose type is a pointer to a fixed-size array `double (*x)[3]` rather than a pointer to a pointer `double **x`.

There is one important instance of a more sophisticated use of arrays of pointers; the allocation of the atomic-site, site-force, centre-of-mass force and torque arrays in the main timestep loop function `do_step()`. These syntactically resemble 3D arrays declared, *e.g.* `site[nspecies][nsites][3]` but because the number of sites differs between molecular species they do not map onto "rectangular" 3D arrays. However the "array of pointers" construct does allow for rows of different lengths and this is easily set up. These pseudo-arrays can again be passed straightforwardly as function arguments and used with the obvious and intuitive indexing scheme at the cost of a little extra code to allocate and initialize the pointers.

5.1.3 System Variables and Data Structures

Moldy makes consistent use of C language structures or "structs" to combine related pieces of data into a single variable. This may then be passed as a single function argument avoiding long and cumbersome argument lists and the consequent risk of programming error. All of the major struct types used in this way are defined in the header file structs.h where each type is carefully annotated with the meaning of its members.

The struct `control` is shared between almost all modules in the program using external linkage.[4] It contains the values of the parameters from the control file — the exact mapping is defined by the array (of structs) `match[]` declared in source file startup.c. The values are read from the control file by function `read_control()` in source file input.c and adjusted by `start_up()` in startup.c where the timestep-related parameters are updated and the floating-point values are transformed into program units. These are the only functions which alter values in `control`.

Most of the information needed to describe the system is stored in the struct `system` and the array of structs `species[]`. Struct `system` contains counters to record the total numbers of species (`nspecies`), atomic sites (`nsites`), molecules (`nmols`), polyatomic molecules[5] (`nmols_r`) and pointers to the arrays of dynamical variables. Its counterpart `species[]` is a dynamically allocated array, length `nspecies`, of structs of type `spec_mt` which contains individual data for each molecular or atomic species. This includes the mass, moments of inertia, dipole moment, and charge, the number of sites belonging to this species and the number of molecules. It also contains pointers to the array of site identification numbers, and Cartesian site co-ordinates (expressed in the molecular principal frame) for this species and also to the arrays of dynamical variables.

The dynamical variables are stored in dynamically-allocated arrays of total length `nmols` and `nmols_r` which may be regarded as the concatenation of individual arrays containing variables belonging to each molecular species. The pointers in `species[i]` locate the beginning of the array of variables belonging to species `i` and those in `system` to the beginning of the entire array. The variables are manipulated by either route as is most convenient (but never both within a single function).

The potential parameters are stored separately in the array `potpar` since they refer to pairs of site types (*site-id's*). This is an array of structs of type `pot_mt`, laid out as a 2-dimensional symmetric matrix of dimension `system.max_id` with rows and columns indexed by the site identifiers as given in the system specification file. In fact it is stored in a one-dimensional array of length $(\texttt{system.max\_id})^2$ and the index calculation is performed explicitly when it is referenced. Structure type `pot_mt` contains an array `p` for the actual parameter values. This is a fixed-length array with `NPOTP` members, where the constant `NPOTP` is defined in the header file defs.h. If a new potential with more than 7 parameters is to be added to *Moldy* it will be necessary to increase its value.

The above primary storage of the potential parameters is convenient for reading in and storing and rereading restart files. However it can not be indexed efficiently to retrieve the potential parameter values in the critical innermost loops. The data is therefore copied into an expanded 3-dimensional array `potp[max_id][NPOTP][nsites]` in the force evaluation function `forces()` before being used. The masses and charges of the various sites are stored in another array of structs `site_info[]`, simply indexed by the site identifier. As with the potential parameters this is not convenient for access within the inner force loop, so the data are expanded in `do_step()` to fill an array `chg[nsites]` to allow direct indexing by the loop counter.

The two final data structures of importance are those containing accumulated data for computing the usual averages and the radial distribution functions. Both databases are considered private to their respective source modules values.c and rdf.c and are only accessed using the function interfaces provided in those modules.

The averages database provides a general and extensible scheme for storing partial sums and computing rolling- and time- averages of instantaneous values such as temperature, kinetic energy *etc.* It consists of two parts, a linear array `av[]` of structs of type `av_mt` (defined in values.c) and an array of classified types of data which contains pointers to `av[]`, the numbers of components, and to format strings and units conversion factors for output formatting and printing. This is the compile-time struct array `av_info[]`, again defined in values.c. To compute averages of a different quantity a new entry should be added to the array `av_info`, and another corresponding enum type to `enum av_n` in defs.h. The storage, retrieval and averages computation functions will then recognize the new type and may be called from within `values()` and `averages()`.

The array `av` itself is set up using the "struct hack" to allocate space for the rolling average data. The structure type `av_mt` contains as its final entry an array with one element `roll[1]`. When storage for `av` is allocated, the amount requested is calculated *as if* the member array was declared `roll[n]` with *n* equal to the rolling-average interval. The array member `roll[]` may then be used as if it had *n* members. This does mean that the array `av` can not be simply indexed using pointer arithmetic or array subscripting since the compiler's notion of the size of a pointer of type `av_mt *` is wrong. It is instead accessed solely via the pointers contained in array `av_info`.

[4]This is analogous to a COMMON block in FORTRAN.

[5]strictly speaking molecules with rotational degrees of freedom.

The general functions `add_average()` *etc.* in values.c provide the means to store and retrieve values and compute averages. The function `av_ptr()` is provided to make the size and base address of the averages database available to the restart file reading and writing functions in restart.c.

Radial distribution functions are calculated between all pairs of site types (defined by the site identifier given in the system specification file). Storage for the accumulating binned pair distances is provided by a 3D pseudo-array `rdf[idi][idj][bin]` with two indices for the site identifiers of the pair and one for the bin. This does not have the geometry of a true 3D array but is set up by function `init_rdf()` to economise on memory by making `rdf[idi][idj][bin]` an alias for `rdf[idj][idi][bin]`. Both the pointers and the actual data area are of course dynamically allocated. The latter is in a single block of memory whose address is stored in `rdf_base` so that it may be easily stored in and retrieved from a restart file. The function `rdf_ptr()` is provided to make the size and base address of the RDF database available to the restart file reading and writing functions in restart.c.

5.2 Files and Functions

5.2.1 Source Files

The source files consist of 8 ".h" header files and 23 ".c" code files which contain functions grouped in a modular fashion. A detailed description of the purpose and interface to each function is given in the source code in the form of comments, which should be regarded as the primary documentation for the function. An annotated list of files follows.

structs.h Definitions of structure types used throughout the program.

defs.h Main *Moldy* header file to be included in all ".c" files. It contains machine and operating-system configuration macros., physical constants, unit definitions and global (non-structure) type definitions.

string.h, time.h, stddef.h, stdlib.h These files are replacements for the ANSI C library header files of the same name, included here for portability to pre-ANSI environments.

messages.h Error messages file containing messages in the form of preprocessor macros. This may allow for non-English language customization.

xdr.h Includes the system headers for the usual External Data Representation (XDR) functions and prototypes of additional functions for XDR reading and writing of *Moldy*–specific structure types.

accel.c Contains `do_step()`, which implements the major part of the main MD timestep procedure plus functions for velocity rescaling and thermostatting.

algorith.c Contains functions to implement algorithms and computations related to the dynamical equations of motion. This includes the computation of molecular centre-of-mass forces, torques, the Newton-Euler equations and the constant–stress and –temperature functions. These functions all have an interface which makes no reference to the `system` or `species[]` structs.

alloc.c Contains functions for memory management; allocation, (`talloc()`) freeing (`tfree()`) and setting up of pointer-based multi-dimensional arrays (`arralloc()`).

ansi.c Replacements for certain functions required in any ANSI C standard library but missing from pre-ANSI environments.

auxil.c Machine- and OS-dependent support functions. Contains *(a)* Fast vector arithmetic functions, including interface to various vector libraries as well as pure "C" versions. *(b)* OS-dependent time and file manipulation functions.

beeman.c Functions implementing the separate stages of the modified Beeman algorithm, including basic steps and the updating of all the necessary dynamical variables.

convert.c Functions for converting the potential parameters and control file parameters from input units to program units and vice versa.

dump.c Functions for managing periodic trajectory *etc.* dumps.

ewald.c Evaluates the reciprocal-space part of the Ewald sum for the long-ranged Coulombic forces.

force.c Implements the calculation of the short-ranged forces and the real-space part of the Ewald sum using the Link Cell algorithm. This excludes the actual evaluation of the potential which is contained in `kernel()` in kernel.c.

input.c Functions for reading the control and system specification files and allocating the structures to hold the data read in from them.

eigens.c Matrix diagonalizer from the netlib archive.

kernel.c Contains `kernel()` which actually evaluates the potential given a vector of pair distances.

main.c Function `main()` is the main program which controls the set up and contains the main MD timestep loop that calls `do_step()` to do most of the work of a timestep.

matrix.c Functions for manipulating 3×3 matrices and applying to $3 \times n$ arrays of co-ordinates *etc.*

output.c Main output functions responsible for regular output sent to main output file. Also contains error-handling function `message()` which may terminate simulation.

parallel.c Interface between *Moldy* and various parallel message-passing libraries. Also contains functions to copy *Moldy*'s data structures from the input/output node to other nodes using the parallel interface.

quaterns.c Functions for manipulating arrays of quaternions.

rdf.c Contains `init_rdf()` which sets up RDF database, `rdf_accum()` which periodically bins pair distances and `rdf_out()` which calculates and prints all the radial distribution functions.

restart.c Functions for reading and writing the restart and backup files.

startup.c Primary initialization function `start_up()` plus subsidiary functions `allocate_dynamics()` to create dynamic variable arrays and `initialise_sysdef` to compute molecular and whole-system properties. Also contains functions to create initial configuration for system.

values.c Function `init_averages()` allocates and sets up pointers to averages database. Other functions store and retrieve instantaneous values of thermodynamic quantities and compute the averages thereof.

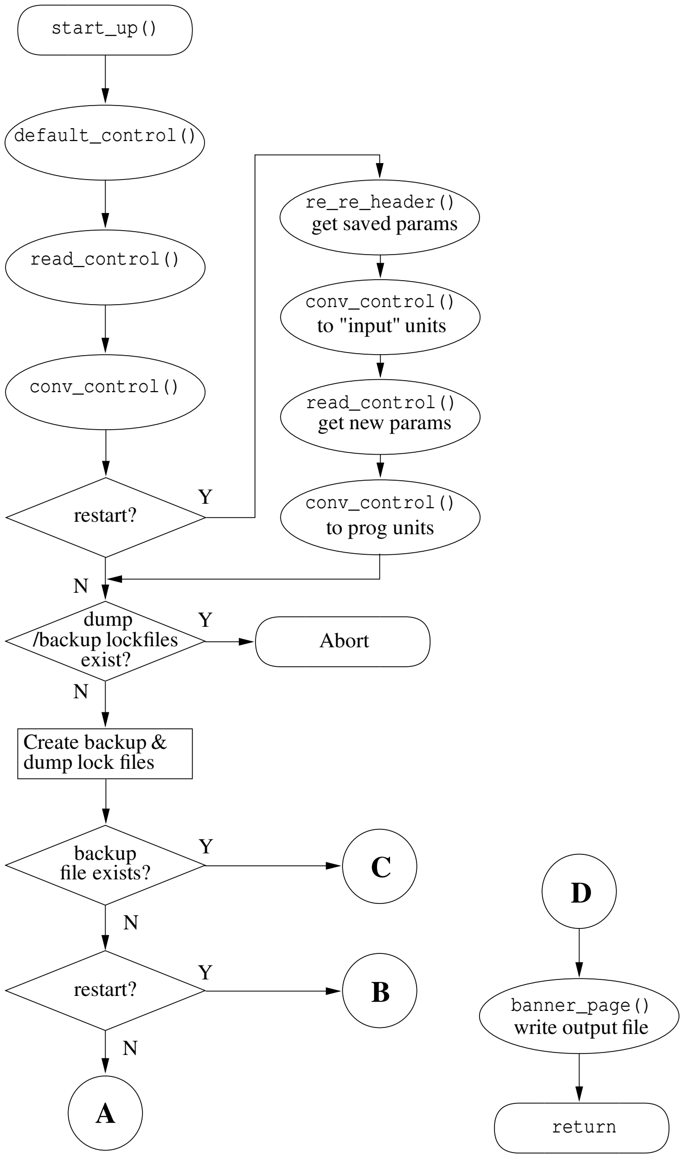
xdr.c Functions to read and write *Moldy*'s own struct types using the External Data Representation (XDR) library.

5.3 Flow of Control

5.3.1 Input and Initialization

The initialization and input file reading stage of *Moldy* is rather more complicated than is usual in a molecular-dynamics simulation program because

- of the use of dynamic data structures. The size of the arrays required can only be determined after the input files have been partially read in, but they must be allocated before the read is complete. Thus reading of input files and array allocation must be interspersed.
- of the general nature of the systems *Moldy* is able to simulate. Many structures that might otherwise be hard-coded must here be read from the input files and set up dynamically.
- *Moldy* has three different modes of start-up; an initial run, a restart and a backup restart. The use of the stored values of the control parameters as defaults on a restart run is an added complication.
- the ability to specify the input units the potential parameters are expressed in requires the appropriate conversion to be performed at start-up.



Functions called by start_up()

`default_control()` Initialize struct `control` with default parameter values.

`read_control()` Read parameters from control file and store in struct `control`.

`convert_control()` Convert physical values in `control` between input and program units.

`read_sysdef()` Read the system specification file. Allocate storage for arrays `species`, `site_info` and `potpar` and copy in values from file.

`initialise_sysdef()` Finish set up of `system` and `species` structs.

`allocate_dynamics()` Allocate memory for the dynamic variable arrays and set up pointers in `system` and `species`

`skew_start()` Set up skew-cyclic initial state.

`lattice_start()` Read crystal structure and set up co-ordinates, quaternions and cell vectors.

`init_cutoffs()` Determine optimum parameters for Ewald sum.

`re_re_header()` Read the `restart_header` and `control` structs from the restart file.

`re_re_sysdef()` Read the system specification (structs/arrays `system`, `species`, `site_info` and `potpar`) from the restart file.

`read_restart()` Read dynamic variables and the RDF and averages databases from the restart file.

`check_sysdef()` Check new system specification is consistent with old.

`thermalise()` Set up Maxwell-Boltzmann velocity distribution

`init_rdf()` Prepare to bin RDFs. Allocate memory and pointers

`init_averages()` Allocate space for and initialize the averages database.

`convert_averages()` Update averages database if roll_interval changed or if old-style restart file.

`conv_potentials()` Convert potential params between "input" and "program" units.

Figure 5.2(a): Block diagram of the initialization function `start_up()` and a list of the functions called. Continued in Figure 5.2(b).

Initialization is controlled by function `start_up()` which is called directly from `main()`. It calls subsidiary functions to read the control and system specification, restart or backup files. It controls the allocation of dynamic memory for the system description and potential and dynamic variable arrays, and the RDF and averages databases. It oversees the computation of quantities derived from the input parameters and system specification and generally transforms the input data from a form useful for human preparation to one useful for machine computation. Figures 5.2(a) and 5.2(b) show a block diagram of `start_up()` and a brief description of the functions it calls.

Parameters are read from the control file by `read_control()` which assigns their values to the corresponding member of struct `control`. The default values were previously set by `default_control()`. In the case of a restart the saved parameters in the restart file are restored to `control` by function `re_re_header()`, overwriting all of the current values. In this way the saved values become the defaults for a second call of `read_control()` which rereads the control file and assigns any new values. The repeated read is necessary because the *name* of the restart file is supplied by the control file. Note that those parameters representing physical quantities must be converted to and fro between input and program units for consistency since they are stored in program units in the restart file.

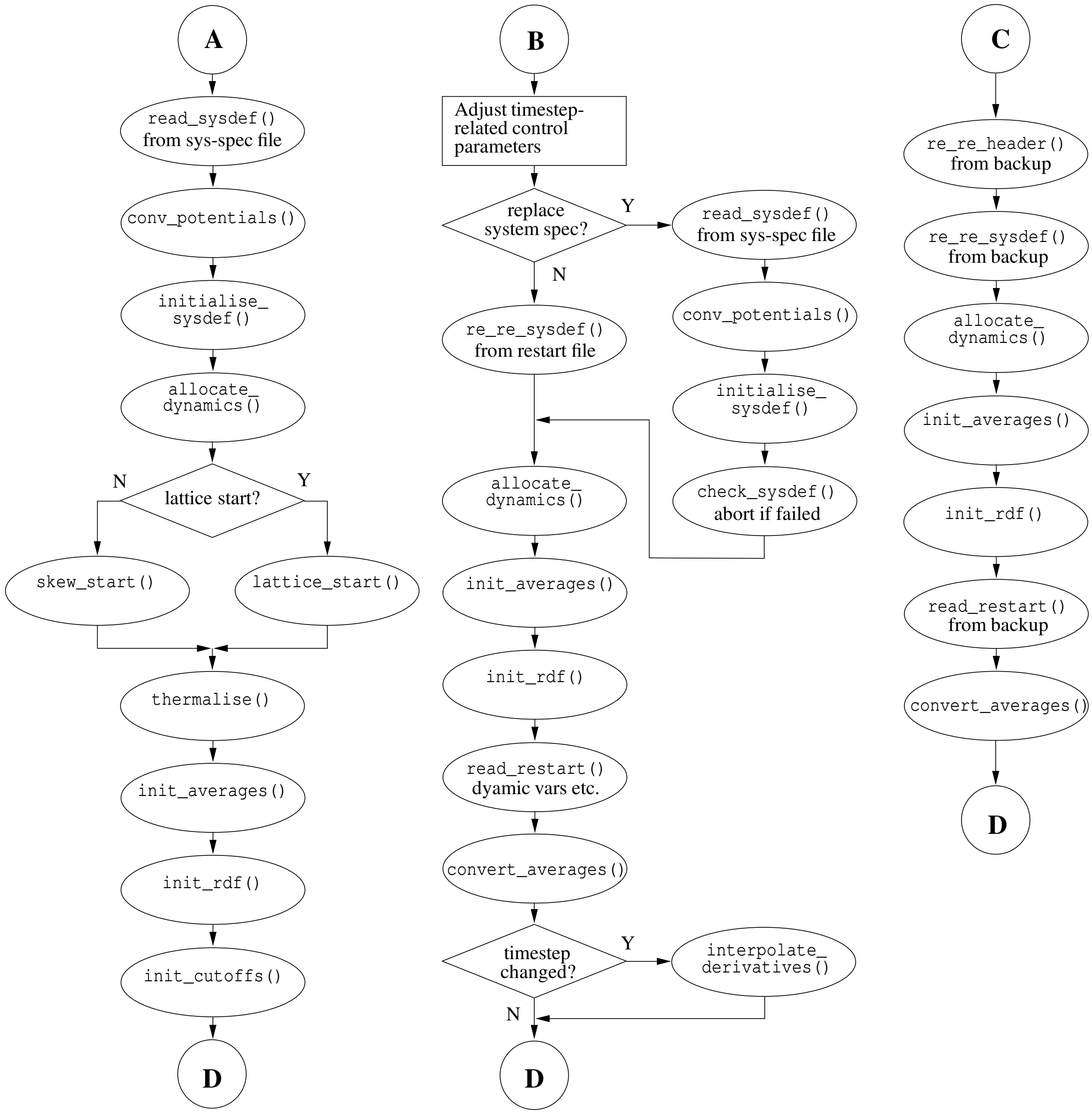


Figure 5.2(b): Block diagram of the initialization function `start_up()` and a list of the functions called. Continued from Figure 5.2(a). The paths beginning at **A**, **B** and **C** are for a new run, a restart from a save file, and a restart from a backup file respectively.

The three alternative routes within `start_up()` for reading the system specification and initializing the dynamic variables *etc.* are shown in Figure 5.2(b). The case of a new run is reasonably straightforward. Memory for the "system specification" arrays `species[]`, `potpar[]` and `site_info[]` is allocated as the system specification is read in by `read_sysdef()`. The raw atomic site co-ordinates are then shifted to the molecular centre-of-mass frame and rotated to the principal frame of the inertia tensor by `initialise_sysdef()`, which also computes a number of other molecular quantities and completes the set up of the system specification. Next the dynamic-variable arrays are allocated and initialized and finally the RDF and averages databases are installed.

If the run is a restart then those control parameters whose value is interpreted relative to the current timestep (see section 3.1) must have its value added to theirs. This is only done if the parameter was explicitly specified in the new control file, otherwise its saved value is untouched. Function `re_re_sysdef()` reads the system specification variables and arrays `system`, `species[]`, `potpar[]` and `site_info[]` from the restart file. These were stored in their final form after being set up by the previous run so a call to `initialise_sysdef()` is unnecessary. Alternatively a new system specification file may be used in which case a similar sequence to that for a new run is followed. Memory for the dynamic variable arrays and the RDF and averages databases must be allocated before their saved values are restored by `read_restart()`.

The simplest startup mode is that from a backup file. Once it has been determined that a backup file exists, setup proceeds much as in the restart case except that the parameters and system specification are restored unchanged. Thus the run continues exactly from the point the backup file was written. A restart file is still opened, but only to obtain the name of a backup file to search for.

The final act of `start_up()` is to call `banner_page()` to print out a nicely formatted page of information to record the simulation details in the output file. Control is then transferred back to the main program where everything is now ready for the molecular dynamics to begin.

5.3.2 Main timestep loop

The initial entry point into *Moldy* is into function `main()`, which calls `start_up()` (section 5.3.1) and then proceeds into the main loop over molecular-dynamics timesteps (Figure 5.3). This loop calls `do_step()` which computes the forces and advances the co-ordinates, velocities and their angular counterparts by one timestep. Most of the periodic tasks not directly associated with progressing the simulation, such as the accumulation of running averages, velocity rescaling and writing the output files are called directly from `main()`. The exceptions are the binning of site-pair distances for computing the radial distribution functions which is integrated into the site forces evaluation, and the writing of trajectory data to the dump file which is called from `do_step()` since the force arrays are local to that function.

The timestep loop continues until either the requested number of steps have been completed, a `SIGTERM` or `SIGXCPU` signal is received or one more step would exceed the CPU time limit set in `control.cpu_limit`. Only the normal termination case is shown in Figure 5.3. In the case of an early termination `write_restart()` is called to write out a backup file so the simulation may be restarted later. The signal handling functions `shutdown()` for `SIGTERM` and `SIGXCPU` and `siglock()` for other signals are installed in `main()` following the call to `start_up()`. Function `shutdown()` simply sets a flag which is checked at the end of every iteration of the timestep loop. This allows for an orderly shutdown if the CPU limit is exceeded or if the run must be interrupted for some other reason. For conditions which require immediate program exit, function `siglock()` is called to delete the lock files before exiting.

The tasks of calculating the forces, implementing the equations of motion and updating the dynamic variables each timestep are managed by function `do_step()`. The conceptually simple calculation of forces and stepping of co-ordinates is made lengthy and less transparent by the need to handle the rigid-molecule equations of motion and the constant-pressure and -temperature extended system equations. The procedure is set out in figures 5.4(a) and 5.4(b). The main stages of the calculation are

a. update all of the centre-of-mass co-ordinates, quaternions and extended-system variables according to steps *i* and *ii* of equations 2.13. This is done by function `step_1()` which also applies the quaternion normalization and constraints of equations 2.6 and 2.11.

b. call `force_calc()` and `ewald()` to evaluate the potential energy and the forces on all atomic sites. The site co-ordinates must themselves be computed from the centre-of mass co-ordinates and the quaternions by function `make_sites()`.

c. calculate the molecular centre-of-mass forces and torques using equations 2.1 and 2.2 which are implemented in functions `mol_force()` and `mol_torque()`. At this point the accelerations from the previous timestep are "shuffled down" from the `acc`, `qddot` *etc.* arrays in `system` and `species[]` to `acco`, `qddoto` to make way for the new accelerations computed in this timestep.

d. calculate the internal stress (equation 2.32) and the "accelerations" of the MD cell matrix from the Parrinello-Rahman equations of motion. Function `rahman()` implements equation 2.31.

e. calculate the new centre-of-mass and extended-system variable accelerations including the velocity-dependent parts which arise in a constant-pressure or -temperature calculation. Iterate steps *iii*–*v* of equations 2.13 until the velocities have converged.

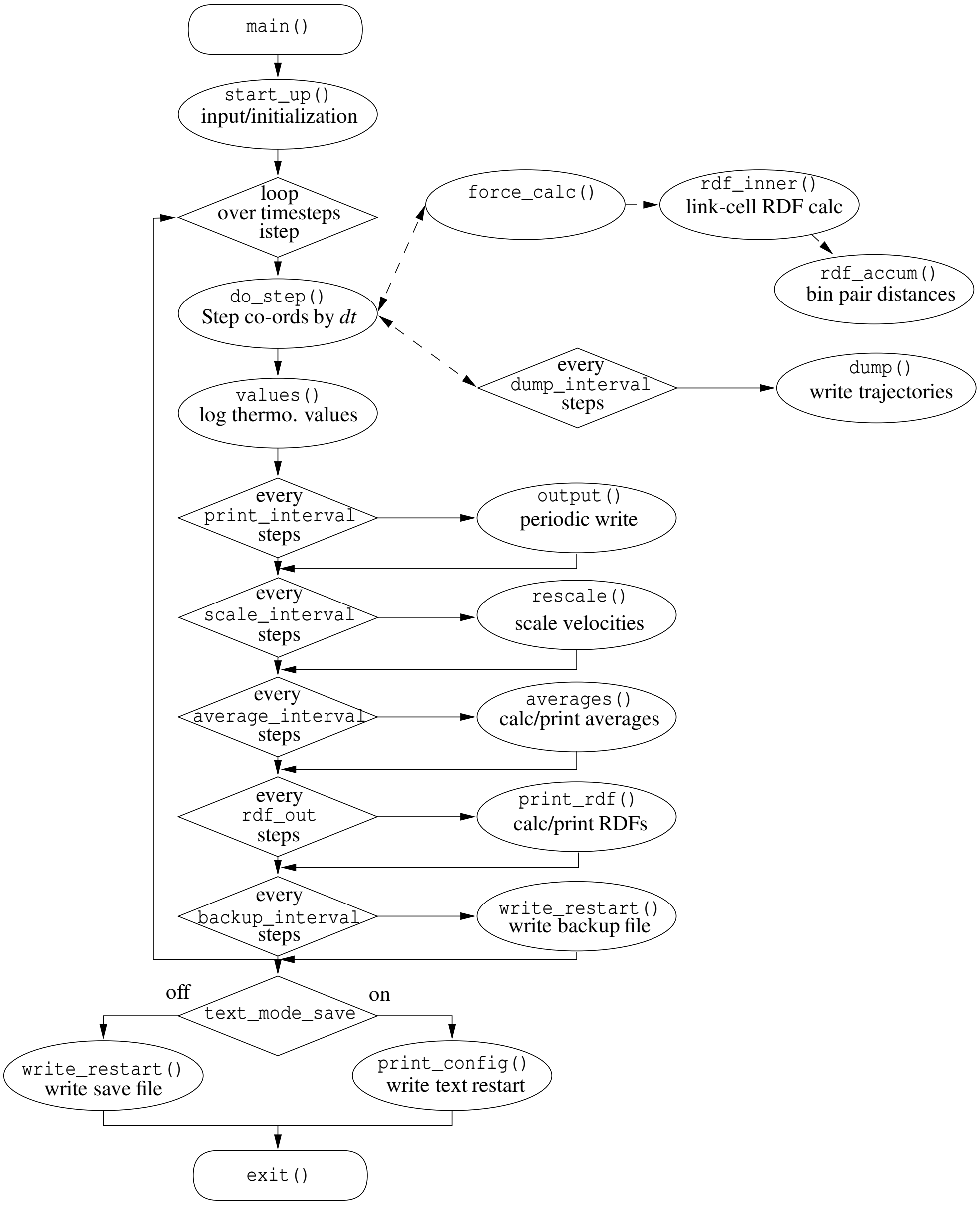


Figure 5.3: The main timestep loop in file main.c showing the outermost control structures. Periodic analysis, trajectory dump or output tasks are all shown here including the accumulation of the radial distribution function data which is integrated into the link cell force calculation.

f. calculate the new angular and rotational thermostat accelerations. The Euler equations (2.4) and equation 2.10 are used by function `euler()` to evaluate the second time-derivatives of the molecular quaternions. Iterate until the corresponding angular velocities have converged.

g. call `step_2()` to apply the final step *iv* of equations 2.13.

h. write requested co-ordinates *etc.* to the dump file.

The loops of steps *e* and *f* iterate until the molecular centre-of-mass velocities and quaternion time derivatives have adequately converged (in practice about 3 or 4 iterations). No additional iteration is applied to converge the extended-system dynamic variables, since in any practical system these should be very slowly-varying compared to molecular motion and any error introduced will be very small.

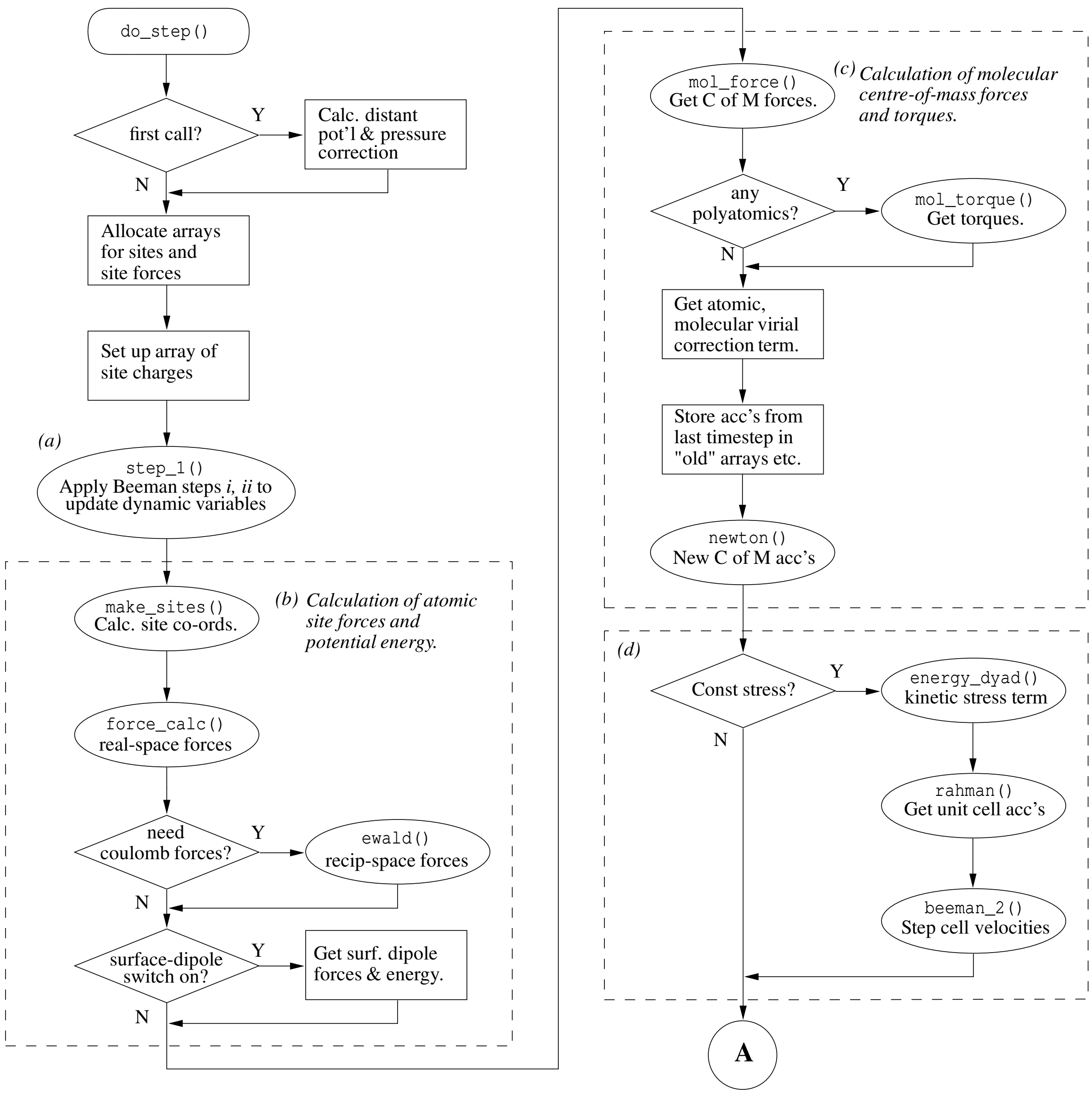


Figure 5.4(a): Flow diagram of function `do_step()` which performs a single timestep. (*continued in Figure 5.4(b)*)

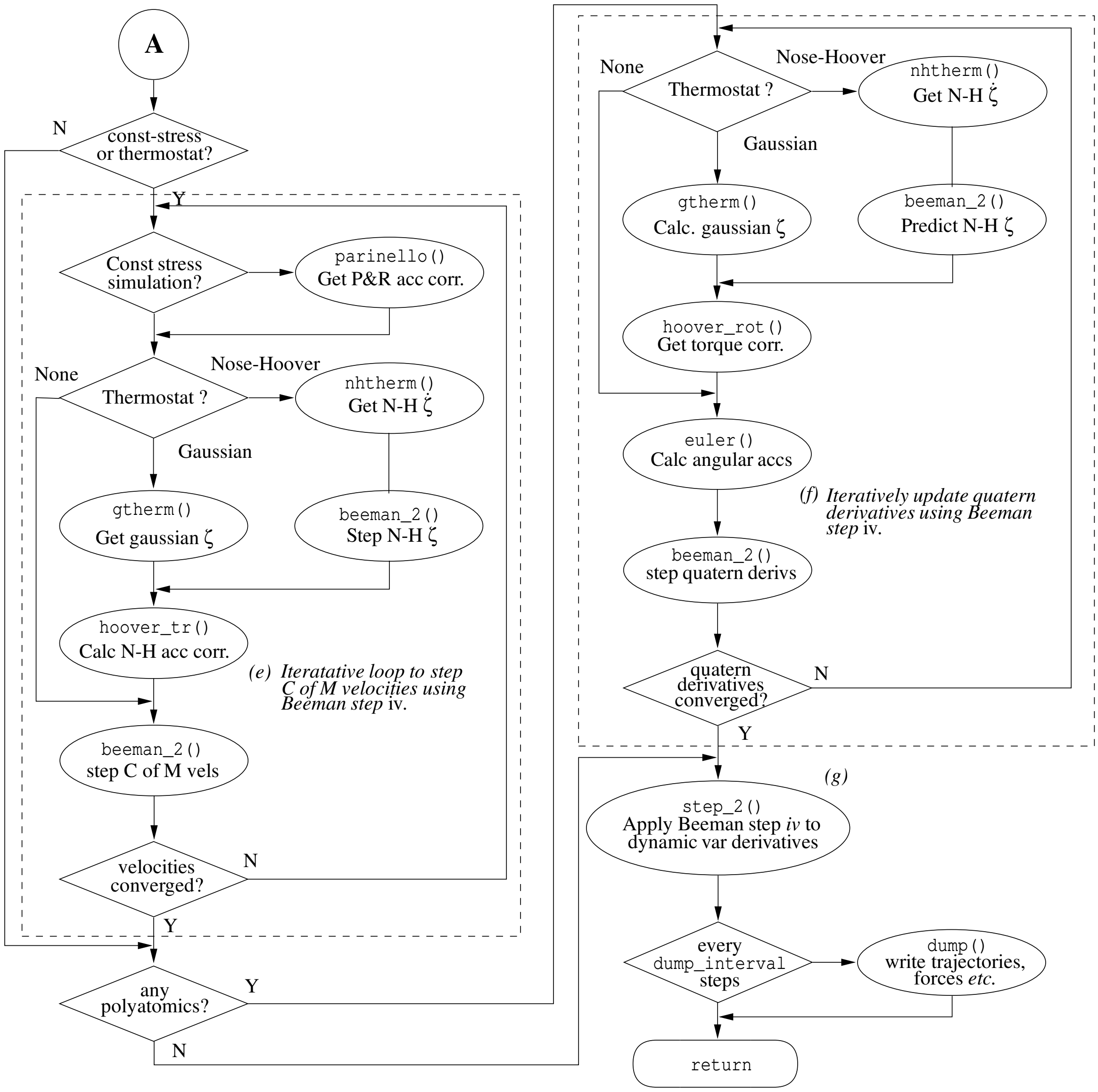


Figure 5.4(b): Flow diagram of function `do_step()` which performs a single timestep. (*continued from Figure 5.4(a)*)

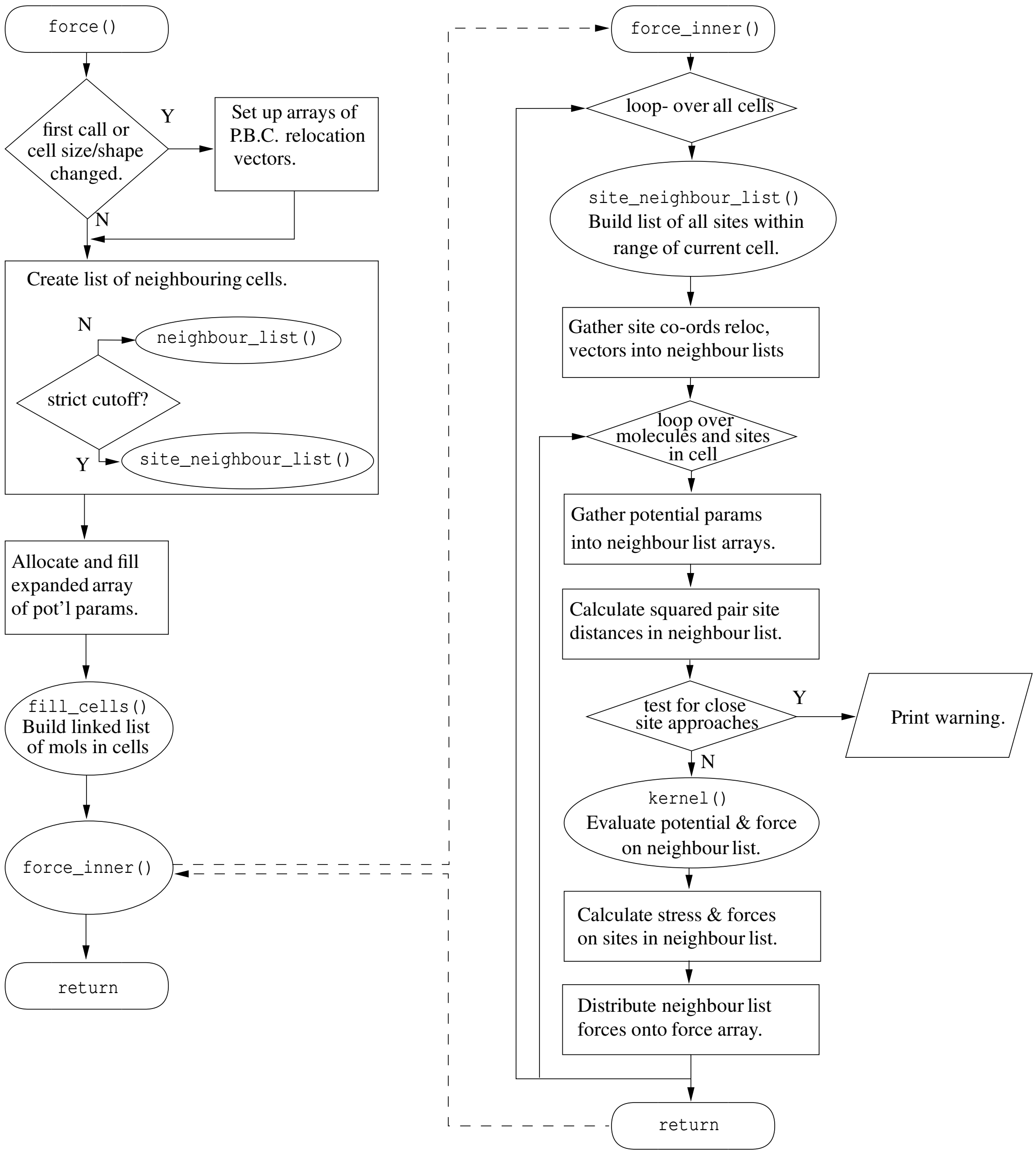


Figure 5.5: The Link-Cell short-range force calculation

5.3.3 The Site-Forces Calculation

The evaluation of the forces on atomic sites comprises almost all of the run-time of any reasonably sized simulation — over 95% for systems of only a few hundred atoms. Efficiency of execution is therefore a primary consideration in this part of the code. It is frequently the case that a straightforward implementation of an algorithm is not optimally fast, which regrettably means that the optimized code is not as transparent to read as for other, less critical parts of the program.

The link cell algorithm for the short-ranged part of the forces (section 2.5) is one which scales very well to large systems. In its original form [41] the MD cell is divided into a number of "subcells" and a linked-list structure [27] lists the molecules or atoms belonging to each subcell. The inner loops of the force calculation must therefore use an *indirect* indexing operation to obtain the co-ordinates on every iteration: the index of a molecule on each iteration depends on the index of the previous iteration. This kind of loop is inherently un-vectorizable which

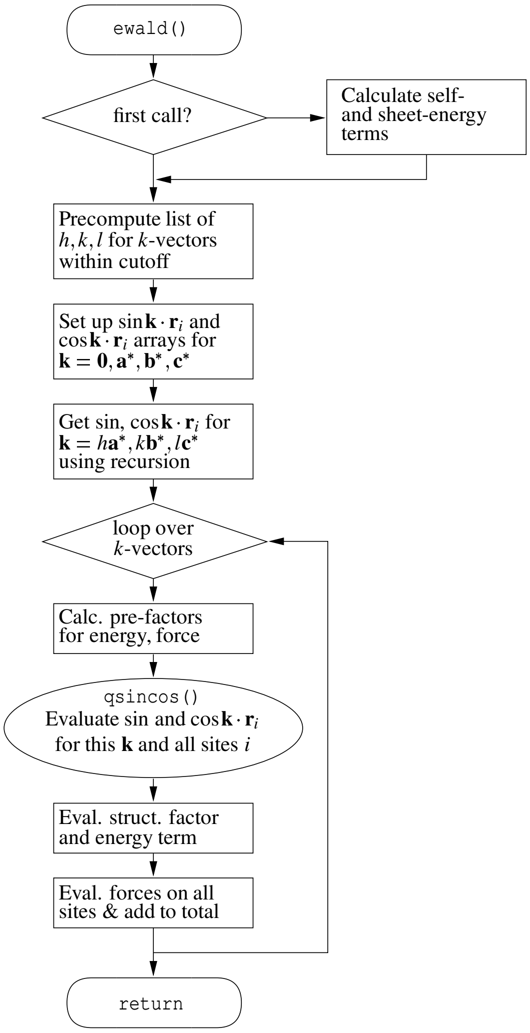


Figure 5.6: Evaluation of the reciprocal-space parts of the ewald sum

grossly limits the efficiency on vector computers.

To overcome this limitation a further step is needed, as suggested by Heyes and Smith [19]. Inside the loop over subcells, the linked list is pre-scanned and used to construct an array containing the indices of the molecules in the list. This is known as the *neighbour list* since it points to the molecules in the region of the subcell under consideration. The co-ordinates are then assembled into a contiguous array using a *gather* operation. The inner force loop then becomes a straightforward DO loop over this temporary array which *is* vectorizable. In addition to the co-ordinates, a corresponding gather operation using the same neighbour list operates on the electric charge and potential parameter arrays. The corresponding temporary arrays are also accessed in the inner loop over neighbour sites with a single array subscript. Finally the computed forces are added to the main site force arrays using a *scatter* operation, inverse to the gather.

Although this algorithm was designed for vector machines, it is also highly efficient on modern RISC processors which universally employ a cache-based memory architecture. Since adjacent iterations of the inner loop access adjacent locations in memory, loading a cache line will bring the operands of the next few iterations into cache. Otherwise a (very slow) cache load would be necessary on every loop iteration.

The link cell functions are all in `force.c` (figure 5.5). The outer-level function is `force_calc()`. Function `neighbour_list()` constructs a list of subcells within the cutoff radius of a reference cell. By adding an offset and applying periodic boundary conditions this list yields the neighbour cells of *any* subcell. This should not be confused with the neighbour *sites* list above. In fact the list contains only a hemisphere of cells since Newton's third law is exploited to halve the computational cost. A more rigorous list may be computed instead by `strict_neighbour_list()` which is selected in strict cutoff mode (see section 2.5.2). Next an array of potential parameters called `potp` is constructed. This has an innermost dimension of the number of sites, which maps one-to-one onto the co-ordinates array. It will be used as the argument of the gather operation using the site neighbour list. Function `fill_cells()` constructs the linked list of molecules in each subcell.

The loops over cells are all contained in `force_inner()`. Within the loop over subcells the neighbour list of *sites* for this cell is constructed from the list of neighbour *cells* by `site_neighbour_list()`. This list is then used as the index array to gather the co-ordinates, charges, potential parameters and any periodic boundary vectors. The innermost loops compute the site pair distances, the potential and the forces on the neighbour list sites. The actual potential and scalar-force evaluation is delegated to function `kernel()` for ease of comprehension and modification. This takes as its arguments an array of squared pair distances and corresponding arrays of potential parameters and site charges and returns the potential energy and an array containing the scalar part of the force, $\phi'(r_{ij})/r_{ij}$. The structure of `kernel()` is designed to evaluate all the different kinds of potential functions (see section 2.3.3) as efficiently as possible. It contains separate versions of a vector loop for each distinct potential type, both with and without electrostatic charges. For the charged case, the complementary error function is approximated to an accuracy of approximately 1 part in 10^{-6} by the formula given in Abramowitz and Stegun[1, section 7.1.26].

Pair distances for the calculation of radial distribution functions are evaluated by `rdf_inner()` whose structure resembles a simplified `force_inner()`. It does not call `kernel()` or compute forces but instead calls `rdf_accum()` which bins the computed pair distances. The separate function allows the use of a larger cutoff than for the force and potential evaluation.

The reciprocal-space part of the Ewald sum is evaluated in function `ewald()`, shown in figure 5.6. This is a fairly straightforward implementation of the k-space terms of equations 2.19, 2.20 and 2.24, and differs from common implementations only in the respect that it must correctly handle parallelepiped-shaped MD cells and therefore a triclinic grid of k-vectors. A list of vectors satisfying $\mathbf{k} > \mathbf{0}$ and $|\mathbf{k}| < k_c$ is pre-computed to simplify the control condition of the main k-vector loop.

Again, computational efficiency dictates much of the structure of `ewald()`. The values of $\cos(\mathbf{k}\cdot\mathbf{r}_i)$ and $\sin(\mathbf{k}\cdot\mathbf{r}_i)$ must be computed for every site i and every vector $\mathbf{k} = h\mathbf{a}^* + k\mathbf{b}^* + l\mathbf{c}^*$ with $|\mathbf{k}| < k_c$. Since the cosine and sine functions are relatively expensive to evaluate, the values of $\cos(h\mathbf{a}^*\cdot\mathbf{r}_i)$, $\cos(k\mathbf{b}^*\cdot\mathbf{r}_i)$ *etc.* are precomputed for each site i and $h = 0, 1, \ldots h_{\text{max}}$, $k = 0, 1, \ldots k_{\text{max}}$ and stored in arrays of dimension `[hmax][nsites]` *etc.* Then within the loop over k-vectors the trigonometric addition formulae are used to construct $q_i\cos(\mathbf{k}\cdot\mathbf{r}_i)$ and $q_i\sin(\mathbf{k}\cdot\mathbf{r}_i)$ using only arithmetic operations. That task is delegated to function `qsincos()` since certain compilers can optimize it better that way. The self-energy and charged-system terms of equations 2.19 and 2.24 are also evaluated in `ewald()`.

5.4 Parallelization

It has been predicted since the early 1980s that parallel computers would offer the very highest performance for scientific computing. That vision has been somewhat slow in coming about, in part due to the difficulty of writing programs to explicitly exploit unique and idiosyncratic architectures and the lack of portability or re-usability of the resulting code. It was often necessary to write programs in machine-specific languages [37, 5], use proprietary and obscure library calls [8] and be designed around a specific communications topology. Nevertheless molecular-dynamics simulation was one of the earliest applications of parallel computers and many simulations have been undertaken despite the difficulties [37, 8, 42].

The emergence and widespread adoption of the single-program multiple-data (SPMD) programming model and the standardization of parallel communications libraries in the 1990s has much improved the situation. In this model a parallel program is treated as a set of copies of a single program executing asynchronously and independently on different processors and which communicate using library calls. The program is written in a standard language and arranged to divide the calculation among the processors in some way so that each handles a subset of the operations. The model is completed by the addition of a library of communications routines which allow the separate parts of the calculation to be assembled to give the full, desired result. A significant advance was the availability of parallel libraries such as TCGMSG [18], PVM [15], BSP [31] and the new standard MPI [14]. Together these developments allow a program to be both parallel and sufficiently independent of architectural details to be portable to a wide range of parallel environments. These may include shared-memory and distributed-memory multiprocessors, and even networks of workstations.

Of course formal portability is no guarantee of good performance on hardware with widely differing processor and communications performance. That is a function of the granularity of the problem with respect to communications latency and volume of data with respect to communications bandwidth. Both of these depend on the nature of the problem and the parallelization strategy.

5.4.1 Parallel Strategy

Moldy uses the "replicated data" approach to parallel molecular dynamics [8, 46] whereby each processor maintains a complete set of dynamical variable arrays for all particles. The short-ranged and reciprocal-space force calculations are divided among the processors and a global sum operation is used to add the separate contributions to the forces and propagate a copy of the complete force array to every processor. This "parallel" part of the calculation dominates the execution time and shows a close to linear speed-up. It takes over 95% of the CPU time even for quite small systems of a few hundred particles. The remainder of the calculation, in particular the integration of the equations of motion is not parallelized but is performed redundantly on every processor simultaneously.[6] This "serial" part of the code will therefore limit the parallel speedup on large numbers of processors by Amdahl's law. However this part of the computational work only grows as N compared with $N^{\frac{3}{2}}$ for the forces calculation (see section 2.4.1). Consequently the "serial" part becomes an increasingly insignificant fraction of the computation as the system size increases. Since one might reasonably expect that in practice N is scaled with the number of processors the serial part does not seriously limit the parallel performance.

The advantages of the replicated data strategy are firstly simplicity of programming. Most of the code is identical to the serial program which greatly eases maintenance. Secondly load-balancing is much more straightforward than with domain decomposition methods. This will be discussed below. Thirdly the communications granularity is coarse, consisting of one global summation per timestep. This makes it very suitable for loosely-coupled processors such as a workstation cluster where the communications *latency* is high. Finally it is very efficient for small and medium-sized systems.

The big disadvantage is that the scaling to very large numbers of processors is poor, which will eventually limit the size of system which can be simulated. This is because both the amount of memory used per processor and the total amount of data to be communicated per processor increase linearly with N, the number of particles in the system. However in comparison with other supercomputing applications, molecular dynamics simulation uses relatively little memory and most supercomputers can easily accommodate simulations of many thousands of particles. Though scaling to multi-million particle systems would be desirable, most practical simulations of solids or liquids can be accomplished using systems of a few tens of thousands of particles or fewer.

5.4.2 Implementation

Two of the design goals are that the serial and parallel versions of *Moldy* can be built from the same code, and that the code is easily portable to a number of communications libraries. The C preprocessor macro `SPMD` is used to conditionally include the parallel code. All calls to the parallel library interface are protected by conditional-compilation and if `SPMD` is undefined at compile time then the serial version is built. These calls are limited to a very few modules — main.c for the parallel set-up, `do_step()` in accel.c for the global summation of the forces, virial and potential energy and `message()` in output.c for error handling. Furthermore the communications library is not called directly, but through interface functions defined in file parallel.c (these are listed in section 4.3.2). This means that only parallel.c need ever be modified if a port to a different communications library is needed, as the rest of the code sees a common interface. As supplied parallel.c contains implementations for the MPI library, the TCGMSG library, the Oxford BSP library and the Cray SHMEM library for the T3D/T3E series machines

[6] It is essential that the particle co-ordinates are identical on every processor and remain so throughout the simulation run. Since the equations of motion are chaotic the tiniest numerical difference in the forces between processors will cause the trajectories to diverge. It is not difficult to arrange that a global sum returns results identical to the last decimal place on all processors and this is guaranteed by the BSP, and cray SHMEM implementations and strongly recommended by the MPI standard. *Moldy* relies on this behaviour to ensure that no divergence occurs. If this were not the case it would be necessary to add a step to synchronize the co-ordinates periodically. This may also be an issue for running on networks of workstations - it is assumed that the numerical properties of all the processors are identical. The equality of the separate copies of the co-ordinates is checked periodically and the simulation exits if divergence is detected.

(see section 4.1.2). One of these is selected by conditional compilation of parallel.c with one of the preprocessor symbols MPI, TCGMSG, BSP or SHMEM defined.

5.4.3 Input/Output and startup

Since each processor runs an identical copy of the program executable, steps must be taken to ensure that input and output are handled by a single processor. Otherwise a parallel run would print P copies of every line if running on P processors. Indeed there are some parallel systems on which only one processor is allowed to perform input or output. It follows that all parts of the code which perform I/O must be aware of which processor this instance is running on. This is done using a global integer variable, ithread which contains the index (or "rank") of the processor. Another variable nthreads contains the value of P. These are set up in function par_begin() called from main() and passed by external linkage to every module which needs them. Processor 0 performs all input/output, which is arranged by testing ithread prior to any I/O call, *e.g.*

```
if( ithread == 0 && control.istep % control.print_interval == 0)
       output();
```

which is the code in main() used to print the regular output. These variables are also present in the serial code, where they are harmlessly redundant. In that case ithread is always set to 0 and nthreads to 1.

Parallel.c also contains the function replicate() to pass the system specification and dynamical variables from processor 0 to all of the others. This is called from main() after start_up() has returned. It allocates memory for the system specification and dynamical variables on all other processors using the same functions as start_up() itself and calls copy_sysdef() and copy_dynamics() to broadcast the data from processor 0 to all of the others. The function par_broadcast() defined earlier in parallel.c calls the underlying communications library to broadcast the data globally. On exit from replicate() *every* processor now contains a complete set of data and is ready to begin the run.

Error or warning conditions are handled by function message() in output.c if called with the FATAL or WARNING flags set in its argument list. In the serial case this prints out a failure message and calls exit() to terminate the run. In a parallel run, an error condition may occur on just one processor or on all simultaneously depending on precisely what caused the problem. Furthermore, if just one processor is involved, it may not be the I/O processor (rank 0). The approach of printing the message only from processor 0 is inadequate since there would be no indication that something has gone wrong. But neither is it appropriate to print on all, since that would result in a multiplicity of messages in some cases. The approach taken is that message() always prints the error irrespective of which processor is executing it, but that calls to message() are made conditional on ithread == 0 for those conditions known to occur on all processors simultaneously. If the condition signalled was FATAL then message() finally calls par_abort() which terminates the run on all processors.

5.4.4 Distributed forces calculation

The two parts of the forces calculation which must be parallelized are the short-ranged forces calculation and the reciprocal-space part of the Ewald sum. The outermost loop of the link cell forces calculation in function force_inner() runs over all subcells within the simulation cell (see figure 5.5). In the serial case, it has the form

```
for( icell = 0; icell < nx*ny*nz; icell++)
{
      [Evaluate forces on particles in cell icell]
}.
```

The iterations of this loop are distributed over processors by rewriting it as

```
for( icell = ithread; icell < nx*ny*nz; icell += nthreads)
{
      [Evaluate forces on particles in cell icell]
}.
```

where the integer variables ithread and nthreads contain the rank of the processor and the total number of processors respectively. Since ithread has a different value on each processor, the separate copies of the force arrays contain only the contributions from the set of subcells assigned to that processor. After completion of the forces calculation the individual arrays are summed in function `do_step()`. Similar partial contributions to the potential energy and the stress are also globally summed. Provided that the number of subcells is rather greater than the number of processors and the number of particles per subcell is fairly constant, this algorithm automatically ensures that the computation is divided evenly between the processors.

The reciprocal-space sum may be parallelized either by distributing the loop over k-vectors over processors [8, 47] or by partitioning the particles between processors. Smith [47] called the former strategy the "reduced k-vector list" (RKL) method and the latter the "reduced ion list" (RIL) method. *Moldy* contains implementations of both methods. In the RKL method, it is necessary to pre-compute a list of k-vectors within the cutoff sphere (see figure 5.6) which is stored in the array `hkl[]`. (Each element of `hkl[]` is a struct containing the values of h, k, l and the vector components of $\mathbf{k}$.) The loop over k-vectors then takes the form of a single loop over this list, which is easily sliced over processors exactly as the loop over subcells above.

```
for(phkl = hkl+ithread; phkl < hkl+nhkl; phkl += nthreads)
{
        [Compute the contribution of k-vector in phkl]
}.
```

At the end of this loop the force arrays contain the contributions to the force on each site from the particular set of k-vectors handled by that processor. The total forces are obtained by globally summing these arrays over processors. In fact since the same arrays are used to store both the short-ranged and long-ranged contributions to the force then only one global sum is needed for the forces. Since the contribution at each k-point is a sum of fixed length and takes exactly the same time to evaluate as any other, this algorithm is always optimally load-balanced.

There is also an implementation of the alternative, RIL parallelization strategy in file ewald-RIL.c, which is a direct replacement for ewald.c. In this case each processor handles a subset of the particles and all k-vectors. This involves an extra global sum within the loop over k-vectors to evaluate the total structure-factor for each $\mathbf{k}$.

5.4.5 Radial Distribution Functions

The accumulation of distances for the radial distribution function calculation is potentially costly since it is a site-pair property. It is therefore handled using the parallel link cell method exactly as for the site forces calculation by functions `rdf_inner()` and `rdf_accum()`. However the contributions from different processors do *not* need to be summed every time `rdf_inner()` is called, but may be left to accumulate separately. The data is globally summed by function `par_isum()` only when `rdf_out()` is called to calculate and print the RDFs or just before a restart file is written.

Appendix A

Example System Specifications

A.1 Argon

```
# LJ Argon - about as simple as you can get
# Parameters from Allen and Tildesley Table 1.1
Argon 108
1          0           0          0    39.948 0 Ar
end
Lennard-Jones
1 1 3.984 3.41
```

A.2 TIPS2 Water

This is the four-site water model of Jorgensen[22]. Only the oxygen site interacts via the Lennard-Jones potential, and the charge site, M, is displaced 0.15Å from the Oxygen.

```
# Modified TIPS2 water
Water 64
1          0            0          0 16    0 O
2  0.7569503            0 -0.5858822 1 0.535 H
2 -0.7569503            0 -0.5858822
3          0            0      -0.15 0 -1.07 M
end
lennard-jones
1 1 0.51799  3.2407
end
```

A.3 Aqueous $MgCl_2$ Solution

This is a three-component system consisting of MCY water[30], magnesium and chloride ions. The Mg^{2+} potential was fitted to the SCF calculations of Dietz and Heinzinger[9] and the Cl^- to the calculations of Kistenmacher, Popkie and Clementi[26]. Note that the potential parameters are expressed in kcal mol^{-1}, and the control file must set the parameter `time-unit=4.8888213e-14`.

```
# MCY Water/ Mg2+ / Cl - solution
Water 200
1          0            0          0 16    0 O
2  0.7569503            0 -0.5858822 1     0.717484 H
2 -0.7569503            0 -0.5858822
3          0            0    -0.2677 0 -1.434968 M
```

```
Magnesium 4
4         0            0          0 24.31 2 Mg2+
Chloride  8
5         0            0          0 35.45 -1 Cl-
end
MCY
1 1 1088213.2   5.152712        0               0
1 2 1455.427    2.961895        273.5954        2.233264
2 2 666.3373    2.760844        0               0
1 4 47750.0     3.836           546.3           1.253 # New values of Mg potl
2 4 111.0       1.06            0               1.0
1 5 198855.0    3.910           0               0
2 5 1857.0      2.408           77.94           1.369
4 5 28325.5     2.65            0               0
end
```

A.4 Quartz

```
# Quartz parameters from Van Beest, Kramer and Van Santen
# Physical Review Letters 64,(16) p1955 (1990)
# Units are eV, A, el chg. so time-unit=1.0181e-14
Oxygen 384
1       0       0       0       16      -1.2    O
Silicon 192
2       0       0       0       28.0855 2.4     Si
end
buckingham
1 1     175.0000        1388.7730       2.76000
1 2     133.5381        18003.7572      4.87318
2 2     0.0             0.0             0.0
end
4.903 4.903 5.393 90 90 120 4 4 4
Oxygen   0.415000     0.272000     0.120000
Oxygen   0.857000     0.5850000    0.453300
Oxygen   0.728000     0.143000     0.453300
Oxygen   0.143000     0.728000     0.880000
Oxygen   0.272000     0.415000     0.546700
Oxygen   0.5850000    0.857000     0.213300
Silicon  0.465000     0            0
Silicon  0.535000     0.535000     0.333300
Silicon  0            0.465000     0.666700
end
```

Appendix B

Utility Programs

Five utility programs are included in the *Moldy* package, mostly for manipulation and analysis of dump data (see section 3.7). They are easily compiled on unix systems using the makefile: the command is "`make` *progname*" for each program or "`make utilities`" to make the lot. Apart from *mdshak* which must be linked with several modules from *Moldy* itself they are all self-contained in a single source file. They are written with unix systems in mind using unix-style option arguments, but ought to compile and run under VMS if defined as foreign commands.

Several of them require you to specify lists of (integer) numbers, for example selected molecules, timeslices *etc.*, which share a common syntax for such options. The numbers 1, 5, 17 to 20 inclusive and 34,44,54...94 are selected by the command-line argument `-t 1,5,17-20,34-100:10`. Selectors are separated by commas and each one may be a range separated by a hyphen with an optional increment following a colon.

B.1 Moldyext

It is usual to plot the instantaneous values of energy, temperature *etc.* during the course of a simulation, for example to monitor the approach to equilibrium. *Moldyext* processes the periodic output produced by *Moldy* (see section 3.5) and extracts this information from each timestep recorded. It is presented in tabular form for input into plotting programs. The command is

`moldyext -f` *fields output-file1 output-file2 ...*

where *fields* is a list in the selector format above. The numbering runs from left to right, ignoring newlines and blank columns so that the translational KE's in figure 3.1 are fields 1 and 14 and the σ_{zz} component of the stress tensor is 30.

Moldyext can currently only extract information from the "instantaneous" section of the output, not the rolling averages or standard deviations.

B.2 Dumpanal

Dumpanal examines the dump files given as arguments and prints out the header information to help with identification. For example

`dumpanal` *dump-file1 dump-file2 ...*

B.3 Dumpconv

Dumpconv is a tool for moving binary dump files between computers of different architectures. It has mostly been superseded by the portable XDR format dump files introduced in version 2.1 (see section 3.3.2) but is retained in case of machines for which no XDR implementation is available. The command

`dumpconv` *binary-dump-file text-dump-file* creates a straightforward ASCII text file with a representation of the dump information in the binary file including the header. The command

`dumpconv -d` *text-dump-file binary-dump-file*

converts it back. Seven significant decimals are used which ought to retain almost all of the precision of the single precision binary floating-point numbers. You can also convert an old "native" format dump into XDR format using the `-x` option, *viz*

`dumpconv -x` *native-dump-file xdr-dump-file*.

The `-x` and `-d` options may be combined if the input is a text format dump.

B.4 Dumpext

Dumps are designed so that *Moldy* can take care of all the bookkeeping, perform data security checks and to divide a lot of data into manageable portions. It is therefore not in a convenient form for reading by other programs, especially FORTRAN ones. *Dumpext* is a program which processes dump files, extracts a subset of the information and outputs it in a form more suitable for reading by other programs. It is invoked with the command:

`dumpext -R` *nmols* `-Q` *nquats* `-c` *components* `[-t` *timeslices* `]`
`[-m` *molecules* `] [-o` *output-file* `] [-b]` *dump-file1 dump-file2* ...

The meanings of its arguments are

- `-R` the total number of molecules. This argument is compulsory.
- `-Q` the number of polyatomic molecules. This argument is compulsory.
- `-c` which pieces ("components") of the information in dump record to extract. This is a selector-format list and may list any combination of

  1. C of M positions
  2. quaternions
  3. unit cell matrix
  4. potential energy
  5. C of M velocities
  6. quaternion derivatives
  7. unit cell velocities
  8. C of M accelerations
  9. quaternion accelerations
  10. unit cell accelerations
  11. C of M forces
  12. torques
  13. stress tensor

  This argument is compulsory.
- `-t` which timeslices (or dump records) to extract. This is a selector format list and is the index in the whole dump sequence, not just a single file. Timeslices or dump records are sequentially numbered in the dump files from 1. Defaults to all timeslices.
- `-m` extract information for selected molecules. This is a selector list specifying the molecule index. Defaults to all molecules.
- `-o` name of optional output file. Defaults to standard output.
- `-b` selects binary output in single-precision floating point numbers. Defaults to ASCII formatted numbers.

If any of the compulsory arguments are omitted, you will be prompted to supply a value. In particular you must *always* supply the number of molecules and polyatomic molecules. (This information is not recorded in the dump header and is needed to determine the number of co-ordinates and quaternions in each dump record.) You must specify which pieces of information to extract using the `-c` option.

The dump files must, of course, be part of the same dump sequence; this is carefully checked. They may be supplied as arguments in any order; *dumpext* automatically determines the sequence from the information in the headers. This is not as pointless as it sounds, since the list may be generated using unix shell wild-cards which arrange them in alphabetical rather than numeric order.

The output is arranged in columns, one line per time slice. So if, for example you wish to extract positions and quaternions there will be 7 columns corresponding to $x, y, z, q_0, q_1, q_2, q_3$. Multiple components are printed in the integer order of the component, *not* the order specified with `-c`. If binary output is asked for with the `-b` option the order is the same. The numbers are written sequentially as single-precision floating-point numbers (in machine native format) without record delimiters. The records may be easily read in C with an `fread()` call with the appropriate number of bytes or from FORTRAN77 using direct-access unformatted read[1] of the appropriate length records[2] `OPEN(...,ACCESS=DIRECT,LRECL=nnn)`. In the above example there are $3 \times 4 + 4 \times 4 = 28$ bytes in each record.

B.5 Mdshak/Mdxyz

Mdshak was written as an interface between *Moldy* configurations and a molecular graphics program called SCHAKAL[23]. It will also, optionally, write the output in XYZ form, suitable for input to the freely available viewers *Xmol* and *RasMol*. The output format is sufficiently transparent that it should not be hard to modify it for any other molecular graphics package, but note that the *babel* package, also freely available on the Internet will convert between the XYZ file format and many others. There is also a SCHAKAL file viewer called "read_shak" written by the author for use with AVS, the general purpose visualization program, and available free from the International AVS Center.[3]

Mdshak writes a SCHAKAL file containing the co-ordinates of a single or of multiple time slices during a MD run. It can read configurations from (*a*) a system specification file plus lattice start, (*b*) a system specification file plus dump file (*c*) a restart file or (*d*) a restart file plus dump file. It can be driven either interactively or using command-line arguments. If the executable file is named mdshak then the default output is in SCHAKAL file format. If it is renamed to mdxyz then the default is XYZ file format.

In interactive mode you are prompted for the source of the system specification (sys-spec or restart file) and the configurational information (restart or dump file). But you must either redirect the output (using > on unix) or specify an output file with `-o` or the output will go to the screen!

Alternatively, the command line options are

`mdshak [-s` *system-specification* `]` | `[-r` *restart-file* `] [-d` *dump-file-format* `] [-t` *dump-range* `]`
`[-c] [-x]` | `[-b] [-i` *extra-text* `] [-o` *output-file* `]`

where the meanings of the options are

- `-s` read a system specification file
- `-r` read a restart file. Only one of `-s` or `-r` may be given.
- `-c` if reading a system specification file, skip any preceding control parameter info.
- `-d` read the configurational information from a dump file. The argument is a dump prototype name containing a `printf()` format string – see section 3.7.
- `-t` range of records in dump file, specified in "selector" format.
- `-i` this inserts its argument into the output file and is used to add extra SCHAKAL commands, such as BOX.

[1]The usual FORTRAN sequential unformatted read is not suitable as this expects record size information to be present. *Dumpext* can not write this as its format is entirely determined by the whim of the compiler writer. FORTRAN unformatted sequential files are not guaranteed to be portable even between different compilers on the same machine.

[2]Be aware that the FORTRAN77 standard does not guarantee what units the record length is specified in. Predictably some manufacturers' compilers use bytes and others words. Consult the documentation!

[3]It may be obtained using anonymous ftp from `avs.ncsc.org` or by writing to The International AVS Center, PO Box 12889, 3021 Cornwallis Road, RTP, NC 27709, USA.

- `-x` Write an output file in XYZ format suitable for *Xmol* or *RasMol*
- `-b` Write the atom co-ordinates in raw binary (unformatted) format.

Mdshak may be useful for more than just visualization purposes as it extracts atomic co-ordinates from the system specification, positions and quaternions. It is written so that only the output routine (called `schakal_out()`) is SCHAKAL specific. This is quite short and can easily be replaced with one tailored to a different purpose.

Bibliography

[1] M. Abramowitz and I. A. Stegun, *Handbook of mathematical functions*, Dover, New York, 1970.

[2] M. P. Allen and D. J. Tildesley, *Computer simulation of liquids*, Clarendon Press, Oxford, 1987.

[3] D. Beeman, *Some multistep methods for use in molecular dynamics calculations*, Journal of Computational Physics **20** (1976), 130–139.

[4] F. Berthaut, *L'énergie électrostatique de rèseaux ioniques*, J. Phys. Radium **13** (1952), 499–505.

[5] K. C. Bowler, R. D. Kennaway, G. S. Pawley, and D. Roweth, *An introduction to OCCAM-2 programming*, Chartwell-Bratt, 1987.

[6] K. Cho and J. D. Joannopoulos, *Ergodicity and dynamical properties of constant-temperature molecular dynamics*, Physical Review A **45** (1992), no. 10, 7089–7097.

[7] K. Cho, J. D. Joannopoulos, and L. Kleinman, *Constant-temperature molecular dynamics with momentum conservation*, Physical Review E **47** (1993), no. 5, 3145–3151.

[8] E. Clementi, G. Corongiu, and J. H. Detrich, *Parallelism in computations in quantum and statistical mechanics*, Computer Physics Communications **37** (1985), 287–294.

[9] W. Deitz, W. O. Riede, and K. Heinzinger, *Molecular dynamics simulation of an aqueous $MgCl_2$ solution*, Z. Naturforsch **37a** (1982), 1038–1048.

[10] D. J. Evans, *On the representation of orientation space*, Molecular Physics **34** (1977), no. 2, 317–325.

[11] D. J. Evans and S. Murad, *Singularity free algorithm for molecular dynamics simulation of rigid polyatomics*, Molecular Physics **34** (1977), no. 2, 327–331.

[12] D. Fincham, *Rotational verlet*, Information Quarterly - CCP5 **5** (1981), 6.

[13] D. Fincham, *Optimization of the Ewald sum for large systems*, Molecular Simulation **13** (1994), no. 1, 1–9.

[14] Message Passing Interface Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputer Applications **8** (1994), no. 3-4, 165.

[15] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam, *PVM: Parallel virtual machines, a user's guide and tutorial for networked parallel computing*, MIT Press, Cambridge, MA, 1994.

[16] H. Goldstein, *Classical mechanics*, 2nd ed., Addison–Wesley, Reading, MA, 1980.

[17] J. P. Hansen and I. R. McDonald, *Theory of simple liquids*, 2nd ed., Academic Press, London, 1986.

[18] Robert J. Harrison, *Tcgmsg send/receive subroutines*, Battelle Pacific Northwest Laboratory, 1994, Available by internet ftp from url ftp://ftp.tcg.anl.gov/pub/tcgmst/tcgmsg-4.04.tar.Z.

[19] D. M. Heyes, *Large numbers of particles on the Cray-1*, Information Quarterly - CCP5 **25** (1987), 57–61.

[20] W. G. Hoover, *Canonical dynamics: equilibrium phase-space distributions*, Physical Review A **31** (1985), 1695–1697.

[21] Sun Microsystems Inc, *External data representation: Protocol specification*, Tech. Report RFC1014, 1987.

[22] W. L. Jorgensen, *Revised TIPS model for simulations of liquid water and aqueous solutions*, Journal of Chemical Physics **77** (1982), 4156–63.

[23] E. Keller, *Neues von SCHAKAL*, Chemie in unserer Zeit **20** (1986), no. 6, 178–181.

[24] B. W. Kernighan and D. Ritchie, *The C programming language*, 1st ed., Prentice Hall, Cambridge, 1978.

[25] ———, *The C programming language*, second ed., Prentice Hall, Cambridge, 1988.

[26] H. Kistenmacher, H. Popkie, and E. Clementi, *Study of the structure of molecular complexes III. Energy surface of a water molecule in the field of a fluorine or chlorine anion.*, Journal of Chemical Physics **58** (1973), no. 4, 5842.

[27] Donald E. Knuth, Fundamental Algorithms, The Art of Computer Programming, vol. 1, section 1.2, The Art of Computer Programming, Addison-Wesley, second ed., 1973.

[28] A. Laakonsen, *Computer simulation package for liquids and solids with polar interactions. I. McMOL-DYN/H2O: aqueous systems*, Internal Report KGN-41, IBM Corporation, Kingston, N.Y.12401, USA, 1985.

[29] S. W. De Leeuw, J. W. Perram, and E. R. Smith, *Simulation or electrostatic systems in periodic boundary conditions. I Lattice sums and dielectric constant*, Proceedings of the Royal Society **A373** (1980), 27–56.

[30] O. Matsuoka, E. Clementi, and M. Yoshimine, *CI study of the water dimer potential surface*, Journal of Chemical Physics **64** (1976), no. 4, 1351–1361.

[31] Richard Miller, *A library for bulk-synchronous parallel computing*, Parallel Processing Specialist Group workshop on General Purpose Parallel Computing, British Computer Society, 1993.

[32] S. Nosé, *A molecular dynamics method for simulations in the canonical ensemble*, Molecular Physics **52** (1984), 255–268.

[33] ———, *Molecular dynamics simulations at constant temperature and pressure*, Computer Simulation in Materials Science (M. Meyer and V. Pontikis, eds.), vol. E205, Kluwer, Dordrecht, 1991, NATO ASI, pp. 21–42.

[34] S. Nosé and M. L. Klein, *Constant pressure molecular dynamics for molecular systems*, Molecular Physics **50(5)** (1983), 1055–1076.

[35] M. Parrinello and A. Rahman, *Polymorphic transitions in single crystals: A new molecular dynamics method*, Journal of Applied Physics **52** (1981), no. 12, 7182–7190.

[36] G. S. Pawley, *Molecular dynamics simulation of the plastic phase; a model for SF_6*, Molecular Physics **43** (1981), no. 6, 1321–1330.

[37] ———, *Computer simulation of the plastic-to-crystalline phase transition in SF6*, Physical Review Letters **48** (1982), 410–413.

[38] G. S. Pawley and M. T. Dove, *Quaternion-based reorientation conditions for molecular dynamics analyses*, Molecular Physics **55** (1985), no. 5, 1147–1157.

[39] J. G. Powles, W. A. B. Evans, E. McGrath, K. E. Gubbins, and S. Murad, *A computer simulation for a simple model of liquid hydrogen chloride*, Molecular Physics **38** (1979), 893–908.

[40] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in C the art of scientific computing*, 2nd ed., Cambridge University Press, 1992.

[41] B. Quentrec and C. Brot, *New methods for searching for neighbours in molecular dynamics computations*, Journal of Computational Physics **13** (1975), 430–432.

[42] D.C. Rapaport, *Large-scale molecular-dynamics simulation using vector and parallel computers*, Computer Physics Reports **9** (1988), no. 1, 1–53.

[43] K. Refson, *Molecular dynamics simulation of solid* n*-butane*, Physica **131B** (1985), 256–266.

[44] K. Refson and G. S. Pawley, *Molecular dynamics studies of the condensed phases of* n*-butane and their transitions, I Techniques and model results*, Molecular Physics **61** (1987), no. 3, 669–692.

[45] P. M. Rodger, *On the accuracy of some common molecular dynamics algorithms*, Molecular Simulations **3** (1989), 263–269.

[46] W. Smith, *Molecular dynamics on hypercube parallel computers*, Computer Physics Communications **62** (1991), 229–248.

[47] ———, *A replicated-data molecular dynamics strategy for the parallel Ewald sum*, Computer Physics Communications **67** (1992), 392–406.

[48] W. Smith and D. Fincham, *The program MDMPOL*, SERC, Daresbury Laboratory, 1982, CCP5 Program Library.

[49] R. Sonnenschein, *An improved algorithm for molecular dynamics simulation of rigid molecules*, Journal of Computational Physics **59** (1985), no. 2, 347–350.

[50] P. Du Val, *Homographies, quaternions and rotations*, Oxford Mathematical Monograph, 1964.

[51] L. Verlet, *Computer 'experiments' on classical fluids. I. thermodynamical properties of lennard-jones molecules*, Physical Review **165** (1967), 201–214.