Self-Assembly of Polymeric Chains under a new Radially Symmetric Potential

A thesis submitted towards partial fulfilment of
BS-MS Dual Degree Programme

by

Alex Abraham

under the guidance of

Dr. Apratim Chatterji

Associate Professor

Indian Institute of Science Education and Research
Pune

Certificate

This is to certify that this thesis entitled "Self-Assembly of Polymeric chains under a new Radially Symmetric Potential" submitted towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research Pune represents original research carried out by Alex Abraham at IISER Pune, under the supervision of Dr. Apratim Chatterji during the academic year 2015-2016.

Student
ALEX ABRAHAM

Supervisor
DR. APRATIM CHATTERJI

30 March, 2016

Declaration

I hereby declare that the matter embodied in the report entitled "Self-Assembly of a System of Particles under a new Interacting Potential" are the results of the investigations carried out by me at the Department of Physics, IISER Pune, under the supervision of Dr. Apartim Chatterji and the same has not been submitted elsewhere for any other degree.

Student
ALEX ABRAHAM

Supervisor
DR. APRATIM CHATTERJI
30 March, 2016

Acknowledgements

I would like thank my supervisor Dr. Apratim Chatterji for all the guidance.
I thank my family and my dear friends for their relentless support.

Abstract

Self Assembly of particles under a radially symmetric two-body potential was studied. The new potential was designed by us with the aim of modeling polymeric chains. The self assembly, implemented through the off-lattice Monte Carlo scheme, produced chain-like structures. The structures were found to have branching, which, although remained after various attempts to eliminate, was brought down to a minimum. The set of parameters of potential was then optimized for maximum chain length with least branching. The clustersize distribution was analyzed and found to have exponential distribution. As a measure to reduce branching of the chains, the system was confined between walls and was subjected to weak constant shear stress, implemented using Molecular Dynamics scheme. The response of the system was analyzed. The variation of the branching and clustersize when the system was sheared with walls of various degrees of attraction was studied.

Contents

1 Introduction **4**

1.1 Motivation 5

1.2 Thesis outline 5

2 Theory **8**

2.1 Polymers 8

2.1.1 Ideal chains: Freely jointed model 8

2.1.2 Ideal chains: Worm-like chain model 9

3 Methods and Models **10**

3.1 Simulation of Self-Assembly 10

3.1.1 Metropolis Algorithm 11

3.1.2 Periodic Boundary conditions and the Truncated Potential 11

3.1.3 Clustersize and Branching estimation 12

3.1.4 Validation of MC method for Self-Assembly 13

3.2 Simulation of the system under shear 14

3.2.1 Molecular Dynamics of the free particles 14

3.2.2 Confined system and shearing 17

3.2.3 Steady state and Measurements 18
3.3 Validation of the Molecular Dynamics 19
3.3.1 MD of non-confined system 19
3.3.2 MD of confined system 20

4 Results & Discussions **23**
4.1 Self Assembly of particles: MC simulations 23
4.2 Shearing the system 30
4.2.1 Shearing the system of particles under LJ potential . . 30
4.2.2 Shearing the system of particles under the new potential 31
4.3 Conclusion 37

5 Future Prospects **38**

References **38**

A Simulation Codes **41**
A.1 Off-lattice Monte-Carlo simulation of self-assembly and additional analysis 41
A.2 Molecular Dynamics of shearing the system 52

Chapter 1

Introduction

In Soft Matter Physics, chain-like structures are commonplace in a variety of systems. They keep reappearing in different phenomena involving micelles, polymers, colloids etc. Indeed, there have been numerous studies over the years through lab experiments as well as computer modeling and simulations, to understand the properties and dynamics of such structures. The computational modeling has in great deal been a successful tool in the design, synthesis and in aiding successful applications of these materials.

Choice of model is dependent upon on the particular aspect of the concerned entity we would like to explore. In the case of polymers, the ideal chain models serve the purpose of explaining the general properties and provide insights into the physics of polymers at the macroscopic level. But when the microscopic details of the polymers are needed to be taken into account, one moves towards more precise models. The first successful molecular model of their dynamic behavior was described by P. Rouse, where he provided a spring and bead description. For the polymer dynamics in dilute solutions, where hydrodynamic interactions are strong, the Zimm model is relevant. There are other models which describe more features of polymers, like the entangled polymer dynamics models. [10]

In micellar systems, we come across a variety of chain-like structures. The wormlike micelles, which are elongated flexible self-assembled structures, are of particular interest. Depending on their concentrations and other parameters a variety of properties can be observed in them, including viscoelasticity, shear banding transition, nematic ordering etc [3, 13, 12, 4, 2, 9]. The theories on these structures can explain only some of the rich experimental rheological data that has been produced over the years. At this point, the simulations

provide a path to understand their formation, structural features and particularly, their peculiar rheology. Such simulations belong to three categories in general. (i) The atomistic simulations, where we have accurate and well understood force fields and which provides insights about the overall structure of the micelles, along with fluctuations in its shape. (ii) The coarse grained simulations, which follows from the principles of previous method but have greater pace, due to the approximations used. (iii) Higher level models like FENE-C and MESOWORM, where the focus is more on understanding the rheological properties. [14]

Chain-like structures that can be observed under optical microscopy have been reported by K. Guruswamy [6] in colloidal systems. This is a convenient feature, since this allows us to learn in greater detail the dynamics of the system and provides better control over the self assembly.

1.1 Motivation

The various computational methods that exist today to model a chain-like structure, more specifically one which can model and give control over properties like semi-flexibility of the chains, have used forms of potential which has angular dependence or 3-body interactions. As the most time consuming part of a simulation is the estimation of these inter atomic interactions, all these models are computationally demanding when compared to a interacting potential like Lennard-Jones. In this context, we aim to design a radially symmetric two-body potential, which would be computationally cheap to evaluate, and can model self assembly of particle into chains, which are possibly semi flexible in nature. We try and obtain control over various polymer properties like persistence length etc. by varying suitable parameters in the two body potential.

1.2 Thesis outline

The two body potential that was investigated in this study is,

$$U(r) = \varepsilon \left[\alpha \left(\left(\frac{\sigma}{r} \right)^{2p} - \left(\frac{\sigma}{r} \right)^{p} \right) + \beta \left(\frac{e^{\frac{-(r-r_0)}{\eta}}}{r} \right) \right] \tag{1.1}$$

The peak after the potential minimum is crucial in giving rise to chain-like

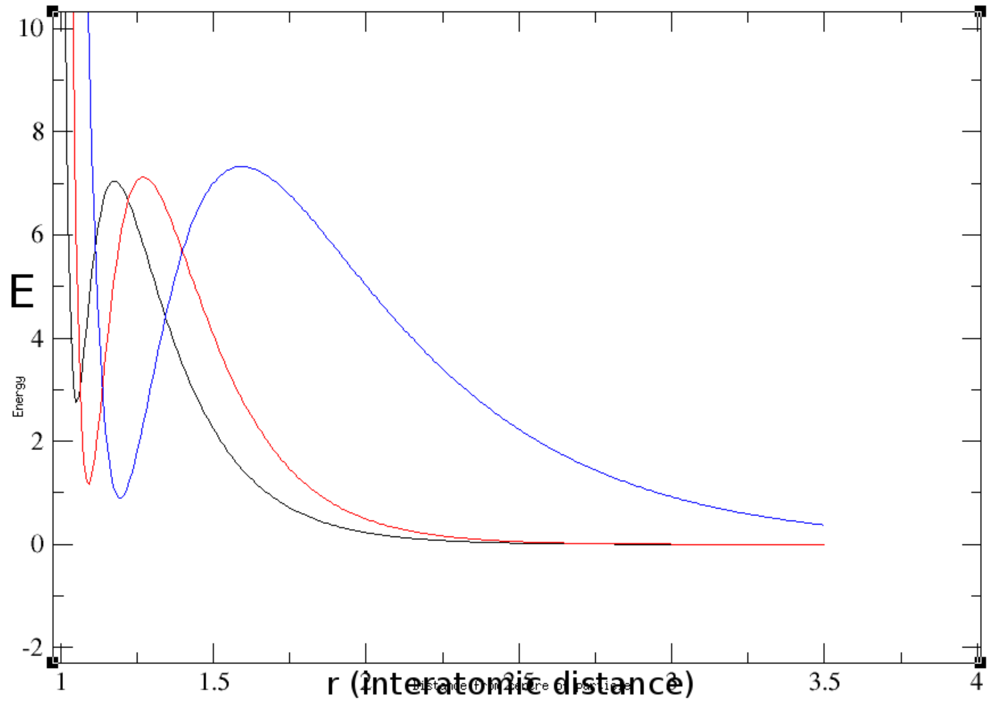


Figure 1.1: Typical plots of the three classes of potential. Green: $p = 18$. Red: $p = 12$. Blue: $p = 6$.

structures (Fig. 1.1). A particle, when bonded to another particle, will be at a distance less than the peak position. The presence of the peak makes this new bond stable; it, in way, ”locks” the particles. Now, when a third particle approaches to form a bond with any of these two, the presence of the peak makes it difficult for it attach to either particle, perpendicular to the existing bond, making it favorable only to attach from the either ends of the dimer. Ideally, if a particle tries to make third bond, the combined repulsion due the peak in potential of the particles around would deter it.

We focused on three classes of this potential; $p = 6$, $p = 12$ and $p = 18$ (Fig. 1.1). We start with a preliminary study on understanding the kind of structures that are formed and optimizing it to produce better chains. Then we shear the structures obtained to improve the chains and reduce any branching. Our model belong to the iiird category where rather than trying to include the atomistic details of the structure we are interested in, we simply try to model the macroscopic properties of the material.

The following the general structure of the Thesis:

Chapter 2 presents a quick overview of the physics of the kind of sub-

stances we have tried to model.

Chapter 3 provides the details of the model we have used and the techniques and schemes that were used in the study. This includes MC simulation of the self assembly and the MD simulation of the confined system and its shearing. The details regarding the validation of each scheme are also provided.

In Chapter 4, the results obtained are presented along with the discussions entailing them. The results and the related discussion have been supported with relevant analysis.

Chapter 5 provides an overview of the future directions in the study of the model.

Chapter 2

Theory

2.1 Polymers

Polymers, structures formed from repeating units known a monomers, have been widely theoretically modeled. The simplest of such models are the ideal chains.

2.1.1 Ideal chains: Freely jointed model

The ideal chain models describe the polymer as simple connections of monomers which has no interaction with each other when they are far apart. Consider an ideal polymer chain with n bonds (therefore, n+1 particles) and of equal bond length l. The angle between the bond vector of the ith and j^{th} particles is θ_{ij}. r_i is the bond vector between the ith and the $(i-1)^{th}$ particles. Let $\vec{R}_n$ be the end-to-end vector, the sum of all n bond vectors. Now, the mean-square end-to-end distance denoted by $\langle R^2 \rangle$ is,

$$\begin{aligned} \left\langle R^2 \right\rangle &= \left\langle \vec{R}_n \cdot \vec{R}_n \right\rangle \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} \langle \vec{r}_i \cdot \vec{r}_j \rangle \\ &= l^2 \sum_{i=1}^{n} \sum_{j=1}^{n} \langle \cos(\theta_{ij}) \rangle \end{aligned} \tag{2.1}$$

For a freely jointed chain, $\cos(\theta_{ij}) = 0$ for $i \neq j$ and 1 otherwise. Therefore, $\langle R^2 \rangle = nl^2$ for such a chain. But for an ideal polymer chain it would have

an extra factor $C_n(> 1)$ (Flory's Characteristic ratio) or C_∞ as n tends to infinity, which accounts for the correlations among the bond vectors.

In order keep things simple, any ideal polymer chain can be associated with an equivalent *free joint* chain, which can describe many universal properties of the polymer. The equivalent freely jointed chain of a polymer will have same mean-square end-to-end distance $\langle R^2 \rangle$ and the maximum end-to-end distance R_{max} as the actual polymer, but will have N effective bonds of length b. This length is called the *Kuhn length*. It can be shown that,

$$b = \frac{C_\infty n l^2}{R_{max}} \tag{2.2}$$

Now, we can see that a chain whose difference between its Kuhn length and the actual bond length is small, will be more flexible, and vice versa. Therefore, the Kuhn length indeed tells us a great deal about the flexibility of the chain.

2.1.2 Ideal chains: Worm-like chain model

A special case of the free rotating chain model (where one assumes equal probabilities for all torsion angles of bonds), is the $worm-like\ chain\ model$, where we consider small bond angles. Now, if s_p is the number of bonds in a persistence segment, the scale at which the local correlations between the bonds decay,

$$s_p = -\frac{1}{\ln(\cos(\theta))} \cong \frac{2}{\theta} \tag{2.3}$$

The $persistence\ length$ is the length of this persistence segment,

$$l_p = s_p l = l\frac{2}{\theta^2} \tag{2.4}$$

l being the constant bond length. This persistence length is related to Kuhn length b as,

$$b \cong 2l_p \tag{2.5}$$

The major difference between the two kind of the chains that has been mentioned is that in freely jointed model each bond of Kuhn length b is considered to be completely stiff, whereas, in worm-like chains this stiffness is within length scales shorter than the Kuhn length, allowing them fluctuate and bend more. This difference leads to different dynamical properties. Reference: [10].

Chapter 3

Methods and Models

In this chapter, we introduce the simulation models and methods used for the various studies on the potential. The initial simulations of the self-assembly and the resulting structures were done using the Off-Lattice Monte-Carlo (OLMC) simulation method. Once the optimized potential was found, a weak constant shear stress (boundary-driven) was applied on the system, simulated using Molecular Dynamics (MD). The confining walls in this model were modeled as a single triangular lattice of particles, interacting with the system particles through Lennard-Jones potential. The validity of the each system was ensured by matching several of their physical properties with known theoretical and experimental results.

3.1 Simulation of Self-Assembly

The primary step of study was to simulate the self-assembly of particles interacting through the potential. The off-lattice Monte-Carlo simulation method was deployed for this. The method is suitable for a preliminary study on the potential as the objective at this step was to make a computationally cheap assessment of the structures produced by the potential.

In the OLMC simulation, the system (a $L_x \times L_y \times L_z$ box), is filled with certain number of particles and their positions randomly assigned, under the condition that the distance between any two particles should be greater than the diameter of particle. The particle volume density is a controllable variable and the number of particles is decided accordingly. Next, the time evolution

of the particle system is performed as successive Monte-Carlo steps. In each step, every particle in the system is visited exactly once and their positions are updated using the Metropolis algorithm [8].

3.1.1 Metropolis Algorithm

In the Metropolis Algorithm, a random change in the position of the particle is proposed. We accept it if the resulting change in the energy of system ΔE is negative. Otherwise, the proposed change is accepted with a probability $e^{\frac{\Delta E}{k_B T}}$. In order to do this, we generate a number using a uniform random number generator from the interval [0,1] and if the number is less than the factor $e^{\frac{\Delta E}{k_B T}}$, the change in position will be accepted.

3.1.2 Periodic Boundary conditions and the Truncated Potential

Even though the simulation is expected to provide accurate information about the macroscopic sample, the limitations pertaining to the present-day computers allow us to deal with only a smaller number of degrees of freedom. In order to overcome this limitation we have imposed periodic boundary conditions (PBC) on the system. This helps us go around the problem and mimic the presence of an infinite bulk engulfing our model system containing a few thousand particles. The the box containing the system particles is considered as a primitive cell which is part of a infinite periodic lattice of identical cells. This could lead to artificial dynamics in the system (for example, if you have a fluctuation of wavelength greater than the cell size). In order to minimize such finite size effects, sufficiently large box ($30 \times 30 \times 30$) has been used.

In principle, the PBC lets any particle in the the primitive cell to interact with infinite number of particles (including its own periodic images). But the limited computational power means that we have to restrict the interactions to a limited volume around the particle concerned. Also, the potential we are dealing with (most intermolecular interaction potentials, for that matter) is a short-range interaction potential. Therefore, we can truncate all intermolecular interactions beyond a cutoff distance.

Truncated Potential

In order to truncate the interactions, we define a cutoff distance r_c. It is chosen such that the error in ignoring the interactions with particles at distances larger than r_c is sufficiently small. Since $U(r) = 0$ for $r > r_c$, the potential will become a discontinuous at r_c. Therefore, for $r < r_c$ we define a truncated and shifted potential,

$$U(r) = \varepsilon \left[\alpha \left(\left(\frac{\sigma}{r}\right)^{2p} - \left(\frac{\sigma}{r}\right)^{p} \right) + \beta \left(\frac{e^{\frac{-(r-r_0)}{\eta}}}{r} \right) \right] - U(r_c) \tag{3.1}$$

as the net potential used in the simulation, where,

$$U(r_c) = \varepsilon \left[\alpha \left(\left(\frac{\sigma}{r_c}\right)^{2p} - \left(\frac{\sigma}{r_c}\right)^{p} \right) + \beta \left(\frac{e^{\frac{-(r_c-r_0)}{\eta}}}{r_c} \right) \right] \tag{3.2}$$

3.1.3 Clustersize and Branching estimation

The set of parameters for an efficient potential was found on the basis the average chain size (or rather clustersize) and average branch number of the configuration arises out of the self-assembly. These quantities were calculated from the cluster sampling done during the runtime. Any group of particle in which, each particle is within the bond length, i.e ”bonded”, with at least one other particle of the group, is considered as a cluster. The bond length is defined as the position of the point of energy peak in the potential. A branch is defined as a third bond arising from a particle. Note that due to this definition if a bond shared between two particles is the third bond for both of them, then the number of branches counted due to that bond is two, not one.

The system can contain all kinds of clusters other than chains. Also, many chains could have several branches on them too. It is very difficult to come up with a criteria to consider a cluster as a chain. Therefore, instead of exclusively picking chains and classifying them according to their size, we have included all kind of clusters. The number of branches were also recorded in each sampling. We then looked for set of parameters that corresponds to maximum clustersize but minimum number of branches, since it would imply the the clusters in the system are predominantly chain-like.

The cluster sampling was done at regular intervals once the system had reached the equilibrium. During sampling, every particle will be visited exactly once. When a particle is visited we will visit all particles of the same cluster starting from it and in the process count the number of particles, which is nothing but the size of cluster. In the meantime, if any of those particle is found to have more than two bonds the number of branches in the system is incremented by one. From this data, we calculate the average number of branches and the average cluster size, which is,

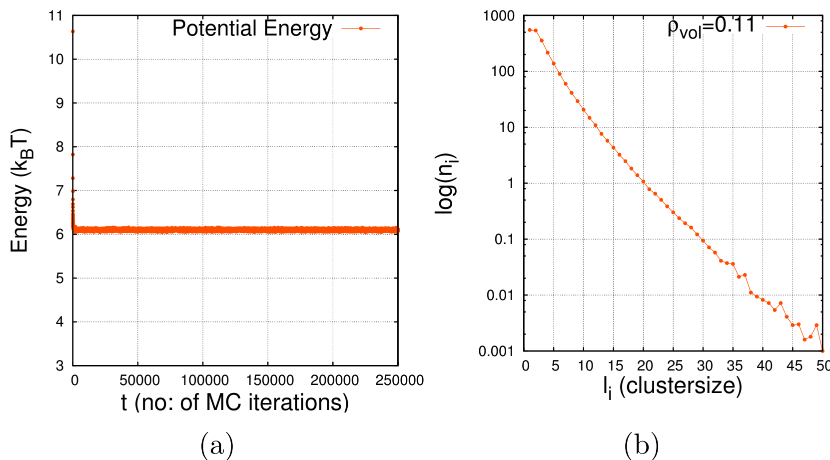
$$l_{avg} = \frac{\sum_i l_i \times n_i}{\sum_i n_i} \tag{3.3}$$

where n_i is the number of clusters with size l_i per sample.

3.1.4 Validation of MC method for Self-Assembly

As an initial check of the MC scheme, self-assembly of particles interacting through the potential of interest produced energy curves as expected for an system in equilibrium (Fig. 3.1a).

As a further check, we looked for the features of clustersize distribution (Fig. 3.1b). The clustersizes followed an exponential distribution, which is



(a) (b)

Figure 3.1: **(a)** The energy curve for the potential of interest. The dip in potential energy is almost unnoticeable, indicating a quick equilibration of the system. **(b)** Logarithmic plot of clustersize vs. number of clusters in each size category. The log plot is clearly linear in nature. l_i is the clustersize and n_i is the number of clusters belonging to the size category l_i.

expected for a self-assembled system[7, 11].

3.2 Simulation of the system under shear

The study of the system of particles under the application of shear stress required a technique that takes into account the intermolecular forces. The obvious choice was Molecular Dynamics (MD). The technique gives a view of the dynamics of the system of particles by simplifying the interatomic interactions as classical interactions. This method is deterministic as compared to Monte-Carlo method, which was stochastic.

3.2.1 Molecular Dynamics of the free particles

The Molecular Dynamics involves numerically solving the Newton's equations of motion for the system of particles and determining their trajectory in the system. To begin with, we initialize the system with particles (number calculated from the predefined volume density) at random positions, such that none of them overlap any other particle, as we had done for the Monte-Carlo simulation (see Sec. 3.1). We also initialize velocities for them such that the system has the required initial temperature. First, we calculate the v_{rms} corresponding to the initial temperature T. We then assign each velocity component a value belonging to a uniform distribution on the interval $[-v_{rms}, v_{rms}]$. By choosing the velocities from this particular interval we are guaranteed to get the desired temperature, which can be shown as follows. In thermal equilibrium, the following relation holds,

$$v_{rms}^2 = \frac{3k_BT}{m} \tag{3.4}$$

Throughout the study we have set $m = 1$ and $k_B = 1$ (see Sec. 3.3.2). Therefore,

$$T = \frac{v_{rms}}{\sqrt{3}} \tag{3.5}$$

Let $v'_{ix}, v'_{iy}, v'_{iz}$ be the three velocity components for the ith particle chosen from the above mentioned interval. Then,

$$v'_{rms} = \sqrt{\frac{1}{n}\sum_i \left({v'_{ix}}^2 + {v'_{iy}}^2 + {v'_{iz}}^2\right)} \tag{3.6}$$

Now, for the uniform distributions on the interval of the kind $[-x, x]$ the mean is zero and therefore, the v'_{rms} is nothing but $\sqrt{3}$ times standard deviation of the distribution (as we have used $3n$ samples from the intervel to calculate v'_{rms}). Therefore,

$$
\begin{aligned}
Std.\,Dev. &= \sqrt{\frac{1}{12}\,(b-a)^2} \quad \text{for a uniform distribution in the interval } [a,b] \\
\therefore v'_{rms} &= \sqrt{3}\,\sqrt{\frac{1}{12}\,(b-a)^2} \quad \text{where, } a = -v_{rms},\ b = v_{rms} \\
&= v_{rms}
\end{aligned}
\tag{3.7}
$$

This ensures that the temperature is T. Additionally, we shift all the velocities such that the net linear momentum of the system is zero.

We now proceed to calculate the forces acting on every particle in the system. In our case, the interatomic forces are dictated by the potential through which the particles are interacting. We have,

$$
\begin{aligned}
\vec{F}(r) &= -\vec{\nabla}U(r) \\
&= \varepsilon\left[2p\alpha\left(\left(\frac{\sigma^{2p}}{r^{2p+1}}\right) - 0.5\left(\frac{\sigma^{p}}{r^{p+1}}\right)\right) + \beta\, e^{\frac{-(r-r_0)}{\eta}}\left(\frac{1}{r^2} + \frac{1}{\eta r}\right)\right]\hat{n}
\end{aligned}
\tag{3.8}
$$

Once we have the forces corresponding to each particle we can start integrating the Newton's equations of motion in order to obtain the trajectory that describes the position and velocity for the time forward. We used the Velocity-Verlet algorithm for this purpose [5]. Starting with the Taylor expansions of the $r(t+\Delta t)$ and $r(t-\Delta t)$, this algorithm gives the updated position and velocity of a particle after a timestep Δt by,

$$
r(t+\Delta t) = r(t) + v(t)\Delta t + \frac{f(t)}{2m}\Delta t^2 \tag{3.9}
$$

$$
v(t+\Delta t) = v(t) + \frac{f(t+\Delta t) + f(t)}{2m}\Delta t \tag{3.10}
$$

Once the new positions and velocities are calculated, we update these as the current coordinates in the phase-space and repeats the above calculations for the future time steps. As in the case of Monte-Carlo simulations, we use Periodic Boundary conditions and the truncated and shifted potentials here as well, for the same reasons as before (see Sec. 3.1.2). The force we use here is truncated and shifted as well,

$$F(r) = \varepsilon \left[2p\alpha \left(\left(\frac{\sigma^{2p}}{r^{2p+1}} \right) - 0.5 \left(\frac{\sigma^{p}}{r^{p+1}} \right) \right) + \beta\, e^{\frac{-(r-r_0)}{\eta}} \left(\frac{1}{r^2} + \frac{1}{\eta r} \right) \right] - F(r_c) \tag{3.11}$$

where, $F(r_c)$ is the force value at the cutoff distance r_c. Since we are dealing with forces also, the form of potential used in the MD simulations will have a extra term $F(r_c)\, r$ (where r is distance between the particles) added to it, in order to have a proper shifted potential.

Thermostat

The Molecular Dynamics we are carrying out is done in isothermal conditions. Even though we had carefully set the initial temperature at the desired value, there is no guarantee that it will stay constant throughout. Since there is little chance that the total initial potential energy of the system will be at the equilibrium value, it would give out or draw in from the kinetic energy of the system, which will lead to changes in the temperature. Therefore, it is essential to take away or provide extra energy in order to keep the system at constant temperature. In real experiments, the environment of the setup usually takes care of this. In simulations, we employ a virtual thermostat to maintain constant temperature in system. At sufficiently large but equal intervals, the thermostat will be applied, which rescales the velocities of particles such that the kinetic energy per particle is in agreement with desired temperature.

Later, when we introduce shear stress to the system through confining walls, the thermostat is to play an important role in driving the system towards steady state and maintaining the system in it.

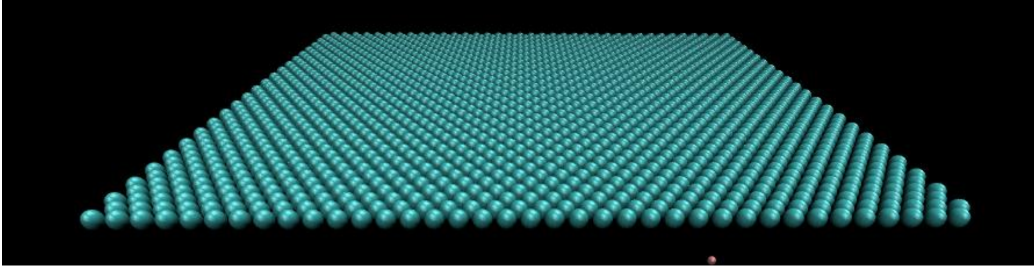


Figure 3.2: Particles arranged in triangular lattices serve as walls

3.2.2 Confined system and shearing

In order to shear the system, the system is confined between two parallel walls, which will then be moved in opposite directions. As first step, we introduce walls at the boundaries perpendicular to the z-axis (i.e. at the two x-y boundaries of the box). The walls are triangular lattices of particles (Fig. 3.2), which interact with the system particles through the simple Lennard-Jones potential,

$$U(r) = 4\varepsilon_2 \left[\left(\frac{1}{r}\right)^{12} - \left(\frac{1}{r}\right)^{6} \right] \tag{3.12}$$

which, of course, will be cut off at a distance r_c^{wall} and shifted by $U(r_c^{wall})$.

No momentum will be transferred to the wall particles from the free system particles; only free particles will feel the interacting forces from walls. They have an constant but independent motion parallel to the system. The wall at $z = 0$ move in the $+x$ direction and the other wall at $z = l_z$ (length of simulation box along the z-axis) move in -x direction, with constant velocity (Fig. 3.3). This wall velocity will induce shear stress in the system. The nature of the wall (attractive, weakly attractive, repelling etc.) can be controlled by varying ε_2 value of the potential and the cutoff distance r_c^{wall}.

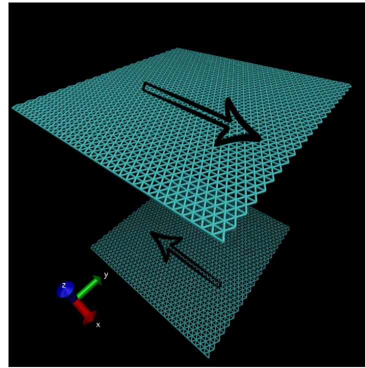


Figure 3.3

The periodic boundary conditions have to be slightly modified when we

introduce the walls. Since presence of walls means that the particles can no longer move across the corresponding x-y boundaries, no periodic boundary condition will be imposed along the z-axis. In the extremely rare situation of a particle crossing the walls, steps will be taken to remove it from simulation altogether, as would have happened in a real experiment.

Once the walls are setup, the simulation is initiated and the next step is to do the required measurements on the system after reasonable time. The phenomena we are interested in are not transient in nature. Therefore, meaningful measurements can only be done once the system has at least evolved past the transient period.

3.2.3 Steady state and Measurements

Shearing the system is a non-equilibrium process, since we are putting in energy continuously. As in the case of equilibrium processes, this process too have a transient period, where various physical quantities of the system varies. The system then attains a steady state, where the physical quantities of the system is no more changing with time, i.e, for a quantity of interest $w(x,t)$,

$$\frac{\partial w}{\partial t} = 0 \tag{3.13}$$

We need to make sure that the system has reached the steady state before we start sampling data for the measurement of any quantity. The determination of the steady state in the system is based the variation in the profile of x-component of particle velocities, along the z-axis of the system.

In order to determine this velocity profile of particles (and for several other calculations), we divide the box into small layers (compared to l_z) along the z-direction. We then identify the particles that fall within each of these layers and an average v_x for each layer is calculated. Repeating this in regular intervals of timesteps, a v_x profile averaged over a fairly long period of time (long enough to average out fluctuations) is obtained. A second estimation of v_x profile is done in the next same length of time in similar fashion. These two profiles are then compared and if the variation between two profiles is within a certain predefined error bar, we conclude that the system has reached steady state.

Once the system has reached steady state, we can commence measurement of the quantities we are interested in.

Temperature profiling: The temperature of each layer is calculated using the average velocities of the layers, from the principles of Kinetic theory: $\frac{3}{2}k_BT = \left\langle \frac{1}{2}mv^2 \right\rangle$. Since there is flow in x-direction, we have to subtract the flow velocity from the v_x in order to obtain the thermal velocity. Therefore, we calculate the average KE as,

$$\left\langle \frac{1}{2}mv^2 \right\rangle = \frac{1}{n_{layer}} \sum_{i}^{n_{layer}} \frac{1}{2}m \left((v_x - v_x^{mean})^2 + v_y^2 + v_z^2 \right)_i \quad (3.14)$$

where n_{layer} is the number of particles in the concerned layer. This is averaged over several samples in order to get a reliable temperature estimate (see Fig. 3.6b).

Density profiling: The number of particles in each layer is counted and the volume of one particle is then multiplied with it to get the total volume occupied by particles. We then divide this by the total volume of the layer (except for the end layers, where the presence of the wall particles calls for correction in available layer volume) to obtain the volume density profile (see Fig. 3.6a).

$\mathbf{v_x}$ profiling: The v_x profiling involves simply averaging the x-component of the velocity of particles in a layer. Similar profiling have been done for other components of velocity as well.

3.3 Validation of the Molecular Dynamics

3.3.1 MD of non-confined system

The simulation system developed was initially tested on Lennard-Jones particles. The simulation scheme indeed took the system to a lower energy state as desired. The system was equilibriated and maintained at that state with the help of the thermostat, which was called every 200 iterations (Fig. 3.4a). Further, the energy plots for two different values of Δts (0.001 and 0.002) were compared and the fluctuation in energy for $\Delta t = 0.002$ was four times as that of $\Delta t = 0.001$, i.e, the fluctuations were proportional to Δt^2 which

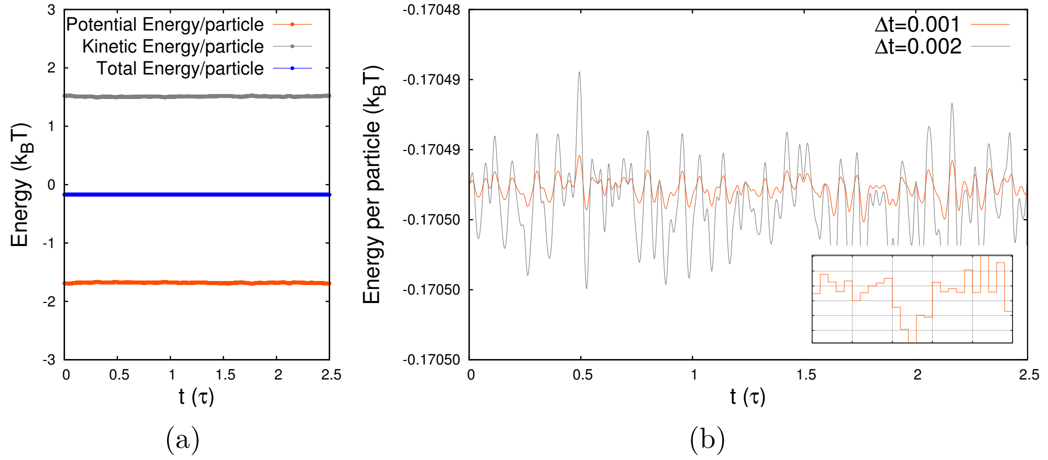


Figure 3.4: **(a)** Energy plot of the system at T=1 **(b)** Energy plot for $\Delta t = 0.001$ and $\Delta t = 0.002$. The fluctuation in $\Delta t = 0.002$ energy is four times more that of $\Delta t = 0.001$, i.e., it is proportional to Δt^2. (**Inset:** Plot of total energy per particle is zoomed up. Note that the energy is constant in regular inetervels between thermostating.)

is what is expected (see Fig. 3.4b) [5]. Also, the energy of the system stays constant until the thermostat is applied.

3.3.2 MD of confined system

To establish the validity of the system under shear stress we had to rely on various physical quantities other than the different energies, since we have a non-equilibrium system in hand. Lennard-Jones particles were used as test case. The temperature profile is even throughtout the system along the z-axis, except near the walls. This irregularity (not shown in the figures here) in temperature profile can be explained by the free particles crowding next

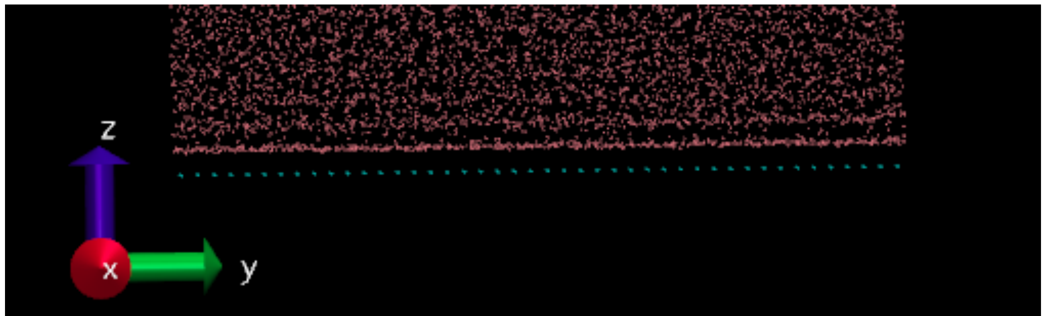


Figure 3.5: The crowded band of particles near the walls. The bluish-green dots are the walls rows (seen from their ends) and the others are the free particles.

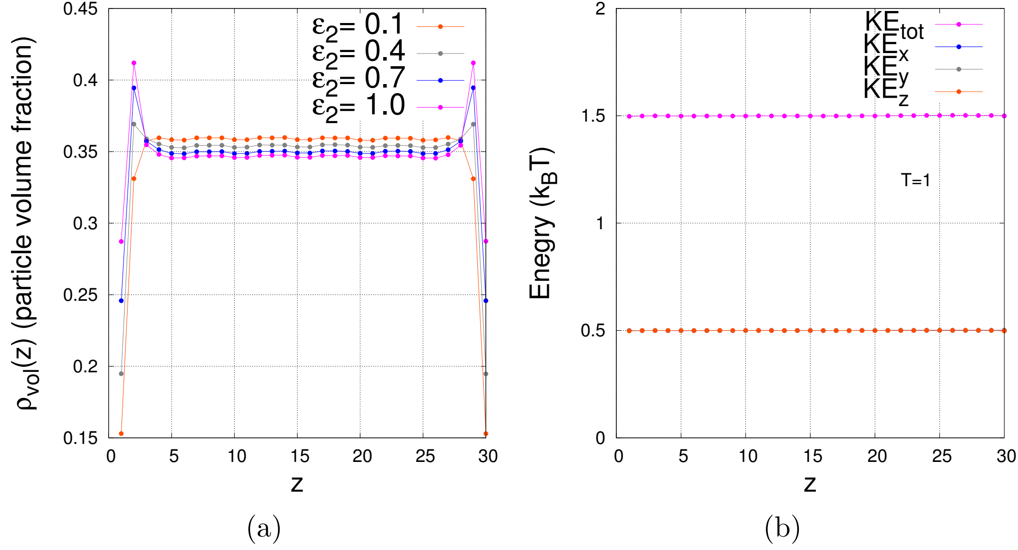


Figure 3.6: **(a)** Density profile of system with walls $v_{wall} = 0.3$, with various ε_2. The peaks near the walls vanishes and then dips as wall strength becomes weaker. **(b)** The plot of total Kinetic Energy and the share of KE for each degree of freedom. The total KE at T=1 is 1.5 and each d.o.f has a KE share of 0.5 (hence the overlap of plots at y=0.5)

to the walls, as the potential minima of the wall particles are facilitating a low energy laminar region parallel to the walls (Fig. 3.5).

The boundary crowding effect is more visible in the density profile, which is otherwise even throughout (Fig. 3.6a). Also, there is a reversal in this effect as the wall interaction strength is brought down. The system also abides to the law of equipartition since the thermal kinetic energy is distributed equally

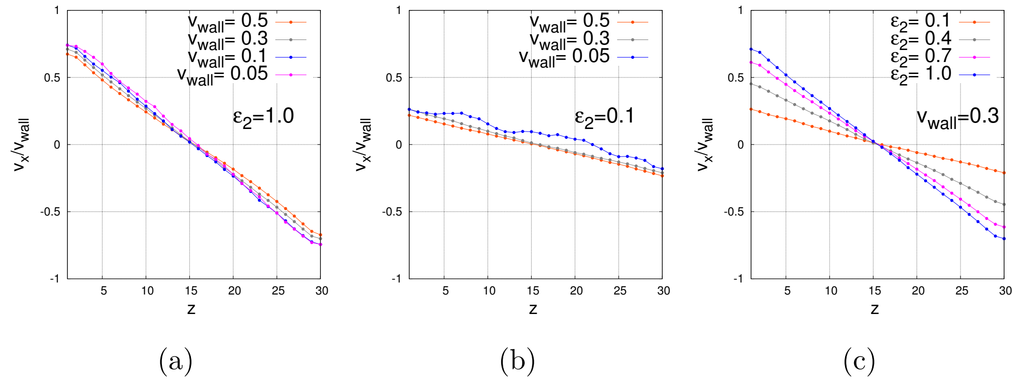


Figure 3.7: **(a)** v_x profile for $\varepsilon_2 = 1.0$. **(b)** v_x profile for $\varepsilon_2 = 0.1$. A deviation (blue line) was observed when the wall was only weakly attractive and shear rate was low. **(c)** The v_x profiles for various ε_2 when $v_{wall} = 0.3$ was applied. All the runs are 6000 LJ time units long. The units of v_{wall} and ε_2 are a/τ and $k_B T$, respectively.

in all three degrees of freedom per particle (Fig. 3.6b).

The v_x profile of a set of Lennard-Jones particles under constant shearing would be linear in the direction perpendicular the direction of shearing (z-direction in our case) when relatively low shear rate is applied [1]. This was replicated in our system (Fig. 3.7). Also, the slope of the profile seem to decrease with reduction in the potential strength (i.e. the ε_2 value).

A Note on units

Throughout the simulations and study, we have used reduced units for convenience. Setting a as the unit of length we have scaled all lengths accordingly. For a given $k_B T$, T being the temperature, we then fix the unit of time as τ, where $\tau = \sqrt{m_p a^2 / k_B T}$, with $m_p = 1, k_B = 1$ and $a = 1$. Any value a physical qauntity, including time and temperature, discussed anywhere in this thesis have been scaled accordingly, if not mentioned otherwise.

Chapter 4

Results & Discussions

In this chapter, we present the results obtained on from the study of properties shown by self-assembled structures formed under the radially symmetric two-body potential. Further in, we enlist the effects of confinement and shearing on these structures.

4.1 Self Assembly of particles: MC simulations

The preliminary study of self-assembly of the particles under the potential,

$$U(r) = \varepsilon \left[\alpha \left(\left(\frac{\sigma}{r} \right)^{2p} - \left(\frac{\sigma}{r} \right)^{p} \right) + \beta \left(\frac{e^{\frac{-(r-r_0)}{\eta}}}{r} \right) \right] \tag{4.1}$$

was intended to get an overview of the kind of the structures it could produce. The MC simulation provided prima facie evidence that the potential indeed gives rise to chain-like structures (Fig. 4.2), although with some branching. We then optimized the potential for maximum yield of chains with minimum branching.

The optimization study showed that set parameters of the following potential maximized the chain yield while keeping the branches minimum,

$$U(r) = 12 \left[5.8 \left(\left(\frac{\sigma}{r} \right)^{36} - \left(\frac{\sigma}{r} \right)^{18} \right) + 1.3 \left(\frac{e^{\frac{-(r-1.12)}{0.25}}}{r} \right) \right] \tag{4.2}$$

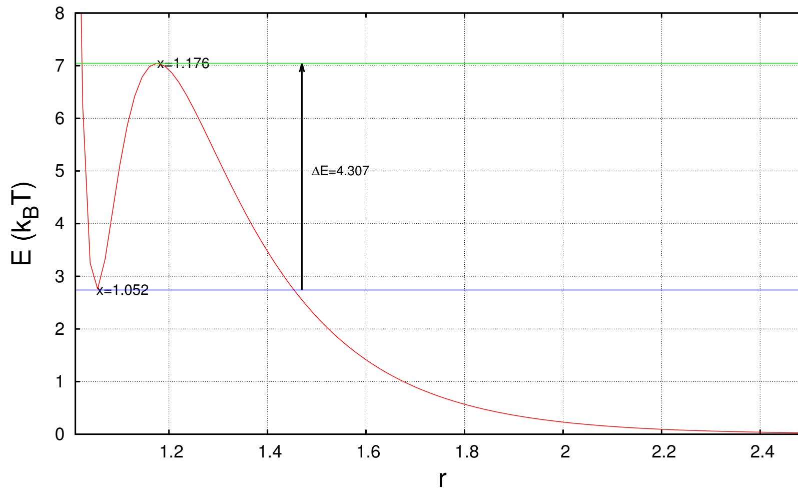


Figure 4.1: Plot of the optimal potential (Eqn. 4.2)

The average clustersize of this is potential l_{avg} is 2.25 and number of branches per unit volume is 6.14×10^{-3}. Note that the particle diameter σ has been set to $1\,a$, i.e, one unit of length.

We looked at various features of the potential in reaching at the above form. The difference in the energy value of the peak in the potential and the dip, ΔE, is crucial in giving stability to the chains formed under this potential. At the particle volume densities around 0.15, particles on average would be at a distance ≈ 1.5. Now, we want the energy peak to be scalable by a particle and be able to form a bond, and at the same time, we want the bonds to be stable, which would be decided by the height of the peak from the minimum of potential well. A careful analysis showed that $\Delta E \approx 4$ would be ideal.

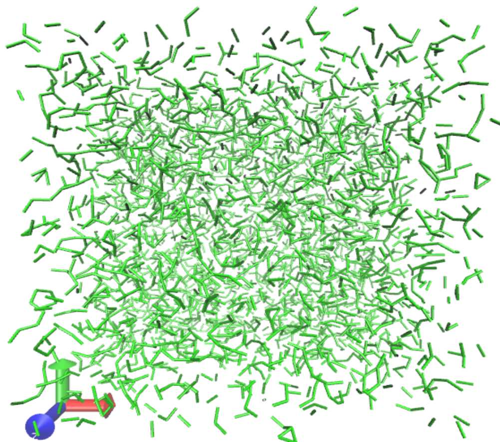


Figure 4.2: The snapshot of the final configuration formed under the optimal potential [4.2].

One can see that the peak is steeper inside the potential well, making it difficult for a bond to break, since even a small displacement means a larger change in energy. This is not the case for a particle that is approaching the peak from the outside the bond, where it

is less steep, making it relatively easier for a bond to form than to break off. Now, if a particle approaches to form a bond with another particle which already have two bonds emanating from it, the repulsions from the other two particles should be strong enough to deter the new particle from creating a third bond and thus, a branch.

Analysis of chains

The average branching was observed to decrease as change the p value (Eqn. 4.1) was changed from 6 to 12 to 18, without much loss in the clustersize (Ref. Appendix A). The increase in sharpness of the peak in potential is the general change that occurs when one make such a change (Fig. 4.3), indicating that the sharpness of the potential peak is a favorable aspect in producing better chains. Therefore, potentials of the class $p = 18$ were used further studies.

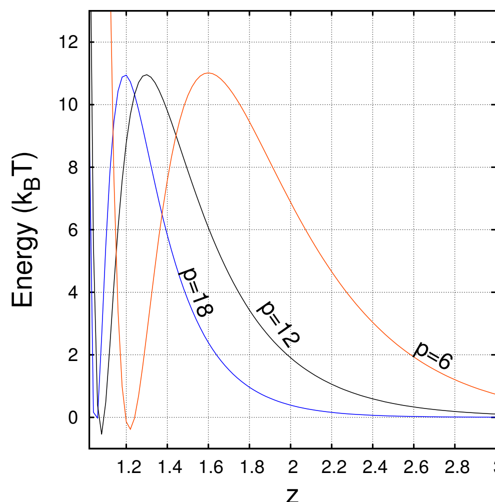


Figure 4.3: Variation in the potential as the p value varies from 6 to 12 t 18. The sharpness of the peak increases with the increment in p. Note the variation of interaction a particle at a given distance would experience as this change happens.

The analysis of the clustersizes shows that their distribution is exponential (Fig. 4.6a), in agreement with literature [11]. The particles started to coalesce to form large network-like structures when the densities were increased. This was marked by the deviation from the exponential distribution (Fig. 4.6b) (Note that the distributions shown here are of the potential [4.2]). The variation of the average clustersize and average branching as the energy difference between the potential energy peak and the dip varies, was also looked at(Fig. 4.7).

The following is the MC simulation data for various sets of parameters of potential. Note that for $p = 6$, all final configurations were dominant with large non-chain clusters, leading to high clustersizes.

r12-6 (*p*-value = 6)

Sl. no.	epsilon	alpha	beta	eta	r0	b2	Density	Particles	niter	Bond length	Avg. cluster size	Avg. branching	Branching/ unit^3	Box
30	40	6.6	2	0.666	1.12	0	0.2	10318	2.5	1.6	NaN	972	36	30x30x30
31	40	6.6	2	0.666	1.12	0	0.18	9286	2.5	1.6	NaN	905.2	33.5	30x30x30
32	40	6.6	2	0.666	1.12	0	0.15	7738	2.5	1.6	NaN	841.5	31.2	30x30x30
33	40	6.6	2	0.666	1.12	0	0.12	773	2.5	1.6	9	89	26.4	15x15x15
34	40	6.6	2	0.666	1.12	0	0.1	644	2.5	1.6	15.89	79.67	23.6	15x15x15
35	30	6.5	2	0.666	1.12	0	0.12	773	2.5	1.6	12.43	90.6	26.8	15x15x15
36	30	6.5	2	0.666	1.12	0	0.1	644	2.5	1.6	13.78	63	18.6	15x15x15

r36-18 (*p*-value = 18)

Sl. no.	epsilon	alpha	beta	eta	r0	b2	Density	Particles	niter	Bond length	Avg. cluster size	Avg. branching	Branching/ unit^3	Box
37	12	5.8	1.3	0.25	1.12	0	0.15	7738	2.5	1.2	2.25	165.7	6.14	30x30x30
38	12	5.8	1.3	0.25	1.12	0.1	0.15	7738	2.5	1.2	2.27	170.5	6.31	30x30x30
39	12	5.8	1.3	0.25	1.12	0.5	0.15	7738	2.5	1.2	2.39	192.3	7.12	30x30x30
40	12	5.8	1.3	0.25	1.12	1	0.15	7738	2.5	1.2	2.56	220.7	8.17	30x30x30
41	12	5.8	1.3	0.25	1.12	2	0.15	7738	2.5	1.2	3.14	263.9	9.77	30x30x30
42	12	5.8	1.3	0.25	1.12	0	0.18	9286	2.5	1.2	5.58	1033.8	38.3	30x30x30
43	12	5.8	1.3	0.25	1.12	0.5	0.18	9286	2.5	1.2	6	1121.3	41.5	30x30x30
44	12	5.8	1.3	0.25	1.12	1	0.18	9286	2.5	1.2	6.34	1192.9	44.2	30x30x30

Figure 4.4: MC simulation results for $p = 6$ and $p = 18$. The final average clustersizes are large for $p = 6$ since majority of the self assembled structures were large highly interconnected clusters of particles.

r24-12 (*p*-value = 12)

Sl. no.	epsilon	alpha	beta	eta	r0	b2	Density	Particles	niter (x10^6)	Bond length	Avg. cluster size	Avg. branching	Branching/ unit^3(x10^{-3})	Box
1	36	5.25	1.2	0.25	1.12	0	0.1	644	1	1.3	2.59	7.15	2.12	15x15x15
2	36	5.25	1.2	0.25	1.12	0	0.11	709	1	1.3	3.574	21.15	6.27	15x15x15
3	30	5.25	1.2	0.25	1.12	0	0.1	644	1	1.3	2.4	8.57	2.54	15x15x15
4	30	5.25	1.2	0.25	1.12	0	0.12	773	1	1.3	4.56	41.92	12.4	15x15x15
5	30	5.25	1.2	0.25	1.12	0	0.13	838	1	1.3	7.46	81.32	24.1	15x15x15
6	30	5.25	1.2	0.25	1.12	0	0.14	902	1	1.3	6.59	125.9	37.3	15x15x15
7	30	5.25	1.2	0.25	1.12	0	0.15	967	1	1.3	3.95	153.7	45.5	15x15x15
8	45	5.25	1.2	0.25	1.12	0	0.2	1289	1	1.3	1.0	181.2	53.7	15x15x15
9	40	5.25	1.2	0.25	1.12	0	0.2	1289	1	1.3	1.0	186.5	55.3	15x15x15
10	50	5.25	1.2	0.25	1.12	0	0.2	1289	1	1.3	NaN	186.4	55.2	15x15x15
11	60	5.25	1.2	0.25	1.12	0	0.2	1289	1	1.3	NaN	180.5	53.5	15x15x15
12	40	5.25	1.2	0.25	1.12	0	0.15	967	1	1.3	2.91	149.0	44.1	15x15x15
13	50	5.25	1.2	0.25	1.12	0	0.15	967	1	1.3	2.95	130.7	38.7	15x15x15
14	60	5.25	1.2	0.25	1.12	0	0.15	967	1	1.3	3.25	149.6	44.3	15x15x15
15	120	5.25	1.2	0.25	1.12	0	0.15	967	1	1.3	4.7	158.8	47.1	15x15x15
16	150	5.25	1.2	0.25	1.12	0	0.15	967	1	1.3	4.8	62.6	18.5	15x15x15
17	180	5.25	1.2	0.25	1.12	0	0.15	967	1	1.3	3.84	47	13.9	15x15x15
18	75	5.0	1.2	0.4	1.12	0	0.15	967	1	1.3	3.39	159.4	47.2	15x15x15
19	50	4.5	1.2	0.666	1.12	0	0.15	967	5	1.3	NaN	135.9	40.3	15x15x15
20	75	5.0	1.2	0.4	1.12	0	0.18	1160	1	1.3	5.02	165.3	49	15x15x15
21	50	4.5	1.2	0.666	1.12	0	0.18	1160	2.5	1.3	NaN	164	48.6	15x15x15
22	32	5.6	1.2	0.25	1.12	0	0.12	773	2	1.3	7.93	103.7	30.7	15x15x15
23	31	5.45	1.2	0.25	1.12	0	0.12	773	2	1.3	6.7	77.79	23.0	15x15x15
24	30	5.25	1.2	0.25	1.12	2	0.15	15477	2.5	1.3	4.07	2145	39.7	30x30x60
25	21.5	5.25	1.2	0.25	1.12	2	0.15	15477	2.5	1.3	3.73	2455	45.5	30x30x60
26	30	5.25	1.2	0.25	1.12	0.5	0.12	12382	2.5	1.3	4.95	771.8	14.3	30x30x60
27	30	5.25	1.2	0.25	1.12	0.5	0.15	15477	2.5	1.3	3.79	2470	45.7	30x30x60
28	30	5.25	1.2	0.25	1.12	0.5	0.18	18573	2.5	1.3	1.74	2849	52.8	30x30x60
29	30	5.25	1.2	0.25	1.12	0.5	0.2	1289	2.5	1.3	1.10	189	56	15x15x15

Figure 4.5: MC simulation data for $p = 12$.

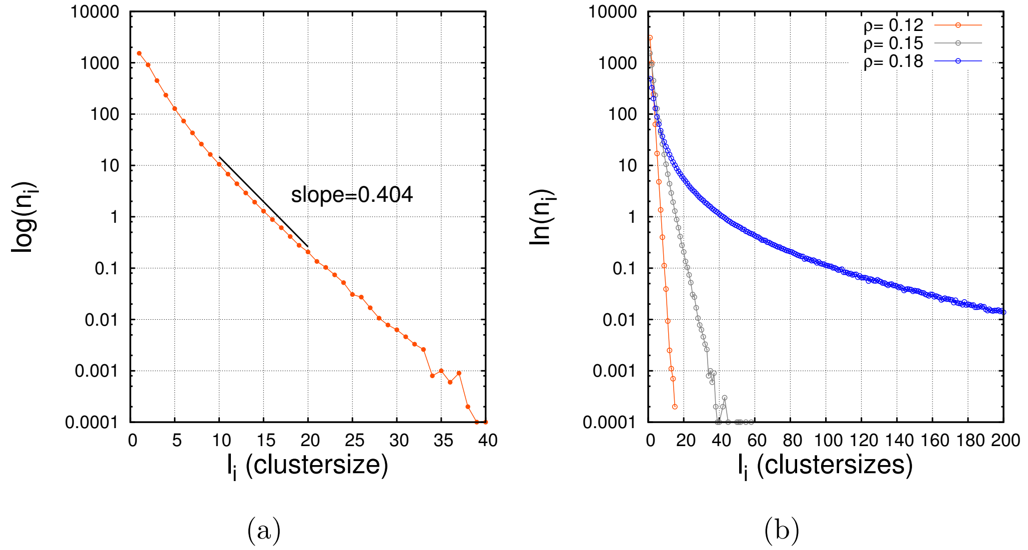


Figure 4.6: (a) Log plot of clustersize distribution. The clustersize is distributed as $842.59\, exp(-0.404 l_i)$. l_i is clustersize and n_i is number of clusters of size l_i. $\rho = 0.15$. (b) Log plot of clustersize distribution for various densities. The distribution for $\rho = 0.18$ shows a deviation from exponential distribution as the particles started to coalesce and form large non-chain structures.

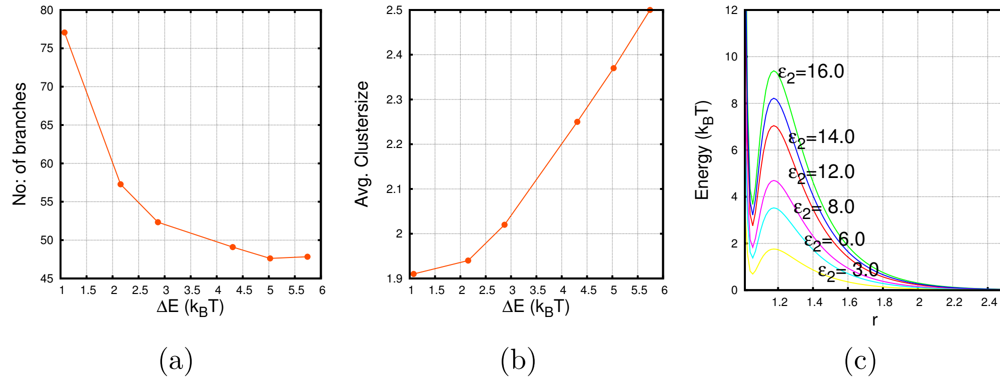


Figure 4.7: (a) Variation in the avg. branches as the the height of the potential peak from the energy dip. The ε_2 of the eqn. 4.2 is varied to obtain potentials of different ΔE. (b) Variation of the avg. clustersize as we change the same energy difference. (c) The plots of potentials corresponding to each of the ΔE that has been mentioned the above plots.

Applicaton of weak bias

As a measure to possibly reduce the amount of branching, we had tried introducing a weak unidirectional bias to help align the chains. This was expected to help reduce branching in high density systems and higher densities are essential to obtain longer chains. The bias was introduced by adding a relatively weak extra energy, $-b^2(\vec{r}_{ij} \cdot \hat{z})^2$, to the bonded particles, favoring

the bonds in a unique direction (here z). We expected that this would help align the chains in a single direction by overcoming the hindrances, if at all they have a tendency to align. The bonds not aligned in the bias direction were expected to be reduced and effectively reduce the branching, especially in system of higher densities.

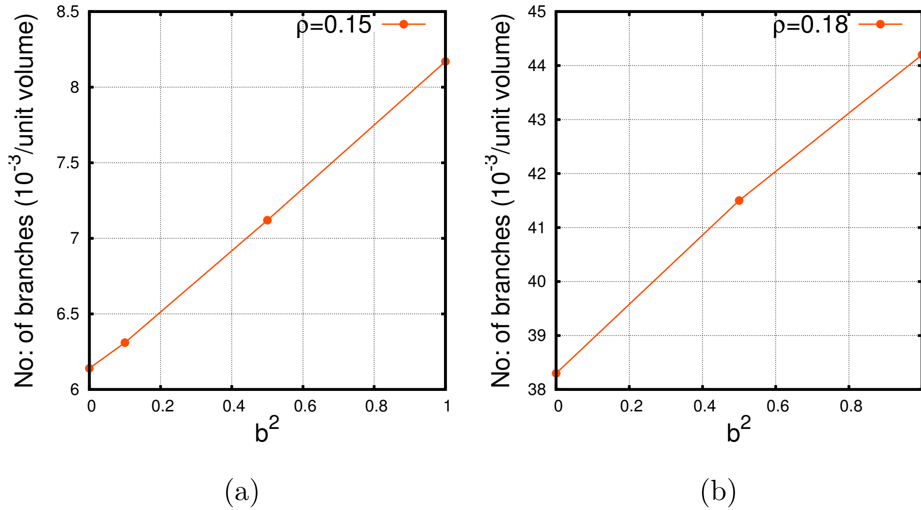


Figure 4.8: The Average branches per unit volume vs. strength of the bias. For both densities the avg. branch number increased with the increase in b^2.

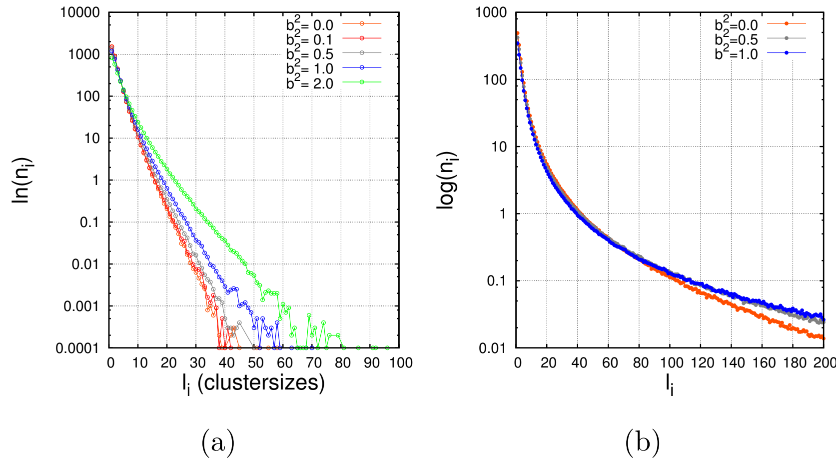


Figure 4.9: (a) The clustersize distribution deviated from the expected exponential distribution as the bias strength was increased. This can be related to the gradual increase in the bulky structures in the system, as seen in high density systems. (b) The networked structures were not broken down by the weak bias in the high density system. Instead, it caused an increment, reflected in the plot as increased deviation from the non-biased distribution.

The attempt didn't help in reducing the branch number, instead it rather increased the overall amount of bonding, which helped both branching and chain formation (Fig. 4.8a,4.8b). For high density system, the bias increased the clusters and the deviation from the exponential distribution also increased (Fig. 4.9b). A possible reason for this is the following. Although we applied the bias in favor of the bonds tending towards the z-direction, the extra energy introduced by the bias, helped increase and better stabilize the bonds leaning towards the direction of bias, in both the chains and the branches, and thereby not reducing the number of branches.

4.2 Shearing the system

4.2.1 Shearing the system of particles under LJ potential

As a measure to create longer chains and possibly, reduce branching, we decided to shear the system of particles. Schematic diagram Fig. 4.10 illustrates the process. By inducing a shear flow in the system we expect ordering in the direction of flow and thus, possibly obtain longer chains. As an initial study on the system that was setup, the shearing was done to particles interacting through Lennard-Jones potential. A constant temperature was maintained by applying a thermostat. Once the MD scheme was validated, v_x profiling along the axis perpendicular the wall velocity was done. The shear rates applied varied from 6.67×10^{-4} ($v_{wall} = 0.01$) to 3.33×10^{-2} ($v_{wall} = 0.5$).

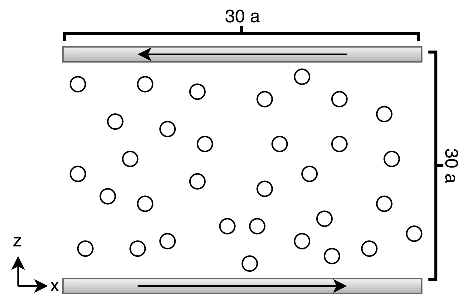


Figure 4.10

As expected, a linear v_x profile was obtained (Fig. 4.11, 4.12), at least for the high shear rates. The normalized v_x gradient is solely dependent on the nature of the wall that was used for shearing, i.e, the ε_2 value of the wall-particle interaction, which is in agreement with the known results. The slip at the boundaries increased as the wall strength ε_2 was decreased.

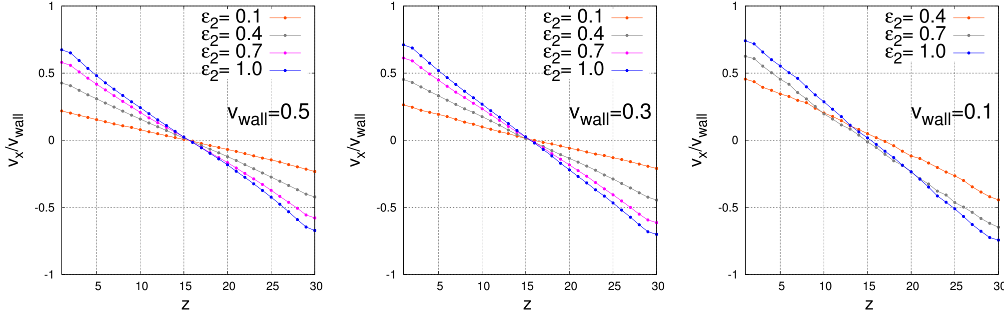


Figure 4.11: The normalized velocity gradient for particular shear rates. It increases as expected, as the wall strength ε_2 is increased. The units of v_{wall} and ε_2 are a/τ and $k_B T$, respectively.

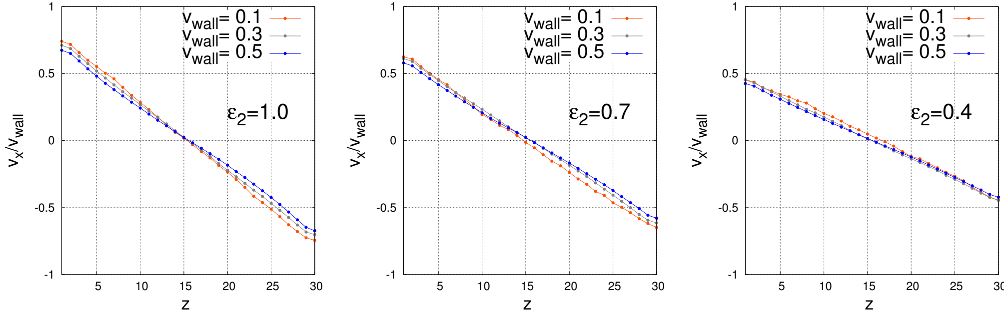


Figure 4.12: The normalized velocity gradient is same for same wall-particle interaction, i.e, ε_2. The units of v_{wall} and ε_2 are a/τ and $k_B T$, respectively.

4.2.2 Shearing the system of particles under the new potential

The system of particles was subjected to varying degrees of shear rate. We used shear rates ranging from 3.33×10^{-2} to 6.67×10^{-4}, with ε_2 values from 1.0 to 0.1 (attractive wall to weakly attractive wall). A thermostat was applied every 200 iterations. We had used a time step $\Delta t = 0.004$ for the simulations. The system maintained constant temperature throughout the shearing (Fig. 4.13a) and the density was even too (except, as expected, near the walls) (Fig. 4.13b).

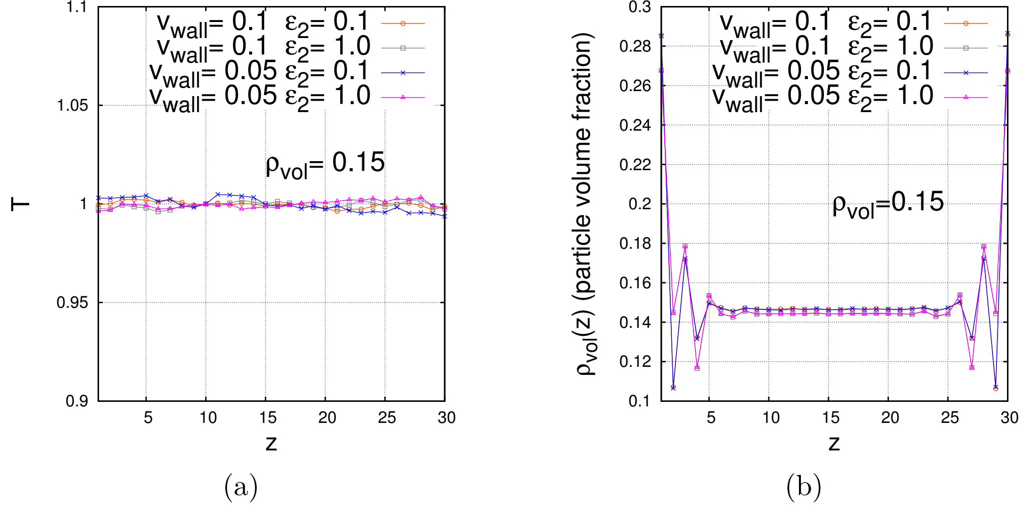


Figure 4.13: **(a)** The temperature profile of the system with a particle volume density 0.15. The fluctuations in temperature is less than 0.3%. **(b)** The density profile shows the huge fluctuations near the wall due to the attraction. This crowd of particles arranged in bands affects other properties in this region as well, slightly, even the v_x profile as well. The units of v_{wall} and ε_2 are a/τ and k_BT, respectively.

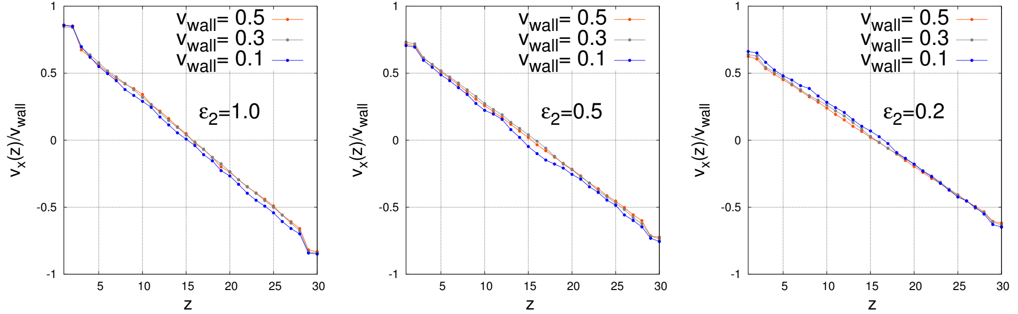


Figure 4.14: The normalized velocity gradient is found to be same for same wall strength ε_2 irrespective of the shear rate applied. The units of v_{wall} and ε_2 are a/τ and k_BT, respectively.

The v_x profiles obtained were linear (Fig. 4.14, 4.15). The normalized velocity gradient was same for same value of ε_2 indicating that the slip at the wall-bulk boundary is determined by the nature of the wall. This slip was large when the wall interaction was weaker. But the variation of slip between strongly and weekly attracting walls was small as compared to that of a standard system like Lennard-Jones, which means that the system under consideration is relatively resilient to shearing. The fluctuations were increased when lower shear rates were applied, which could be the indication of the thermal energy taking over the velocity behavior of the system particles (Note that the average energy per particle is around $6\,k_BT$). The

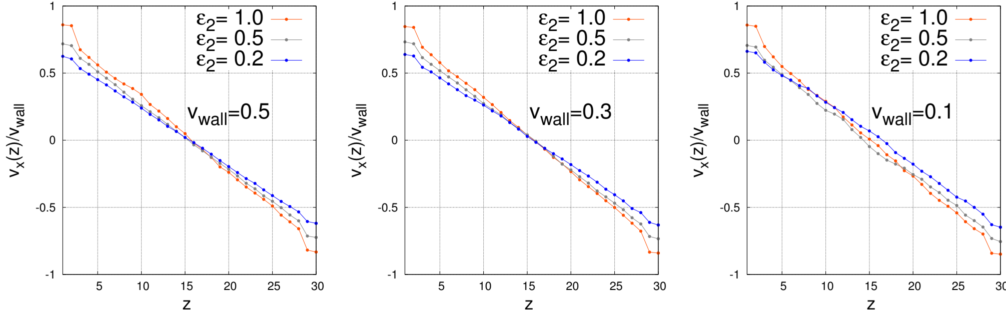


Figure 4.15: The normalized velocity gradient increases as expected, as the wall strength ε_2 is increased. But the fact that the change in this gradient is comparatively small even as the wall is varied from strongly attractive to weakly attractive, hints that the system is fairly resilient to shearing. The units of v_{wall} and ε_2 are a/τ and k_BT, respectively.

system responded to shear rates up to the order $\sim 10^{-4}$ (Fig. 4.16). Below this limit, no significant shear response was observed in normal time scales of computer simulation.

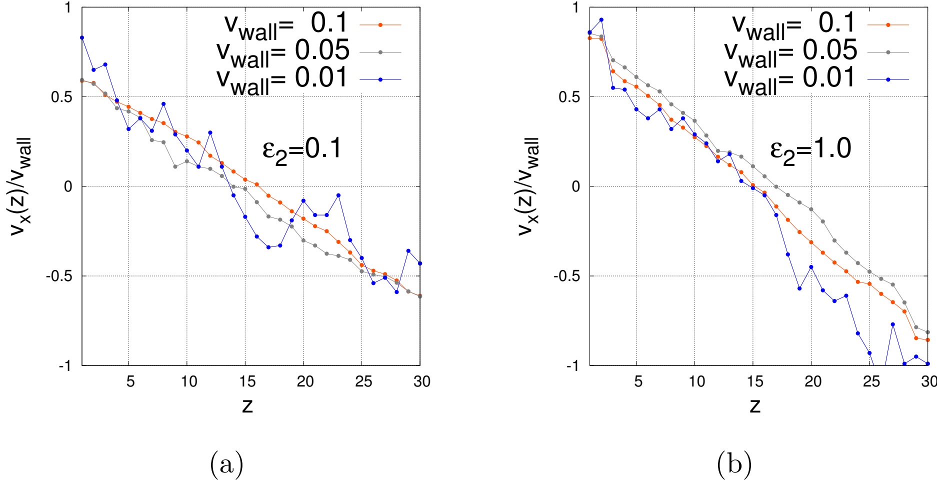
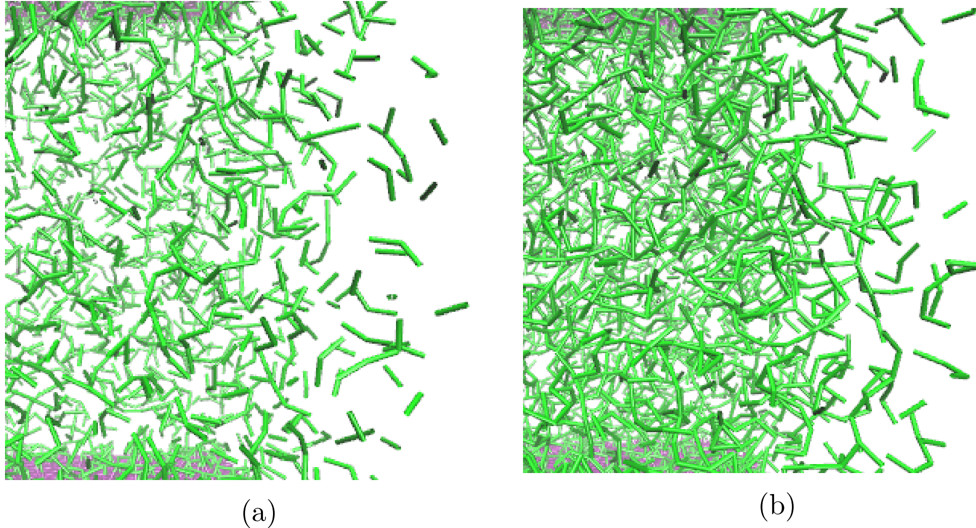


Figure 4.16: **(a)** v_x profile for various shear rates for $\varepsilon_2 = 0.1$ at density 0.15. **(b)** v_x profile for various shear rates for $\varepsilon_2 = 1.0$ at density 0.15. The velocity gradient slowly vanishes as the shear rates approaches the 10^{-4} mark. For wall velocities of the order 0.001 the response was null even after 7000 LJ time units (τ) of simulation. The units of v_{wall} and ε_2 are a/τ and k_BT, respectively.

The system retained chain structures under shearing. But there was a noticeable reduction in chain size in bulk, at density 0.15 (Fig. 4.17a). For example, when a shear rate of 6.67×10^{-3} ($v_{wall} = 0.1$) was applied with wall interaction strength $\varepsilon_2 = 0.1$, the l_{avg} was 2.04, as opposed to the 2.25 for the same potential in an unconfined system. This small change is

significant considering the fact that the clustersize distribution is exponential. At the same time, the system also showed a reduction in branching with average branches per unit volume 4.2×10^{-3}. Interestingly, the same changes



(a) (b)

Figure 4.17: **(a)** A part of the final configuration of the simulation with optimal potential under density 0.15. When compared to 4.2 a reduction in clustersize might be noticeable. **(b)** Final configuration of system under same potential but at 0.18 density. There are lot of branching, but not as much as a non-sheared system of same parameters would normally have.

in average clustersize and branching were observed in a confined but non-sheared system. With $\varepsilon_2 = 0.1$ and no wall velocity, a $l_{avg} = 2.03$ and average branches per unit volume $= 4.2 \times 10^{-3}$ was obtained. This indicates that the confinement of the system of particles rather than the shear applied on them, might be the cause of reduction in branching as well as the change in average clustersize. Now, when the ε_2 value was increased to 1.0, the l_{avg} increased to 2.12 and average branch number to 6.75×10^{-3}, the later being same as the average branches in the unconfined system. In effect, when compared to the unconfined situation the clustersize reduced but the branching remained the same. Therefore, confinement of the of the system not necessarily increase the clustersize to branching ratio, from an unconfined situation.

The following shows the change in branching and average clustersize as wall strength and also as the wall velocity varies. Both the clustersize and branching increases as the wall velocity (i.e, shear rate) is increased.

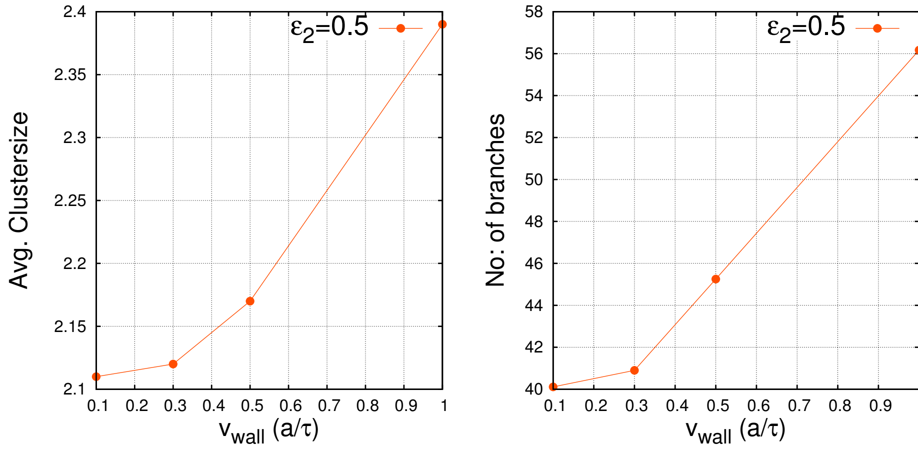


Figure 4.18: Variation of branching and clustersize with the wall velocity.

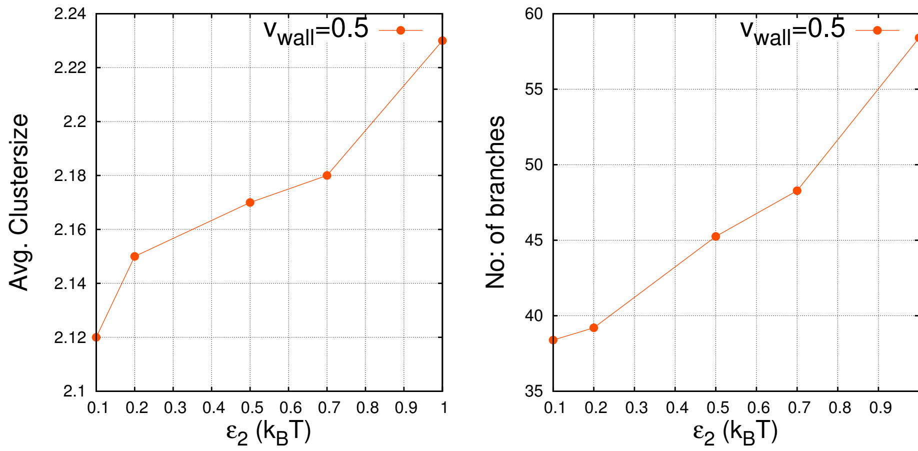
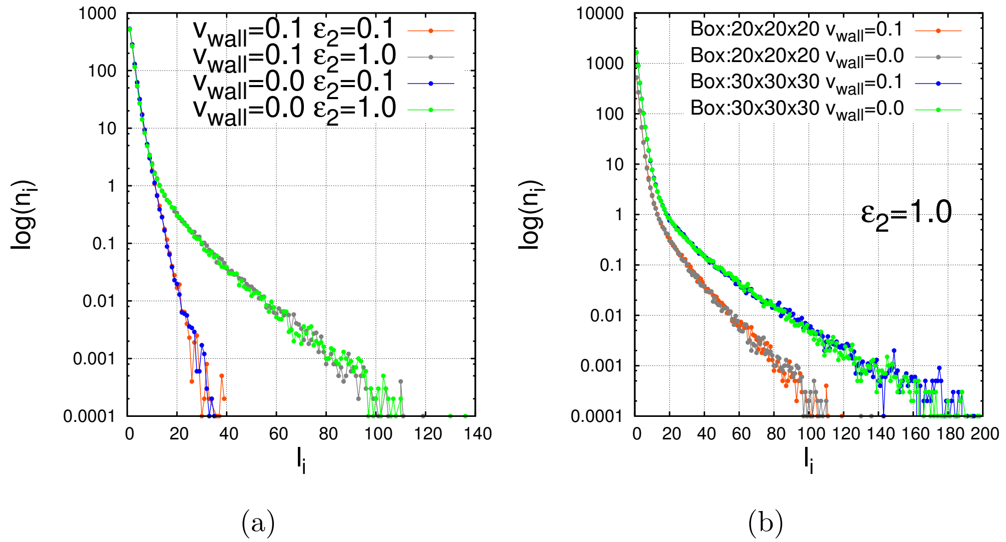


Figure 4.19: Variation of branching and clustersize with the wall strength ε_2.

Clustersize distribution and its anomalous deviation

The clustersize distribution of the confined system shows a deviation from the usual simple exponential distribution. The occurrence of the behavior is independent of the shear rate, rather it depends on the ε_2 value, as seen in Fig. 4.20a. The distribution for high ε_2 value has slope same as that of a weak wall until around $l_i = 9$ and then deviates and settles down with a different exponential distribution. Looking at the structures formed on the walls of strong attraction (Fig. 4.21), one can argue that the higher amount



(a) (b)

Figure 4.20: **(a)** The log plot of length distribution for various v_{wall} s and ε_2 s. The plot shows that there is significant change in distribution when the ε_2 is varied and almost no dependence on the wall velocity. **(b)** The clustersize distributions for the of system in two different box sizes. The wall strength $\varepsilon_2 = 1.0$.

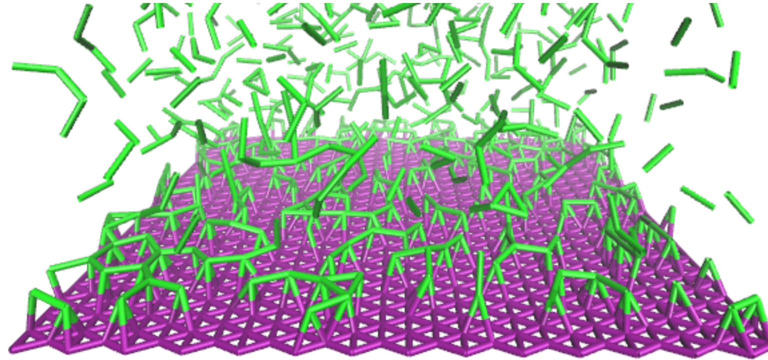


Figure 4.21: Final configuraton of a system with $\varepsilon_2 = 1.0$ and $v_{wall} = 0.1$ The strutures grown on the wall can be seen.

of long chains as seen in the distribution graph is due to the particles sticking to walls and forming chains over there. The comparison of the distributions (Fig. 4.20b) in boxes of different sizes reinforces this argument. Now, this could as well be finite box size effect but when we look at this along with the fact that such a deviation happens only when we have a strongly attractive wall, there is little doubt that this deviation is nothing but an effect of the walls. Nonethless, an explanation of the seemingly good agreement of the deviated size distribution with the size distribution of weak wall system in lower size regime and the sudden divergence afterwards, is still difficult.

The density profile shows a high concentration of particles near walls, especially when ε_2 value is large, and a different density leads to different kind of cluster forms. The high dense region near the walls and the remaining bulk, therefore, could have different kinetics of clustering. The kinetics near the walls might be favoring large clusters and the bulk obviously smaller clusters. If this was the case, it would lead to the kind of distribution as seen above.

4.3 Conclusion

The study of the radially symmetric two-body potential concludes that the potential has ability cause self assembly of particles to forms chains, albeit with some branching. The clustersizes follow a exponential distribution. An optimized set of parameters has been found which gives maximum yield of chains with minimum branching. The confinement and shearing of the system of particles gives rise interesting dynamics of cluster formation. The deviation of clustersize distribution indicates a possibility of two separate cluster formation kinetics, one close to the walls and the other in bulk, to be happening simultaneously in the same system.

The thermal stability of the chains formed is yet to be ascertained. The shear stress application, for example, seems to decrease bonding in the system. We have not yet found any evidence of a shear band separation in the system when sheared.

Chapter 5

Future Prospects

As the study on the two-body potential has provided evidence of self assembly of particles into chains, the next obvious goal is to improve the quality of chains obtained by reducing the branching and possibly increasing the chain length. The viscoelastic properties of the structures formed needs a thorough analysis. The persistence lengths and the Kuhn lengths of the chains have to be analyzed in detail. Until now the focus has been on the self assembly in temperature $T = 1$. It will be an interesting prospect to learn the dynamics and the variations of chain length at different temperature settings. Varying the temperature during the simulation will open up new possibilities in tailoring better chains.

We also propose a alternative scheme of interaction, in which the system contains two kind particles. The particles of same kind interacts through a attractive potential and the unlike ones interacts through a repulsive potential. We expect formation of chains were the two kind of particles are alternated.

References

[1] W. T. Ashurst and W. G. Hoover. Argon shear viscosity via a lennard-jones potential with equilibrium and nonequilibrium molecular dynamics. *Phys. Rev. Lett.*, 31:206–208, Jul 1973.

[2] J.-F. Berret. Rheology of Wormlike Micelles : Equilibrium Properties and Shear Banding Transition. *eprint arXiv:cond-mat/0406681*, June 2004.

[3] Cecile A. Dreiss. Wormlike micelles: where do we stand? recent developments, linear rheology and scattering techniques. *Soft Matter*, 3:956–970, 2007.

[4] M. A. Fardin, T. Divoux, M. A. Guedeau-Boudeville, I. Buchet-Maulien, J. Browaeys, G. H. McKinley, S. Manneville, and S. Lerouge. Shear-banding in surfactant wormlike micelles: elastic instabilities and wall slip. *Soft Matter*, 8:2535–2553, 2012.

[5] Daan Frenkel and Berend Smit. Chapter 4 - molecular dynamics simulations. In Daan Frenkel and Berend Smit, editors, *Understanding Molecular Simulation (Second Edition)*, pages 63 – 107. Academic Press, San Diego, second edition edition, 2002.

[6] Guruswamy Kumaraswamy, Bipul Biswas, and Chandan Kumar Choudhury. Colloidal assembly by ice templating. *Faraday Discuss.*, pages –, 2016.

[7] J. S. Marshall and W. Mc K. Palmer. The distribution of raindrops with size. *Journal of Meteorology*, 5(4):165–166, 1948.

[8] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6), 1953.

[9] Shaikh Mubeena and Apratim Chatterji. Hierarchical self-assembly: Self-organized nanostructures in a nematically ordered matrix of self-assembled polymeric chains. *Phys. Rev. E*, 91:032602, Mar 2015.

[10] M. Rubinstein and R.H. Colby. *Polymer Physics*. OUP Oxford, 2003.

[11] P. G. J. van Dongen and M. H. Ernst. Dynamic scaling in the kinetics of clustering. *Phys. Rev. Lett.*, 54:1396–1399, Apr 1985.

[12] Lynn M Walker. Rheology and structure of worm-like micelles. *Current Opinion in Colloid and Interface Science*, 6(5âĂŞ6):451 – 456, 2001.

[13] Jiang Yang. Viscoelastic wormlike micelles and their applications. *Current Opinion in Colloid Interface Science*, 7(5âĂŞ6):276 – 281, 2002.

[14] R. Zana and E.W. Kaler. *Giant Micelles: Properties and Applications*. Surfactant Science. CRC Press, 2007.

Appendix A

Simulation Codes

A.1 Off-lattice Monte-Carlo simulation of self-assembly and additional analysis

(Fortran 90)

```
!==========================================================================================!
!                                                                                          !
!                                off-lattice_monte-carlo                                   !
!                                                                                          !
!             V(r)= eps*(alpha*((sig/r)**36-(sig/r)**18)+((beta*exp(-(r-r0)/eta))/r))  !
!                   with bias  ((rij.z)**2)*(B**2)                                         !
!==========================================================================================!


!==================================parameter_module====================================1

module olmc_para
implicit none

! model_parameters
integer,parameter :: lx=20, ly=20, lz=20
real*8 :: llx, lly, llz
real*8 :: llxby2, llyby2, llzby2
real*8 :: den= 0.15d0 !density
real*8,parameter :: eps= 24.0d0, kbt=1.0d0, sig= 1.0d0, mass= 1.0d0,tem= 1.0d0,rc= 2.5d0&
                  &,beta= 1.3d0, eta=0.25d0, b2= 0.0d0, eps2= 1.0d0
real*8,parameter :: alpha= 5.8d0, r0= 1.12d0, ngh_trim= 0.8d0
          ! epsilon, boltz. const., particle_diameter, mass, temperature, potentialrange

! simulation_variables
 character(len=30) :: label= "713"
integer, parameter :: niter= 250000, iter_equil= 100000, nghupdate= 10, count_intervel=10
integer, parameter :: maxclustsize= 3000, maxboxcontent= 500, maxnghbours= 2500
integer :: npart !no: of particles
```

```
integer :: nbox, boxsidex, boxsidey, boxsidez  !no: boxes and no: of boxes along eachedge
integer*8 :: clustersize, tot_clust, avg_clustsize, branchnum, accptd= 0, rjctd= 0
integer*8 :: tot_clust2, avg_clustsize2
integer*8, dimension(:), allocatable :: nghbours, clust_count, ifvisited, numofbonds
integer*8, dimension(:,:), allocatable :: nghlist, boxnghlist
real*8 :: vrc
real*8 :: delx, dely, delz, dist, de, pot_e, r_ngh, bond_l= 1.2d0, rij
real*8 :: min_ini_dist= 1.0d0
real*8 :: box_size= 5.0d0
real*8, parameter ::  dd= 0.25d0
real*8, dimension(:), allocatable :: pos

integer :: zzz=-4271245, zzzz=65902043

! index_variables
integer :: i, j, k, m, n, temp, iter, dummy

! timer_variables
real*8 :: start, ends !start time, end time

! others
 character(len=20) :: den_str
 character(len=60) :: filename
real*8 :: r, init_pot, final_pot,tempx, tempy, tempz, posx, posy, posz, rc2, r_ngh2, bon&
&d_l2

end module olmc_para


!===============================main_program=========================================2
program chained
use olmc_para
implicit none
integer :: i_time
real*8 :: junk,zz1,ran1,vol

call cpu_time(start)

zz1=ran1(zzz)
junk=ran1(zzzz)

!calculating various derived parameters of the model

llx=dfloat(lx); lly=dfloat(ly); llz= dfloat(lz)
llxby2=llx/2.0d0; llyby2=lly/2.0d0; llzby2=llz/2.0d0
vol = 4.0d0*3.14d0*(sig*sig*sig/24.0d0)
npart= int(den*llx*lly*llz/vol)
vrc= eps*(alpha*((sig/rc)**36-(sig/rc)**18)+((beta*exp(-(rc-r0)/eta))/rc))
!eps2*(((sig/rc)**12-(sig/rc)**6))
r_ngh= (rc+(sqrt(((dd/2.0d0)**2)*3))*nghupdate*ngh_trim)
boxsidex= int(lx/box_size)
boxsidey= int(ly/box_size)
boxsidez= int(lz/box_size)
nbox= boxsidex*boxsidey*boxsidez
branchnum= 0
rc2= rc*rc
r_ngh2= r_ngh*r_ngh
bond_l2= bond_l*bond_l
write(den_str, '(1F5.3)') den

print*, 'npart:',npart
print*, 'r_ngh:', r_ngh
```

```
print*, 'nbox :', nbox

allocate(pos(3*npart))
allocate(nghbours(npart))
allocate(nghlist(maxnghbours,npart))
allocate(ifvisited(npart))
allocate(clust_count(maxclustsize)); clust_count=0
allocate(boxnghlist(27, nbox))
allocate(numofbonds(npart))

call boxnghbours         !assigns_ngbours_for_each_box
call initpos             !ensemble_initialisation
call init_potential      !initial_potential_calculation


!monte_carlo_steps

filename= trim(den_str)//"_E_"//trim(label)//".dat"
 open (95, file=filename,status='unknown',form='formatted')
do iter= 0, niter

    if(mod(iter, count_intervel)==0) write(95, '(1I41F20.4)') iter, pot_e/dfloat(npart)

    if(mod(iter,1000)==0) print*, iter
    if(mod(iter,5000)==0) call poscopy
    if(mod(iter,nghupdate)==0) call update_nghlist    !updates_neighbourlist
    if(iter .eq. iter_equil) then
        accptd=0
        rjctd=0
    endif
    if((mod(iter, count_intervel)==0) .and. (iter>iter_equil)) call clust_counter
    if(mod(iter,10000)==2000) then
        call cpu_time(ends)
        write(*,'(1A15,1I2,1A,1I2,1A,1I2)'), &
        &'Time remaining:', int((niter-iter)*(ends-start)/(iter*3600)), & !hours
        &      ':', mod(int((niter-iter)*(ends-start)/(iter*60)),60), & !minutes
        &      ':', mod(int((niter-iter)*(ends-start)/iter),60)         !seconds
    endif

    call pos_update      !updates_positions

enddo
 close (95)


!writes_the_length_distribution_to_file

tot_clust= 0
avg_clustsize= 0
filename= trim(den_str)//"_len_distri_"//trim(label)//".dat"
 open(98, file=filename, status='unknown', form='formatted')
do i= 1, 200
    if (clust_count(i) .ne. 0) write(98, '(1I41F20.4)') i,&
    &dfloat(clust_count(i))/dfloat((niter-iter_equil)/count_intervel)
    avg_clustsize= avg_clustsize+ i*clust_count(i)
    tot_clust= tot_clust+ clust_count(i)
enddo
 close(98)

tot_clust2= 0
avg_clustsize2= 0
filename= trim(den_str)//"_len_distri2_"//trim(label)//".dat"
```

```
 open(97, file=filename, status='unknown', form= 'formatted')
do i= 4, 200
    if(clust_count(i) .ne. 0) write(97, '(1I41F20.4)') i,&
    &dfloat(clust_count(i))/dfloat((niter-iter_equil)/count_intervel)
    avg_clustsize2= avg_clustsize2+ i*clust_count(i)
    tot_clust2= tot_clust2+ clust_count(i)
enddo
 close(97)

call poscopy             !writes_final_positions_to_a_file
call cpu_time(ends)

print*, 'density : ', den
print*, 'particles : ', npart
print*, 'avg. cluster size : ', dfloat(avg_clustsize)/dfloat(tot_clust)
print*, 'avg. no: of branching: ', dfloat(branchnum)/dfloat((niter-iter_equil)/count_int&
&ervel)
print*, 'acceptance: ', dfloat(accptd)/dfloat(accptd+rjctd)
print*, 'rejection : ', dfloat(rjctd)/dfloat(accptd+rjctd)
print*, 'runtime: ', int((ends-start)/60),':',mod((ends- start),60.0)
call writerundetails      !write the results and the parameters used for the run


deallocate(pos)
deallocate(nghbours)
deallocate(nghlist)
deallocate(ifvisited)
deallocate(clust_count)
deallocate(boxnghlist)
deallocate(numofbonds)

end program chained

!=======================position_initialisation_subroutine===========================3

subroutine initpos
use olmc_para
implicit none

real*8 :: ran1

open(90, file='posi.dat',status='unknown',form='formatted')

r = ran1(zzzz)
pos(1)= r*llx
r = ran1(zzzz)
pos(2)= r*lly
r = ran1(zzzz)
pos(3)= r*llz
write(90, '(3F20.4)') pos(1), pos(2), pos(3)

i= 2

do
    if(i>npart) exit
    dummy = 0
    r = ran1(zzzz)
    tempx= r*llx
    r = ran1(zzzz)
    tempy= r*lly
    r = ran1(zzzz)
    tempz= r*llz
```

```
    do j= 1, i-1
        delx= abs(tempx-pos(3*j-2))
        if (delx >= llxby2) delx= llx-delx
        dely= abs(tempy-pos(3*j-1))
        if (dely >= llyby2) dely= lly-dely
        delz= abs(tempz-pos(3*j  ))
        if (delz >= llzby2) delz= llz-delz

        dist= dsqrt((delx*delx)+(dely*dely)+(delz*delz))

        if (dist<= min_ini_dist*sig) dummy=1
    enddo

    if(dummy==0)  then
        pos(3*i-2) = tempx;  pos(3*i-1) = tempy;  pos(3*i  ) = tempz
        write(90, '(3F20.4)') pos(3*i-2), pos(3*i-1), pos(3*i)
        i= i+1
    endif
enddo ; close(90)

end subroutine initpos

!===============================initial_energy_calculation==============================4

subroutine init_potential
use olmc_para
implicit none

pot_e =0.0d0
do i= 1, npart - 1
    do j= i+1, npart
        delx= abs(pos(3*i-2)-pos(3*j-2))
        if (delx >= llxby2) delx= llx-delx
        dely= abs(pos(3*i-1)-pos(3*j-1))
        if (dely >= llyby2) dely= lly-dely
        delz= abs(pos(3*i  )-pos(3*j  ))
        if (delz >= llzby2) delz= llz-delz

        dist= (delx*delx)+(dely*dely)+(delz*delz)

        if (dist <= rc2) then    !checks_if_particle_is_within_in_the_range
            dist= dsqrt(dist)
            pot_e= pot_e + eps*(alpha*(((sig/dist)**36)-((sig/dist)**18))+&
            &((beta*exp(-(dist-r0)/eta))/dist))-vrc
        endif
    enddo
enddo

end subroutine init_potential

!============================position_update_subroutine================================5

subroutine pos_update
use olmc_para
implicit none
real*8 :: dx, dy, dz, ran1

do i= 1, npart

    init_pot=0.0d0;final_pot=0.0d0
    posx= pos(3*i-2)
```

```
posy= pos(3*i-1)
posz= pos(3*i  )

!this loop calculates the initial potential
do j =1, nghbours(i)
    k= nghlist(j,i)
    delx= abs(posx -pos(3*k-2)); if (delx >= llxby2) delx= llx-delx
    dely= abs(posy -pos(3*k-1)); if (dely >= llyby2) dely= lly-dely
    delz= abs(posz -pos(3*k  )); if (delz >= llxby2) delz= llz-delz

    dist= (delx*delx)+(dely*dely)+(delz*delz)
    if (dist<=rc2) then
        dist= dsqrt(dist)
        if (dist<=bond_l) then
            init_pot= init_pot+ eps*(alpha*(((sig/dist)**36)-((sig/dist)**18))+((bet&
            &a*exp(-(dist-r0)/eta))/dist))-&
            &((delz/dist)**2)*b2-vrc
        else
            init_pot= init_pot+ eps*(alpha*(((sig/dist)**36)-((sig/dist)**18))+((bet&
            &a*exp(-(dist-r0)/eta))/dist))-vrc
        endif
    endif
enddo

!random_increments
r=ran1(zzzz)
dx= (0.5d0-r)*dd
r=ran1(zzzz)
dy= (0.5d0-r)*dd
r=ran1(zzzz)
dz= (0.5d0-r)*dd

!new_positions_after_the_above_increments
tempx= posx +dx
if (tempx>llx) tempx= tempx- llx
if (tempx<=0) tempx= tempx+ llx
tempy= posy +dy
if (tempy>lly) tempy= tempy- lly
if (tempy<=0) tempy= tempy+ lly
tempz= posz +dz
if (tempz>llz) tempz= tempz- llz
if (tempz<=0) tempz= tempz+ llz

!this_loop_calculates_the_changed_potential_for_the_particle
do j =1, nghbours(i)
    k= nghlist(j,i)
    delx= abs(tempx-pos(3*k-2)); if (delx >= llxby2) delx= llx-delx
    dely= abs(tempy-pos(3*k-1)); if (dely >= llyby2) dely= lly-dely
    delz= abs(tempz-pos(3*k  )); if (delz >= llxby2) delz= llz-delz

    dist= (delx*delx)+(dely*dely)+(delz*delz)
    if (dist<=rc2)then
        dist= dsqrt(dist)
        if (dist<=bond_l) then
            final_pot= final_pot+ eps*(alpha*(((sig/dist)**36)-((sig/dist)**18))+((b&
            &eta*exp(-(dist-r0)/eta))/dist))-&
            &((delz/dist)**2)*b2-vrc
        else
            final_pot= final_pot+ eps*(alpha*(((sig/dist)**36)-((sig/dist)**18))+((b&
            &eta*exp(-(dist-r0)/eta))/dist))-vrc
        endif
    endif
```

```
    enddo

    de= final_pot - init_pot
    !acceptance_or_rejectance_of_the_new_position_w.r.t._change_in_potential
    if (de< 0.0d0) then
        pot_e = pot_e + de
        pos(3*i-2)= tempx
        pos(3*i-1)= tempy
        pos(3*i  )= tempz
        accptd= accptd+1
    else
       call random_number(r)
       if (r.le. exp(-de/(kbt))) then
           pot_e = pot_e + de
           pos(3*i-2)= tempx
           pos(3*i-1)= tempy
           pos(3*i  )= tempz
           accptd= accptd+1
       else
           rjctd= rjctd+1
       endif
    endif

enddo !  do i= 1, npart

end subroutine pos_update

!========================assigning_neighbours_for_each_box===============================6

subroutine boxnghbours
use olmc_para
implicit none
integer :: bx, by, bz, tempbx, tempby, tempbz


do i= 1, nbox !assigning_neighbours_to_each_box
    bz= (i-1)/(boxsidey*boxsidex)
    by= (modulo(i-1,boxsidey*boxsidex)/boxsidex)
    bx= (modulo(i-1,boxsidex))

    m=0
    do j= 1, 3
        do k= 1, 3
            do n= 1, 3
                tempbx= (j-2)*1+ bx
                if(tempbx .ge. boxsidex) tempbx= tempbx- boxsidex
                if(tempbx .lt. 0) tempbx= tempbx+ boxsidex
                tempby= (k-2)*1+ by
                if(tempby .ge. boxsidey) tempby= tempby- boxsidey
                if(tempby .lt. 0) tempby= tempby+ boxsidey
                tempbz= (n-2)*1+ bz
                if(tempbz .ge. boxsidez) tempbz= tempbz- boxsidez
                if(tempbz .lt. 0) tempbz= tempbz+ boxsidez

                temp = (tempbz*llx*lly/(box_size**2))+(tempby*llx/box_size)+(tempbx)+ 1
                if(temp .ne. i) then
                    m=m+1
                    boxnghlist(m,i)= temp
                endif
            enddo
        enddo
    enddo
```

```
enddo   !i= 1, nbox


end subroutine boxnghbours

!========================updating_the_neighbourlist_cells==============================7

subroutine update_nghlist
use olmc_para
implicit none
integer *8 :: tmpx, tmpy, tmpz, p, ibox, ktemp
integer *8 :: n_box_mono(nbox+2)
integer *8 :: box_mono(maxboxcontent,nbox+2)

nghlist=0
nghbours=0
n_box_mono=0
box_mono=0

do i= 1, npart
    tmpx= pos(3*i-2)/box_size
    tmpy= pos(3*i-1)/box_size
    tmpz= pos(3*i  )/box_size
    temp = (tmpz*llx*lly/(box_size*box_size))+(tmpy*llx/box_size)+tmpx+1

    n_box_mono(temp)= n_box_mono(temp)+ 1
    if(n_box_mono(temp)>maxboxcontent) print*, "Err: The max box content exceeded!"
    box_mono(n_box_mono(temp),temp)= i
enddo

do i= 1, npart
    p=0
    tmpx= pos(3*i-2)/box_size
    tmpy= pos(3*i-1)/box_size
    tmpz= pos(3*i  )/box_size
    temp = (tmpz*llx*lly/(box_size*box_size))+(tmpy*llx/box_size)+tmpx+1

    do j= 1, n_box_mono(temp)
        dummy= box_mono(j,temp)
        if(dummy .ne. i) then
            delx= abs(pos(3*i-2)-pos(3*dummy-2)); if (delx >= llxby2) delx= llx-delx
            dely= abs(pos(3*i-1)-pos(3*dummy-1)); if (dely >= llyby2) dely= lly-dely
            delz= abs(pos(3*i  )-pos(3*dummy  )); if (delz >= llxby2) delz= llz-delz
            dist= (delx*delx)+(dely*dely)+(delz*delz)

            if(dist<r_ngh2) then
                p= p+1
                if(p>maxnghbours) print*, "Err: maxnghbours exceeded!"
                nghlist(p,i)= dummy
            endif
        endif
    enddo

    do j= 1, 26
        ibox= boxnghlist(j,temp)
        dummy= n_box_mono(ibox)
        if(dummy>0) then
            do k= 1, dummy
                ktemp= box_mono(k,ibox)
                delx= abs(pos(3*i-2)-pos(3*ktemp-2)); if (delx >= llxby2) delx= llx-delx
                dely= abs(pos(3*i-1)-pos(3*ktemp-1)); if (dely >= llyby2) dely= lly-dely
                delz= abs(pos(3*i  )-pos(3*ktemp  )); if (delz >= llxby2) delz= llz-delz
```

```
                dist= (delx*delx)+(dely*dely)+(delz*delz)

                if(dist<r_ngh2) then
                    p=p+1
                    if(p>maxnghbours) print*, "Err: maxnghbours exceeded!"
                    nghlist(p,i)= ktemp
                endif
            enddo
        endif
    enddo
    nghbours(i)= p
enddo !i= 1, npart

end subroutine update_nghlist

!=======================upadating_the_neighbourlist_basic==============================8

subroutine update_nghlist_basic
use olmc_para
implicit none
integer *8 :: a, b

nghlist=0
nghbours=0
do i= 1, npart-1
    do j= i+1, npart
        delx= abs(pos(3*i-2)-pos(3*j-2)); if (delx >= llxby2) delx= llx-delx
        dely= abs(pos(3*i-1)-pos(3*j-1)); if (dely >= llyby2) dely= lly-dely
        delz= abs(pos(3*i  )-pos(3*j  )); if (delz >= llxby2) delz= llz-delz

        dist= dsqrt((delx**2)+(dely**2)+(delz**2))
        if(dist<r_ngh) then
            nghbours(i)= nghbours(i)+1
            if(nghbours(i)>maxnghbours) print*, "Err: maxnghbours exceeded!"
            a= nghbours(i)
            nghbours(j)= nghbours(j)+1
            if(nghbours(i)>maxnghbours) print*, "Err: maxnghbours exceeded!"
            b= nghbours(j)
            nghlist(a,i)=j
            nghlist(b,j)=i
        endif
    enddo
enddo

end subroutine update_nghlist_basic

!============================counting_the_clusters_and_their_size======================9

subroutine clust_counter
use olmc_para
implicit none

ifvisited= 0
numofbonds= 0

do k= 1, npart
    clustersize= 0
    if(ifvisited(k) .eq. 0) then
        clustersize=1
        call counter(k)
    endif
    if(clustersize>maxclustsize) print*, "Err: maxclustsize exceeded!"
```

```
    clust_count(clustersize)= clust_count(clustersize) +1
enddo

end subroutine clust_counter

!===========================counter===================================================10

recursive subroutine counter(l)
use olmc_para
implicit none
integer :: l, g, h

ifvisited(l)=1

do h= 1, nghbours(l)
    g= nghlist(h,l)
    if(ifvisited(g) .eq. 0) then
        delx= abs(pos(3*l-2)-pos(3*g-2)); if (delx >= llxby2) delx= llx-delx
        dely= abs(pos(3*l-1)-pos(3*g-1)); if (dely >= llyby2) dely= lly-dely
        delz= abs(pos(3*l  )-pos(3*g  )); if (delz >= llxby2) delz= llz-delz
        dist= (delx*delx)+(dely*dely)+(delz*delz)

        if(dist < bond_l2) then
            clustersize= clustersize+ 1
            numofbonds(l)= numofbonds(l)+1
            numofbonds(g)= numofbonds(g)+1
            if(numofbonds(l)>2) branchnum= branchnum+1
            if(numofbonds(g)>2) branchnum= branchnum+1
            call counter(g)
        endif
    endif
enddo

end subroutine counter

!====================writing_final_positions==========================================11

subroutine poscopy
use olmc_para
implicit none

filename= trim(den_str)//"_"//trim(label)//".xyz"
open (91,file=filename,status='unknown', form='formatted')
write (91, '(I5)') npart
write (91, '(1A7)') 'Chained'
do i= 1, npart

  if(i/10000 >= 1) then
    write(91, '(1A,1I5,3F20.4)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
    else
    if(i/1000 >= 1) then
      write(91, '(1A,1I4,3F20.4)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
      else
      if(i/100 >= 1) then
        write(91, '(1A,1I3,3F20.4)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        else
        if(i/10 >= 1) then
          write(91, '(1A,1I2,3F20.4)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
          else
            write(91, '(1A,1I1,3F20.4)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        endif
      endif
```

```
    endif
  endif
enddo

close (91)
end subroutine poscopy

!==================writing_run_details=================================================12

subroutine writerundetails
use olmc_para
implicit none

filename= trim(den_str)//"_para&res_"//trim(label)//".dat"
open(80, file=filename, status='unknown', form='formatted')

write(80, '(1A28,1F8.4)')'density                    :', den
write(80, '(1A28,1I8)')  'particles                  :', npart
write(80, '(1A28,1F8.2)')'acceptance                 :', dfloat(accptd)/dfloat(accptd+rjctd)
write(80, '(1A28,1F8.2)')'rejection                  :', dfloat(rjctd)/dfloat(accptd+rjctd)
write(80, '(1A28,1F8.2)')'total clusters             :', dfloat(tot_clust)/dfloat((niter-it&
&er_equil)/count_intervel)
write(80, '(1A28,1F8.2)')'avg. cluster size          :', dfloat(avg_clustsize)/dfloat(tot_c&
&lust)
write(80, '(1A28,1F8.2)')'avg. no: of branching      :', dfloat(branchnum)/dfloat((niter-it&
&er_equil)/count_intervel)
write(80, '(1A28,1F8.2)')'total clusters2            :', dfloat(tot_clust2)/dfloat((niter-i&
&ter_equil)/count_intervel)
write(80, '(1A28,1F8.2)')'avg. cluster size2         :', dfloat(avg_clustsize2)/dfloat(tot_&
&clust2)
write(80, '(1A36)')      'PARAMETERS OF POTENTIAL=========>'
write(80, '(1A28,1F8.4)')'epsilon2                   :', eps2
write(80, '(1A28,1F8.4)')'epsilon                    :', eps
write(80, '(1A28,1F8.4)')'alpha                      :', alpha
write(80, '(1A28,1F8.4)')'beta                       :', beta
write(80, '(1A28,1F8.4)')'eta                        :', eta
write(80, '(1A28,1F8.4)')'r0                         :', r0
write(80, '(1A28,1F8.4)')'B^2                        :', b2
write(80, '(1A36)')      'OTHERS=========================>'
write(80, '(1A28,1I8)')  'no: of iterations          :', niter
write(80, '(1A28,1I8)')  'equilibrium iteration      :', iter_equil
write(80, '(1A28,1F8.4)')'bond length                :', bond_l
write(80, '(1A28,1F8.4)')'range of potential         :', rc
write(80, '(1A28,1F8.4)')'max. displacement radius:', dd/2.0d0
write(80, '(1A28,1F8.4)')'neighbourhood radius       :', r_ngh
write(80, '(1A28,1F8.4)')'minimum initial distance:', min_ini_dist
write(80, '(1A7,1I3,1A10,1I3,1A10,1I3)')'   lx: ', lx, ' ly: ', ly, ' lz: ', lz
write(80, '(1A18,1I4,1A5,1F5.2,1A4)')'code runtime  :', &
&int((ends-start)/60),' min:',mod((ends- start),60.0),' sec'

 close(80)

end subroutine writerundetails


!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FUNCTION ran1(IDUM)
implicit none

!  RAN1 returns a unifom random deviate on the interval [0,1]
! ----------------------------------------------------------------
!
```

```
INTEGER :: IDUM
REAL*8 :: RAN2,ran1
integer,parameter :: IM1=2147483563,IM2=2147483399
integer,parameter :: IMM1=IM1-1,                                   &
IA1=40014,IA2=40692,IQ1=53668,IQ2=52774,IR1=12211,IR2=3791, &
           NTAB=32
        integer,parameter :: NDIV=1+IMM1/NTAB
         real*8,parameter :: EPS=1.2e-7,RNMX=1.-EPS,AM=1./IM1
        INTEGER :: IDUM2,J,K,IV(NTAB),IY
        DATA IDUM2/123456789/, iv/NTAB*0/, iy/0/
        IF (IDUM.LE.0) THEN
          IDUM=MAX(-IDUM,1)
          IDUM2=IDUM
          DO  J=NTAB+8,1,-1
             K=IDUM/IQ1
             IDUM=IA1*(IDUM-K*IQ1)-K*IR1
             IF (IDUM.LT.0) IDUM=IDUM+IM1
             IF (J.LE.NTAB) IV(J)=IDUM
          end do
          IY=IV(1)
        ENDIF
        K=IDUM/IQ1
        IDUM=IA1*(IDUM-K*IQ1)-K*IR1
        IF (IDUM.LT.0) IDUM=IDUM+IM1
        K=IDUM2/IQ2
        IDUM2=IA2*(IDUM2-K*IQ2)-K*IR2
        IF (IDUM2.LT.0) IDUM2=IDUM2+IM2
        J=1+IY/NDIV
        IY=IV(J)-IDUM2
        IV(J)=IDUM
        IF(IY.LT.1)IY=IY+IMM1
        RAN1=MIN(AM*IY,RNMX)

        END function ran1

!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.2 Molecular Dynamics of shearing the system

(Fortran 90)

```
!==========================================================================================!
!                                                                                          !
!                                   Molecular Dynamics                                     !
!                                                                                          !
!        V(r)= eps*(alpha*((sig/r)**36-(sig/r)**18)+((beta*exp(-(r-r0)/eta))/r))           !
!                          with walls of lj interaction                                    !
!                                                                                          !
!==========================================================================================!

module md_para
implicit none
```

```
! model_parameters
integer,parameter :: lx=20, ly=20, lz=30
real*8 :: llx, lly, llz
real*8 :: llxby2, llyby2, llzby2
real*8 :: den= 0.135d0 !density
real*8,parameter :: eps= 24.0d0, kbt= 1.0d0, sig= 1.0d0, mass= 1.0d0,tem= 0.5d0,rc=2.5d0&
                   &,beta= 1.3d0, eta=0.25d0, b2= 0.0d0, eps2= 1.0d0, rc_wall= 2.5d0
                   ! If the rc_wall is being set higher than rc, then modify forceup too
real*8,parameter :: alpha= 5.8d0, r0= 1.12d0, ngh_trim= 0.8d0     !minimum of lj potential
! epsilon, boltz. const., particle_diameter, mass, temperature, potential_range etc.

! simulation_variables
 character(len=20) :: label= '744mdshear'
integer, parameter :: niter= 550000, iter_equil= 180000, nghupdate= 10,&
                     & count_intervel= 10, stabilise_intervel= 200,&
                     sample_intervel= 20, check_intervel= 90000
integer, parameter :: maxclustsize= 8000, maxboxcontent= 3000, maxnghbours= 2500
integer :: npart, nwall, nwallby2, nwallx, nwally, ntot, leaked=0, layernum
integer :: nbox, boxsidex, boxsidey, boxsidez  !no: boxes and no: of boxes along each edge
integer*8 :: clustersize, tot_clust, avg_clustsize, branchnum= 0, iter_steadystate= 0
integer*8, dimension(:), allocatable :: nghbours, boxnghbournum, n_box_mono
integer*8, dimension(:), allocatable :: clust_count, ifvisited, numofbonds
integer*8, dimension(:,:), allocatable :: nghlist, boxnghlist, box_mono
real*8 :: pot_e, kin_e, tot_e
real*8 :: frc, vrc, frc_wall, vrc_wall, v0, wall_dx, vwall= 0.3d0, shearrate
real*8 :: vdev, vdev100= 20.0d0
real*8 :: sliplen, sliplen2, a, b      !ax + b linear fit coefficients
real*8 :: xnwalldist, ynwalldist, ynwallcorrection, wallscalefactor= 0.8d0
real*8 :: realn, vol, delx, dely, delz, dist, r_ngh, rij, f, bond_l= 1.2d0
real*8 :: decimals= 2.0d0, exp10, layervol, endlayervol, ljruntime
real*8 :: box_size= 5.0d0, layerbin= 1.0d0 !should be a value that can divide lz
                                          !and can a give a integer quotient,
                                          !shouldnt be larger than box_size
real*8, parameter ::  dt= 0.004d0
real*8, dimension(:), allocatable :: prev_vx, curr_vx, densityprof, temprof, temprof2, t&
&emprofx, temprofy, temprofz
real*8, dimension(:), allocatable :: pos, vel, force, oldforce, layertemp, xveloprof, xv&
&eloprof2, yveloprof, zveloprof

integer :: zzz=-4271246, zzzz=65902043

! index_variables
integer  :: i, j, k, m, n, temp, iter, dummy, tmpx, tmpy, tmpz, daycount

! others
 character(len=60) :: filename
 character(len=10) :: den_str, daycount_str
integer :: tempmetercalls= 0, steadycalls= 0, tempmetercalls2=0
real*8 :: r, tempx, tempy, tempz, posx, posy, posz, halfdt, halfddt
real*8 :: rc2, rc_wall2, r_ngh2, bond_l2

! timer_variables
real*8 :: start, ends, endstemp !start time, end time

end module

!=======================================mainprogram=====================================1

program chained2
use md_para
implicit none
real*8 :: junk,zz1,ran1
```

```
call cpu_time(start)
call initcalc

print*, 'npart  :', npart
print*, 'nwall  :', nwall
print*, 'ntot   :', ntot
print*, 'nbox   :', nbox
print*, 'vwall  :', vwall
print*, 'dt     :', dt
print*, 'wall_dx:', wall_dx

allocate(pos(3*ntot))
allocate(vel(3*npart))
allocate(force(3*npart)); force=0.0d0
allocate(oldforce(3*npart))
allocate(nghbours(npart))
allocate(nghlist(maxnghbours,npart))
allocate(ifvisited(npart))
allocate(clust_count(maxclustsize)); clust_count=0
allocate(n_box_mono(nbox+2))
allocate(box_mono(maxboxcontent,nbox+2))
allocate(boxnghbournum(nbox))
allocate(boxnghlist(27,nbox))
allocate(numofbonds(npart))
allocate(layertemp(layernum+2)); layertemp=0.0d0
allocate(xveloprof(layernum+2)); xveloprof=0.0d0
allocate(xveloprof2(layernum+2)); xveloprof2=0.0d0
allocate(yveloprof(layernum+2)); yveloprof=0.0d0
allocate(zveloprof(layernum+2)); zveloprof=0.0d0
allocate(prev_vx(layernum+2)); prev_vx=0.0d0
allocate(curr_vx(layernum+2)); curr_vx=0.0d0
allocate(densityprof(layernum+2)); densityprof=0.0d0
allocate(temprof(layernum+2)); temprof=0.0d0
allocate(temprof2(layernum+2)); temprof2=0.0d0
allocate(temprofx(layernum+2)); temprofx=0.0d0
allocate(temprofy(layernum+2)); temprofy=0.0d0
allocate(temprofz(layernum+2)); temprofz=0.0d0


call boxnghbours           !assigns_nghbours_for_each_box
call initpos               !ensemble_initialisation
call fullforceup           !calculates forces
call initvel               !calculating initial velocities
print*, "Initial simulation setup done..."


!md iterations
filename= trim(den_str)//'_Elist_'//trim(label)//'.dat'
 open (94, file=filename,status='unknown',form='formatted')
do iter= 0, niter
    call energymeter
    write(94, *) iter, dt*iter, pot_e, kin_e, tot_e

    if(mod(iter,nghupdate)==0) call update_nghlist        !updates_neighbourlist
    if((mod(iter, sample_intervel)==0) .and. (iter.gt.iter_equil)) call tempmeter
    call update                        !updates positions, velocities and forces

    if((mod(iter-1, check_intervel)==0) .and. (iter_steadystate .gt. niter)&
    & .and. (iter-1 .gt.iter_equil)) call steadystate_determiner
                                        !checks if the system is in steady state
```

```
    if(mod(iter, stabilise_intervel)==0) call thermostat

    if(mod(iter,500)==0) print*, iter, leaked
    if(mod(iter,10000)==2000) then
        call cpu_time(ends)
        write(*,'(1A15,1I2,1A,1I2,1A,1I2)'), &
        &'Time remaining:', int((niter-iter)*(ends-start)/(iter*3600)), & !hours
        &      ':', mod(int((niter-iter)*(ends-start)/(iter*60)),60), & !minutes
        &      ':', mod(int((niter-iter)*(ends-start)/iter),60)         !seconds
    endif
    if(mod(iter,10000)==0) call poscopy
    call cpu_time(endstemp)
    if(int((endstemp-start)/(24*60*60)) .gt. daycount) then
        daycount= daycount+1
        write(daycount_str, '(1I2)') daycount
        call poscopy2
        call writerundetails2
    endif
enddo
    call energymeter
    write(94, *) iter, dt*iter, pot_e, kin_e, tot_e
 close (94)

do i= 1, layernum
    if((i.eq.1) .or. (i.eq.layernum)) then
        densityprof(i)= vol*densityprof(i)/(endlayervol*dfloat(tempmetercalls2))
    else
        densityprof(i)= vol*densityprof(i)/(layervol*dfloat(tempmetercalls2))
    endif
    xveloprof2(i)= xveloprof2(i)/dfloat(tempmetercalls2)
    yveloprof(i)= yveloprof(i)/dfloat(tempmetercalls2)
    zveloprof(i)= zveloprof(i)/dfloat(tempmetercalls2)
    layertemp(i)= (mass*layertemp(i))/(2.0d0*dfloat(tempmetercalls2))

    temprof(i)= temprof(i)*mass/(3.0d0*kbt*dfloat(tempmetercalls2))
    temprof2(i)= temprof2(i)*mass/(3.0d0*kbt*dfloat(tempmetercalls2))
    temprofx(i)= temprofx(i)*mass/(3.0d0*kbt*dfloat(tempmetercalls2))
    temprofy(i)= temprofy(i)*mass/(3.0d0*kbt*dfloat(tempmetercalls2))
    temprofz(i)= temprofz(i)*mass/(3.0d0*kbt*dfloat(tempmetercalls2))
    !here kbt is boltzmns const.
enddo

call linearfitter

sliplen= dt*(vwall-((abs(xveloprof2(1))+&
         &abs(xveloprof2(layernum)))/(2.0d0)))
sliplen2= 1.0d0-((vwall-b)/a)    !position of vwall - point at which the linear
                                 !fit has same value

call cpu_time(ends)
call poscopy                                  !writes final positions to a file
call writerundetails   !writes the results, data and parameters used for the run

print*, 'runtime: ', int((ends-start)/3600),':',mod(int((ends-start)/60.0),60),':',mod((&
&ends- start),60.0)
print*, 'density: ', den
print*, 'npart:', npart
print*, 'nwall:', nwall
print*, 'ntot :', ntot
print*, 'vwall:', vwall
print*, 'leaked:', leaked
print*, 'sliplength:', sliplen
```

```
print*, 'sliplength 2:', sliplen2
print*, 'linear fit for xvelo:', a, 'x + ', b


deallocate(pos)
deallocate(vel)
deallocate(force)
deallocate(oldforce)
deallocate(nghbours)
deallocate(nghlist)
deallocate(ifvisited)
deallocate(clust_count)
deallocate(n_box_mono)
deallocate(box_mono)
deallocate(boxnghbournum)
deallocate(boxnghlist)
deallocate(numofbonds)
deallocate(layertemp)
deallocate(xveloprof)
deallocate(xveloprof2)
deallocate(yveloprof)
deallocate(zveloprof)
deallocate(prev_vx)
deallocate(curr_vx)
deallocate(densityprof)
deallocate(temprof)
deallocate(temprof2)
deallocate(temprofx)
deallocate(temprofy)
deallocate(temprofz)


end program chained2

!===============================initial_calculations===================================2

subroutine initcalc
use md_para
implicit none


llx=dfloat(lx); lly=dfloat(ly); llz= dfloat(lz)  !sidelengths to dfloat
llxby2=llx/2.0d0; llyby2=lly/2.0d0; llzby2=llz/2.0d0  !half of side lengths
halfdt= 0.5d0*dt
halfddt= 0.5d0*dt*dt
rc2= rc*rc
rc_wall2= rc_wall*rc_wall
bond_l2= bond_l*bond_l

daycount=0
iter_steadystate=niter+2

wall_dx= vwall*dt
ljruntime= niter*dt
shearrate= 2.0d0*vwall/llz
exp10= 10.0d0**decimals

xnwalldist= 1.0d0*wallscalefactor
ynwalldist= (dsqrt(3.0d0)/2.0d0)*wallscalefactor
nwallx= int(llx/xnwalldist)
nwally= int(lly/ynwalldist)
if(mod(nwally,2)==1) nwally= nwally+1
```

```
xnwalldist= llx/dfloat(nwallx)
ynwalldist= lly/dfloat(nwally)
print*, 'nwallx: ', nwallx, 'nwally: ', nwally
nwallby2= nwallx*nwally
nwall= 2*nwallby2
vol = 4.0d0*3.14d0*(sig*sig*sig/24.0d0)  !volume of a single particle
npart= int((den*(llx*lly*llz-nwallby2*vol))/vol)  !calculating no: of particles
!-nwallby2*vol bcoz the hemispheres of nwall particles take up nwall*vol/2=nwallby2*vol
ntot= npart+nwall
realn= dfloat(npart)
v0= dsqrt(3.0d0*tem)
layernum= int(llz/layerbin)
layervol= layerbin*llx*lly
if(layerbin .le. (sig/2.0d0) ) then
    print*, "Boundary layers not accesible to the free particles. Modify the code!"
else
    endlayervol= (layerbin*llx*lly)-(nwallby2*vol*0.5d0)
                 !correcting for vol occupied by the wall particles
endif

write(den_str, '(1F5.3)') den

frc_wall= 4.0d0*eps2*( 12.0d0*( (sig**12.0d0)/(rc_wall**13.0d0) )-&
                   & ( 6.0d0*( (sig** 6.0d0)/(rc_wall** 7.0d0) )) )
vrc_wall= 4.0d0*eps2*(((sig/rc_wall)**12.0d0)-((sig/rc_wall)**6.0d0))+frc_wall*rc_wall
frc= eps*(alpha*(36*((sig**36)/(rc**37))-18*((sig**18)/(rc**19)))+&
       &(beta*((exp(-(rc-r0)/eta)/(eta*rc))+(exp(-(rc-r0)/eta)/(rc*rc)))))
vrc= eps*(alpha*((sig/rc)**36-(sig/rc)**18)+((beta*exp(-(rc-r0)/eta))/rc))+(frc*rc)
r_ngh= 3.5d0
r_ngh2= r_ngh*r_ngh

boxsidex= int(lx/box_size)
boxsidey= int(ly/box_size)
boxsidez= int(lz/box_size)
nbox= boxsidex*boxsidey*boxsidez

end subroutine initcalc

!========================position_initialisation_subroutine============================3

subroutine initpos
use md_para
implicit none

real*8 :: ran1

open(90, file='posi.dat',status='unknown',form='formatted')

call initpos_wall_tri

r = ran1(zzzz)
pos(1)= r*llx
r = ran1(zzzz)
pos(2)= r*lly
r = ran1(zzzz)
pos(3)= r*llz
write(90, '(3F20.4)') pos(1), pos(2), pos(3)

i= 2

do
    if(i>npart) exit
```

```
    dummy = 0
    r = ran1(zzzz)
    tempx= r*llx
    r = ran1(zzzz)
    tempy= r*lly
    r = ran1(zzzz)
    tempz= r*llz

    j=1
    do while(j .le. ntot)
        delx= abs(tempx-pos(3*j-2))
        if (delx >= llxby2) delx= llx-delx
        dely= abs(tempy-pos(3*j-1))
        if (dely >= llyby2) dely= lly-dely
        delz= abs(tempz-pos(3*j  ))

        dist= dsqrt((delx*delx)+(dely*dely)+(delz*delz))

        if (dist<= 1.2*sig) dummy=1
        j= j+1
        if (j==i) j= npart+1
    enddo

    if(dummy==0)  then
        pos(3*i-2) = tempx;  pos(3*i-1) = tempy;  pos(3*i  ) = tempz
        write(90, '(3F20.4)') pos(3*i-2), pos(3*i-1), pos(3*i)
        i= i+1
    endif
enddo ; close(90)

end subroutine initpos

!================initializes_the_wall_particles_in_square_lattice=======================4

subroutine initpos_wall_sq
use md_para
implicit none

pos=0.0d0
i=npart+1
do j= 1, nwallx
    do k= 1, nwally
        pos(3*i-2)= dfloat(j-1)*xnwalldist+ xnwalldist/2.0d0
        pos(3*i-1)= dfloat(k-1)*ynwalldist+ ynwalldist/2.0d0
        pos(3*i  )= 0.000001d0
        write(90, '(3F20.4)') pos(3*i-2), pos(3*i-1), pos(3*i)
        i= i+1
    enddo
enddo

do j= 1, nwallx
    do k= 1, nwally
        pos(3*i-2)= dfloat(j-1)*xnwalldist+ xnwalldist/2.0d0
        pos(3*i-1)= dfloat(k-1)*ynwalldist+ ynwalldist/2.0d0
        pos(3*i  )= llz- 0.000001d0
        write(90, '(3F20.4)') pos(3*i-2), pos(3*i-1), pos(3*i)
        i= i+1
    enddo
enddo

end subroutine initpos_wall_sq
```

```
!================initializes_the_wall_particles_in_triangular_lattice===================5

subroutine initpos_wall_tri
use md_para
implicit none

pos= 0.0d0
i= npart+1
do j= 1, nwallx
    do k= 1, nwally
        if (mod(k,2) == 0) then
            pos(3*i-2)= dfloat(j-1)*(xnwalldist)+ xnwalldist- 0.000001d0
        else
            pos(3*i-2)= dfloat(j-1)*(xnwalldist)+ (xnwalldist/2.0d0)- 0.000001d0
        endif
        pos(3*i-1)= dfloat(k-1)*(ynwalldist)+ ynwalldist/2.0d0
        pos(3*i  )= 0.000001d0
        write(90, '(3F20.4)') pos(3*i-2), pos(3*i-1), pos(3*i)
        i= i+1
    enddo
enddo

do j= 1, nwallx
    do k= 1, nwally
        if (mod(k,2) == 0) then
            pos(3*i-2)= dfloat(j-1)*(xnwalldist)+ (xnwalldist)- 0.000001d0
        else
            pos(3*i-2)= dfloat(j-1)*(xnwalldist)+ (xnwalldist/2.0d0)- 0.000001d0
        endif
        pos(3*i-1)= dfloat(k-1)*(ynwalldist)+ ynwalldist/2.0d0
        pos(3*i  )= llz- 0.000001d0
        write(90, '(3F20.4)') pos(3*i-2), pos(3*i-1), pos(3*i)
        i= i+1
    enddo
enddo

end subroutine initpos_wall_tri

!================================initializes_the_velocities============================6

subroutine initvel
use md_para
implicit none
real*8 :: ran1
real*8 :: xv, yv, zv, vsqr= 0.0d0
xv=0.0d0; yv=0.0d0; zv=0.0d0

do i= 1, npart
    r=ran1(zzzz)
    vel(3*i-2)= 2.0d0*(r-0.5d0)*v0
    r=ran1(zzzz)
    vel(3*i-1)= 2.0d0*(r-0.5d0)*v0
    r=ran1(zzzz)
    vel(3*i  )= 2.0d0*(r-0.5d0)*v0
    xv= xv+ vel(3*i-2)
    yv= yv+ vel(3*i-1)
    zv= zv+ vel(3*i  )
    vsqr= vsqr+vel(3*i-2)*vel(3*i-2)+vel(3*i-1)*vel(3*i-1)+vel(3*i)*vel(3*i)
enddo

xv= xv/realn; yv= yv/realn; zv= zv/realn
vsqr= 0.0d0
```

```
do i= 1, npart
    vel(3*i-2)= vel(3*i-2)-xv
    vel(3*i-1)= vel(3*i-1)-yv
    vel(3*i  )= vel(3*i  )-zv
    vsqr= vsqr+vel(3*i-2)*vel(3*i-2)+vel(3*i-1)*vel(3*i-1)+vel(3*i)*vel(3*i)
enddo

end subroutine initvel

!===================updates_positions,force_and_velocities===========================7

subroutine update
use md_para
implicit none

!updating positions

do i= 1, npart
    posx= pos(3*i-2); posy= pos(3*i-1); posz= pos(3*i)

    posx= posx+ (vel(3*i-2)*dt)+ (force(3*i-2)*halfddt)
    posx= modulo(posx,llx)
    posy= posy+ (vel(3*i-1)*dt)+ (force(3*i-1)*halfddt)
    posy= modulo(posy,lly)
    posz= posz+ (vel(3*i  )*dt)+ (force(3*i  )*halfddt)
    if((posz.gt.llz) .or. (posz.lt. 0.0d0)) then
        leaked= leaked+1
        print*, "Particle ", i, "has been leaked out!"
        pos(i:)= eoshift(pos(i:),shift=1)
        vel(i:)= eoshift(vel(i:),shift=1)
        force(i:)= eoshift(force(i:),shift=1)
        oldforce(i:)= eoshift(oldforce(i:),shift=1)
        npart= npart-1
        ntot= npart+ nwall
        call update_nghlist
    endif
    pos(3*i-2)= posx; pos(3*i-1)= posy; pos(3*i)= posz
enddo

do i= npart+1, npart+nwallby2 !z-position= 0.0
    posx= pos(3*i-2)
    posx= posx+ wall_dx
    if(posx .ge. llx) posx= posx-llx
    pos(3*i-2)= posx
enddo
do i= npart+nwallby2+1, ntot !z-position= 30.0
    posx= pos(3*i-2)
    posx= posx- wall_dx
    if(posx .lt. 0.0d0) posx= llx+posx
    pos(3*i-2)= posx
enddo

oldforce= force

call forceup  !updating forces, using nghlist

vel= vel+ (oldforce+force)*halfdt  !updating velocities

end subroutine update

!==========================updates_the_forces========================================8
```

```
subroutine forceup
use md_para
implicit none
real*8 ::junk3

pot_e= 0.0d0
force= 0.0d0

do i= 1, npart
    do j= 1, nghbours(i)
        k= nghlist(j,i)
        delx= (pos(3*i-2) -pos(3*k-2))
        if (abs(delx) > llxby2) delx= (-1.0d0*delx/abs(delx))*(llx-abs(delx))
        dely= (pos(3*i-1) -pos(3*k-1))
        if (abs(dely) > llyby2) dely= (-1.0d0*dely/abs(dely))*(lly-abs(dely))
        delz= (pos(3*i  ) -pos(3*k  ))

        dist= (delx*delx)+(dely*dely)+(delz*delz)
        if (dist .le. rc2) then
            if (k .le. npart) then
                dist= dsqrt(dist)
                f= eps*(alpha*(36.0d0*((sig**36.0d0)/(dist**37.0d0))-&
                &             18.0d0*((sig**18.0d0)/(dist**19.0d0)))+&
                &     (beta*((exp(-(dist-r0)/eta)/(eta*dist))+&
                &            (exp(-(dist-r0)/eta)/(dist*dist)))))-frc
                force(3*i-2)= force(3*i-2)+(delx/dist)*f
                force(3*i-1)= force(3*i-1)+(dely/dist)*f
                force(3*i  )= force(3*i  )+(delz/dist)*f
                force(3*k-2)= force(3*k-2)-(delx/dist)*f
                force(3*k-1)= force(3*k-1)-(dely/dist)*f
                force(3*k  )= force(3*k  )-(delz/dist)*f
                pot_e= pot_e + eps*(alpha*(((sig/dist)**36.0d0)-((sig/dist)**18.0d0))+&
                                  &((beta*exp(-(dist-r0)/eta))/dist))+(frc*dist)-vrc
            else
                if (dist .le. rc_wall2) then
                    dist= dsqrt(dist)
                    f= 4.0d0*eps2*((12.0d0*((sig**12.0d0)/(dist**13.0d0)))-&
                    &              ( 6.0d0*((sig** 6.0d0)/(dist** 7.0d0))))-frc_wall
                    force(3*i-2)= force(3*i-2)+(f*delx/dist)
                    force(3*i-1)= force(3*i-1)+(f*dely/dist)
                    force(3*i  )= force(3*i  )+(f*delz/dist)
                    junk3= 4.0d0*eps2*(((sig/dist)**12.0d0)-&
                    &((sig/dist)**6.0d0))+frc_wall*dist-vrc_wall
                    pot_e= pot_e+ junk3
                endif
            endif
        endif
    enddo
enddo

end subroutine forceup

!===============updates_positions,force_and_velocities,no_nghlist======================9

subroutine fullupdate
use md_para
implicit none

!updating positions

do i= 1, npart
```

```
    posx= pos(3*i-2); posy= pos(3*i-1); posz= pos(3*i)
    posx= posx+ (vel(3*i-2)*dt)+ (force(3*i-2)*halfddt)
    posx= modulo(posx,llx)
    posy= posy+ (vel(3*i-1)*dt)+ (force(3*i-1)*halfddt)
    posy= modulo(posy,lly)
    posz= posz+ (vel(3*i  )*dt)+ (force(3*i  )*halfddt)
    if((posz.gt.llz) .or. (posz.lt. 0.0d0)) then
        leaked= leaked+1
        print*, "Particle ", i, "has been leaked out!"
        pos(i:)= eoshift(pos(i:),shift=1)
        vel(i:)= eoshift(vel(i:),shift=1)
        force(i:)= eoshift(force(i:),shift=1)
        oldforce(i:)= eoshift(oldforce(i:),shift=1)
        npart= npart-1
        ntot= npart+ nwall
        call update_nghlist
    endif
    pos(3*i-2)= posx; pos(3*i-1)= posy; pos(3*i)= posz
enddo
do i= npart+1, nwallby2+npart
    posx= pos(3*i-2)
    posx= posx+ wall_dx
    posx= modulo(posx,llx)
    pos(3*i-2)= posx
enddo
do i= nwallby2+npart+1, ntot
    posx= pos(3*i-2)
    posx= posx- wall_dx
    posx= modulo(posx,llx)
    pos(3*i-2)= posx
enddo

oldforce= force

call fullforceup  !updating forces, with no nghlist

vel= vel+ (oldforce+force)*halfdt  !updating velocities

end subroutine fullupdate

!===========================updates_the_forces,no_nghlist==============================10

subroutine fullforceup
use md_para
implicit none

pot_e= 0.0d0
force= 0.0d0

do i= 1, npart
    do k= i+1, ntot
        delx= (pos(3*i-2) -pos(3*k-2))
        if (abs(delx) > llxby2) delx= (-1.0d0*delx/abs(delx))*(llx-abs(delx))
        dely= (pos(3*i-1) -pos(3*k-1))
        if (abs(dely) > llyby2) dely= (-1.0d0*dely/abs(dely))*(lly-abs(dely))
        delz= (pos(3*i  ) -pos(3*k  ))

        dist= (delx*delx)+(dely*dely)+(delz*delz)
        if (dist .le. rc2) then
            if (k .le. npart) then
                dist= dsqrt(dist)
                f= eps*(alpha*(36.0d0*((sig**36.0d0)/(dist**37.0d0))-&
```

```
                &              18.0d0*((sig**18.0d0)/(dist**19.0d0)))+&
                &      (beta*((exp(-(dist-r0)/eta)/(eta*dist))+&
                &             (exp(-(dist-r0)/eta)/(dist*dist)))))-frc
                force(3*i-2)= force(3*i-2)+(f*delx/dist)
                force(3*i-1)= force(3*i-1)+(f*dely/dist)
                force(3*i  )= force(3*i  )+(f*delz/dist)
                force(3*k-2)= force(3*k-2)-(f*delx/dist)
                force(3*k-1)= force(3*k-1)-(f*dely/dist)
                force(3*k  )= force(3*k  )-(f*delz/dist)
                pot_e= pot_e + eps*(alpha*(((sig/dist)**36.0d0)-((sig/dist)**18.0d0))+&
                               &((beta*exp(-(dist-r0)/eta))/dist))+(frc*dist)-vrc
            else
                if (dist .le. rc_wall2) then
                    dist= dsqrt(dist)
                    f= 4.0d0*eps2*((12.0d0*((sig**12.0d0)/(dist**13.0d0)))-&
                    &            (6.0d0*((sig**6.0d0)/(dist**7.0d0))))-frc_wall
                    force(3*i-2)= force(3*i-2)+(f*delx/dist)
                    force(3*i-1)= force(3*i-1)+(f*dely/dist)
                    force(3*i  )= force(3*i  )+(f*delz/dist)
                    pot_e= pot_e+ 4.0d0*eps2*(((sig/dist)**12.0d0)-&
                    &((sig/dist)**6.0d0))+frc_wall*dist-vrc_wall
                endif
            endif
        endif
    enddo
enddo

end subroutine fullforceup

!==================================thermostat==========================================11

subroutine thermostat
use md_para
implicit none
real*8 :: scalefactorx, scalefactor, vsqx, vsq

vsqx= 0.0d0
vsq= 0.0d0
do i= 1, npart
    vsq= vsq+ (vel(3*i-1)*vel(3*i-1))+(vel(3*i)*vel(3*i))
    vsqx= vsqx+ (vel(3*i-2)*vel(3*i-2))
enddo

scalefactor= dsqrt(2.0d0*realn/vsq)
scalefactorx= dsqrt(1.0d0*realn/vsqx)

do i= 1, npart
    vel(3*i-2)= vel(3*i-2)*scalefactorx
    vel(3*i-1)= vel(3*i-1)*scalefactor
    vel(3*i  )= vel(3*i  )*scalefactor
enddo

end subroutine thermostat

!==============================calculates_KE,PE_and_TE================================12

subroutine energymeter
use md_para
implicit none

kin_e= 0.0d0; tot_e= 0.0d0
```

```
do i= 1, npart
    kin_e= kin_e + &
    &0.5d0*((vel(3*i-2)*vel(3*i-2))+(vel(3*i-1)*vel(3*i-1))+(vel(3*i)*vel(3*i)))
enddo

pot_e= pot_e/realn
kin_e= kin_e/realn
tot_e= pot_e + kin_e

end subroutine energymeter

!=========================determines_the_temperature_gradient==========================13

subroutine tempmeter
use md_para
implicit none
integer :: layercount, layerpart, itemp, q
real*8 :: layerpos, uplim, lowlim, junk4
real*8 :: v2, v2c_1, v2c_2, v2c_x, v2c_y, v2c_z, vx, vy, vz, vxbar, vybar, vzbar
real*8, dimension(1:1200) :: vxpart, vypart, vzpart

layercount= 1
layerpos= layerbin/2.0d0
lowlim= 0.0d0; uplim= layerbin

do while(layerpos .lt. llz)
layerpart= 0
v2= 0.0d0; v2c_1= 0.0d0; v2c_2= 0.0d0; v2c_x= 0.d0; v2c_y= 0.0d0; v2c_z= 0.0d0
vx= 0.0d0; vy= 0.0d0; vz= 0.0d0; vxpart=0.0d0; vypart= 0.0d0; vzpart= 0.0d0
tempy= box_size/2.0d0
do while(tempy .lt. lly)
tempx= box_size/2.0d0
do while(tempx .lt. llx)
tmpx= tempx/box_size
     tmpy= tempy/box_size
     tmpz= layerpos/box_size
     temp= (tmpz*llx*lly/(box_size*box_size))+(tmpy*llx/box_size)+tmpx+1

     do i= 1, n_box_mono(temp)
     itemp= box_mono(i,temp)
     if(itemp .le. npart) then
     if((pos(3*itemp) .lt. uplim) .and.&
       &(pos(3*itemp) .ge. lowlim)) then
     layerpart= layerpart+1
         vx= vx+ vel(3*itemp-2)
         vy= vy+ vel(3*itemp-1)
         vz= vz+ vel(3*itemp  )
         vxpart(layerpart)= vel(3*itemp-2)
         vypart(layerpart)= vel(3*itemp-1)
         vzpart(layerpart)= vel(3*itemp  )
         junk4= ((vel(3*itemp-2)*vel(3*itemp-2))+&
         &(vel(3*itemp-1)*vel(3*itemp-1))+(vel(3*itemp)*vel(3*itemp)))
     v2= v2+ junk4
     endif
     endif
     enddo
     tempx= tempx+box_size
enddo
tempy= tempy+box_size
enddo
if (layerpart .ne. 0) then
    vxbar= vx/dfloat(layerpart)
```

```
    vybar= vy/dfloat(layerpart)
    vzbar= vz/dfloat(layerpart)
    xveloprof2(layercount)= xveloprof2(layercount)+ vxbar
       xveloprof(layercount)= xveloprof(layercount)+ vxbar
     yveloprof(layercount)= yveloprof(layercount)+ vybar
     zveloprof(layercount)= zveloprof(layercount)+ vzbar
     layertemp(layercount)= layertemp(layercount)+ (v2/dfloat(layerpart))
     densityprof(layercount)= densityprof(layercount)+ (layerpart)
     do i= 1, layerpart
         v2c_x= v2c_x+ (vxpart(i)-curr_vx(layercount))*(vxpart(i)-curr_vx(layercount))
         v2c_y= v2c_y+ (vypart(i))*(vypart(i))
         v2c_z= v2c_z+ (vzpart(i))*(vzpart(i))
         v2c_1=v2c_1+((vxpart(i)-curr_vx(layercount))*(vxpart(i)-curr_vx(layercount)))&
                     &+((vypart(i))*(vypart(i)))+&
                     &((vzpart(i))*(vzpart(i)))
         v2c_2= v2c_2+ ((vxpart(i)-vxbar)*(vxpart(i)-vxbar))+&
                      &(vypart(i)*vypart(i))+ (vzpart(i)*vzpart(i))
     enddo
     temprof(layercount)= temprof(layercount)+ (v2c_1/dfloat(layerpart))
     temprof2(layercount)= temprof2(layercount)+ (v2c_2/dfloat(layerpart))
     temprofx(layercount)= temprofx(layercount)+ (v2c_x/dfloat(layerpart))
     temprofy(layercount)= temprofy(layercount)+ (v2c_y/dfloat(layerpart))
     temprofz(layercount)= temprofz(layercount)+ (v2c_z/dfloat(layerpart))
    endif
   layercount= layercount+1
layerpos= layerpos+ layerbin; lowlim= lowlim+ layerbin; uplim= uplim+ layerbin
enddo
tempmetercalls2= tempmetercalls2+ 1
tempmetercalls= tempmetercalls+ 1

end subroutine tempmeter

!======================checks_if_the_system_acheived_steady_state======================14

subroutine steadystate_determiner
use md_para
implicit none
integer:: flag
real*8 :: maxvel(1), minvel(1)

if(iter_steadystate .lt. niter) print*, "Already determined!"

flag=1
steadycalls= steadycalls+ 1
maxvel= maxloc(xveloprof)
minvel= minloc(xveloprof)
vdev= (maxvel(1)-minvel(1))*vdev100/(100.0d0*2.0d0*dfloat(tempmetercalls))

do i= 1, layernum
    if(tempmetercalls .ne. 0) curr_vx(i)= xveloprof(i)/dfloat(tempmetercalls)
     if(abs(prev_vx(i)-curr_vx(i)) .gt. vdev) then
        flag=0
    endif
enddo
xveloprof= 0.0d0
tempmetercalls= 0

filename= trim(den_str)//'_prev_curr_vx_'//trim(label)//'.dat'
 open(68, file=filename, status='unknown', form='formatted')
do i= 1, layernum
    write(68, '(1I6,2F10.4)') i, prev_vx(i), curr_vx(i)
enddo
```

```
 close(68)

if(flag==1) then
    iter_steadystate= iter
    print*, "Reached steadystate at: ", iter
    xveloprof2= 0.0d0; tempmetercalls2= 0;
    densityprof= 0.0d0; temprof= 0.0d0; temprof2= 0.0d0
    yveloprof= 0.0d0; zveloprof= 0.0d0; layertemp= 0.0d0
else
    prev_vx= curr_vx
endif

end subroutine steadystate_determiner

!===================determines_the_best_linear_fit_for_the_xveloprof===================15
subroutine linearfitter
use md_para
implicit none
real*8 :: sumx, sumx2, sumxy, sumy
sumx= 0.0d0; sumx2= 0.0d0; sumxy= 0.d0 ; sumy= 0.0d0

do i= 1, layernum
    sumy= sumy+xveloprof2(i)
    sumxy= sumxy+(xveloprof2(i)*dfloat(i))
    sumx2= sumx2+(dfloat(i)*dfloat(i))
    sumx= sumx+dfloat(i)
enddo

a= (sumxy-(sumx*sumy/dfloat(layernum)))/(sumx2-(sumx*sumx/dfloat(layernum)))
b= (sumy-(a*sumx))/dfloat(layernum)

end subroutine linearfitter

!=======================assigning_neighbours_for_each_box==============================16

subroutine boxnghbours
use md_para
implicit none
integer :: bx, by, bz, tempbx, tempby, tempbz


do i= 1, nbox !assigning_neighbours_to_each_box
    bz= (i-1)/(boxsidey*boxsidex)
    by= (modulo(i-1,boxsidey*boxsidex)/boxsidex)
    bx= (modulo(i-1,boxsidex))

    m=0
    do j= 1, 3
        do k= 1, 3
            do n= 1, 3
                tempbx= (j-2)*1+ bx
                if(tempbx .ge. boxsidex) tempbx= tempbx- boxsidex
                if(tempbx .lt. 0) tempbx= tempbx+ boxsidex
                tempby= (k-2)*1+ by
                if(tempby .ge. boxsidey) tempby= tempby- boxsidey
                if(tempby .lt. 0) tempby= tempby+ boxsidey
                tempbz= (n-2)*1+ bz
                if(tempbz .ge. boxsidez) cycle
                if(tempbz .lt. 0) cycle

                temp = (tempbz*llx*lly/(box_size**2))+(tempby*llx/box_size)+(tempbx)+ 1
                if(temp .ne. i) then
```

```
                        m=m+1
                        boxnghlist(m,i)= temp
                endif
            enddo
        enddo
    enddo
    boxnghbournum(i)= m
enddo   !i= 1, nbox

end subroutine boxnghbours

!=======================updating_the_neighbourlist_cells==============================17

subroutine update_nghlist
use md_para
implicit none
integer *8 :: p, ibox, ktemp, nmonotemp

nghlist=0
nghbours=0
n_box_mono=0
box_mono=0

do i= 1, ntot
    tmpx= pos(3*i-2)/box_size
    tmpy= pos(3*i-1)/box_size
    tmpz= pos(3*i  )/box_size
    temp = (tmpz*llx*lly/(box_size*box_size))+(tmpy*llx/box_size)+tmpx+1

    nmonotemp= n_box_mono(temp)
    nmonotemp= nmonotemp+ 1
    if(nmonotemp>maxboxcontent) print*, "Err: The max box content exceeded!", temp, i, nmonotemp
    box_mono(nmonotemp,temp)= i
    n_box_mono(temp)= nmonotemp
enddo

do i= 1, npart
    p=0
    tmpx= pos(3*i-2)/box_size
    tmpy= pos(3*i-1)/box_size
    tmpz= pos(3*i  )/box_size
    temp = (tmpz*llx*lly/(box_size*box_size))+(tmpy*llx/box_size)+tmpx+1

    do j= 1, n_box_mono(temp)
        dummy= box_mono(j,temp)
        if(dummy .gt. i) then   !all nghbr dummy less than i wud have i as nghbr
            delx= abs(pos(3*i-2)-pos(3*dummy-2)); if (delx >= llxby2) delx= llx-delx
            dely= abs(pos(3*i-1)-pos(3*dummy-1)); if (dely >= llyby2) dely= lly-dely
            delz= abs(pos(3*i  )-pos(3*dummy  ))
            dist= (delx*delx)+(dely*dely)+(delz*delz)

            if(dist<r_ngh2) then
                p= p+1
                if(p>maxnghbours) print*, "Err: maxnghbours exceeded!"
                nghlist(p,i)= dummy
            endif
        endif
    enddo

    do j= 1, boxnghbournum(temp)
        ibox= boxnghlist(j,temp)
        dummy= n_box_mono(ibox)
```

```
        if(dummy>0) then
            do k= 1, dummy
                ktemp= box_mono(k,ibox)
                if(ktemp .gt. i) then
                    delx= abs(pos(3*i-2)-pos(3*ktemp-2)); if (delx >= llxby2) delx= llx-delx
                    dely= abs(pos(3*i-1)-pos(3*ktemp-1)); if (dely >= llyby2) dely= lly-dely
                    delz= abs(pos(3*i  )-pos(3*ktemp  ))
                    dist= (delx*delx)+(dely*dely)+(delz*delz)

                    if(dist<r_ngh2) then
                        p=p+1
                        if(p>maxnghbours) print*, "Err: maxnghbours exceeded!"
                        nghlist(p,i)= ktemp
                    endif
                endif
            enddo
        endif
    enddo
    nghbours(i)= p
enddo !i= 1, npart

end subroutine update_nghlist

!=======================upadating_the_neighbourlist_basic==============================18

subroutine update_nghlist_basic
use md_para
implicit none
integer *8 :: c, nmonotemp

nghlist=0
nghbours=0
n_box_mono=0
box_mono=0

do i= 1, ntot
    tmpx= pos(3*i-2)/box_size
    tmpy= pos(3*i-1)/box_size
    tmpz= pos(3*i  )/box_size
    temp = (tmpz*llx*lly/(box_size*box_size))+(tmpy*llx/box_size)+tmpx+1

    nmonotemp= n_box_mono(temp)
    nmonotemp= nmonotemp+ 1
    if(nmonotemp>maxboxcontent) print*, "Err: The max box content exceeded!", temp, i, n&
    &monotemp
    box_mono(nmonotemp,temp)= i
    n_box_mono(temp)= nmonotemp
enddo

do i= 1, npart
    c=0
    do j= i+1, ntot
        delx= abs(pos(3*i-2)-pos(3*j-2)); if (delx >= llxby2) delx= llx-delx
        dely= abs(pos(3*i-1)-pos(3*j-1)); if (dely >= llyby2) dely= lly-dely
        delz= abs(pos(3*i  )-pos(3*j  ))

        dist= (delx**2)+(dely**2)+(delz**2)
        if(dist<r_ngh2) then
            c= c+1
            nghlist(c,i)= j
        endif
    enddo
```

```
    nghbours(i)= c
enddo

end subroutine update_nghlist_basic

!============================counting_the_clusters_and_their_size======================19

subroutine clust_counter
use md_para
implicit none

ifvisited= 0
numofbonds= 0

do k= 1, npart
    clustersize= 0
    if(ifvisited(k) .eq. 0) then
        clustersize=1
        call counter(k)
    endif
    if(clustersize>maxclustsize) print*, "Err: maxclustsize exceeded!"
    clust_count(clustersize)= clust_count(clustersize) +1
enddo

end subroutine clust_counter

!===========================counter===================================================20

recursive subroutine counter(l)
use md_para
implicit none
integer :: l, g, h

ifvisited(l)=1

do h= 1, nghbours(l)
    g= nghlist(h,l)
    if(ifvisited(g) .eq. 0) then
        delx= abs(pos(3*l-2)-pos(3*g-2)); if (delx >= llxby2) delx= llx-delx
        dely= abs(pos(3*l-1)-pos(3*g-1)); if (dely >= llyby2) dely= lly-dely
        delz= abs(pos(3*l  )-pos(3*g  ))
        dist= (delx**2)+(dely**2)+(delz**2)

        if(dist < bond_l2) then
            clustersize= clustersize+ 1
            numofbonds(l)= numofbonds(l)+1
            numofbonds(g)= numofbonds(g)+1
            if(numofbonds(l)>2) branchnum= branchnum+1
            if(numofbonds(g)>2) branchnum= branchnum+1
            call counter(g)
        endif
    endif
enddo

end subroutine counter

!====================writing_final_positions===========================================21

subroutine poscopy
use md_para
implicit none
```

```
filename= trim(den_str)//'_'//trim(label)//'.xyz'
open (91,file=filename,status='unknown', form='formatted')

write (91, '(I5)') ntot
write (91, '(1A7)') 'Chained'
do i= 1, npart
  if(i/10000 >= 1) then
    write(91, '(1A,1I5,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
    else
    if(i/1000 >= 1) then
      write(91, '(1A,1I4,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
      else
      if(i/100 >= 1) then
        write(91, '(1A,1I3,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        else
        if(i/10 >= 1) then
          write(91, '(1A,1I2,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
          else
            write(91, '(1A,1I1,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        endif
      endif
    endif
  endif
enddo

do i= npart+1, ntot
  if(i/10000 >= 1) then
    write(91, '(1A,1I5,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
    else
    if(i/1000 >= 1) then
      write(91, '(1A,1I4,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
      else
      if(i/100 >= 1) then
        write(91, '(1A,1I3,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        else
        if(i/10 >= 1) then
          write(91, '(1A,1I2,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
          else
            write(91, '(1A,1I1,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        endif
      endif
    endif
  endif
enddo

 close (91)
end subroutine poscopy

!===================writing_final_positions2==========================================21

subroutine poscopy2
use md_para
implicit none

filename= trim(den_str)//'_'//trim(label)//'_day_'//trim(daycount_str)//'.xyz'
open (91,file=filename,status='unknown', form='formatted')

write (91, '(I5)') ntot
write (91, '(1A7)') 'Chained'
do i= 1, npart
  if(i/10000 >= 1) then
    write(91, '(1A,1I5,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
```

```
    else
    if(i/1000 >= 1) then
      write(91, '(1A,1I4,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
      else
      if(i/100 >= 1) then
        write(91, '(1A,1I3,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        else
        if(i/10 >= 1) then
          write(91, '(1A,1I2,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
          else
            write(91, '(1A,1I1,3F20.16)') 'A', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        endif
      endif
    endif
  endif
enddo

do i= npart+1, ntot
  if(i/10000 >= 1) then
    write(91, '(1A,1I5,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
    else
    if(i/1000 >= 1) then
      write(91, '(1A,1I4,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
      else
      if(i/100 >= 1) then
        write(91, '(1A,1I3,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        else
        if(i/10 >= 1) then
          write(91, '(1A,1I2,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
          else
            write(91, '(1A,1I1,3F20.16)') 'W', i, pos(3*i-2), pos(3*i-1), pos(3*i)
        endif
      endif
    endif
  endif
enddo

 close (91)
end subroutine poscopy2

!=================writing_run_details===============================================22

subroutine writerundetails
use md_para
implicit none

!writes the final velocity  data to file
filename= trim(den_str)//'_velocities_'//trim(label)//'.dat'
 open(84, file=filename, status='unknown', form='formatted')
do i= 1, npart
write(84, '(1I5,3F20.16)') i, vel(3*i-2), vel(3*i-1), vel(3*i)
enddo
 close(85)

!writes the temperature data of layers to file
filename= trim(den_str)//'_KEgrad_'//trim(label)//'.dat'
 open(85, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(85, '(1I31F10.4)') i, layertemp(i)
enddo
 close(85)
```

```
!writes the avg x velocity data of layers to file
filename= trim(den_str)//'_xveloprof_'//trim(label)//'.dat'
 open(86, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(86, '(1I31F10.4)') i, xveloprof2(i)
enddo
 close(86)

!writes the avg y velocity data of layers to file
filename= trim(den_str)//'_yveloprof_'//trim(label)//'.dat'
 open(87, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(87, '(1I31F10.4)') i, yveloprof(i)
enddo
 close(87)

!writes the avg z velocity data of layers to file
filename= trim(den_str)//'_zveloprof_'//trim(label)//'.dat'
 open(88, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(88, '(1I31F10.4)') i, zveloprof(i)
enddo
 close(88)

!writes the density data of layers to file
filename= trim(den_str)//'_densityprof_'//trim(label)//'.dat'
 open(89, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(89, '(1I31F10.4)') i, densityprof(i)
enddo
 close(89)

!writes the temperature data of layers to file
filename= trim(den_str)//'_tempprof_'//trim(label)//'.dat'
 open(90, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(90, '(1I31F10.4)') i, temprof(i)
enddo
 close(90)

!writes the temperature-of-2nd-kind data of layers to file
filename= trim(den_str)//'_tempprof2_'//trim(label)//'.dat'
 open(91, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(91, '(1I31F10.4)') i, temprof2(i)
enddo
 close(91)

filename= trim(den_str)//'_para&res_'//trim(label)//'.dat'
open(80, file=filename, status='unknown', form='formatted')

write(80, '(1A28,1F10.4)')'density                 :', den
write(80, '(1A28,1I10)')  'particles               :', ntot
write(80, '(1A28,1I10)')  'free particles          :', npart
write(80, '(1A28,1I10)')  'wall particles          :', nwall
write(80, '(1A28,1F10.2)')'avg. cluster size       :', dfloat(avg_clustsize)/dfloat(tot_&
&clust)
write(80, '(1A28,1F10.2)')'avg. no: of branching   :', dfloat(branchnum)/dfloat((niter-i&
&ter_equil)/count_intervel)
write(80, '(1A28,1I10)'  )'steady state reached at :', iter_steadystate
write(80, '(1A28,1F10.6)')'slip length             :', sliplen
write(80, '(1A28,1F10.6)')'slip length 2           :', sliplen2
```

```
write(80, '(1A28,1F10.4)')'vxprof linearfit coeff a:', a
write(80, '(1A28,1F10.4)')'vxprof linearfit coeff b:', b
write(80, '(1A38)')      'PARAMETERS OF POTENTIAL==========>'
write(80, '(1A28,1F10.4)')'epsilon                 :', eps
write(80, '(1A28,1F10.4)')'alpha                   :', alpha
write(80, '(1A28,1F10.4)')'beta                    :', beta
write(80, '(1A28,1F10.4)')'eta                     :', eta
write(80, '(1A28,1F10.4)')'r0                      :', r0
write(80, '(1A28,1F10.4)')'B^2                     :', b2
write(80, '(1A28,1F10.4)')'epsilon2                :', eps2
write(80, '(1A28,1F10.4)')'wall velocity           :', vwall
write(80, '(1A38)')      'OTHERS===========================>'
write(80, '(1A28,1I10)')  'no: of iterations       :', niter
write(80, '(1A28,1I10)')  'equilibrium iteration   :', iter_equil
write(80, '(1A28,1F10.4)')'bond length             :', bond_l
write(80, '(1A28,1F10.4)')'range of potential      :', rc
write(80, '(1A28,1F10.4)')'range of potential_wall :', rc_wall
write(80, '(1A28,1F10.4)')'timestep                :', dt
write(80, '(1A28,1F10.4)')'temperature             :', kbt
write(80, '(1A28,1F10.4)')'runtime in LJ time units:', ljruntime
write(80, '(1A28,1F10.4)')'wall_dx                 :', wall_dx
write(80, '(1A28,1F10.4)')'shear rate              :', shearrate
write(80, '(1A28,1F10.4)')'layerbin                :', layerbin
write(80, '(1A28,1F10.4)')'xvelo tolernce(v_dev %) :', vdev100
write(80, '(1A28,1I10)')  'velo. sampling intervel :', sample_intervel
write(80, '(1A28,1I10)')  'Steadystate check intrvl:', check_intervel
write(80, '(1A28,1F10.4)')'neighbourhood radius    :', r_ngh
write(80, '(1A7,1I3,1A11,1I3,1A11,1I3)')'   lx: ', lx, ' ly: ', ly, ' lz: ', lz
write(80, '(1A20,1I4,1A5,1F5.2,1A4)')'code runtime    :', &
&int((ends-start)/60),' min:',mod((ends- start),60.0),' sec'

 close(80)

end subroutine writerundetails

!=================writing_run_details2===============================================22

subroutine writerundetails2
use md_para
implicit none

!writes the final velocity  data to file
filename= trim(den_str)//'_velocities_'//trim(label)//'_day_'//trim(daycount_str)//'.dat'
 open(84, file=filename, status='unknown', form='formatted')
do i= 1, npart
write(84, '(1I5,3F20.16)') i, vel(3*i-2), vel(3*i-1), vel(3*i)
enddo
 close(85)

!writes the temperature data of layers to file
filename= trim(den_str)//'_KEgrad_'//trim(label)//'_day_'//trim(daycount_str)//'.dat'
 open(85, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(85, '(1I31F10.4)') i, layertemp(i)
enddo
 close(85)

!writes the avg x velocity data of layers to file
filename= trim(den_str)//'_xveloprof_'//trim(label)//'_day'//trim(daycount_str)//'.dat'
 open(86, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(86, '(1I31F10.4)') i, xveloprof2(i)
```

```
enddo
 close(86)

!writes the avg y velocity data of layers to file
filename= trim(den_str)//'_yveloprof_'//trim(label)//'_day_'//trim(daycount_str)//'.dat'
 open(87, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(87, '(1I31F10.4)') i, yveloprof(i)
!iter_steadystate)/sample_intervel))
enddo
 close(87)

!writes the avg z velocity data of layers to file
filename= trim(den_str)//'_zveloprof_'//trim(label)//'_day_'//trim(daycount_str)//'.dat'
 open(88, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(88, '(1I31F10.4)') i, zveloprof(i)
!iter_steadystate)/sample_intervel))
enddo
 close(88)

!writes the density data of layers to file
filename= trim(den_str)//'_densityprof_'//trim(label)//'_day_'//trim(daycount_str)//'.dat'
 open(89, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(89, '(1I31F10.4)') i, densityprof(i)
enddo
 close(89)

!writes the temperature data of layers to file
filename= trim(den_str)//'_tempprof_'//trim(label)//'_day_'//trim(daycount_str)//'.dat'
 open(90, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(90, '(1I31F10.4)') i, temprof(i)
enddo
 close(90)

!writes the temperature-of-2nd-kind data of layers to file
filename= trim(den_str)//'_tempprof2_'//trim(label)//'_day_'//trim(daycount_str)//'.dat'
 open(91, file=filename, status='unknown', form='formatted')
do i= 1, int(llz/layerbin)
write(91, '(1I31F10.4)') i, temprof2(i)
enddo
 close(91)

end subroutine writerundetails2

!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FUNCTION ran1(IDUM)
implicit none

!  RAN1 returns a unifom random deviate on the interval [0,1]
! ------------------------------------------------------------------
!
INTEGER :: IDUM
REAL*8 :: RAN2,ran1
integer,parameter :: IM1=2147483563,IM2=2147483399
integer,parameter :: IMM1=IM1-1,                                &
IA1=40014,IA2=40692,IQ1=53668,IQ2=52774,IR1=12211,IR2=3791, &
           NTAB=32
        integer,parameter :: NDIV=1+IMM1/NTAB
         real*8,parameter :: EPS=1.2e-7,RNMX=1.-EPS,AM=1./IM1
```

```
        INTEGER :: IDUM2,J,K,IV(NTAB),IY
        DATA IDUM2/123456789/, iv/NTAB*0/, iy/0/
        IF (IDUM.LE.0) THEN
          IDUM=MAX(-IDUM,1)
          IDUM2=IDUM
          DO  J=NTAB+8,1,-1
             K=IDUM/IQ1
             IDUM=IA1*(IDUM-K*IQ1)-K*IR1
             IF (IDUM.LT.0) IDUM=IDUM+IM1
             IF (J.LE.NTAB) IV(J)=IDUM
          end do
          IY=IV(1)
        ENDIF
        K=IDUM/IQ1
        IDUM=IA1*(IDUM-K*IQ1)-K*IR1
        IF (IDUM.LT.0) IDUM=IDUM+IM1
        K=IDUM2/IQ2
        IDUM2=IA2*(IDUM2-K*IQ2)-K*IR2
        IF (IDUM2.LT.0) IDUM2=IDUM2+IM2
        J=1+IY/NDIV
        IY=IV(J)-IDUM2
        IV(J)=IDUM
        IF(IY.LT.1)IY=IY+IMM1
        RAN1=MIN(AM*IY,RNMX)

        END function ran1

!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```