The Working Person's Guide to Molecular Dynamics Simulations

David Keffer
Department of Chemical Engineering
University of Tennessee, Knoxville
Begun: October 29, 2001
Last Updated: March 3, 2002

Table of Contents

I. Introduction

The objective of these notes is to provide a self-contained tutorial for people who wish to learn the basics of molecular dynamics simulations. The final goal of these notes is that the student be able to sit down at a machine and execute a molecular dynamics code describing a single-component fluid at a specified temperature and density.

These notes are intended for people who have the following characteristics:

- a basic understanding of classical mechanics,
- a basic understanding of the molecular nature of matter,
- a sound understanding of calculus and ordinary differential equations,
- an understanding of the basic concepts underlying the numerical methods involved in computationally solving ordinary differential equations,
- a working familiarity with a structured programming language, such as FORTRAN, and
- a fundamental drive to understand the world from a methodical point of view.

Molecular Dynamics (MD) is a method for computationally evaluating the thermodynamic and transport properties of materials by solving the classical equations of motion at the molecular level. Molecular Dynamics is not the only method available for achieving this task. Nor is MD the most accurate, nor in some instances, the most efficient. However, MD is the most conceptually straightforward method, and an altogether very utilitarian method for obtaining thermodynamic and transport properties.

The purpose of these notes is to provide a brief hands-on introduction to the molecular dynamics. Several textbooks to aid in this endeavor have been published. Haile's "Molecular Dynamics Simulation" provides a good basic introduction to molecular dynamics.[1] It is the text from which I learned molecular dynamics. Allen and Tildesley's "Computer Simulation of Liquids" provides an excellent introduction to both molecular dynamics and Monte Carlo methods, including brief exposure to more advanced methods, such as the treatment of internal degrees of freedom.[2] Frenkel and Smit's "Understanding Molecular Simulation" has also proven to be a useful resource, supplementing gaps in the other two texts.[3]

II. Classical Mechanics: Newton's Equation of Motion

The classical mechanics that we need to know is due to Sir Isaac Newton. The physical world is described by classical mechanics when things move slow enough and are small enough that we don't have to worry about relativity and things move fast enough and are large enough that we don't have to worry about quantum effects. From one point of view, classical mechanics describes an infinitely large set of problems. From another point of view, classical mechanics fail to describe an infinitely large set of problems.

Newton's equations of motion, generally thought of as equating force, F, to the product of mass, m, and acceleration, a,

$$F = ma \tag{1}$$

is one statement of the classical equations of motion. A second and completely equivalent statement can be obtained from a function called the Lagrangian. In this case, we start with the Lagrangian, which is the difference between the kinetic and potential energy.

$$L(\underline{r},\dot{\underline{r}}) = T(\dot{\underline{r}}) - U(\underline{r}) \tag{2}$$

where L is the Langrangian, T is the kinetic energy and U is the potential energy. We have assumed that the kinetic energy is only a function of velocity and the potential energy is only a function of position. We don't have to assume this, but this is how it usually turns out. The Lagrangian is a function of position, $\underline{r}$, and velocity, $\dot{\underline{r}}$. The equation of motion is obtained by evaluating

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\underline{r}}}\right) - \left(\frac{\partial L}{\partial \underline{r}}\right) = 0 \tag{3}$$

If we consider a spherical particle with mass m, then the translational kinetic energy is

$$T = \frac{1}{2} m |\dot{\underline{r}}|^2 \tag{4}$$

By definition the force is the negative of the gradient of the potential,

$$\underline{F} \equiv -\frac{\partial U}{\partial \underline{r}} \tag{5}$$

Substituting equation (4) and (5) into equation (3) and evaluating the derivatives, we have

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\underline{r}}}\right) - \left(\frac{\partial L}{\partial \underline{r}}\right) = \frac{d}{dt}\left(\frac{\partial T}{\partial \dot{\underline{r}}}\right) + \left(\frac{\partial U}{\partial \underline{r}}\right) = \frac{d}{dt}\left(m\dot{\underline{r}}\right) - F = m\ddot{\underline{r}} - F = 0 \tag{6}$$

Equation (6) is just a restatement of Newton's second law in terms of the kinetic energy and potential energy. This reformulation is necessary, since generally, we are given the potential energy function, rather than the force.

We recognize that equation (6) is a system of second-order (generally nonlinear) ordinary differential equation. It is a system because the equation is a vector equation, with an equation for each dimension of three-dimensional space, i.e. 3 equations, one each for x, y, and z. As such we can solve it if we have two initial conditions per dimension,

$$\underline{r}(t = t_o) = \underline{r}_o \qquad \text{and} \qquad \underline{\dot{r}}(t = t_o) = \underline{\dot{r}}_o \tag{7}$$

In other words, we need to know the position and velocity in each dimension at an initial time.

Newton's equations of motion are called "deterministic", because given the position and velocity at a given time, all past and future positions and velocities can be uniquely determined.

Generally, we solve this system of ordinary differential equations for N molecules. In this case, we have 3N second-order ordinary differential equations and 6N initial conditions. In typical molecular dynamics simulations, N can vary from 10^2 to 10^9, depending upon the computational resources available.

A good and time-tested reference for brushing up on your classical mechanics is reference 4.

III. Intermolecular Potentials

The only piece of information that is lacking in equation (6) is a functional form of the potential energy function, U. The functional form of the potential energy has to be derived from a subatomic theory of physics and chemistry based on the specific identity of the molecule. Invariably, the potential includes parameters that are fit either to (1) experimental data or (2) quantum calculations. Both procedures are used.

The potential energy is generally split up into intermolecular and intramolecular components. Intermolecular interactions include (i) electron cloud repulsion, (ii) attractive van der Waal's dispersion, and (iii) Coulombic interactions between distinct molecules or atoms of the same molecule separated by a substantial distance (such as atoms in different parts of a polymer chain). Intramolecular forces include (i) chemical bond-stretching, (ii) bond-bending, and (iii) bond torsion around a dihedral angle. We only need to consider intramolecular forces when the molecules have non-negligible internal degrees of freedom. Internal degrees of freedom include vibration and rotation about a chemical bond. Monatomic fluids like Ar have no internal degrees of freedom. Polyatomic fluids like O_2, N_2, and CH_4 can be considered as "pseudo-atoms" or "united atoms", that is we can neglect the internal degrees of freedom. For larger atoms, we cannot neglect the internal degrees of freedom and must include an intramolecular potential energy. In this introduction to molecular dynamics, we concern ourselves only with species for which we can neglect internal degrees of freedom.

For simple fluids, we are interested only in the intermolecular potential. These potentials are generally characterized as pair-wise potentials or many-body potentials. Pair-wise potentials assume that the total potential energy can be obtained by summing up the interactions between all combinations of pairs of atoms. Fluids can be well described by pair-wise potentials.

Many-body potentials include, in addition to pair-wise terms, contributions stemming from the interaction of three or more atoms. The Embedded-atom model, used to describe Cu and many other metals, is one example of a many-body potential, because it included 3-body terms. Without the inclusion of these higher-body terms, a potential cannot correctly predict the crystal structure of Cu and other metals.

In this hand-out, we are going to restrict ourselves to materials which can be described by pair-wise potentials. There are a wide variety of pair-wise potentials in the literature. One that is commonly used is the Lennard-Jones potential, defined as

$$U_{LJ}\left(r_{ij}\right) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{8}$$

where r_{ij} is the magnitude of the distance between the centers of atom i and j,

$$r_{ij} = \left|\underline{r}_i - \underline{r}_j\right| = \sqrt{\left(r_{x,i} - r_{x,j}\right)^2 + \left(r_{y,i} - r_{y,j}\right)^2 + \left(r_{y,i} - r_{y,j}\right)^2} \tag{9}$$

and where ε is the well-depth of the interaction potential, and σ is the collision diameter. A plot of the Lennard Jones potential for Argon (ε/k_b= 119.8 K and σ = 3.405 Å) is shown in Figure 1.

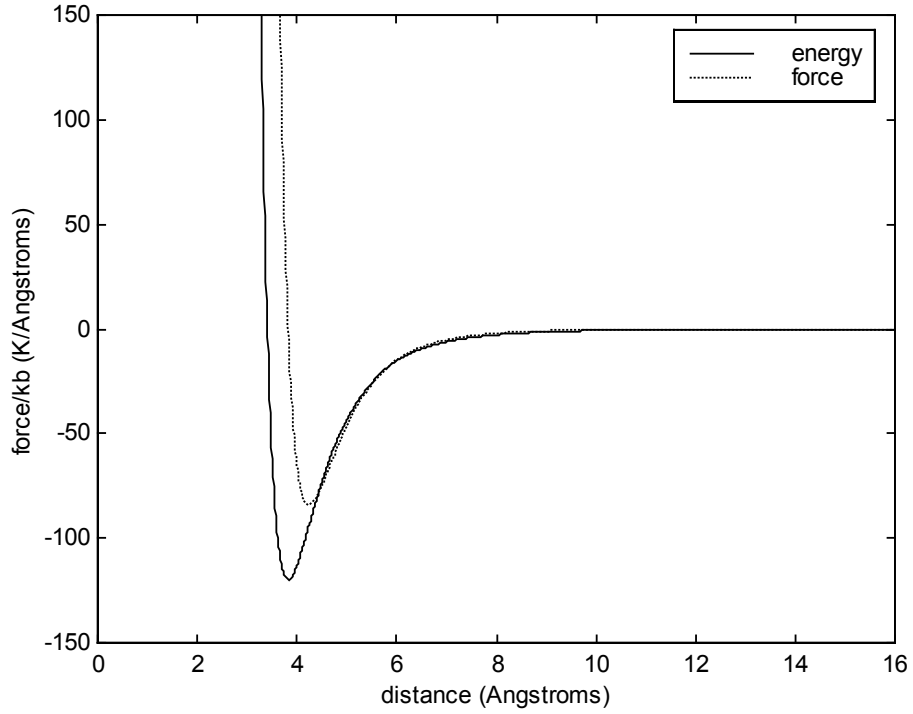

Figure 1. The Lennard-Jones pair-wise potential for Argon.

The Lennard Jones potential has the correct qualitative features. At infinite separation, the interaction potential is zero, because the atoms are too far away to interact appreciably. As the atoms approach each other, "van der Waal's dispersion forces" cause a attractive interaction (as indicated by a negative energy). As the atoms get too far, electron cloud repulsion causes a steep repulsive interaction. The minimum in the energy has a value of ε.

This potential is only for a single pair of atoms. The entire potential energy is given by the sum over all pairs of atoms, given by

$$U = \frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{N} U_{LJ}(r_{ij}) = \sum_{i=1}^{N-1}\sum_{j>i}^{N} U_{LJ}(r_{ij}) \tag{9}$$

where N is the total number of atoms (or pseudo-atoms). The second formulation of the summation yields the same result but is computationally more efficient.

The force on an atom due to the pairwise interaction is determined by the definition of the force in equation (5).

$$\underline{F}_{LJ,ij} = -\frac{\partial U_{LJ}(r_{ij})}{\partial \underline{r}_i} = \frac{24\varepsilon}{r_{ij}}\left[2\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right]\left(\frac{\partial r_{ij}}{\partial \underline{r}_i}\right) = \frac{24\varepsilon}{r_{ij}}\left[2\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right]\left(\frac{\underline{r}_{ij}}{r_{ij}}\right) \tag{10}$$

The force on particle j for the same pairwise interaction is the opposite of the force on particle I,

$$\underline{F}_{LJ,ji} = -\underline{F}_{LJ,ij} \tag{11}$$

The total force on a particle i is simply given by applying the gradient in equation (10) through the entire summation in equation (9), yielding

$$\underline{F}_i = \sum_{\substack{j=1 \\ j\neq i}}^{N} \underline{F}_{LJ,ij} \tag{12}$$

Thus we have the force necessary to complete the ordinary differential equations which state the classical equations of motion.

The calculation of the force and the potential energy is performed in the subroutine funk_force.f (FORTRAN 90) and the function funk_force.m (MATLAB) included in the appendices at the end of this hand-out .

An old and well-loved resource for obtaining parameters for the Lennard-Jones potential is “Molecular Theory of Gases and Liquids” by Hirschfelder, Curtis, and Bird.[5] Bird, Stewart, & Lightfoot has some Lennard-Jones parameters as well.[6]

IV. Initial Conditions

In order to solve the ordinary differential equations, we need to have an initial condition for the position and velocity of each atom (or pseudo-atom) in each dimension, typically 3 for 3-dimensional simulations. One can begin the simulations in any old configuration. By configuration, we mean combination of 3N positions and 3N velocities. The equilibrium state of the system will not depend on the initial conditions unless there are local minima in the Gibbs Free Energy (metastable states). However, there are some standard initial conditions that simulators use to initialize molecular dynamics simulations.

For the simulation of fluids, generally we begin with the system in a perfect simple cubic (SC) or face-centered-cubic (FCC) lattice. The lattice constants of this fictitious crystalline structure are such that the atoms are equally distributed through-out the entire simulation volume. As long as the temperature of the simulation is above the melting temperature of the material, the material will gradually "melt" out of this initial configuration. This initial configuration is commonly used for two reasons. First, it is a well-defined and easily reproducible configuration. Second, it has an advantage over random initial placement in that it makes sure not to overlap atoms. Such overlap in an initial configuration can give rise to highly repulsive forces which cause the simulation to "blow up" in the first few time steps.

The subroutine funk_ipos.f (FORTRAN 90) and the function funk_ipos.m (MATLAB) included in the appendices at the end of this hand-out place the atoms in a simple cubic lattice.

We also need the velocities of each particle defined at the starting time. Generally, we randomly initialize velocities and enforce two or three stipulations on the velocities. The first stipulation is that the translation momentum must be conserved. The second stipulation is that the kinetic energy must be related to the thermodynamic equipartition theorem, namely

$$\frac{1}{2}\sum_{i=1}^{N} m_i \sum_{\alpha=x,y,z} v_{\alpha,i}^2 = \frac{3}{2} N k_b T \tag{13}$$

where the T is specified by the user.

These first two constraints are required. The third stipulation is optional and says that the velocity distribution should follow the Maxwell-Boltzmann distribution. As the system equilibrates from the original configuration, this last stipulation will be automatically fulfilled. Satisfying it initially might speed up the process of equilibration.

The subroutine funk_ivel.f (FORTRAN 90) and the function funk_ivel.m (MATLAB) included in the appendices at the end of this hand-out assign initial velocities generated from a random number generator, then scaled to have zero net momentum and scaled again to the equilibrium temperature.

V. Boundary Conditions

In general, ordinary differential equation do not require boundary conditions. They only require initial conditions, which we have just stipulated above. The boundary conditions discussed here provide a way to simulate an infinite system (at least on the molecular scale) with a few hundred or thousand atoms. If we consider our simulation volume to be the unit cell of a system that is periodically replicated in all three dimensions, then our simulation could be considered as an infinite system.

This periodicity of the system is implemented by what are called periodic boundary conditions. Periodic boundary conditions are manifested in two ways. First they are manifested in the trajectories of particles. Second they are manifested in the "minimum image convention", in the computation of pair-wise separation distances used to evaluate the energy and forces. References [1] and [2] have good discussions of periodic boundary conditions and the minimum image convention.

If we consider our simulation volume as a cube in space, with all the N atoms inside the cube, then during the course of the simulation, some of the atoms will follow trajectories that take it outside the simulation cube. In order to maintain a constant number of particles, a new particle must enter the simulation volume. In order to ensure conservation of momentum and kinetic energy, the new particle must have the same velocity and potential energy as the old particle. Periodic boundary conditions satisfy all of these constraints.

Assume our simulation volume is a cube centered (x,y,z) = (L/2,L/2,L/2) with sides of length L. When a particle leaves through a face of the cube at length x=L, a new particle enters at x=0 with the same value of y and z positions and the same velocity in all 3 components. Analogous statements hold for the y and z dimensions. Since we only care about the particles in our cube, we forget about the old particle and only follow the new particle now. See Figure 2.

Periodic boundary conditions are applied at the end of each time step by the routine pbc.f or pbc.m.

The minimum image convention is a procedure that ensures that the potential energy of the new particle is the same as the potential energy of the old particle, thus ensuring the conservation of energy. Because our simulation volume is periodically repeated, the nearest image of an atom j to an atom i, may not be the atom j lying in the simulation volume. Rather, it may be an image of j lying in a nearby periodic cell. See Figure 3.

In the computation of the energy and the forces (per equations 8-12), one needs to use the separation distance between atom I and the nearest image of atom j, in order to conserve energy. The minimum image convention is implemented directly in the evaluation of the energy and forces. It does not require an explicit call to pbc.f.

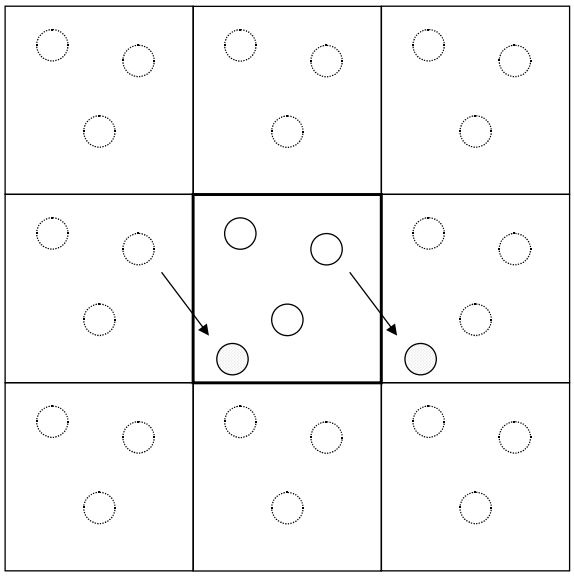

Figure 2. Periodic Boundary Conditions. When a molecule leaves the simulation volume, an image of the molecule enters the simulation volume.

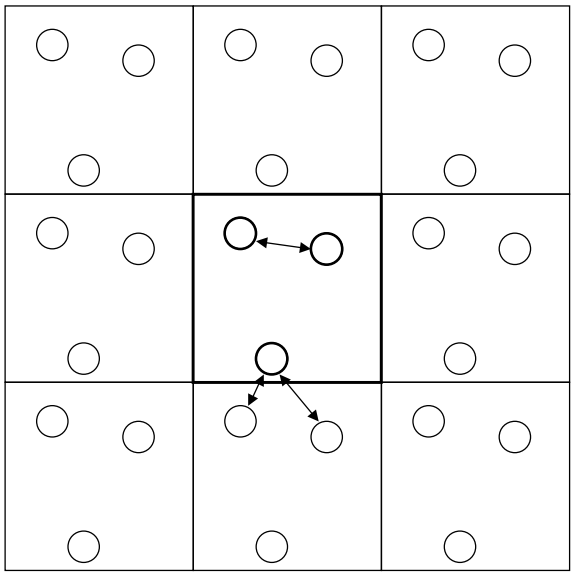

Figure 3. Minimum Image Convention. Here we have a central simulation volume with replicated images. N=3 and there are N(N-1)/2 = 3 neighbor pairs. The separation of these pairs is defined by the minimum separation between atom i and any image of atom j.

VI. Gear Predictor Corrector

We could use any method we want to numerically solve the system of coupled second-order nonlinear ordinary differential equations with initial conditions. For example, we could use a method we are already familiar with, like the Euler method or the Classical Fourth-Order Runge-Kutta method. However, we do not use these methods for reasons of accuracy, stability, and computational efficiency. In order to solve a second-order ODE using one of these methods, it must be rewritten as a system of 2 first order ODEs. Also, in order to obtain the higher-order accuracy of the Classical Fourth-Order Runge-Kutta, we have to evaluate the ODE (in this case the forces) four times. There are better methods than these, in terms of accuracy, stability, and computational efficiency. Allen & Tildesley provide a good discussion of the most common methods used in molecular dynamics.[2] Here we discuss only one such method.

The Gear predictor-corrector (GPC) method is a method that allows us to numerically solve a second-order ODE without converting it into a system of first order ODEs. Moreover, the GPC method requires only one evaluation of the forces per time step. This is of chief importance because we will find that the evaluation of forces is the most computationally expensive part of the molecular dynamics simulation.

Here we describe a fifth-order Gear Predictor-Corrector method appropriate for solving a second-order ODE. The ODE is second order because it has a second derivative in time. The order is fifth order because it is based on a Taylor series that includes all terms out to the fifth derivative.

Consider a vector of the position and its first five time derivatives, multiplied by the factor that will appear before it in a Taylor Series Expansion,

$$\underline{q}_{i,\alpha} = \left[r_{i,\alpha} \quad \Delta t \frac{dr_{i,\alpha}}{dt} \quad \frac{\Delta t^2}{2}\frac{d^2 r_{i,\alpha}}{dt^2} \quad \frac{\Delta t3}{6}\frac{d^3 r_{i,\alpha}}{dt^3} \quad \frac{\Delta t^4}{24}\frac{d^4 r_{i,\alpha}}{dt^4} \quad \frac{\Delta t^5}{120}\frac{d^5 r_{i,\alpha}}{dt^5} \right]^T \tag{14}$$

where i runs over all molecules from 1 to N, and α can be x, y, or z. This vector gives the position and each derivative so that they all have units of length. It is not necessary to write the equations in this form, but it is a common convention.[1,2,7,8] A Taylor series expansion of $\underline{q}_{i,\alpha}$ is used to predict new values, $\underline{q}^p_{i,\alpha}$, of the position and the derivatives based on the old values, $\underline{q}^o_{i,\alpha}$. This Taylor series expansion can be written in matrix form as

$$\underline{q}^{p}_{i,\alpha} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 1 & 3 & 6 & 10 \\ 0 & 0 & 0 & 1 & 4 & 10 \\ 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \underline{q}^{o}_{i,\alpha} \tag{15}$$

The matrix in equation (15) is a Pascal triangle. One should note that equation (15) has no information from the ODE in it. It is purely a Taylor series expansion based on what the position and its derivatives were previously doing.

The forces are evaluated at the predicted positions using equations (8)-(12). These forces are used to correct the position and its derivatives, $\underline{q}^{c}_{i\alpha}$ based on the predicted positions.

$$\underline{q}^{c}_{i,\alpha} = \underline{q}^{p}_{i,\alpha} + \underline{c}\left[\frac{\Delta t^2}{2}\frac{F_{i,\alpha}}{m_i} - \frac{\Delta t^2}{2}\frac{d^2 r^{p}_{i,\alpha}}{dt^2}\right] \tag{16}$$

The vector $\underline{c}$ is simply a vector of constant corrector coefficients. This vector is multiplied by the difference between the acceleration obtained from the evaluation of the forces and the predicted acceleration, which is the third element of the vector $\underline{q}^{p}_{i,\alpha}$.

The values of the corrector coefficients in $\underline{c}$ depend upon the order of the ODE, the order of the GPC method, and the functional form of the ODE. They are selected so as to maximize stability and minimize error. For a second-order ODE with a fifth-order GPC method, where the forces are strictly a function of position (and not velocity) the corrector coefficients are

$$\underline{c} = \left[\frac{3}{20} \quad \frac{251}{360} \quad 1 \quad \frac{11}{18} \quad \frac{1}{6} \quad \frac{1}{60}\right]^{T} \tag{17}$$

If the velocity does appear explicitly in the ODE (i.e. the force is a function of velocity), then the factor of 3/20 should be replaced by a factor of 3/16.

The subroutines predictor.f and corrector.f (FORTRAN 90) and the functions predictor.m and corrector.m (MATLAB) included in the appendices at the end of this hand-out perform the prediction and correction steps outlined above.

The family of methods to which the Gear predictor-corrector method belongs is rigorously derived in the following two references by Gear.[7,8] You should be warned that these references are not for the faint of heart. Even repeating Gear's derivation of the corrector coefficients is an extremely nontrivial task. The values of the corrector coefficients are also available elsewhere.[2]

VII. Force Evaluation and Neighbor Lists

The calculation of the force and the potential energy is performed in the subroutine funk_force.f (FORTRAN 90) and the function funk_force.m (MATLAB) included in the appendices at the end of this hand-out .

There are two main points in which the implementation of the force and potential energy evaluation differ from the expressions in equations (8) to (12).

The first difference is that equation (9) contains a double summation but the subroutines contain a single summation over a neighbor list. The use of a neighbor list increases computational efficiency. The basic idea is as follows. For interactions between pairs of molecules separated by less than a distance r_{cut}, we calculate the forces and energies explicitly. For molecules separated by a distance of r_{cut} or greater, we assume a mean-field. Thus the potential energy is a sum of the short-range intermolecular forces and the long-range approximation.

$$U = U_{SR} + U_{LR} = \frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i \\ r_{ij}\leq r_{cut}}}^{N} U_{LJ}\left(r_{ij}\right) + U_{LR} \tag{18}$$

The long-range contribution to the potential energy of the entire system can be explicitly calculated if we assume that the fluid is isotropic,

$$U_{LR} = \frac{1}{2}\frac{N^2}{V}\int_0^{2\pi}\int_0^{\pi}\int_{r_{cut}}^{\infty} U_{LJ}(r)\sin\theta r^2 dr d\theta d\phi = \frac{1}{2}\frac{N^2}{V}\left[\frac{16\pi\varepsilon}{9}\left(\frac{\sigma^{12}}{r_{cut}^9} - 3\frac{\sigma^6}{r_{cut}^3}\right)\right] \tag{19}$$

For given values of N and V, this long-range term is a constant. As such, there is no force associated with it.

Since we have already taken into account the long-range forces, we can now reduce the force and energy calculation only to those pairs with a separation within r_{cut}. In order to use a neighbor list, we need a scalar, Nnbr, which is the total number of pairs with separation within r_{cut}. Second, we need a matrix, Nnbrlist, which has Nnbr rows and 2 columns. The first column gives the identification number of one molecule which forms the pair and the second column gives the identification number of the other molecule in the pair. In this way, by evaluating the forces and energies of the pairs in the Nbrlist, we have all the forces and energies we need to integrate the ODEs. Again, the motivation behind doing this is based on computational efficiency.

Since it takes some time to generate the neighbor list, we don't want to do it every step. instead, we define a distance r_{nbr}, which is generally a few Angstroms larger than r_{cut}. We include the molecules in the neighbor list if the separation is less than r_{nbr}, but we don't include them in the force evaluation unless their separation is less than r_{cut}. Then we only create a new neighbor list every k_{nbr} time steps. The idea is that every pair that could possibly be inside a distance r_{cut} within the next k_{nbr} time steps is included in the neighbor list. As the simulation system becomes large, this neighbor list is much smaller than the total number of possible

neighbors, which is N(N-1)/2. (We will see that using the neighbor list, the number of neighbors scales with N rather than N^2.)

The particular values of r_{cut}, r_{nbr}, k_{nbr} and are chosen by experience so that the approximation of a mean field potential at long-range is not a bad one. Sample values are given in the code below.

The neighbor list is created by the subroutine funk_mknbr.f (FORTRAN 90) and the function funk_mknbr.m (MATLAB) included in the appendices at the end of this hand-out .

The second difference between the implementation of the energy and force evaluation given in the codes below and the equations is the presence of the minimum image convention. We have explained the purpose of the minimum image convention above. The following lines implement the minimum image convention in FORTRAN 90, where our simulation volume is a cube with sides of length, `side`, and where `sideh` is half of `side`.

```
dis(1:3) = r(i,1:3) - r(j,1:3)
do k = 1, 3, 1
      if (dis(k) .gt.  sideh) dis(k) = dis(k) - side
      if (dis(k) .lt. -sideh) dis(k) = dis(k) + side
enddo
```

VIII. Sampling and Property Evaluation

We all know that the solution to a system of N m^{th}-order ODEs is Nm functions of time. These functions represent particle positions and velocities in time. However, this is too much data for us to make sense of. All we want are familiar functions of the data, that we can make sense of. Therefore, we calculate these functions at every k_{samp} time step and compute average values and standard deviations of the functions.

These functions can be whatever we want. In the code below, we sample $n_{prop} = 8$ properties: kinetic energy, potential energy, total energy, temperature, total x-, total y-, total z-momentum, and pressure. We choose these properties because they will help us identify whether the system is at equilibrium and whether the conservation of energy and momentum are being obeyed.

The kinetic energy of the system is simply the sum of the individual kinetic energies. The potential energy is the sum of pairwise potential energies, plus the long-range correction as shown in equation (18). The total energy is the sum of the kinetic and potential. The temperature is determined from the kinetic energy, using the equipartition function given in equation (13). The total momenta are simply the sum of the individual momenta. The pressure is the only property which is not obvious. The pressure can be calculated by

$$P = \rho\left(k_b T - \frac{W}{3N}\right) = \rho\left(k_b T - \frac{W_{SR}}{3N} - \frac{W_{LR}}{3N}\right) \tag{20}$$

where W is the virial coefficient, W_{SR} is the short-range component of the virial coefficient and W_{LR} is the long-range component of the virial coefficient. The short-range component of the virial coefficient is given by[1]

$$W_{SR} = \sum_{i=1}^{N-1}\sum_{j>i}^{N}\left[\underline{F}_{LJ,ij}\cdot\left(\frac{r_{ij}^2}{\underline{r}_{ij}}\right)\right] \tag{21}$$

and the long-range component of the virial coefficient is given by[1]

$$W_{LR} = -\frac{1}{2}\frac{N^2}{V}\left[\frac{96\pi\varepsilon}{9}\left(2\frac{\sigma^{12}}{r_{cut}^9} - 3\frac{\sigma^6}{r_{cut}^3}\right)\right] \tag{22}$$

Because we want both the average value and the standard deviation of the properties, we need to keep track not only of the cumulative sum of the properties, but also the cumulative sum of the squares of the properties, because we will use the following "mean of the square less the square of the mean" formula to calculate the standard deviation of the property

$$\sigma_x = \sqrt{\sigma_x^2} = \sqrt{\mu_{x2} - (\mu_x)^2} \tag{23}$$

For simplicity, we store the properties in a matrix that is dimensioned n_{prop} by 6. The six columns correspond to (i) the instantaneous value of the property, (ii) the current cumulative sum, (iii) the current cumulative sum of the squares, (iv) the average, (v) the variance, and (vi) the standard deviation.

The properties are cumulative sums are updated every k_{samp} time steps by the subroutine funk_getprops.f (FORTRAN 90) and the function funk_getprops.m (MATLAB) included in the appendices at the end of this hand-out. At the end of the equilibration stage (explained in the next section) and at the end of the data production stage, we use the subroutine funk_report.f (FORTRAN 90) and the function funk_report.m (MATLAB) to calculate the averages and the standard deviations of each of the n_{prop} properties.

Diffusion Coefficients

If we would like to calculate diffusion coefficients from this data, then we need mean square displacement (MSD) data as a function of time. The MSD vs time data can be calculated from molecule positions. Of course, we don't generally have enough memory to save the positions every time step, nor is it statistically necessary. We only need to save the positions every k_{msd} time steps. A separate code will be used to calculate a self diffusion coefficient from this data, after the simulation is finished.

The MSD calculations have to be generated from positions that have never had periodic boundary conditions (PBCs) applied to them. Therefore, in the code below, we save the positions in two vectors. The first is `r`, which has the PBCs applied. The second vector is `rwopbc` and stores positions WithOut PBCs.

The `rwopbc` positions are regularly written to a file by the subroutine funk_msd.f (FORTRAN 90) and the function funk_msd.m (MATLAB) included in the appendices at the end of this hand-out.

IX. Initialization, Equilibration and Data Production

The main program to run a molecular dynamics simulation is called mddriver.f or mddriver.m. It is divided up into three main sections: initialization, equilibration, and data production.

Initialization

The portion of the code labeled initialization performs 5 tasks:

1. Define all simulation parameters, such as thermodynamic conditions (N, V, T), Lennard-Jones parameters (σ, ε), numerical integration constants (Δt, $\underline{c}$), various intervals (k_{samp}, k_{nbr}, k_{msd}), and any other necessary parameters.
2. Assign initial positions.
3. Assign initial velocities.
4. Generate initial neighbor list.
5. Calculate initial potential energy and forces.

These tasks are only performed once.

Equilibration

The initial positions (some kind of lattice presumable) and the initial velocities (random) are not representative of the equilibrium state. The second part of the program solves the ODEs but in order to move from the initial state to an equilibrium state. The program functions during equilibration are for the most part the same as the functions during data production, with a couple exceptions. The main exception is that this data is not equilibrated. Therefore, we do not want to use any of these properties to calculate average properties of the equilibrium system.

The Equilibration portion of the code contains a loop which performs the following task for `maxeqb` time steps:

1. Predict new positions.
2. Calculate potential energy and forces.
3. Correct new positions.
4. Apply periodic boundary conditions.
5. Scale velocities.
6. Update neighbor list every k_{nbr} steps.
7. Sample properties every k_{samp} steps.
8. Generate report of equilibration results when equilibration is over.

We have discussed each of these steps before, with the exception of step 5. We still sample the properties and report them, even though we will not use them to calculate our equilibrium properties, because they can be used to determine if the system has become equilibrated.

Step 5 is where we scale the velocities. Scaling the velocities is a non-rigorous way to maintain a constant temperature. In this ensemble, we conserve N, V, and E. The temperature

may change at each step. If our initial configuration has a higher potential energy than the equilibrium state, then as we equilibrate the potential energy drops, but due to the conservation of energy, the kinetic energy (and consequently the temperature) increases. The opposite is also true; if our initial configuration yields a very low potential energy, then the temperature will drop as we equilibrate. If we would like to simulate about a particular temperature, then we need to scale the velocities so that the temperature is constant. If we scale velocities, we do not conserve energy. But that is okay, we will only scale during equilibration. We are not going to use the equilibration steps for anything anyway.

The subroutine funk_scalev.f (FORTRAN 90) and the function funk_scalev.m (MATLAB) included in the appendices at the end of this hand-out perform this velocity scaling.

Data Production

After we have equilibrated the system, we solve the ODEs and collect those properties of interest, which we will use to compute the equilibrium properties of the system. Since total energy (not kinetic energy) is conserved, the temperature will fluctuate. However, it will fluctuate about whatever temperature we scaled to during the equilibration of the system.

The data production portion of the code contains a loop which performs the following task for `maxstp` time steps:

1. Predict new positions.
2. Calculate potential energy and forces.
3. Correct new positions.
4. Apply periodic boundary conditions.
5. Update neighbor list every k_{nbr} steps.
6. Sample properties every k_{samp} steps.
7. Save positions to a file every k_{msd} steps.
8. Generate report of data production results when the simulation is over.

The only differences between data production and equilibration is that we do not scale the velocities during data production and we now save the positions for the calculation of the self-diffusion coefficient.

X. Checking the Code

Everyone makes typographical errors. When typographical or logical errors occur in code, they are called bugs. Every code must go through a process of debugging before you can trust the results. In this section of the code, we outline a couple suggestions for checking that the code is trustworthy.

The most important check of an MD code is conservation of energy during the data production step. A code can fail to conserve energy for several reasons:

- a bug in the code
- r_{cut} is too small
- r_{nbr} is too small or k_{nbr} is too large
- the time step Δt is too large

In order to determine if we have a bug in the code, we must make sure that the last three problems do not occur. We can do this by looking at a small system, say 125 molecules. In this case, the number of pairs is reasonable 125*124/2 = 7750 and we don't need to use a long range approximation. If our simulation volume is a cube with sides of length `side`, then we set r_{cut} and to r_{nbr} something larger than the maximum possible separation in the system. The maximum possible separation for two points in a cube with periodic boundary conditions is $\frac{\sqrt{3}}{2}$ `side`. In this way, every possible pair is included in the neighbor. Then the interval of updating the neighbor list, k_{nbr}, is irrelevant. We run the code and examine the standard deviations of the kinetic energy, potential energy, and total energy for the data production part of the code. We rerun the code for at least two values of Δt, say 1.0 and 0.1 fs.

Because the total energy is conserved, the standard deviation should ideally be zero. Since all energy is either kinetic or potential, as one decreases the other must increase and vice versa, so the standard deviations of the kinetic energy and potential energy must ideally be the same.

Due to the fact that we are using a numerical algorithm to approximate the solution, we will have some error from these ideal results. However, there are a couple features which we can check for the conservation of energy. First the standard deviation of the total energy, E, should be much lower than the standard deviation of the kinetic, T, or potential, U, energy (which should be about the same). How much lower depends upon the particular system. We will give a couple examples shortly.

$$\sigma_E << \sigma_U \approx \sigma_T \tag{24}$$

When we say much less, typically we mean at least two orders of magnitude smaller.

IF EQUATION (24) IS NOT SATISFIED, THERE IS A BUG IN THE CODE. That's all there is to it. You may try to convince yourself that there is some other problem, but you are just covering up a bug in the code. Every simulation you run with a code that does not satisfy equation (24) is a bogus simulation.

This is not to imply that a code which satisfies equation (24) is bug free. On the contrary, there can be other bugs. But a wide variety of bugs can be discovered by checking the criteria in equation (24).

Further checking can be performed by simulating a system (like a relatively low density gas) that you have experimental data or a reliable theory for. Then checking the particular values of the energy, pressure, and heat capacity can confirm that the simulation is running well.

XI. An Example

Simulate Pure Gas-Phase Methane at T=298 K and P=1 atm

In a standard (microcanonical: specify N, V, & E) molecular dynamics simulation, we do not specify the pressure. Therefore, the best we can do here is specify a density that is close to a pressure of 1 atm. To do this we can, for example, use the van der Waal's equation of state to predict the vapor density (units of molecules per $Å^3$) for methane at T=298K and P = 1atm. Our friend vdW EOS yields $\rho = 2.468 \times 10^{-5}$ molecules/$Å^3$ which for a system of N = 125 molecules yields a system volume = 5.065×10^6 $Å^3$. The side of the cube is 171.7 Å and the max separation between pairs is 148.8 Å.

We equilibrate this system for 10 ps (5000 equilibrium steps @ 2 fs time steps) and produce data for 100 ps (50,000 data production steps @ 2 fs time steps).

Let's examine program output 1. First we see that we have 7750 neighbor pairs, as we knew that we should, since we have made r_{cut} large enough to include every one of the N*(N-1)/2 possible molecule pairs. We are periodically printing out the kinetic energy, potential energy, and total energy every 5000 steps. If we look at the summary of the data for the equilibration period, we see that the average temperature was 298 K and the standard deviation was zero. This is because we scaled the temperature during equilibration. Energy is not conserved with velocity scaling.

Momentum however, should be conserved, even during velocity scaling. The average velocity of a particle can be obtained from equation (13).

$$|\overline{v}| = \sqrt{\frac{k_b T}{m}} \tag{25}$$

This velocity is averaged not only over particles but also over x, y, and z. In the natural units of the code, the average velocity is1.57×10^{-2} Å/fs. The average momentum in natural units is 4.19 aJ*fs/Å, where aJ is an attoJoule, or 10^{-18} Joules. From the data output, we see that the total momentum is of the order of 10^{-12} . Therefore, we have conserved momentum to 12 significant figures. Excellent.

During production we see that the standard deviations of the kinetic energy and the potential energy are of the order of magnitude of 10^{-3} and the standard deviation of the total energy is of the order of magnitude of 10^{-7}. Thus we have satisfied equation (24).

Let's check a thermodynamic property from the code. From statistical mechanics, we know that the total energy of a van der Waal's gas is given by:

$$E = T + U = \frac{3}{2} N k_b T - N \rho a \tag{26}$$

Theoretically, the constant volume heat capacity, Cv, is given by

$$C_v \equiv \left(\frac{\partial E}{\partial T} \right)_V = \frac{3}{2} N k_b \tag{27}$$

One can obtain the heat capacity from the standard deviation of the potential energy

$$\sigma_U^2 = \frac{3}{2}Nk_b^2T^2\left(1-\frac{3}{2}\frac{Nk_b}{C_v}\right) \tag{28}$$

In Table 1, we compare the thermodynamic properties from the van der Waal's theory and the simulation. The result should be pretty good, because we are nearing the ideal gas limit, where the van der Waal's equation of state should do well.

Table 1. Comparison of Theoretical and Simulation Thermodynamic Properties for gaseous Methane at T = 298 K and ρ = 40.98 mole/m^3.

property	theory	simulation (1)	percent error
kinetic energy (aJ)	$7.714\text{x}10^{-1}$	$7.717\text{x}10^{-1}$	$3.913\text{x}10^{-2}$
potential energy (aJ)	$-1.959\text{x}10^{-3}$	$-1.927\text{x}10^{-3}$	1.612
total energy (aJ)	$7.695\text{x}10^{-1}$	$7.695\text{x}10^{-1}$	$4.334\text{x}10^{-2}$
Cv (aJ/K)	$2.589\text{x}10^{-3}$	$2.591\text{x}10^{-3}$	$7.293\text{x}10^{-2}$
Pressure (aJ/Å^3)	$1.013\text{x}10^{-7}$	$1.015\text{x}10^{-7}$	0.1872

The code checks out on all counts.

Simulate Pure Liquid-Phase Methane at T=150 K and P=1 atm

Now let us repeat the example for a liquid. Since the critical temperature of methane is 190.6 K, in order to observe a liquid, we need a temperature less than 190.6 K. We select a temperature of 150 K and a pressure of 1 atm. (Whether the liquid is the stable phase at this combination of T and P is immaterial. We can simulate it anyway.)

Again, in a standard molecular dynamics simulation, we do not specify the pressure. Therefore, the best we can do here is specify a density that is close to a pressure of 1 atm. To do this we can, for example, use the van der Waal's equation of state to predict the liquid density (units of molecules per Å^3) for methane at T=150 K and P = 1 atm. Our friend vdW EOS yields $\rho = 8.832\text{x}10^{-3}$ molecules/Å^3 which for a system of N = 125 molecules yields a system volume = $1.415\text{x}10^4$ Å^3. The side of the cube is 24.2 Å and the max separation between pairs is 21.0Å.

We equilibrate this system for 10 ps (5000 equilibrium steps @ 2 fs time steps) and produce data for 100 ps (50,000 data production steps @ 2 fs time steps).

In program output 2, we see that momentum is still conserved to the same degree that it was in the liquid simulation. We see that equation (24) is satisfied because the standard deviation of the total energy is five orders of magnitude less than the standard deviation of the kinetic or potential energy.

We can also check the thermodynamic properties. In Table 2, the percent errors are fairly large. This is not due to a problem with the simulation code. Rather, this discrepancy is due to the fact that the van der Waal's equation of state (from which we obtain our theoretical values) is not very accurate for liquid phases. But we can see at least, that our values from the simulation are the right order of magnitude.

Table 2. Comparison of Theoretical and Simulation Thermodynamic Properties for liquid Methane at T = 150 K and $\rho = 1.467 \times 10^4$ mole/m^3.

property	theory	simulation (2)	percent error
kinetic energy (aJ)	3.883×10^{-1}	3.825×10^{-1}	1.508
potential energy (aJ)	-7.011×10^{-1}	-8.325×10^{-1}	18.75
total energy (aJ)	-3.127×10^{-1}	-4.501×10^{-1}	43.91
Cv (aJ/K)	2.589×10^{-3}	3.532×10^{-3}	36.45
Pressure (aJ/Å^3)	1.013×10^{-7}	-3.070×10^{-6} *	3131

The code checks out on all counts, except the pressure. Here we have a negative pressure. A negative pressure indicates that the liquid phase is not the stable phase at this temperature. To validate this, we can resolve the vdW EOS for the vapor phase at 150 K and 1 atm. We run the simulation at the vapor density predicted by the van der Waals equation of state, which is 2.020×10^4 Å^3/molecule, and the simulation yields a pressure of 1.021×10^{-7} aJ/Å^3.

We can calculate reasonable pressures for the liquid phase if the liquid phase is the stable phase. In order to check that we were calculating the pressure correctly, we duplicated simulation results reported in reference 1.

* The pressure has a great deal more fluctuations than the energy. In order to obtain the pressure accurately, we reran the simulation with 512 molecules, 10,000 equilibration steps, and 100,000 production steps.

```
 initially we have        7750 neighbor pairs
**********         equilibration Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.77144378E+00   0.77144378E+00   0.00000000E+00
Potential Energy (aJ)   -0.16255502E-02  -0.10080020E-02   0.13536733E-02
Total Energy (aJ)        0.76981822E+00   0.77043577E+00   0.13536733E-02
Temperature (K)          0.29800000E+03   0.29800000E+03   0.00000000E+00
x-Momentum              -0.48290790E-13   0.10353410E-12   0.60053346E-13
y-Momentum              -0.31007770E-12  -0.10467888E-12   0.95734796E-13
z-Momentum              -0.16936435E-12  -0.69707408E-13   0.85013246E-13
**********         production    Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.77404763E+00   0.77174564E+00   0.15209792E-02
Potential Energy (aJ)   -0.42294074E-02  -0.19274171E-02   0.15209793E-02
Total Energy (aJ)        0.76981823E+00   0.76981823E+00   0.38857474E-06
Temperature (K)          0.29900584E+03   0.29811661E+03   0.58753707E+00
x-Momentum               0.50029489E-12   0.32658729E-12   0.21016618E-12
y-Momentum              -0.71304008E-12  -0.31733669E-12   0.24665049E-12
z-Momentum              -0.74065132E-12  -0.56415994E-12   0.26751536E-12
 Program has used   427.074107870460     seconds of CPU time.
 This includes   426.9840    seconds of user time and  9.0129599E-02 seconds of system time.
```

Program Output 1: Checking for conservation of energy in gaseous methane, $\Delta t = 2$ fs.

```
  initially we have        7750 neighbor pairs
**********         equilibration Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.38831062E+00   0.38831063E+00   0.00000000E+00
Potential Energy (aJ)   -0.83837819E+00  -0.82769051E+00   0.21438959E-01
Total Energy (aJ)       -0.45006756E+00  -0.43937988E+00   0.21438959E-01
Temperature (K)          0.15000000E+03   0.15000000E+03   0.00000000E+00
x-Momentum              -0.27264656E-13  -0.32597504E-13   0.12824085E-13
y-Momentum               0.51121231E-13   0.31560640E-13   0.16353686E-13
z-Momentum              -0.84220061E-13  -0.41557694E-13   0.32067904E-13
**********         production    Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.38532780E+00   0.38245442E+00   0.14656641E-01
Potential Energy (aJ)   -0.83539469E+00  -0.83252270E+00   0.14656595E-01
Total Energy (aJ)       -0.45006689E+00  -0.45006828E+00   0.77659571E-06
Temperature (K)          0.14884777E+03   0.14773782E+03   0.56616947E+01
x-Momentum               0.40521518E-13  -0.34630954E-13   0.37970081E-13
y-Momentum               0.20800853E-12   0.21680954E-12   0.74560013E-13
z-Momentum               0.18184139E-12  -0.12192550E-13   0.10744804E-12
 Program has used   423.949600785971     seconds of CPU time.
   This includes   423.8495    seconds of user time and  0.1001440 seconds of system time.
```

Program Output 2: Checking for conservation of energy in liquid methane, $\Delta t = 2.0$ fs.

XII. Benchmarking for Computational Efficiency and Numerical Accuracy

Size of Time step

We want to run our simulation for a long time because the longer we run the simulation, the better our statistical averaging will be. To this end, we wish to maximize the size of the time step. On the other hand, we know that the numerical algorithm used to integrate the ODE will become inaccurate and unstable at large time steps. The trick is to find the largest time step that still yields reasonable accuracy.

We will use the standard deviation of the total, kinetic, and potential energies to gauge the accuracy of the solution, since the first should be zero and the last two should be equal. In Figures 4 and 5, we plot the standard deviations of the kinetic, potential, and total energy for the gas-phase and liquid-phase methane examples simulated above as a function of time step. In all cases, we run 5000 equilibration steps and 50,000 production steps. Also in all cases, we have 125 molecules in the simulation. Since the time step changes, the duration of the simulation changes. For both the gas and the liquid, we see three regimes.

In Figure 4, we see that at very small time steps, the simulation does not run long enough to provide a good statistical representation of equilibrium. There have been no collisions in the simulation so the standard deviation of the kinetic and potential energies are very small. The code cannot report a standard deviation smaller than 10^{-8} because we only have sixteen digits of accuracy and are using the "the mean of the square less the square of the mean" formula to calculate the variance. In the second region, the standard deviation of the total energy is several orders magnitude smaller than the standard deviation of the potential or kinetic energies. Therefore, these time steps provide good energy conservation and good simulations. In the third region, the time step is simply too large to conserve energy.

In Figure 5, we just see regions 2 and 3. We do not observe region 1 since collisions occur in a much shorter time span in the liquid than in the gas phase. If we picked a ridiculously small value for the time step, we should see a region 1.

The ideal time step for computational efficiency is taken from the high side of region 2. Typical values are something like 2 fs.

All of these simulations took 350 to 450 seconds of cpu time on a Pentium III 600 MHz processor. The variation in cpu time is due to other processes running in the background. We will perform a more rigorous timing study shortly.

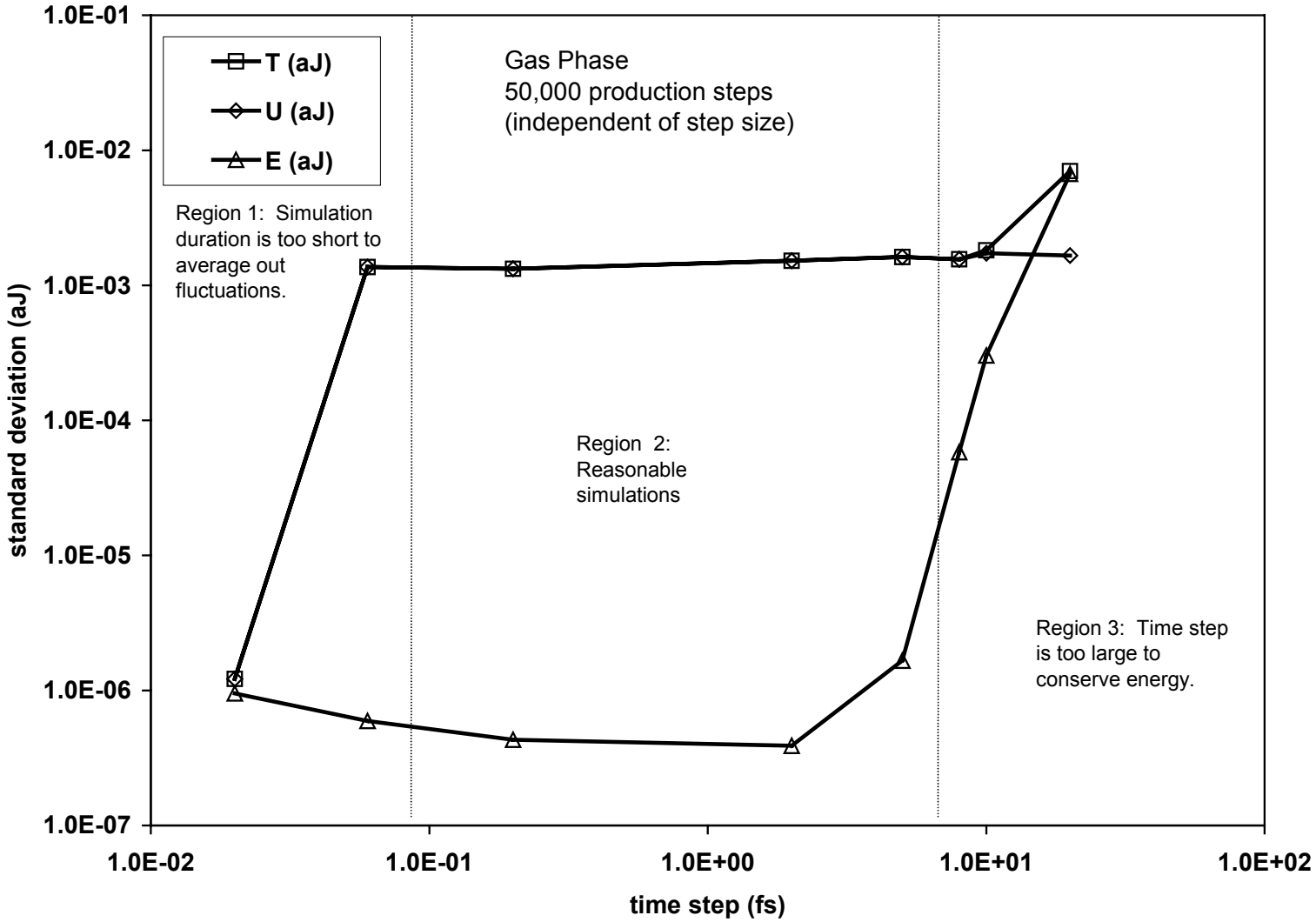

Figure 4. Standard Deviations for total, kinetic, and potential energies as a function of step size for gas-phase methane.

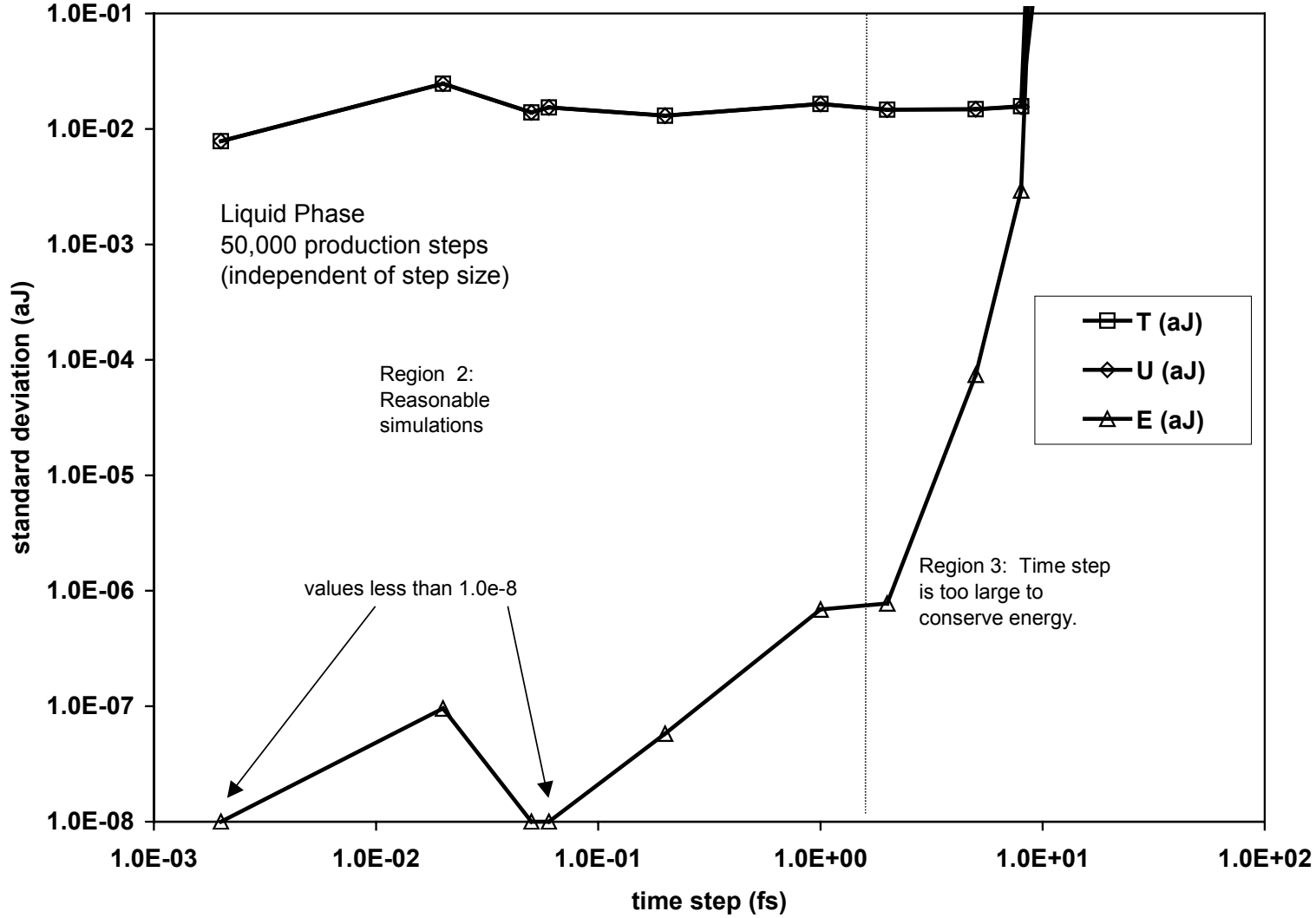

Figure 5. Standard Deviations for total, kinetic, and potential energies as a function of step size for liquid-phase methane.

Number of Time Steps

For a given size of time step, we need to know how long to run the simulation in order to get reasonable results. The answer is system dependent. One should run the simulation for various durations and examine the dependence on the thermodynamic properties. One needs a long enough simulation such that the thermodynamic properties are not a function of the number of steps. Both the equilibration and data production sections of the code need to be run for sufficiently long periods of time.

For a system of N = 125 methane molecules at the liquid phase conditions given in the example above, with a step size of $\Delta t = 2.0$ fs, we run the simulation for various numbers of equilibrium steps and production steps. In Figure 6, we present the potential energy as a function of the number of equilibrium steps and as a function of the number of data production steps.

System Size

The number of molecules in the simulation affects not only the amount of CPU time required but also the accuracy of the results. The number of molecules must be selected large enough that the intensive thermodynamic properties generated by the simulation are no longer a function of the number of molecules. In Figure 8, we plot the potential energy per molecule as a function of the number of molecules in the simulation of liquid methane under conditions described above. We observe that 512 and 1000 molecules yield the same result. Therefore, we could use a simulation with 512 molecules.

The computational cost of increasing N is displayed in Figure 9. As the system becomes large, we will see that the majority of the simulation time is spent evaluating the energy and forces. Without the use of a neighbor list, we see from equation (9) that the number of neighbor pairs scales as N(N-1)/2, thus the computational expense of evaluating the energy scales as N^2. The CPU time for evaluating the energy should be linearly proportional to the number of neighbors and quadratically proportional to N.

Using a neighbor list changes the scaling behavior of the computational time for the evaluation of the energy and forces to linear in both the number of neighbor pairs and N, by eliminating those neighbors too far away to have significant interaction. Figure 9 displays that the large system scaling behavior between the CPU time and the number of molecules in the simulation is indeed linear.

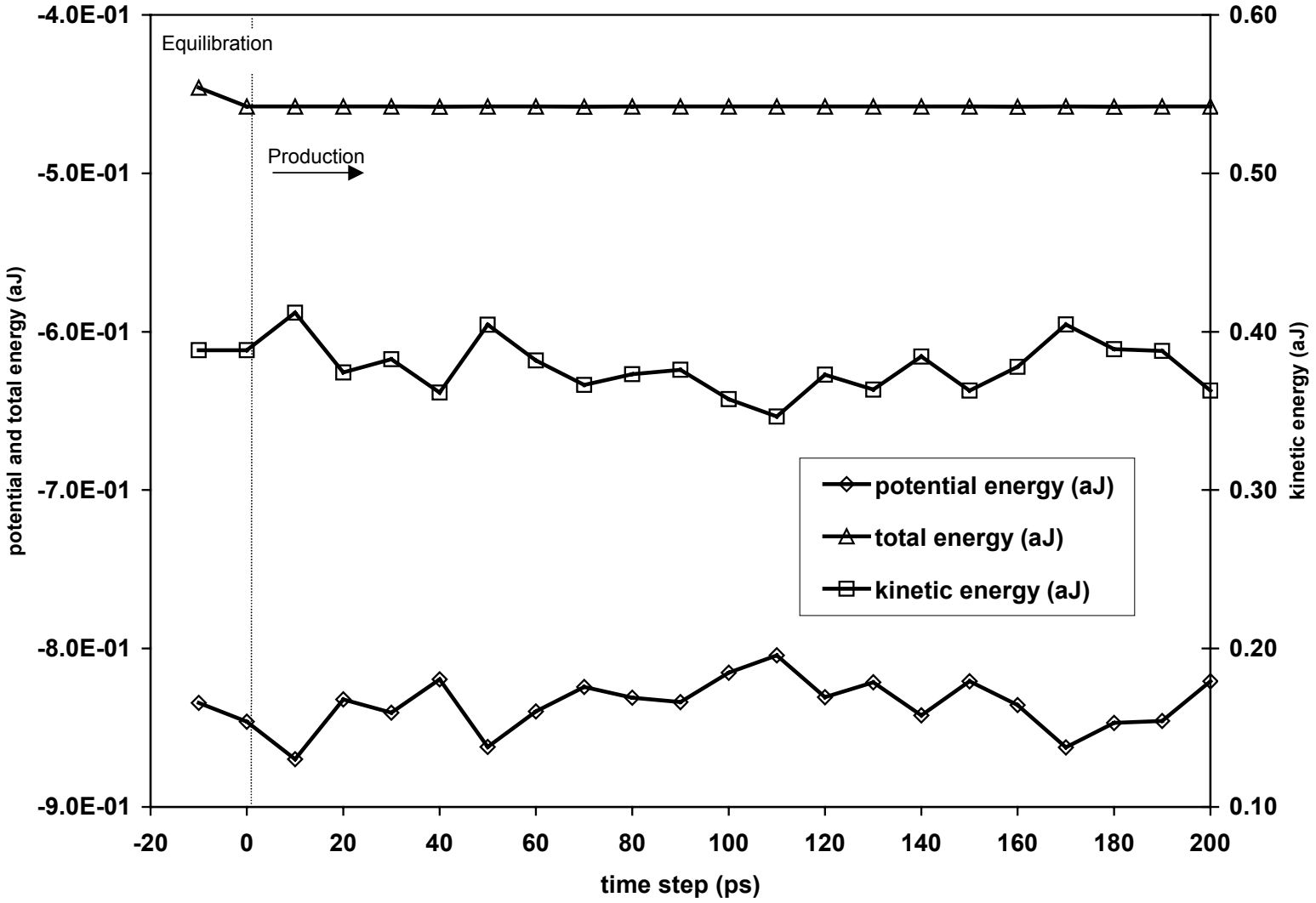

Figure 6. Total, kinetic, and potential energies for a typical simulation of a liquid. The simulation must last long enough to include many fluctuations in the properties.

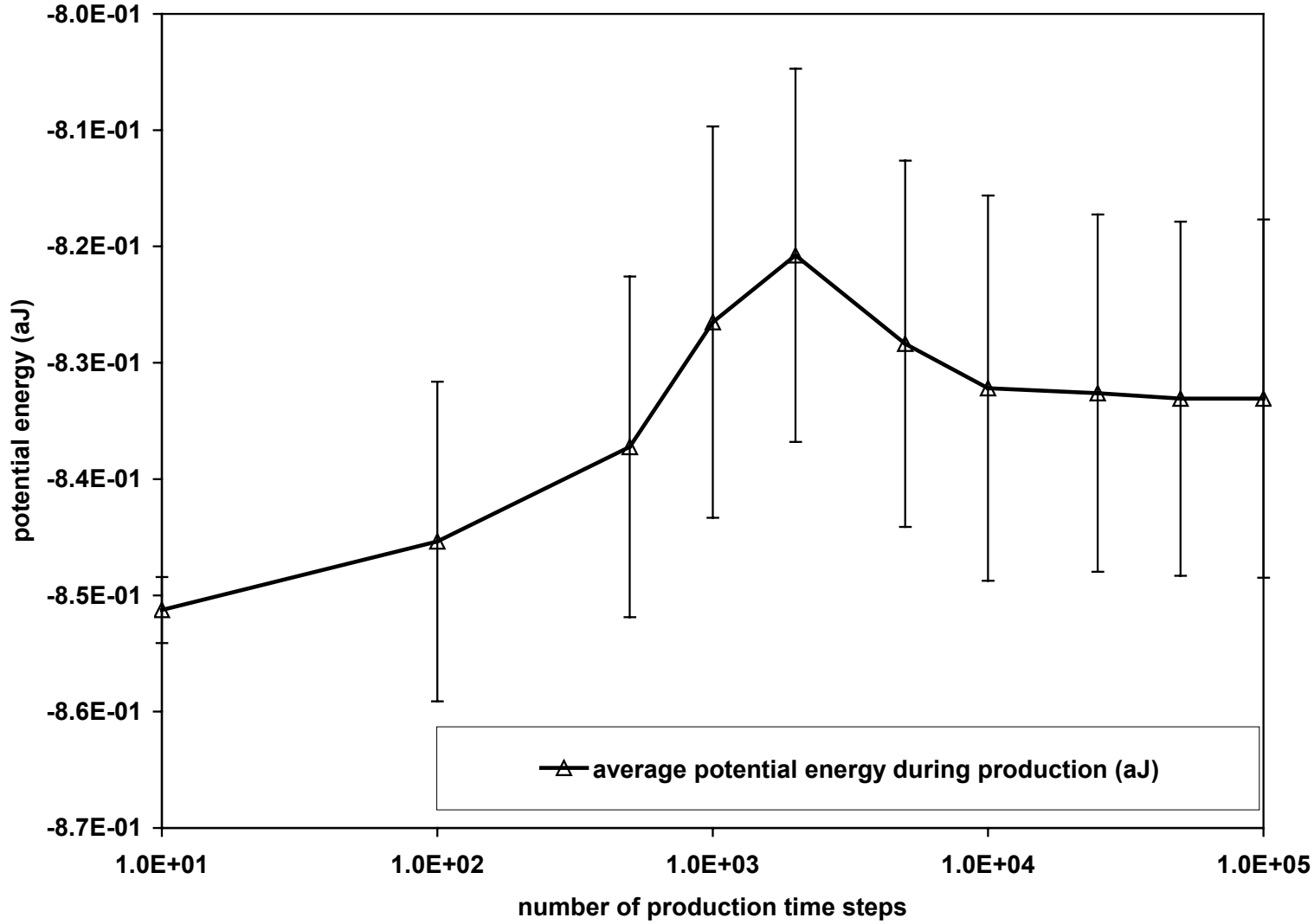

Figure 7. Average potential energy as a function of number of production steps. Here 5000 equilibration steps were run for all simulations with N=125 and $\Delta t = 2.0$ fs.

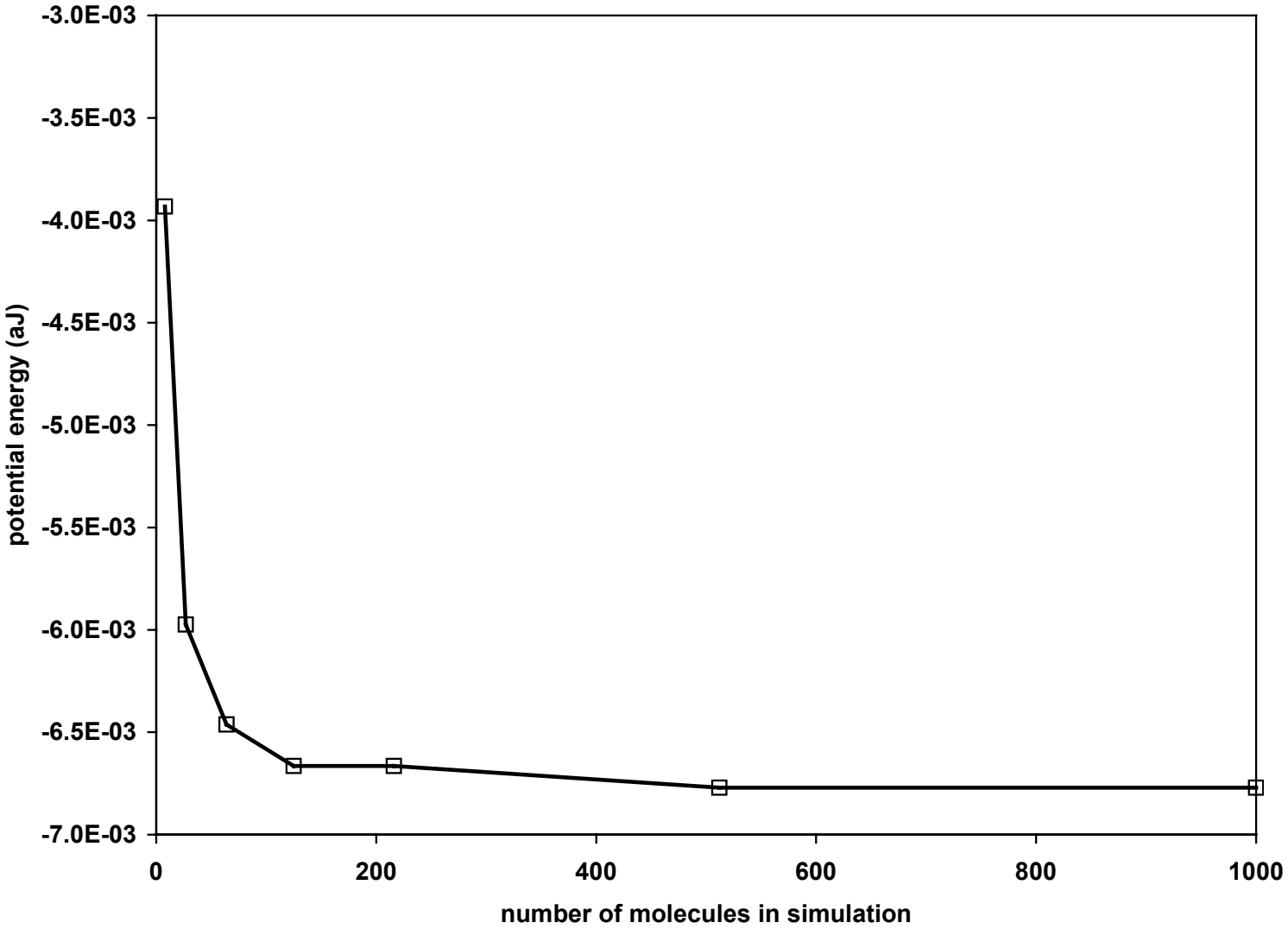

Figure 8. Average potential energy as a function of number of molecules in the simulation. Here 5000 equilibration and 50,000 productino steps were run for all simulations with $\Delta t = 2.0$ fs.

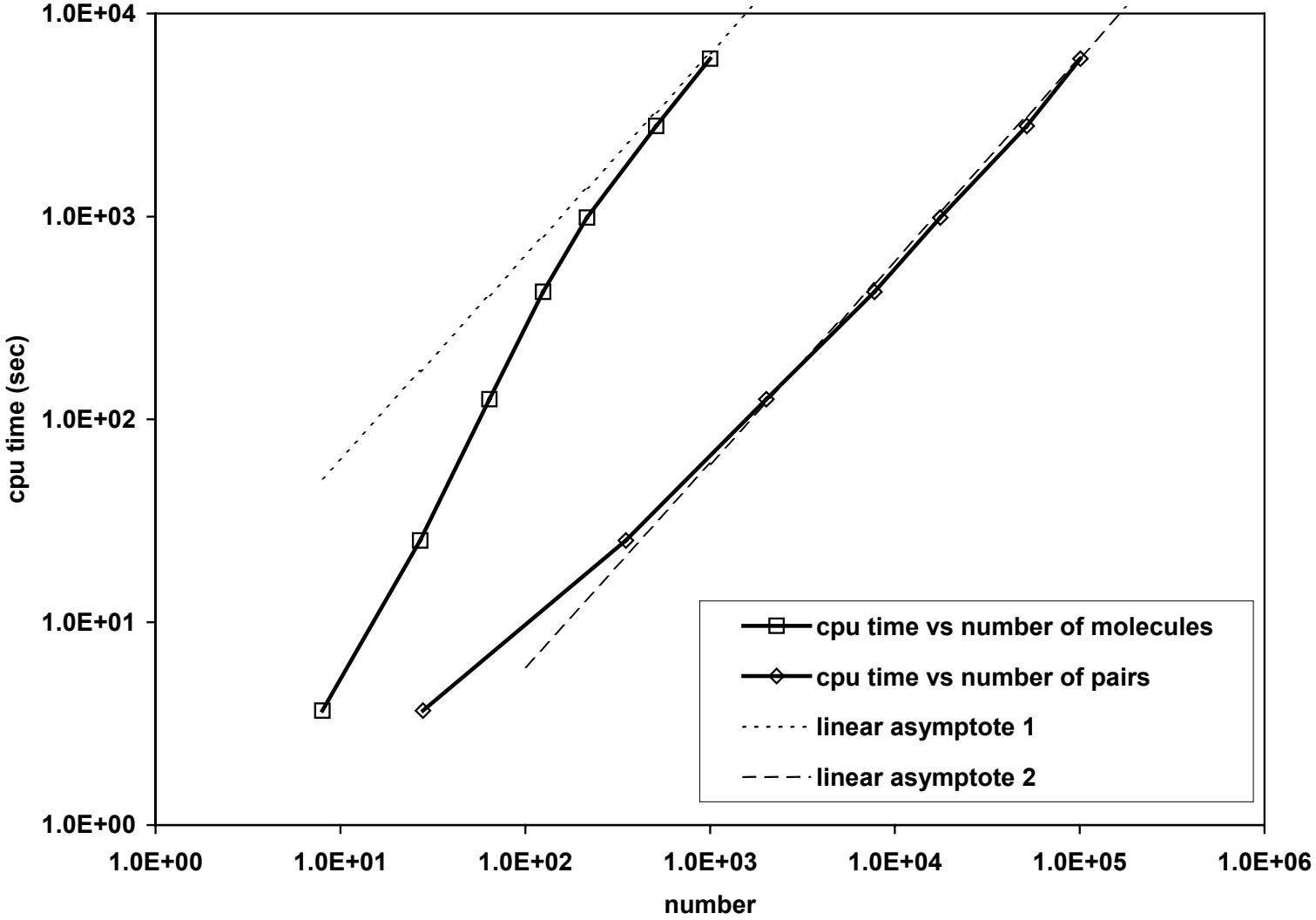

Figure 9. CPU time as a function of number of molecules in the simulation (and corresponding number of neighbor pairs). Here 5000 equilibration and 50,000 production steps were run for all simulations with $\Delta t = 2.0$ fs. The processor was a Pentium III at 600 MHz.

CPU Time per subroutine

The time in a simulation is split between all of the subroutines. In Table 3, we present the division of CPU time for a liquid methane simulation with 125 molecules, 5000 equilibrium steps, and 50,000 data production steps.

Table 3: Division of CPU Time

	without optimization		with optimization	
routine	CPU Time (sec)	fraction of time	CPU Time (sec)	fraction of time
Total	410.37	1.000000	138.60	1.000000
Initialization	0.00	0.000000	0.00	0.000000
Predictor	11.20	0.027284	2.62	0.018930
Forces	364.02	0.887083	126.60	0.913440
Corrector	11.00	0.026796	1.40	0.010115
PBC	1.53	0.003734	0.73	0.005275
Scale Velocities	0.17	0.000415	0.07	0.000506
Make Neighbors	20.89	0.050906	6.50	0.046893
Get Properties	1.45	0.003539	0.54	0.003902
Write Report	0.00	0.000000	0.00	0.000000
Save MSD	0.04	0.000098	0.08	0.000578
Other	0.06	0.000146	0.05	0.000361

We see that 88.7% of the CPU time is spent calculating the forces and energies. This is generally the case. All efforts at optimization of the code should be directed to this subroutine. The neighbor list, which was updated every 10 steps, consumed 5% of the time. The predictor and corrector sum to about 5% of the CPU time. Everything else is trace amounts.

CPU Time and Compiler Optimization

Every decent compiler has the ability to optimize source code during compilation. This optimization rearranges the order of the source lines in ways that allow them to be executed faster. Ideally, optimization does not change the results of the simulation. However, this always needs to be verified manually. The code must be run without compiler optimization and with compiler optimization, before trusting the optimization. MATLAB does not compile code; it does not have an optimizer.

The FORTRAN compiler that we are using in this package of notes is Compaq Visual Fortran Professional Edition Version 6.5.0 on a machine running Windows 2000 operating system.

We run our base case, N = 125, r_{cut} = 15 Å, k_{nbr} = 10, Δt = 2.0 fs, maxeqb = 5000, maxstp = 50,000. Comparing program output 3 and program out 4,we see that every property is identical with the exception of the CPU time. Optimization reduced the CPU time from 425 seconds without optimization to 138 seconds. In under three minutes, you can perform a complete molecular dynamics simulation of a liquid. That is pretty fast. The reduction is 67.5%. This sort of improvement is typical of good compilers.

In Table 4 we show the distribution of CPU time among the various subroutines when the optimization has been turned on. The distribution is very similar to that in the unoptimized case.

```
 funkipos: N =         125 ni =           5
  initially we have         7750 neighbor pairs
**********         equilibration Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.38831063E+00   0.38831063E+00   0.00000000E+00
Potential Energy (aJ)   -0.84616792E+00  -0.82832938E+00   0.18538254E-01
Total Energy (aJ)       -0.45785729E+00  -0.44001876E+00   0.18538254E-01
Temperature (K)          0.15000000E+03   0.15000000E+03   0.00000000E+00
x-Momentum              -0.51178995E-13  -0.16759738E-13   0.36509092E-13
y-Momentum              -0.58283979E-13  -0.30791216E-13   0.19534002E-13
z-Momentum               0.11333317E-12   0.54561265E-13   0.32809937E-13
**********         production    Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.35736819E+00   0.37523125E+00   0.15218465E-01
Potential Energy (aJ)   -0.81525473E+00  -0.83309273E+00   0.15223265E-01
Total Energy (aJ)       -0.45788654E+00  -0.45786148E+00   0.40934097E-04
Temperature (K)          0.13804729E+03   0.14494758E+03   0.58787209E+01
x-Momentum               0.15844693E-12   0.33025018E-13   0.86964724E-13
y-Momentum              -0.11391081E-12  -0.66959452E-13   0.65567218E-13
z-Momentum              -0.38182072E-13   0.30401182E-13   0.64026361E-13
 Program has used   425.551919668913     seconds of CPU time.
   This includes   425.5319    seconds of user time and  2.0028800E-02 seconds of system time.
```

Program Output 3: Output without optimization.

```
 funkipos: N =         125 ni =           5
  initially we have         7750 neighbor pairs
**********         equilibration Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.38831063E+00   0.38831063E+00   0.00000000E+00
Potential Energy (aJ)   -0.84616792E+00  -0.82832938E+00   0.18538254E-01
Total Energy (aJ)       -0.45785729E+00  -0.44001876E+00   0.18538254E-01
Temperature (K)          0.15000000E+03   0.15000000E+03   0.00000000E+00
x-Momentum              -0.51178995E-13  -0.16759738E-13   0.36509092E-13
y-Momentum              -0.58283979E-13  -0.30791216E-13   0.19534002E-13
z-Momentum               0.11333317E-12   0.54561265E-13   0.32809937E-13
**********         production    Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.35736819E+00   0.37523125E+00   0.15218465E-01
Potential Energy (aJ)   -0.81525473E+00  -0.83309273E+00   0.15223265E-01
Total Energy (aJ)       -0.45788654E+00  -0.45786148E+00   0.40934097E-04
Temperature (K)          0.13804729E+03   0.14494758E+03   0.58787209E+01
x-Momentum               0.15844693E-12   0.33025018E-13   0.86964724E-13
y-Momentum              -0.11391081E-12  -0.66959452E-13   0.65567218E-13
z-Momentum              -0.38182072E-13   0.30401182E-13   0.64026361E-13
 Program has used   138.619317531586     seconds of CPU time.
   This includes   138.3690    seconds of user time and  0.2503600 seconds of system time.
```

Program Output 4: Output with optimization.

CPU Time : Matlab vs FORTRAN

FORTRAN is a structured programming language, in which source code (understandable to humans but not to processors) is compiled into a format (the object file) which is understandable to the processor but not to the humans. This compilation process allows the code to run much faster.

Codes written in MATLAB are not compiled. Rather the source code is interpreted from as is. This results in slower code. In fact the code is much slower. To quantify this discrepancy, we can run the same molecular dynamics code in MATLAB and in FORTRAN. We run the base case with N = 125, r_{cut} = 15 Å, k_{nbr} = 10, Δt = 2.0 fs, and maxeqb = 5000. We vary maxstp.

The results are summarized in Table Four.

Table Four. CPU Usage

production steps	equilibrium + production steps	Fortran (with optimization)	Matlab
1000	6000	15 sec	4439 sec
10000	15000	38 sec	~ 13 hours
50000	55000	138 sec	forget it.

We can note a few things from this table. First, we see that MATLAB runs about 1200 times slower than FORTRAN. 1200 times! Can you believe it? MATLAB is not suitable for anything but very small problems.

Second we see that FORTRAN scales linearly with the total number (equilibration and production combined) of time steps. Naturally it should scale this way. Curiously MATLAB gets slower per time step as the number of steps increases. This must be due to some RAM problem with MATLAB. Unbelievable really.

References

1. Haile, J.M., "Molecular Dynamics Simulation", John Wiley & Sons, Inc., New York, 1992.
2. Allen, M.P., Tildesley, D.J., "Computer Simulation of Liquids", Oxford Science Publications, Oxford, 1987.
3. Frenkel, D., Smit B., "Understanding Molecular Simulation", Academic Press, San Diego, 1996.
4. Goldstein Herbert, "Classical Mechanics", 2nd Ed., Addison-Wesley Pub. Co., Reading, MA, 1980 (1st Ed, 1950).
5. Hirschfelder, J.O., Curtiss, C.F., Bird, R.B., "Molecular Theory of Gases and Liquids", John Wiley & Sons, Inc., New York, 1954, pp.1110-1113.
6. Bird, R.B., Stewart, W.E., Lightfoot, E.N., "Transport Phenomena", 2nd Ed., John Wiley & Sons, Inc., New York, 2002, pp.864-865.
7. Gear, C.W., "The Numerical Integration of Ordinary Differential Equations of Various Orders", Argonne National Laboratory, ANL-7126, 1966.
8. Gear, C.W., "Numerical Initial Value Problems in Ordinary Differential Equations", Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1971.

Appendix A. Molecular Dynamics Program in Fortran

```
      program mddriver
c
c  This code performs molecular dynamics simulations
c  in the canonical ensemble (specify T, V, and N)
c
c  Author:  David Keffer
c  Department of Chemical Engineering, University of TN
c  Last Updated:  October 6, 2001
c
cx global maxstp kmsd N dt
c*************************************************************
c  VARIABLE DEFINITIONS AND DIMENSIONS
c*************************************************************
      implicit double precision (a-h, o-z)
      integer, parameter :: N = 512 ! Number of molecules
      integer, parameter :: nprop = 8 ! number of properties
      integer, parameter :: maxnbr = N*N/2 ! number of neighbors
      logical :: lmsd, lscale             ! logical variables
      character*12 :: cmsd, cout          ! character variables
      double precision, dimension(1:nprop,1:6) :: props
      double precision, dimension(1:N,1:3) :: r, v, a, d3, d4, d5
      double precision, dimension(1:N,1:3) :: f, rwopbc
      double precision, dimension(1:5) :: dtv
      double precision, dimension(0:5) :: alpha
      integer, dimension(1:maxnbr,1:2) :: Nnbrlist
      double precision :: kb, MW
      REAL(4), dimension(1:2) :: TA
c*************************************************************
c  PROGRAM INITIALIZATION
c*************************************************************
c
c     This code uses length units of Angstroms (1.0e-10 s)
c     time = fs (1.0e-15 s)
c     xmass = (1.0e-28 kg)
c     energy = aJ (1.0e-18 J)
c     Temperature = K
c
c     Specify thermodynamic state
c
      T = 150.0d0 ! Temperature (K)
      Vn = 1.1323d+2 ! Ang^3/molecule (liq at 150 K & 1 atm)
c
c     Specify Numerical Algorithm Parameters
c
      maxeqb = 10000 ! Number of time steps during equilibration
      maxstp = 100000 ! Number of time steps during data production
      dt = 2.0d0    ! size of time step (fs)
c
c     Specify pairwise potential parameters
c
      sig = 3.884d0     ! collision diameter (Angstroms)
      eps = 137.d0      ! well depth (K)
      MW = 16.0420d0          ! molecular weight (grams/mole)
      rcut = 15.d0      ! cut-off distance for potential (Angstroms)
c
c     Specify sampling intervals
c
```

```
      ksamp = 1          ! sampling interval
      knbr = 10          ! neighbor list update interval
      kwrite = 5000      ! writing interval
      kmsd = 100         ! position save for mean square displacement
      rnbr = rcut + 3.d0
c
c     Logical Variables
c
      lmsd = .false.          ! logical variable for mean square displacement
      lscale = .true.         ! logical variable for temperature scaling
c
c     Character Variables
c
      cmsd = 'md_msd.out'
      cout = 'md_sum.out'
      open(unit=1,file=cout,form='formatted',status='unknown')
      if (lmsd) then
            open(unit=2,file=cmsd,form='formatted',status='unknown')
      endif
c
c     props
c     first index is property
c     property 1:  total kinetic energy
c     property 2:  total potential energy
c     property 3:  total energy
c     property 4:  temperature
c     property 5:  total x-momentum
c     property 6:  total y-momentum
c     property 7:  total z-momentum
c     property 8:  pressure
c
c     second index is
c     1:  instantaneous value
c     2:  sum
c     3:  sum of squares
c     4:  average
c     5:  variance
c     6:  standard deviation
c
      props(:,:) = 0.d0
c
c  Initialize vectors
c
c   first index of r is over molecules
c   second index of r is over dimensionality (x,y,z)
      r(:,:) = 0.d0           ! position
      v(:,:) = 0.d0           ! velocity
      a(:,:) = 0.d0           ! acceleration
      d3(:,:) = 0.d0          ! third derivative
      d4(:,:) = 0.d0          ! fourth derivative
      d5(:,:) = 0.d0          ! fifth derivative
      f(:,:) = 0.d0           ! force
      rwopbc(:,:) = 0.d0      ! position w/o pbc
c
c*************************************************************
c  INITIALIZATION PART TWO
c*************************************************************
c
c     compute a few parameters
c
```

```
      dt2 = dt*dt
      dt2h = 0.5d0*dt2
      Vol = dfloat(N)*Vn            ! total volume (Angstroms**3)
      side = Vol**(1.d0/3.d0) ! length of side of simulation volume (Angstrom)
      sideh = 0.5d0*side            ! half of the side
      density = 1.d0/Vn       ! molar density
      sig6 = sig**6.d0
      sig12 = sig**12.d0
      rcut2 = rcut*rcut
      rnbr2 = rnbr*rnbr
c     stuff for long range energy correction
      rcut3 = rcut**3.d0
      rcut9 = rcut**9.d0
      kb = 1.380660d-5 ! Boltzmann's constant (aJ/molecule/K)
      eps = eps*kb
      pi = 2.d0*dasin(1.d0)
      ulongpre = dfloat(N)*8.d0*eps*pi*density
      ulong = ulongpre*( sig12/(9.d0*rcut9) - sig6/(3.d0*rcut3) )
      vlongpre = 96.d0*eps*pi*density
      vlong = -vlongpre*( sig12/(9.d0*rcut9) - sig6/(6.d0*rcut3) )
c     temperature factor for velocity scaling
      xNav = 6.0220d+23 ! Avogadro's Number
      xmass = MW/xNav/1000.d0*1.0d+28 ! (1e-28*kg/molecule)
      xmassi = 1.d0/xmass
      tfac = 3.d0*float(N)*kb*T*xmassi  ! (Angstrom/fs)**2
c     correction factors for numerical algorithm
      dtv = dt;
      do i = 2, 5, 1
            dtv(i) = dtv(i-1)*dt/dfloat(i)
      enddo
      alpha(0) = 3.d0/20.d0
      alpha(1) = 251.d0/360.d0
      alpha(2) = 1.d0
      alpha(3) = 11.d0/18.d0
      alpha(4) = 1.d0/6.d0
      alpha(5) = 1.d0/60.d0
      fact = 1.d0
      do i = 1, 5, 1
            fact = fact*dfloat(i)
            alpha(i) = alpha(i)*dt**(-dfloat(i))*fact
      enddo
      alpha = alpha*dt2*0.5d0
c
c     assign initial positions of  molecules in FCC crystal structure
c
      call funk_ipos(N, side, r, rwopbc)
c
c     assign initial velocities
c
      call funk_ivel(N,v,T,tfac)
c
c     create neighbor list
c
      call funk_mknbr(N,r,rnbr2,side,sideh, Nnbr, Nnbrlist, maxnbr)
      print *, ' initially we have ', Nnbr, ' neighbor pairs'
c
c     evaluate initial forces and potential energy
c
      call funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,eps,
     & f, U, virial, maxnbr)
```

```
      a = f*xmassi  ! initial acceleration
      call funk_getprops(N,v,xmass,T,kb,U,props,nprop,density,
     & virial,ulong,vlong)
      write (6,1001) 0, props(1:4,1)
      write (1,1001) 0, props(1:4,1)
 1001 format(i7,' KE',e16.8,' PE',e16.8,' E',e16.8,' T',e14.7)
      props = 0.d0
c*************************************************************
c  EQUILIBRATION
c*************************************************************
      do istep = 1, maxeqb, 1
c           predict new positions
            call predictor(N,r,rwopbc,v,a,d3,d4,d5,dtv)
c           evaluate forces and potential energy
            call funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,
     &      eps, f, U, virial, maxnbr)
c           correct new positions
            call corrector(N,r,rwopbc,v,a,d3,d4,d5,f,dt2h,alpha,xmassi)
c           apply periodic boundary conditions
            call pbc(N,r,side)
c           scale velocities
            if (lscale) then
                  call funk_scalev(N,v,T,tfac)
            endif
c           update neighbor list
            if (mod(istep,knbr) .eq. 0) then
                  call funk_mknbr(N,r,rnbr2,side,sideh,Nnbr,Nnbrlist,maxnbr)
            endif
c           sample properties
            if (mod(istep,ksamp) .eq. 0) then
                call funk_getprops(N,v,xmass,T,kb,U,props,nprop,density,
     &        virial,ulong,vlong)
            endif
c           write periodic results
            if (mod(istep,kwrite) .eq. 0) then
                  write (6,1001) istep, props(1:4,1)
                  write (1,1001) istep, props(1:4,1)
            endif
      enddo
c
c     write equilibration results
c
      if (maxeqb .gt. ksamp) then
            call funk_report(N,props,nprop,maxeqb,ksamp,'equilibration')
      endif
c*************************************************************
c  PRODUCTION
c*************************************************************
      props = 0.d0
      lscale = .false.
      if (lmsd) then
            call funk_msd(N,rwopbc)
      endif
      do istep = 1, maxstp, 1
c           predict new positions
            call predictor(N,r,rwopbc,v,a,d3,d4,d5,dtv)
c           evaluate forces and potential energy
            call funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,
     &      eps, f, U, virial, maxnbr)
c           correct new positions
```

```
            call corrector(N,r,rwopbc,v,a,d3,d4,d5,f,dt2h,alpha,xmassi)
c           apply periodic boundary conditions
            call pbc(N,r,side)
c           scale velocities
            if (lscale) then
                  call funk_scalev(N,v,T,tfac)
            endif
c           update neighbor list
            if (mod(istep,knbr) .eq. 0) then
                  call funk_mknbr(N,r,rnbr2,side,sideh,Nnbr,Nnbrlist,maxnbr)
            endif
c           sample properties
            if (mod(istep,ksamp) .eq. 0) then
                call funk_getprops(N,v,xmass,T,kb,U,props,nprop,density,
     &        virial,ulong,vlong)
            endif
c           write periodic results
            if (mod(istep,kwrite) .eq. 0) then
                  write (6,1001) istep, props(1:4,1)
                  write (1,1001) istep, props(1:4,1)
            endif
c           save positions for mean square displacement
            if (lmsd) then
                  if (mod(istep,kmsd) .eq. 0) then
                        call funk_msd(N,rwopbc)
                  endif
            endif
      enddo

c
c     write equilibration results
c
      if (maxstp .gt. ksamp) then
            call funk_report(N,props,nprop,maxstp,ksamp,'production   ')
      endif
c
      ttot = ETIME(TA)
      write(*,*) 'Program has used', ttot, 'seconds of CPU time.'
      write(*,*) '  This includes', TA(1), 'seconds of user time and',
     & TA(2), 'seconds of system time.'
      write(1,*) 'Program has used', ttot, 'seconds of CPU time.'
      write(1,*) '  This includes', TA(1), 'seconds of user time and',
     & TA(2), 'seconds of system time.'
c
      close(unit=1,status='keep')
      if (lmsd) then
            close(unit=2,status='keep')
      endif
c
      stop
      end




c*************************************************************
c  SUBROUTINES
c*************************************************************
```

```
c
c  funk_ipos:  assigns initial positions
c
      subroutine funk_ipos(N,side,r,rwopbc)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N
      double precision, intent(in) :: side
      double precision, intent(out), dimension(1:N,1:3) :: r, rwopbc
      xi = dfloat(N)**(1.d0/3.d0)
      ni = int(xi)
      if (xi - dfloat(ni) .gt. 1.d-14) then
            ni = ni + 1
      endif
      print *, 'funkipos: N = ', N, ' ni = ', ni
      ncount = 0
      dx = side/dfloat(ni)
      do ix = 1, ni, 1
            do iy = 1, ni, 1
                  do iz = 1, ni, 1
                        ncount = ncount + 1
                        if (ncount .le. N) then
                              r(ncount,1) = dx*dfloat(ix)
                              r(ncount,2) = dx*dfloat(iy)
                              r(ncount,3) = dx*dfloat(iz)
                        endif
                  enddo
            enddo
      enddo
      rwopbc = r
      return
      end


c
c  funk_ivel:  assigns initial velocities
c
      subroutine funk_ivel(N,v,T,tfac)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N
      double precision, intent(in) :: T, tfac
      double precision, intent(out), dimension(1:N,1:3) :: v
      double precision, dimension(1:3) :: sumv
c
      call random_number(v)   ! random velocities from 0 to 1
      v = 2.d0*v - 1.d0       ! random velocities from -1 to 1
c     enforce zero net momentum
      do i = 1, 3, 1
            sumv(i) = sum(v(1:N,i))
            v(1:N,i) = v(1:N,i) - sumv(i)/dfloat(N)
      enddo
c     scale initial velocities to set point temperature
      sumvsq = sum(sum(v*v,1) )
      fac = dsqrt(tfac/sumvsq)
      v = v*fac
      return
      end



c
```

```
c  funk_mknbr:  create neighbor list
c
      subroutine funk_mknbr(N,r,rnbr2,side,sideh,Nnbr,Nnbrlist,maxnbr)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N, maxnbr
      double precision, intent(in) :: side, sideh, rnbr2
      double precision, intent(in), dimension(1:N,1:3) :: r
      integer, intent(out) :: Nnbr
      integer, intent(out), dimension(1:maxnbr,1:2) :: Nnbrlist
      double precision, dimension(1:3) :: dis
      Nnbr = 0
      do i = 1, N, 1
            do j = i+1, N, 1
                  dis(1:3) = r(i,1:3) - r(j,1:3)
                  do k = 1, 3, 1
                        if (dis(k) .gt.  sideh) dis(k) = dis(k) - side
                        if (dis(k) .lt. -sideh) dis(k) = dis(k) + side
                  enddo
                  dis2 = sum(dis*dis)
                  if (dis2 .le. rnbr2) then
                        Nnbr = Nnbr + 1
                        Nnbrlist(Nnbr,1) = i
                        Nnbrlist(Nnbr,2) = j
                  endif
            enddo
      enddo
      return
      end



c
c  funk_force:  evaluate forces
c
      subroutine funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,
     & sig12, eps, f, U, virial, maxnbr)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N, maxnbr, Nnbr
      double precision, intent(in) :: side, sideh, rcut2
      double precision, intent(in) :: sig6, sig12, eps
      double precision, intent(in), dimension(1:N,1:3) :: r
      integer, intent(in), dimension(1:maxnbr,1:2) :: Nnbrlist
      double precision, intent(out), dimension(1:N,1:3) :: f
      double precision, intent(out) :: U, virial
      double precision, dimension(1:3) :: dis
      f = 0.d0    ! forces
      U = 0.d0    ! potential energy
      virial = 0.d0 ! virial coefficient
      do m = 1, Nnbr, 1
            i = Nnbrlist(m,1)
            j = Nnbrlist(m,2)
            dis(1:3) = r(i,1:3) - r(j,1:3)
            do k = 1, 3, 1
                  if (dis(k) .gt.  sideh) dis(k) = dis(k) - side
                  if (dis(k) .lt. -sideh) dis(k) = dis(k) + side
            enddo
            dis2 = sum(dis*dis)
            if (dis2 .le. rcut2) then
                  dis2i = 1.d0/dis2
                  dis6i = dis2i*dis2i*dis2i
```

```
                  dis12i = dis6i*dis6i
                  U = U + ( sig12*dis12i - sig6*dis6i )
                  fterm = (2.d0*sig12*dis12i - sig6*dis6i )*dis2i
                  f(i,1:3) = f(i,1:3) + fterm*dis(1:3)
                  f(j,1:3) = f(j,1:3) - fterm*dis(1:3)
                  virial = virial - fterm*dis2
            endif
      enddo
      f = f*24.d0*eps
      U = U*4.d0*eps
      virial=virial*24.d0*eps
      return
      end



c
c predict new positions
c
      subroutine predictor(N,r,rwopbc,v,a,d3,d4,d5,dtv)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N
      double precision, intent(inout), dimension(1:N,1:3) ::
     & r,rwopbc, v, a, d3, d4, d5
      double precision, intent(in), dimension(1:5) :: dtv
      rwopbc = rwopbc + v *dtv(1) + dtv(2)*a  + dtv(3)*d3 + dtv(4)*d4 +
     &                                                      dtv(5)*d5
      r      = r      + v *dtv(1) + dtv(2)*a  + dtv(3)*d3 + dtv(4)*d4 +
     &                                                      dtv(5)*d5
      v      = v      + a *dtv(1) + dtv(2)*d3 + dtv(3)*d4 + dtv(4)*d5
      a      = a      + d3*dtv(1) + dtv(2)*d4 + dtv(3)*d5
      d3     = d3     + d4*dtv(1) + dtv(2)*d5
      d4     = d4     + d5*dtv(1)
      return
      end
c
c correct new positions
c
      subroutine corrector(N,r,rwopbc,v,a,d3,d4,d5,f,dt2h,alpha,xmassi)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N
      double precision, intent(inout), dimension(1:N,1:3) ::
     & r,rwopbc, v, a, d3, d4, d5
      double precision, intent(in), dimension(1:N,1:3) :: f
      double precision, intent(in) :: dt2h, xmassi
      double precision, intent(in), dimension(0:5) :: alpha
      double precision, dimension(1:3) :: errvec
      do i = 1, N, 1
            errvec(1:3) = ( f(i,1:3)*xmassi - a(i,1:3) )
            rwopbc(i,1:3) = rwopbc(i,1:3) + errvec(1:3)*alpha(0)
            r(i,1:3)  = r(i,1:3)  + errvec(1:3)*alpha(0)
            v(i,1:3)  = v(i,1:3)  + errvec(1:3)*alpha(1)
            a(i,1:3)  = a(i,1:3)  + errvec(1:3)*alpha(2)
            d3(i,1:3) = d3(i,1:3) + errvec(1:3)*alpha(3)
            d4(i,1:3) = d4(i,1:3) + errvec(1:3)*alpha(4)
            d5(i,1:3) = d5(i,1:3) + errvec(1:3)*alpha(5)
      enddo
      return
      end
```

```
c
c apply periodic boundary conditions
c
      subroutine pbc(N,r,side)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N
      double precision, intent(inout), dimension(1:N,1:3) :: r
      double precision, intent(in) :: side
      do i = 1, N, 1
            do j = 1, 3, 1
                  if (r(i,j) .gt.  side) r(i,j) = r(i,j) - side
                  if (r(i,j) .lt.  0.0)  r(i,j) = r(i,j) + side
            enddo
      enddo
      return
      end



c
c  funk_scalev:  scale velocities
c
      subroutine funk_scalev(N,v,T,tfac)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N
      double precision, intent(inout), dimension(1:N,1:3) :: v
      double precision, intent(in) :: T, tfac
c     scale velocities to set point temperature
      sumvsq = sum(sum(v*v,1) )
      fac = sqrt(tfac/sumvsq)
      v = v*fac
      return
      end


c
c calculate properties for sampling
c
      subroutine funk_getprops(N,v,xmass,T,kb,U,props,nprop,density,
     & virial,ulong,vlong)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N, nprop
      double precision, intent(in), dimension(1:N,1:3) :: v
      double precision, intent(in) :: T, kb, xmass, U, density, virial
      double precision, intent(in) :: ulong, vlong
      double precision, intent(inout), dimension(1:nprop,1:6) :: props
c     props
c     first index is property
c     property 1:  total kinetic energy
c     property 2:  total potential energy
c     property 3:  total energy
c     property 4:  temperature
c     property 5:  total x-momentum
c     property 6:  total y-momentum
c     property 7:  total z-momentum
c
c     second index is
```

```
c     1:  instantaneous value
c     2:  sum
c     3:  sum of squares
c     4:  average
c     5:  variance
c     6:  standard deviation
c
      sumvsq = sum(sum(v*v,1) )
      xKE = 0.5d0*xmass*sumvsq ! (aJ)
      Ti = 2.d0/(3.d0*dfloat(N)*kb)*xKE
c
c     get instantaneous values
c
      props(1,1) = xKE
      props(2,1) = U + ulong
      props(3,1) = xKE + U + ulong
      props(4,1) = Ti
      props(5,1) = xmass*sum(v(:,1))
      props(6,1) = xmass*sum(v(:,2))
      props(7,1) = xmass*sum(v(:,3))
      props(8,1) = density*(kb*Ti - virial/(3.D0*dfloat(N)) -vlong/3.d0)
c
c     get the cumulative sum and the cumulative sum of the squares
c
      props(1:nprop,2) = props(1:nprop,2) + props(1:nprop,1)
      props(1:nprop,3) = props(1:nprop,3) +
     &                   props(1:nprop,1)*props(1:nprop,1)
      return
      end

c
c calculate and report simulation statistics
c
      subroutine funk_report(N,props,nprop,maxeqb,ksamp,csect)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N, nprop, maxeqb, ksamp
      double precision, intent(inout), dimension(1:nprop,1:6) :: props
      character*13, intent(in) :: csect
      character*22, dimension(1:nprop) :: propname
      den = dfloat(maxeqb/ksamp)
      props(1:nprop,4) = props(1:nprop,2)/den
      props(1:nprop,5) = props(1:nprop,3)/den - props(1:nprop,4)**2.d0
      do i = 1, nprop, 1
            if (props(i,5) .gt. 0.d0) then
                  props(i,6) = dsqrt(props(i,5))
            else
                  props(i,6) = 0.d0
            endif
      enddo
      propname(1) = 'Kinetic Energy (aJ)   '
      propname(2) = 'Potential Energy (aJ) '
      propname(3) = 'Total Energy (aJ)     '
      propname(4) = 'Temperature (K)       '
      propname(5) = 'x-Momentum            '
      propname(6) = 'y-Momentum            '
      propname(7) = 'z-Momentum            '
      propname(8) = 'Pressure aJ/Angstorm^3'
      write(6,1002) csect
      write(1,1002) csect
 1002 format ('********** ', a22 ' Completed **********')
```

```
      write(6,1003)
      write(1,1003)
 1003 format ('property               instant       average       s',
     & 'tandard deviation')
      do i = 1, nprop, 1
            write(6,1004) propname(i), props(i,1),props(i,4),props(i,6)
            write(1,1004) propname(i), props(i,1),props(i,4),props(i,6)
      enddo
 1004 format(a22,3(1x,e16.8))
      return
      end
c
c  save positions for mean square displacement calculations
c
      subroutine funk_msd(N,rwopbc)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N
      double precision, intent(in), dimension(1:N,1:3) :: rwopbc
      do i = 1, N, 1
            write(2,1005) rwopbc(i,1:3)
      enddo
 1005 format(3(e16.8,1x))
      return
      end
```

Appendix B. Molecular Dynamics Program in Matlab

```
function mddriver
%
%  This code performs molecular dynamics simulations
%  in the canonical ensemble (specify T, V, and N)
%
%  Author:  David Keffer
%  Department of Chemical Engineering, University of TN
%  Last Updated:  September 19, 2001
%
%global maxstp kmsd N dt

%*************************************************************
%  PROGRAM INITIALIZATION
%*************************************************************

%
%  This code uses length units of Angstroms (1.0e-10 s)
%  time = fs (1.0e-15 s)
%  mass = (1.0e-28 kg)
%  energy = aJ (1.0e-18 J)
%  Temperature = K
%

%
%  Specify thermodynamic state
%
T = 300;   % Temperature (K)
Vn = 512.0; % Angstroms cubed / molecule
N = 27;   % Number of molecules

%
%  Specify Numerical Algorithm Parameters
%
maxeqb = 2500; % Number of time steps during equilibration
maxstp = 2000; % Number of time steps during data production
dt = 1.0e-0;     % size of time step (fs)

%
%  Specify pairwise potential parameters
%
```

```
sig = 3.884; % collision diameter (Angstroms)
eps = 137;   % well depth (K)
MW = 16.0;   % molecular weight (grams/mole)
rcut = 15;   % cut-off distance for potential (Angstroms)

%
%  Specify sampling intervals
%
nprop = 7;  % number of properties
ksamp = 1; % sampling interval
knbr = 10;  % neighbor list update interval
kwrite = 100;  % writing interval
kmsd = 10; % position save for mean square displacement
rnbr = rcut + 3.0;
fid_msd = fopen('md_msd.out','w');
%
%  props
%  first index is property
%  property 1:  total kinetic energy
%  property 2:  total potential energy
%  property 3:  total energy
%  property 4:  temperature
%  property 5:  total x-momentum
%  property 6:  total y-momentum
%  property 7:  total z-momentum
%
%  second index is
%  1:  instantaneous value
%  2:  sum
%  3:  sum of squares
%  4:  average
%  5:  variance
%  6:  standard deviation
%
props = zeros(nprop,6);

%
%  Initialize vectors
%
%   first index of r is over molecules
%   second index of r is over dimensionality (x,y,z)
r = zeros(N,3);      % position
v = zeros(N,3);      % velocity
```

```
a = zeros(N,3);       % acceleration
d3 = zeros(N,3);      % third derivative
d4 = zeros(N,3);      % fourth derivative
d5 = zeros(N,3);      % fifth derivative
f = zeros(N,3);       % force
rwopbc = zeros(N,3); % position w/o pbc

%
%  additional simulation parameters
%
lmsd = 1;          % logical variable for mean square displacement
lscale = 1;        % logical variable for temperature scaling

%*************************************************************
%  INITIALIZATION PART TWO
%*************************************************************

%
%  compute a few parameters
%
dt2 = dt*dt;
dt2h = 0.5*dt2;
Vol = N*Vn;             % total volume (Angstroms^3)
side = Vol^(1.0/3.0);  % length of side of simulation volume (Angstrom)
sideh = 0.5*side;     % half of the side
density = 1/Vn;       % molar density
sig6 = sig^6;
sig12 = sig^12;
rcut2 = rcut*rcut;
rnbr2 = rnbr*rnbr;
%  stuff for long range energy correction
rcut3 = rcut^3;
rcut9 = rcut^9;
kb = 1.38066e-5; % Boltzmann's constant (aJ/molecule/K) CHECK
eps = eps*kb;
Ulong = N*8*eps*pi*density*( sig12/(9.0*rcut9) - sig6/(3.0*rcut3) ); % CHECK
% temperature factor for velocity scaling
Nav = 6.022e+23; % Avogadro's Number
mass = MW/Nav/1000*1.0e+28; % (1e-28*kg/molecule)
tfac = 3.0*N*kb*T/mass;  % (Angstrom/fs)^2
% correction factors for numerical algorithm
fv = [1 2 6 24 120];  % vector of factorials
dtva = [dt dt*dt dt*dt*dt dt*dt*dt*dt dt*dt*dt*dt*dt];
```

```
dtv = dtva./fv;
%  corrector coefficients for Gear using dimensioned variables
gear = [3.0/20.0 251.0/360.0 1.0 11.0/18.0 1.0/6.0 1.0/60.0];
dtv6 = [1 dt dt*dt dt*dt*dt dt*dt*dt*dt dt*dt*dt*dt*dt];
fv6 = [1 1 2 6 24 120];  % vector of factorials
alpha(1:6) = gear(1:6)./dtv6(1:6).*fv6;
alpha = alpha*dt^2/2;
%
%  assign initial positions of  molecules in FCC crystal structure
%
[r,rwopbc] = funk_ipos(N,side,r,rwopbc);
%
%  assign initial velocities
%
v = funk_ivel(N,v,T,tfac);
%
%  create neighbor list
%
[Nnbr,Nnbrlist] = funk_mknbr(N,r,rnbr2,side,sideh);
%
% evaluate initial forces and potential energy
%
[f,U] = funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,eps);
a = f/mass;  % initial acceleration

[props] = funk_getprops(N,v,mass,T,kb,U,props,nprop);
fprintf(1,'istep %i K %e U %e TOT %e  T %e \n',0,props(1:4,1));
props = zeros(nprop,6);
%*************************************************************
%  EQUILIBRATION
%*************************************************************
for istep = 1:1:maxeqb
   % predict new positions
   [r,v,a,d3,d4,rwopbc] = predictor(N,r,rwopbc,v,a,d3,d4,d5,dtv);
   % evaluate forces and potential energy
   [f,U] = funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,eps);
   % correct new positions
   [r,v,a,d3,d4,d5,rwopbc] = corrector(N,r,rwopbc,v,a,d3,d4,d5,f,dt2h,alpha,mass);
   %a = f/mass;
   % apply periodic boundary conditions
   r = pbc(N,r,side);
   % scale velocities
   if (lscale == 1)
```

```
      v = funk_scalev(N,v,T,tfac);
   end
   % update neighbor list
   if (mod(istep,knbr) == 0)
      [Nnbr,Nnbrlist] = funk_mknbr(N,r,rnbr2,side,sideh);
   end
   % sample properties
   if (mod(istep,ksamp) == 0)
      [props] = funk_getprops(N,v,mass,T,kb,U,props,nprop);
   end
   % save positions for mean square displacement
   if (mod(istep,kwrite) == 0)
      fprintf(1,'istep %i K %e U %e TOT %e  T %e \n',istep,props(1:4,1));
   end
end
%
% write equilibration results
%
if (maxeqb > ksamp)
   [props] = funk_report(N,props,nprop,maxeqb,ksamp);
end
%************************************************************
%  PRODUCTION
%************************************************************
props = zeros(nprop,6);
lscale = 0;
if (lmsd)
   funk_msd(N,rwopbc,fid_msd);
end

for istep = 1:1:maxstp
   % predict new positions
   [r,v,a,d3,d4,rwopbc] = predictor(N,r,rwopbc,v,a,d3,d4,d5,dtv);
   % evaluate forces and potential energy
   [f,U] = funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,eps);
   % correct new positions
   [r,v,a,d3,d4,d5,rwopbc] = corrector(N,r,rwopbc,v,a,d3,d4,d5,f,dt2h,alpha,mass);
%    % apply periodic boundary conditions
   r = pbc(N,r,side);
   % scale velocities
   if (lscale == 1)
%       v = scalev();
   end
```

```
   % update neighbor list
   if (mod(istep,knbr) == 0)
      [Nnbr,Nnbrlist] = funk_mknbr(N,r,rnbr2,side,sideh);
   end
   % sample properties
   if (mod(istep,ksamp) == 0)
      [props] = funk_getprops(N,v,mass,T,kb,U,props,nprop);
   end
   % save positions for mean square displacement
   if (mod(istep,kwrite) == 0)
      fprintf(1,'istep %i K %e U %e TOT %e  T %e \n',istep,props(1:4,1));
   end
   if (lmsd)
      if (mod(istep,kmsd) == 0)
         funk_msd(N,rwopbc,fid_msd);
      end
   end
end
%
if (maxstp > ksamp)
   [props] = funk_report(N,props,nprop,maxstp,ksamp);
end
fclose(fid_msd);




%*************************************************************
%  SUBROUTINES
%*************************************************************

%
%  funk_ipos:  assigns initial positions
%
function [r,rwopbc] = funk_ipos(N,side,r,rwopbc);
ni = ceil(N^(1.0/3.0));
ncount = 0;
dx = side/ni;
for ix = 1:1:ni
   for iy = 1:1:ni
      for iz = 1:1:ni
         ncount = ncount + 1;
```

```
         if (ncount <= N)
            r(ncount,1) = dx*ix;
            r(ncount,2) = dx*iy;
            r(ncount,3) = dx*iz;
         end
      end
   end
end
rwopbc = r;

%
%  funk_ivel:  assigns initial velocities
%
function v = funk_ivel(N,v,T,tfac);
%
v = rand(N,3);     % random velocities from 0 to 1
v = 2*v - 1.0;     % random velocities from -1 to 1
% enforce zero net momentum
for i = 1:1:3
   sumv(i) = sum(v(1:N,i));
   v(1:N,i) = v(1:N,i) - sumv(i)/N;
end
% scale initial velocities to set point temperature
sumvsq = sum(sum(v.*v,1) );
fac = sqrt(tfac/sumvsq);
v = v*fac;

%
%  funk_mknbr:  create neighbor list
%
function [Nnbr,Nnbrlist] = funk_mknbr(N,r,rnbr2,side,sideh);
Nnbr = 0;
for i = 1:1:N
   for j = i+1:1:N
      dis(1:3) = r(i,1:3) - r(j,1:3);
      for k = 1:1:3
         if (dis(k) >  sideh); dis(k) = dis(k) - side; end;
         if (dis(k) < -sideh); dis(k) = dis(k) + side; end;
      end
      dis2 = sum(dis.*dis);
      if (dis2 <= rnbr2)
         Nnbr = Nnbr + 1;
         Nnbrlist(Nnbr,1) = i;
```

```
         Nnbrlist(Nnbr,2) = j;
      end
   end
end
if (Nnbr == 0)
   Nnbrlist = zeros(1,1);
end

%
%  funk_force:  evaluate forces
%
function [f,U] = funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,eps);
f = zeros(N,3);      % force
U = 0.0;             % potential energy
for n = 1:1:Nnbr
   i = Nnbrlist(n,1);
   j = Nnbrlist(n,2);
   dis(1:3) = r(i,1:3) - r(j,1:3);
   for k = 1:1:3
      if (dis(k) >  sideh); dis(k) = dis(k) - side; end;
      if (dis(k) < -sideh); dis(k) = dis(k) + side; end;
   end
   dis2 = sum(dis.*dis);
   if (dis2 <= rcut2)
      dis2i = 1.0/dis2;
      dis6i = dis2i*dis2i*dis2i;
      dis12i = dis6i*dis6i;
      U = U + ( sig12*dis12i - sig6*dis6i );
      fterm = (2.0*sig12*dis12i - sig6*dis6i )*dis2i;
      %fprintf(1, 'n %i i %i j %i dis2 %e fterm %e %e %e\n',n,i,j,dis2,fterm);
      f(i,1:3) = f(i,1:3) + fterm.*dis(1:3);
      f(j,1:3) = f(j,1:3) - fterm.*dis(1:3);
   end
end
f = f*24.0*eps;
U = U*4.0*eps;

%
% predict new positions
%
function [r,v,a,d3,d4,rwopbc] = predictor(N,r,rwopbc,v,a,d3,d4,d5,dtv)
rwopbc = rwopbc + v *dtv(1) + dtv(2)*a  + dtv(3)*d3 + dtv(4)*d4 + dtv(5)*d5;
r      = r      + v *dtv(1) + dtv(2)*a  + dtv(3)*d3 + dtv(4)*d4 + dtv(5)*d5;
```

```
v      = v      + a *dtv(1) + dtv(2)*d3 + dtv(3)*d4 + dtv(4)*d5;
a      = a      + d3*dtv(1) + dtv(2)*d4 + dtv(3)*d5;
d3     = d3     + d4*dtv(1) + dtv(2)*d5;
d4     = d4     + d5*dtv(1);

%
% correct new positions
%
function [r,v,a,d3,d4,d5,rwopbc] = corrector(N,r,rwopbc,v,a,d3,d4,d5,f,dt2h,alpha,mass);
for i = 1:1:N
   errvec(1:3) = ( f(i,1:3)/mass - a(i,1:3) );
   rwopbc(i,1:3) = rwopbc(i,1:3) + errvec(1:3)*alpha(1);
   r(i,1:3)  = r(i,1:3)  + errvec(1:3)*alpha(1);
   v(i,1:3)  = v(i,1:3)  + errvec(1:3)*alpha(2);
   a(i,1:3)  = a(i,1:3)  + errvec(1:3)*alpha(3);
   d3(i,1:3) = d3(i,1:3) + errvec(1:3)*alpha(4);
   d4(i,1:3) = d4(i,1:3) + errvec(1:3)*alpha(5);
   d5(i,1:3) = d5(i,1:3) + errvec(1:3)*alpha(6);
end

%
% apply periodic boundary conditions
%
function r = pbc(N,r,side);
for i = 1:1:N
   for j = 1:1:3
      if (r(i,j) >  side); r(i,j) = r(i,j) - side; end;
      if (r(i,j) <  0.0); r(i,j) = r(i,j) + side; end;
   end
end

%
%  funk_scalev:  scale velocities
%
function v = funk_scalev(N,v,T,tfac);
% scale velocities to set point temperature
sumvsq = sum(sum(v.*v,1) );
fac = sqrt(tfac/sumvsq);
v = v*fac;


%
% calculate properties for sampling
```

```
%
function [props] = funk_getprops(N,v,mass,T,kb,U, props,nprop);
%  props
%  first index is property
%  property 1:  total kinetic energy
%  property 2:  total potential energy
%  property 3:  total energy
%  property 4:  temperature
%  property 5:  total x-momentum
%  property 6:  total y-momentum
%  property 7:  total z-momentum
%
%  second index is
%  1:  instantaneous value
%  2:  sum
%  3:  sum of squares
%  4:  average
%  5:  variance
%  6:  standard deviation
%
sumvsq = sum(sum(v.*v,1) );
KE = 0.5*mass*sumvsq; % (aJ)
T = 2/(3*N*kb)*KE;
%
% get instantaneous values
%
props(1,1) = KE;
props(2,1) = U;
props(3,1) = KE+U;
props(4,1) = T;
props(5,1) = mass*sum(v(:,1));
props(6,1) = mass*sum(v(:,2));
props(7,1) = mass*sum(v(:,3));
props(1:nprop,2) = props(1:nprop,2) + props(1:nprop,1);
props(1:nprop,3) = props(1:nprop,3) + props(1:nprop,1).*props(1:nprop,1);

%
% calculate and report simulation statistics
%
function [props] = funk_report(N,props,nprop,maxeqb,ksamp);
den = floor(maxeqb/ksamp);
props(1:nprop,4) = props(1:nprop,2)/den;
props(1:nprop,5) = props(1:nprop,3)/den - props(1:nprop,4).^2;
```

```
props(1:nprop,6) = sqrt(props(1:nprop,5));
propname{1} = 'Kinetic Energy (aJ)   ';
propname{2} = 'Potential Energy (aJ) ';
propname{3} = 'Total Energy (aJ)     ';
propname{4} = 'Temperature (K)       ';
propname{5} = 'x-Momentum            ';
propname{6} = 'y-Momentum            ';
propname{7} = 'z-Momentum            ';
fprintf(1,' property              instant       average       standard deviation \n');
for i = 1:1:nprop
   fprintf(1,' %s %e %e %e \n', propname{i}, props(i,1),props(i,4),props(i,6))
end

%
%  save positions for mean square displacement calculations
%
function funk_msd(N,rwopbc,fid_msd);
for i = 1:1:N
   fprintf(fid_msd,' %e %e %e \n',rwopbc(i,1:3));
end
```