Molecular Dynamics simulation of self-healing polymers

Andrew Gibbs

March 2011

Abstract

This project aims to introduce and discuss the theory and methods behind Molecular Dynamics simulations of polymers. The underlying aim is to to create a simulation protocol to study the network formation of polymers with self-healing properties. Initially, basic theory is introduced - sufficient to model a single polymer in solution. Following this, the methods for the self-healing model are explained. Finally the cluster formation, functionality and stress modulus of these self-healing polymer networks are analysed.

The following conventions apply throughout:

Differentiation of f with respect to time: $\dot{f} = \dfrac{df}{dt}$

Position is defined by: $\mathbf{r} = (x, y, z)^T$

Velocity is defined by: $\mathbf{v} = \dot{\mathbf{r}}$

Acceleration is defined by: $\mathbf{a} = \dot{\mathbf{v}} = \ddot{\mathbf{r}}$

An ensemble average of a quantity f is represented by: $\langle f \rangle$

Contents

1 Introduction

The concept of a self-healing material is similar to that of a biological organism. For instance, when animal tissue becomes slightly damaged, its body will react to this. The tissue then heals until it is, as far as the animal is concerned, as good as new. Self-healing materials are a man-made attempt at achieving this effect. One example of a self-healing material described by White et al. (2001)[1] is designed for repairing cracks in brittle material such as ceramics and glasses. Microcapsules containing a healing agent are embedded into the structure of the material during synthesis, along with a catalyst capable of polymerizing the healing agent. As the crack grows through the material, it will eventually break open one of these microcapsules, allowing the healing agent to escape, spreading through the crack until it reaches the polymerization catalyst. It will then fill the crack, until the crack has been healed. The drawback with this method is that the healing agent will eventually run out, so each area of the material can only be healed a limited number of times.

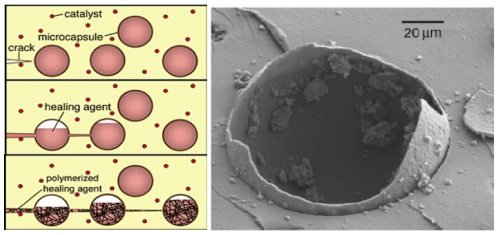


Figure 1: Diagram of the microcapsule self-healing process (left),with an image of a ruptured microcapsule (right). Photos produced by White et al. (2001)[1]

1.1 Self-healing polymers

Due to the limited number of repairs possible in the microcapsule method, Cordier et al. (2008)[2] have synthesized polymers which form chains and cross-links through hydrogen bonding. These materials are elasticized, but unlike rubber if a piece of the material is cut in half, under certain conditions, the pieces can be simply held together and 'healed'. This can take place at room temperature and in some cases, only takes around 15 minutes. After healing, they will still exhibit the same elasticity as before the fracture. It is important that the healing starts soon after the cut, as the hydrogen atoms near the gap will begin to form hydrogen bonds with new partners in their

half of the material immediately. This will then reduce the number of free atoms available for hydrogen bonding when the pieces are held together at a later stage.

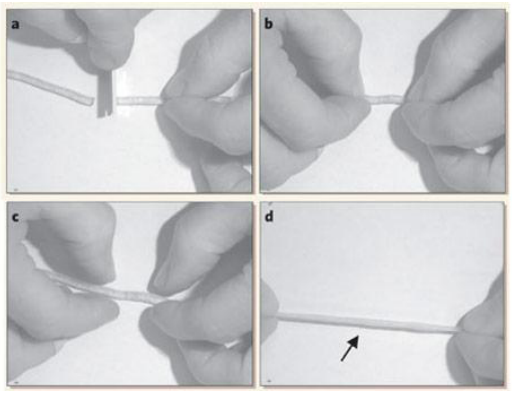


Figure 2: The cut (a), the join (b), the healing (c) and the preserved elasticity (d) of the material. Photos produced by Cordier et al. (2008)[2]

At the University of Reading, Burattini et al. (2010)[3] have recently been experimenting with self-healing polymers, where the healing process occurs as a result of π-π stacking. This is due to π-electron-deficient diimide groups and π-electron-rich pyrenyl units, although hydrogen bonding does still occur. These materials will exhibit 95% of their tensile modulus after healing.

1.2 Project aims

Initially, this project aims to gain a firm understanding of basic polymer physics (§2), and to introduce the basic ideas behind Molecular Dynamics (§3), sufficient to model a single polymer in solution (§4). The development and the theory behind this model serve as an excellent introduction to molecular dynamics and polymer physics. The second (and main) aim is to create a Molecular Dynamics simulation which can be parameterized to model the self-healing polymers synthesized by the groups of both Cordier and Burattini (§6). In particular, we are interested in the stress modulus, and the network formation of the polymers in each case - ideally the results of the simulation of Burattini's polymers will offer greater insight into the results of the *real* experiment. Each simulation is described as a **coarse-grained** model (meaning that certain chemical information is ignored) with no explicit solvent; treating the individual monomers in the chain as spheres, using classical mechanics.

2 Polymer dynamics

In this section the initial aim is to introduce the fundamental ideas behind classical and statistical mechanics, before applying these principles to polymers - with the underlying aim of eventually incorporating this information into a simulation - the details of *how* are described in §3.

2.1 Lagrangian and Hamiltonian mechanics

Before the energies are discussed in detail, it is useful to make some strong general statements that can be applied to any system. For a system of kinetic energy K and potential energy U, Allen & Tildesley (1987)[4] defines the Lagrangian as

$$\mathcal{L}(\mathbf{w}, \mathbf{v}) = K - U, \tag{1}$$

where $\mathbf{w}$[1] and $\mathbf{v}$ are vectors of generalized coordinates and velocities respectively, that are used to describe the system. The Hamiltonian $\mathcal{H}$ is defined as

$$\mathcal{H}(\mathbf{w}, \mathbf{p}) = \sum_i p_i \dot{w}_i - \mathcal{L},$$

where $\mathbf{p}$ is a vector of generalized momenta associated to the system; i denotes the individual components of the vectors $\mathbf{p}$ and $\mathbf{w}$. $\mathcal{L}$ must be redefined to be consistent with the variables used to describe $\mathcal{H}$, for example by substituting $v_i = p_i/m_i$. The Lagrangian equation of motion, for a generalized coordinate w_k is given by

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{w}_k}\right) - \frac{\partial \mathcal{L}}{\partial w_k} = 0, \tag{2}$$

whilst the Hamiltonian equations of motion are defined as

$$\dot{\mathbf{p}} = -\frac{\partial \mathcal{H}}{\partial \mathbf{w}}, \qquad \dot{\mathbf{w}} = \frac{\partial \mathcal{H}}{\partial \mathbf{p}}.$$

2.2 The partition function

In both classical and quantum mechanical canonical ensembles, to find the probability of a system being in any given state, it is common to use the result of the partition function Z, which weights every possible microstate

[1] In literature the generalized coordinate is typically described using q_i instead of w_k, whilst it has been defined differently here to avoid later confusion when dealing with the charge on a particle, also denoted q_i in literature.

$\mathcal{H}_s$ in accordance with its Boltzmann factor. This is given by Frenkel & Smit (1996)[5], as

$$Z = \frac{1}{N!h^{3N}} \int e^{-\frac{1}{k_B \mathcal{T}} \mathcal{H}_s(\mathbf{p}_1,...,\mathbf{p}_N,\mathbf{w}_1,...,\mathbf{w}_N)} \mathrm{d}\mathbf{p}_1...\mathrm{d}\mathbf{p}_N \mathrm{d}\mathbf{w}_1...\mathrm{d}\mathbf{w}_N,$$

where k_B is the Boltzmann constant, $\mathcal{T}$ is the absolute temperature and h is Planck's constant. In quantum mechanical systems, this function takes the form of a summation, as there are only finitely many energy states possible. The probability that a system is in any given state with energy $\mathcal{H}_s$ can be obtained from

$$P(\mathcal{H}_s) = \frac{1}{Z} e^{-\frac{1}{k_B \mathcal{T}} \mathcal{H}_s},$$

where $\frac{1}{Z}$ acts only as a normalisation constant. Marzari, (2005)[6] explains that any average quantity $\langle A \rangle$ can also be computed via the relationship

$$\langle A \rangle = \frac{\int A(\mathbf{p}_1,...,\mathbf{p}_N,\mathbf{w}_1,...,\mathbf{w}_N) e^{-\frac{1}{k_B \mathcal{T}} \mathcal{H}} \mathrm{d}\mathbf{p}_1...\mathrm{d}\mathbf{p}_N \mathrm{d}\mathbf{w}_1...\mathrm{d}\mathbf{w}_N}{\int e^{-\frac{1}{k_B \mathcal{T}} \mathcal{H}} \mathrm{d}\mathbf{p}_1...\mathrm{d}\mathbf{p}_N \mathrm{d}\mathbf{w}_1...\mathrm{d}\mathbf{w}_N}.$$

Based on the dimensional abundance of many systems, computing $\langle A \rangle$ using the above expression would be very difficult - if not impossible. Luckily, due to the numerical nature of computer simulations, we are in the convenient position of being able to measure and record any quantity of our system, whenever we please - an asset absent of *real* experiments. After allowing sufficient time T_{eq} for equilibrium, we can instead appeal to the relationship

$$\langle A(t) \rangle = \lim_{T \to \infty} \frac{1}{T/\delta t} \sum_{k=0}^{T/\delta t} A(T_{eq} + k\delta t),$$

where δt represents the time between recording data, to be averaged over time T. The larger the sample of data T, the more reliable the statistical results become. It is also common to take the time gap between measurements of quantities δt to be around 10 or 100, as measuring every time step will just lead to repeated information (on top of the increased computation cost).

2.3 The Lagrangian of a polymer system

When modeling polymers, there are useful assumptions which can be made. In a system of N particles, each free to move in 3 dimensions, $3N$ generalized coordinates will be required to construct the Lagrangian. However, due to the symmetry of x, y and z in the system considered here, we can simply

choose x_i as our generalized coordinate w_k for now, as equivalent results for y_i, z_i will follow by symmetry. The total kinetic energy of the system is described by

$$K = \sum_{i=1}^{N} \frac{1}{2} m_i \left|\dot{\mathbf{r}}_i\right|^2 = \frac{1}{2} m \sum_{i=1}^{N} \left|\dot{\mathbf{r}}_i\right|^2 \tag{3}$$

where $\frac{1}{2}m$ has been taken outside of the summation, as each monomer (in this system) has equal mass $m_i = m$. This definition can easily be switched to Hamiltonian coordinates using the change of variables $m_i \left|\dot{\mathbf{r}}_i\right|^2 = \frac{1}{m_i} \left|\mathbf{p}_i\right|^2$. The average kinetic energy (in 3 dimensions) is related to the temperature $\mathcal{T}$ of the system via the relationship

$$\langle K \rangle = \frac{3}{2} k_B \mathcal{T} = \frac{1}{2} m \sum_{i=1}^{N} \left|\dot{\mathbf{r}}_i\right|^2, \tag{4}$$

(Frenkel & Smit 1996)[5]. In a simulation of constant temperature, it is common to rescale the kinetic energies of each individual particle to maintain this temperature in the presence of subtle numerical errors. All of the potential energies considered for our polymer systems depend only on position coordinates $\mathbf{r}$, whilst K depends only on $\dot{\mathbf{r}}_i$, so we can make the following two statements:

$$\frac{\partial K}{\partial x_i} = 0, \qquad \frac{\partial U}{\partial \dot{x}_i} = 0. \tag{5}$$

Combining (1) and (2) gives

$$\begin{aligned} 0 &= \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathcal{L}}{\partial \dot{x}_i}\right) - \frac{\partial \mathcal{L}}{\partial x_i} \\ &= \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial (K-U)}{\partial \dot{x}_i}\right) - \frac{\partial (K-U)}{\partial x_i} \\ &= \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial K}{\partial \dot{x}_i} - \frac{\partial U}{\partial \dot{x}_i}\right) - \frac{\partial K}{\partial x_i} + \frac{\partial U}{\partial x_i}. \end{aligned}$$

This equation of motion can now be simplified using (5),

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial K}{\partial \dot{x}_i}\right) + \frac{\partial U}{\partial x_i} = 0. \tag{6}$$

We proceed to derive the corresponding force coordinate F_i on a single par-

ticle i, using 6 and (3),

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial K}{\partial \dot{x}_i}\right) &= \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial}{\partial \dot{x}_i}\left(\frac{1}{2}m\sum_{j=1}^{N}\dot{x_j}^2\right)\right) \\
&= \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{1}{2}m\cdot 2\dot{x}_i\right) \\
&= m\ddot{x}_i \\
\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial K}{\partial \dot{x}_i}\right) &= F_i,
\end{aligned}$$

where the final step is just Newton's second law. Substituting into equation (6) and re-arranging yields

$$F_i = -\frac{\partial U}{\partial x_i}.$$

Written back in 3-dimensional form, this is simply

$$\mathbf{F}_i = -\underset{\mathbf{r}_i}{\nabla} U. \tag{7}$$

The above expression can be used to derive the force on a single particle i, based on the potential U of the system. Furthermore, (7) can be used to provide a relationship which is useful from the perspective of molecular simulations. Since the potential energy depends only on particle positions $\mathbf{r}$, it is straightforward to show that the total energy $U+K$ of *any* such system is constant, as shown (in the 1 dimensional case) by Marzari (2005)[6]. We differentiate the total energy with respect to time,

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\left(U+K\right) &= \frac{\mathrm{d}}{dt}U + \frac{d}{dt}\left(\frac{1}{2}\sum_{i=1}^{N} m_i\dot{\mathbf{r}}_i^2\right) \\
&= \sum_{i=1}^{N}\frac{d\mathbf{r}_i}{dt}\cdot\frac{\partial}{\partial \mathbf{r}_i}U + \sum_{i=1}^{N} m_i\dot{\mathbf{r}}_i\cdot\ddot{\mathbf{r}}_i \\
&= \sum_{i=1}^{N}\left[-\dot{\mathbf{r}}_i\cdot\mathbf{F}_i + m_i\dot{\mathbf{r}}_i\cdot\ddot{\mathbf{r}}_i\right] \\
&= 0,
\end{aligned} \tag{8}$$

where the result from (7) is used in the final step, proving that the total energy of *any* closed, pair-potential system is constant. This is useful for monitoring the error in a simulation of an isolated system; any significant fluctuations in total energy may signify error.

2.4 Langevin dynamics

When molecules are 'in solution', they will be physically affected by collisions with relatively smaller particles (e.g water molecules) of the solution that they are in - described as **Brownian motion**. Rather than additionally modelling large numbers of comparatively tiny molecules, these small forces $\mathbf{F}_R$ can be treated as random and we can assume that the probability of this force being in any direction is the same; $\langle \mathbf{F}_R \rangle = \mathbf{0}$. It follows that the distribution of $\mathbf{F}_R$ is Gaussian - with variance β^2, and is modeled via a white noise function $W_i(t)$.

A particle moving in a direction is more likely to experience random kicks in the opposing direction. This can be considered a frictional force $-\gamma \dot{\mathbf{r}}_i$, against the direction of motion. In summary, the Langevin equation of motion takes the form of Newton's second law, minus a frictional term, plus a random white noise. It is defined[2] by Kremer & Grest (1990)[7] as

$$F(\mathbf{r}_i) = -\frac{\partial U}{\partial \mathbf{r}_i} - \gamma m_i \dot{\mathbf{r}}_i + W_i(t). \tag{9}$$

The **fluctuation-dissipation theorem** is an important result, derived by Nyquist in 1928 and alternatively by Kubo in 1966. Felderhof (1978)[8] provides details of the derivations, but the a key result is that

$$\alpha^2 = 6k_B \mathcal{T} \gamma m_i$$

which can be used to link the relative amplitudes of the random and frictional terms in (9). In the limit of strong friction, it follows that

$$\gamma \dot{\mathbf{r}} m_i = \sqrt{6k_B \mathcal{T} \gamma m_i} W_i(t),$$

which relates the frictional term to the small random forces. This relationship should be maintained during simulation.

2.5 Potential of the polymer system

Here, the potential energies used in our polymer simulations will be introduced and briefly discussed. Aside from those explained here, there are other less significant potentials based on bond angles and conformation properties, but those are not used in these simulations. Each potential considered is a

[2]The Langevin equation of motion is actually defined with a sign error by Kremer & Grest (1990)[7], which has been corrected here.

pair-potential; a function of the distance between two particles. Due to this, it is useful to introduce the following notation:

$$\begin{aligned} \mathbf{r}_{ij} &= \mathbf{r}_i - \mathbf{r}_j, \\ r_{ij} &= |\mathbf{r}_i - \mathbf{r}_j| . \end{aligned}$$

Physically, $\mathbf{r}_{ij}$ represents the separation between particles i and j, whilst r_{ij} is the length of this separation. These definitions are used throughout.

2.5.1 Lennard-Jones potential

The Lennard-Jones (LJ) potential between two particles i and j is given by Frenkel & Smit (1996)[5] (and many other books) as

$$U_{LJ,ij} := 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right]$$

where σ is the particle diameter and ϵ represents the potential depth. It is clear that $r_{ij} = \sigma \Rightarrow U_{LJ} = 0$, which physically corresponds to zero energy when the particles are touching. $(\sigma/r_{ij})^{12}$ is commonly referred to as the *repulsive* part, describing Pauli repulsion, and dominates at shorter ranges. The repulsive part explodes as $r_{ij} \to 0^+$, as this represents overlapping electron orbitals, which (except when dealing with extreme cases such as nuclear fusion) is physically unrealistic. The *attractive* part $(\sigma/r_{ij})^6$ dominates at longer ranges, but becomes less significant as $r_{ij} \to \infty$. This part is theoretically justified, representing dispersion force between particles. Conversely, the repulsive part has no theoretical justification, but is a convenient approximation to close range repulsion in a computational sense. This is because the relationship $r^{12} = (r^6)^2$ can be used, to save calculating both parts from scratch (Allen & Tildesley 1987)[4].

Lennard-Jones is a pair potential, therefore the total LJ energy must consider every possible pair of monomers in the system. This results in $\frac{N(N-1)}{2}$ calculations for potential energy - although certain steps can be taken to reduce this (discussed in §3.2 and §5.2.1). As every particle acts on every other particle, it follows that the total Lennard-Jones energy is given by

$$U_{LJ} = 4\epsilon \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{11}$$

where we have summed over all particle pairs, and taken constants outside the summation. We can calculate the Lennard-Jones force acting on a single

particle i via the relationship (7)

$$\begin{aligned} \mathbf{F}_{i,LJ} &= -\nabla_{\mathbf{r}_i} U_{LJ} \\ &= 24\epsilon \sum_{\substack{j=1 \\ i \neq j}}^{N} \mathbf{r}_{ij} \left[\frac{2\sigma^{12}}{r_{ij}^{14}} - \frac{\sigma^6}{r_{ij}^8} \right]. \end{aligned} \tag{12}$$

The repulsive part of (12) ensures that there will be no significant overlap between particles in a system under U_{LJ}. Rubinstein & Colby (2003)[9] describes this as **excluded volume**, a property of **real chains**. Alternatively, **ideal chains** ignore excluded volume. These are described in detail by Rubinstein & Colby (2003)[9], and are a very useful model for predicting polymer behavior. There are many different ideal chain models, each of which allow particles to overlap and relax the constraint of excluded volume. All simulations discussed here will treat the polymers as real chains.

2.5.2 Entropic spring potential

Kremer & Grest (1990)[7] (amongst others) defines the entropic spring potential for bond energy between neighboring monomers i and j in the chain as

$$U_{Fene,ij} := -\frac{\kappa R_0}{2} \ln \left[1 - \left(\frac{r_{ij}}{R_0} \right)^2 \right],$$

where κ is the spring constant and R_0 is an upper bound on the bond length permitted by $U_{Fene,ij}$. FENE stands for 'finitely extensible nonlinear elastic'. In fact, due to its logarithmic nature the potential is always negative and increasing, thus the corresponding force is always attractive. It follows that without some additional form of repulsion such as Lennard-Jones potential present, the system will collapse as only attractive forces are present. Unlike LJ, for a linear chain only $N-1$ calculations will be required to compute the total energy, as only neighboring monomers are considered. We index such that the total FENE potential is given by

$$U_{Fene} = -\frac{\kappa R_0}{2} \sum_{i=1}^{N-1} \ln \left[1 - \left(\frac{r_{i,i+1}}{R_0} \right)^2 \right],$$

whilst the corresponding force on a single monomer i is given by

$$\begin{aligned}\mathbf{F}_{i,Fene} &= -\underset{\mathbf{r}_i}{\nabla} U_{Fene} \\ &= -\kappa \left[(\mathbf{r}_i - \mathbf{r}_{i+1}) \frac{1}{1 - \left(\frac{r_{i,i+1}}{R_0}\right)^2} + (\mathbf{r}_i - \mathbf{r}_{i-1}) \frac{1}{1 - \left(\frac{r_{i,i-1}}{R_0}\right)^2} \right] \end{aligned} \quad (13)$$

As most terms are killed off by the partial derivative - only neighboring monomers will directly affect the motion of i. However, each of these monomers is affected by its neighbours also, so each monomer in the chain affects each other monomer. This relationship is described as **telechelic** - where information can be passed between any two monomers in the chain.

2.5.3 Coulomb potential

Jackson (1975)[10] gives coulomb energy as

$$U_{c,ij} := -\frac{q_i q_j}{4\pi\varepsilon_0\varepsilon_r r_{ij}}$$

where q_i and q_j are the charge of particle i and j respectively. ε_r is the dielectric constant, representing the electrical permittivity of the solution, and ε_0 is the electrical permittivity in a vacuum. Again, we can describe the total coulomb potential of the system as

$$U_c = -\frac{1}{4\pi\varepsilon_0\varepsilon_r} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{q_i q_j}{r_{ij}}. \quad (14)$$

It follows that the force acting on a single particle i is given by

$$\begin{aligned}\mathbf{F}_{i,c} &= -\underset{\mathbf{r}_i}{\nabla} U_c \\ &= \frac{1}{4\pi\varepsilon_0\varepsilon_r} \sum_{\substack{j=1 \\ i\neq j}}^{N} (\mathbf{r}_i - \mathbf{r}_j) \frac{q_i q_j}{r_{ij}^3}. \end{aligned} \quad (15)$$

Clearly, if q_i and q_j are of the same sign, the force will be repulsive, zero for a neutral charge and attractive if the signs are opposite. When two charges have a separation r_{ij} such that their interaction energy is comparable to $k_B\mathcal{T}$, r_{ij} is referred to as the **Bjerrum length**. This can be formally defined as

$$l_B := \frac{|q_i q_j|}{4\pi\varepsilon_r\varepsilon_0 k_B\mathcal{T}}. \quad (16)$$

The value of this parameter l_B plays a significant role in charged simulations, as is discussed in §6.

3 Molecular Dynamics

Molecular dynamics is a computer simulation method, used to study dynamical behavior of particles on a microscopic scale to understand the macroscopic properties of a system, bridging the gap between experimental and theoretical science, whilst strictly falling into neither category. It has been in use since the 1950s, and has since become recognized as a powerful tool in understanding physical biological and chemical phenomena, which would be unapproachable through experimentation. It can predict how a new material will behave, prior to manufacture, or could predict how an existing material would react under conditions that would be very expensive to replicate in a laboratory. Conversely, rather than using known theories to predict results of experiments, it can test the validity of a new theory or model to simulate an experiment with known results (Frenkel & Smit 1996)[5]. Whilst there are other simulation methods available, the Molecular Dynamics method is well suited to what we are aiming to model.

The general form of any Molecular Dynamics simulation can be broken down into 4 sections, as follows:

1. Assign initial positions and velocities to a set of particles
2. Calculate the force acting on each particle
3. Use a numerical method to calculate positions and velocities at the next time step
4. Loop stages 2 and 3 until the desired results are obtained

As with any numerical method, the implementation of relevant mathematical theory onto a computational platform requires several refinements and alterations to be effective. For instance, computers can only manipulate *numbers*, hence equations for force calculation (stage 2) cannot be left in the general form enjoyed by the previous section (§2). The method of initialization (stage 1) and the force acting on each particle will depend on the type of system we are choosing to model, we make use of a simple method in §4.2, whilst a more advanced method is explained in §5.1. We turn our attention to stage 3.

3.1 Velocity Verlet

In most real life situations, it is difficult or impossible to find the global solution of an initial value problem based on a system of N differential equations. Instead we use a finite difference method and descretise the simulation time $[0, T_f]$ into $M + 1$ timesteps of length $\Delta t = T_f/M$, writing

$$[0, T_f] \rightarrow \{0, \Delta t, 2\Delta t, \ldots, M\Delta t\}\,.$$

There are a few variations of the Verlet integration, each of which are mathematically equivalent. The version used in these simulations is *Velocity Verlet*. Verlet boasts greater stability than the Euler method, and whilst there are more accurate (and naturally more complicated) methods available, the local error of $O(\Delta t^2)$ is sufficient for our needs. This method can be derived using a combination of the following expressions

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \Delta t\dot{\mathbf{r}}(t) + \frac{1}{2}\Delta t^2\ddot{\mathbf{r}}(t) + O(\Delta t^3),$$

$$\dot{\mathbf{r}}(t + \Delta t) = \dot{\mathbf{r}}(t) + \frac{1}{2}\Delta t\left[\ddot{\mathbf{r}}(t) + \ddot{\mathbf{r}}(t + \Delta t)\right] + O(\Delta t^2),$$

where the error can be calculated by Taylor expanding about $(t + \Delta t)$ (see for example, Frenkel & Smit (1996)[5]). Using the relationship $\ddot{\mathbf{r}} = \dot{\mathbf{v}} = \mathbf{a}$, we can derive new information after each time step Δt.

`Step 1 -` Calculate new positions based on current data:

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \Delta t\mathbf{v}(t) + \frac{1}{2}\Delta t^2\mathbf{a}(t).$$

`Step 2 -` Calculate half-step velocities based on current data:

$$\mathbf{v}(t + \frac{1}{2}\Delta t) = \mathbf{v}(t) + \frac{1}{2}\Delta t\mathbf{a}(t).$$

`Step 3 -` Calculate new acceleration, based on the force resultant of new position data:

$$\frac{1}{m}\mathbf{F}(\mathbf{r}(t + \Delta t)) = \mathbf{a}(t + \Delta t).$$

`Step 4 -` Calculate new velocity based on new acceleration and half step velocity data:

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t + \frac{1}{2}\Delta t) + \frac{1}{2}\Delta t\mathbf{a}(t + \Delta t).$$

Note that the calculation of $\mathbf{v}(t + \Delta t)$ in step 4 requires $\mathbf{a}(t + \Delta t)$, which is obtained in step 3. Hence if the force depends on velocity, $\mathbf{F}(t + \Delta t)$ cannot be calculated, since $\mathbf{v}(t+\Delta t)$ is not yet known, because $\mathbf{F}(t+\Delta t)$ is required to calculate it. It follows that the error arising from a single application of Velocity Verlet is $O(\Delta t^2)$.

The method can be made more accurate with a smaller choice of Δt. Allen & Tildesley (1987)[4] suggests Δt must be significantly less than the time it takes for the particle to travel its own diameter, as any initial errors could contribute to much larger errors later in simulation. Typically we take $\Delta t = 0.012$, the significance of this is displayed in figure 3.

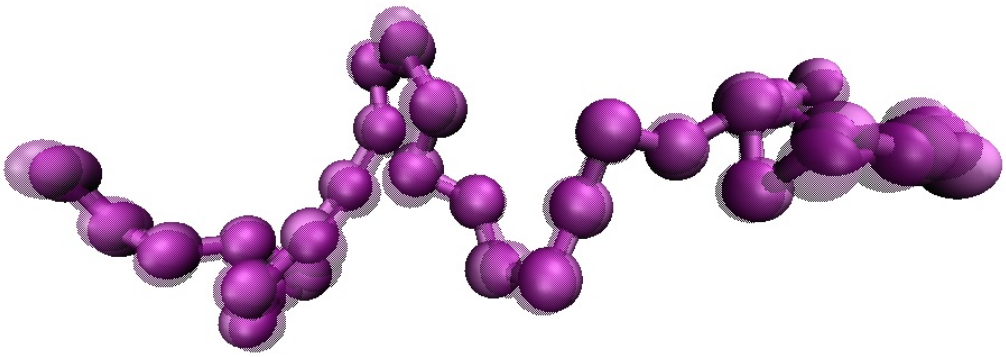


Figure 3: 2 layered images taken 100 time steps apart from the single polymer model (§4), where $\Delta t = 0.0012$

Each of $\mathbf{r}$, $\mathbf{v}$, $\mathbf{a}$, can be overwritten at each new time step (including the half step at stage 2) to save memory. It follows that we require $9N$ variables of space to record these 3 quantities in 3 dimensions for a system of N particles.

3.1.1 Example - testing velocity Verlet

Consider the 1 dimensional harmonic oscillator with the initial conditions

$$\begin{cases} \ddot{r}(t) + r(t) = 0, \quad t \in [0, 100], \\ r(0) = 1, \\ \dot{r}(0) = 2. \end{cases}$$

The exact solution is given by

$$r(t) = \cos(t) + 2\sin(t).$$

We discretize $[0, 100]$ into $M + 1$ points $0, \Delta t, 2\Delta t, \ldots M\Delta t$. The velocity Verlet method is used as an approximate solution, where $r_m = r(m\Delta t)$ with $\Delta t = 100/M$. There are sufficient initial conditions, whilst the acceleration can simply be calculated via $\ddot{r} = -r$. For example, the first step can be calculated using $r_0 = r(0)$ and $\dot{r}_0 = \dot{r}(0)$, yielding

$$\begin{cases} r_1 = r_0 + \Delta t \dot{r}_0 - \frac{1}{2}\Delta t^2 r_0, \\ \dot{r}_{\frac{1}{2}} = \dot{r}_0 - \frac{1}{2}\Delta t r_0, \\ \ddot{r}_1 = -r_1, \\ \dot{r}_1 = \dot{r}_{\frac{1}{2}} - \frac{1}{2}\Delta t r_1, \end{cases}$$

where the substitution $\ddot{r}_m = -r_m$ is used throughout. This process can then be continued (preferably by a computer - the code is given in appendix A.1). We can calculate the error at each time step, by comparing the exact solution with the velocity Verlet solution

$$\tau = |r(t_m) - r_m| .$$

The results are displayed in figure 4 and figure 5, comparing the error for two different values of Δt. From these results, the error that can be incurred by the choice of Δt is clear. It is also worth considering that this is the error on a single particle, in a system of $N = 1000$ particles the error would be far greater; although figure 5 shows $\Delta t = 0.1$ gives a sufficiently close result to the exact solution for this simple oscillator, it would not suffice in the more complex simulations that follow.

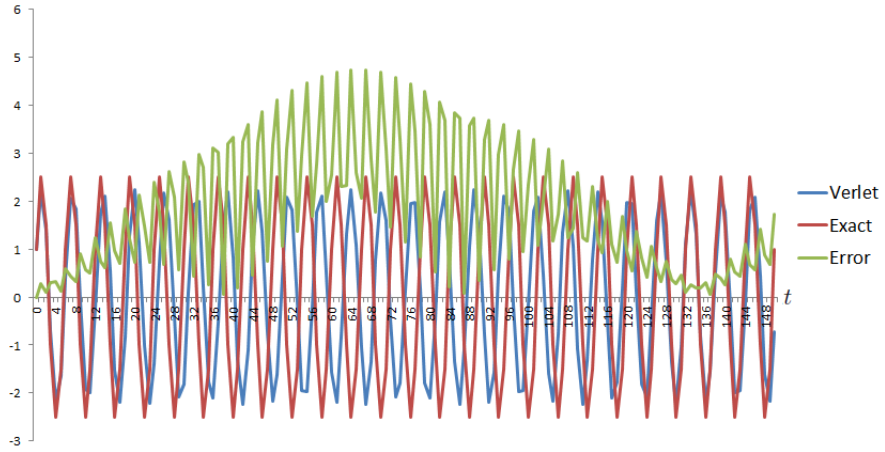


Figure 4: Velocity Verlet approximation for $\Delta t = 1$. The range has been extended to exhibit the oscillatory nature of τ. In a non-periodic system, this oscillatory error would not occur.

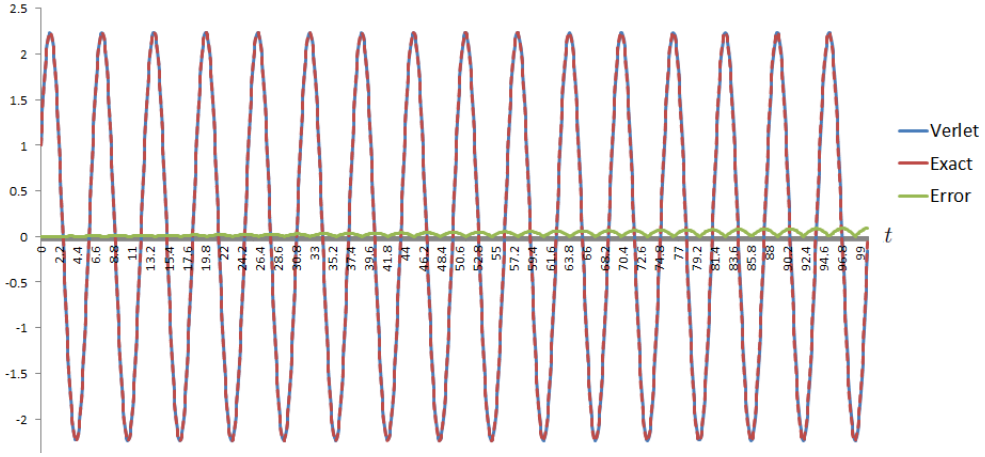


Figure 5: Velocity Verlet approximation for $\Delta t = 0.1$. It is very difficult to distinguish between the two curves here - as the approximation is so effective.

3.2 Lennard-Jones cutoff

The calculation of U_{LJ} and F_{LJ} must consider all possible pairs of particles of the system. This results in $\frac{N(N-1)}{2}$ calculations; it is worthwhile during computation to effectively avoid LJ calculations which will yield relatively insignificant results. A **cutoff radius**, r_c is introduced; whereby only particle interactions such that $r_{ij} \leq r_c$ are considered. However, calculating r_{ij} can also prove costly, as it requires square rooting, an expensive computation,

$$r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}.$$

Where the RHS is the most similar to the algorithm in the code - each dimension is computed separately. This square rooting is easily avoided, simply by comparing against r_c^2 instead;

$$r_c^2 \leq r_{ij}^2 = (\mathbf{r}_i - \mathbf{r}_j) \cdot (\mathbf{r}_i - \mathbf{r}_j) = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2.$$

For exactly the same reason, it is more efficient to calculate the Lennard-Jones force using the form

$$\mathbf{F}_{LJ,ij}(\mathbf{r}_{ij}) = 24\epsilon \sum_{\substack{i=1 \\ j \neq i}}^{N} \mathbf{r}_{ij} \left[2\frac{\sigma^{12}}{(r_{ij}^2)^7} - \frac{\sigma^6}{(r_{ij}^2)^4} \right],$$

as computing r_{ij}^2 first and subbing into the above algorithm is far more efficient. The LJ force between particles i and j can now be re-defined as

$$\mathbf{F}^*_{LJ,ij}(\mathbf{r}_{ij}) = \begin{cases} \mathbf{F}_{LJ,ij}(\mathbf{r}_{ij}), & (r_{ij} \leq r_c), \\ 0, & (r_{ij} > r_c), \end{cases}$$

where outside of the cutoff radius, all LJ interactions are ignored. This cutoff leads to a discontinuity in U_{LJ}, which is corrected by *shifting* the potential,

$$U^*_{LJ,ij}(r_{ij}) = \begin{cases} U_{LJ,ij}(r_{ij}) - U_{LJ,ij}(r_c), & (r_{ij} \leq r_c), \\ 0, & (r_{ij} > r_c). \end{cases}$$

For the case U^*_{LJ}, typically $r_c = 2.5\sigma$, which results in the particles having an attractive tail. Alternatively a 'purely repulsive' cutoff is used, in which $r_c = 2^{1/6}\sigma$. This is obtained by simply finding the turning point in the potential, so we consider

$$\begin{aligned} 0 &= \frac{\partial U_{LJ,ij}(r_c)}{\partial r_{ij}} \\ 0 &= 4\epsilon \left(-12\frac{\sigma^{12}}{{r_c}^{13}} + 6\frac{\sigma^6}{{r_c}^7} \right) \\ \Leftrightarrow r_c &= 2^{1/6}\sigma \end{aligned}$$

as required. This principle of a cutoff radius can applied to any potential considered to become negligible as $r_{ij} \to \infty$.

3.3 Periodic boundary conditions

Despite development of computer technology, the number of particles which can be simulated is still limited to less than 10^6, which is still small compared with Avogadros number[3]. In order to mimic a macroscopic system, **periodic boundary conditions** (PBC) are used, which take a relatively small (usually cubic) sample of the substance being modeled and effectively represents the macroscopic solution by repeating the unit sample periodically in space. The motivation here is to represent the effect of particles surrounding those inside the sample cell, without having to model them explicitly. As an example, Marzari (2005)[6] claims it can be used to effectively model water by repeating the image of just 32 water molecules in the central simulation box.

Under PBC, if a particle should leave the simulation box, it re-enters via the opposite side [4]. As discussed by Allen & Tildesley (1987)[4], to avoid jumps in force calculations, it is also necessary to repeat the image of the simulation box in every direction, so that if two particles are close to the edge of the box, and one jumps from one side to the other, they are still close - at least in a sense of forces between them. Clearly this can not apply to bonded forces such as $\mathbf{F}_{Fene}$. Hence if our simulation box is of length L and is repeated n times in each direction, a particle situated at position $\mathbf{r}_i$ has an image $\mathbf{r}'_i$ in the positions(s)

$$\mathbf{r}'_i = \mathbf{r}_i + \mathbf{n}L, \quad \mathbf{n} \in \{-n, \ldots, -1, 0, 1, \ldots, n\}^3 \tag{17}$$

To see how this will effect motion of particles, consider the total effect of some pairwise force $\mathbf{F}(\mathbf{r}_{ij})$ on a particle i

$$\mathbf{F}_i = \sum_{j=0, j\neq i}^{N} \mathbf{F}(\mathbf{r}_i - \mathbf{r}_j),$$

[3] Avogadro's number relates the number of molecules to the amount of the the substance measured (in moles) and is approximately $6.02214179 \times 10^{23}$. This represents the order of the total number of particles contained in a macroscopic sample.

[4] Frenkel & Smit (1996)[5] describes periodic boundary conditions in 2 dimensions as a mapping from the simulation space to the surface of a sphere - as nothing can ever leave the surface by moving along it in any given way. It is actually topological equivalent to a torus, not a sphere. Similarly, our 3 dimensional case is equivalent to the far less intuitive concept of mapping the simulation space onto a 4 dimensional hyper-torus.

which is re-written under PBC as

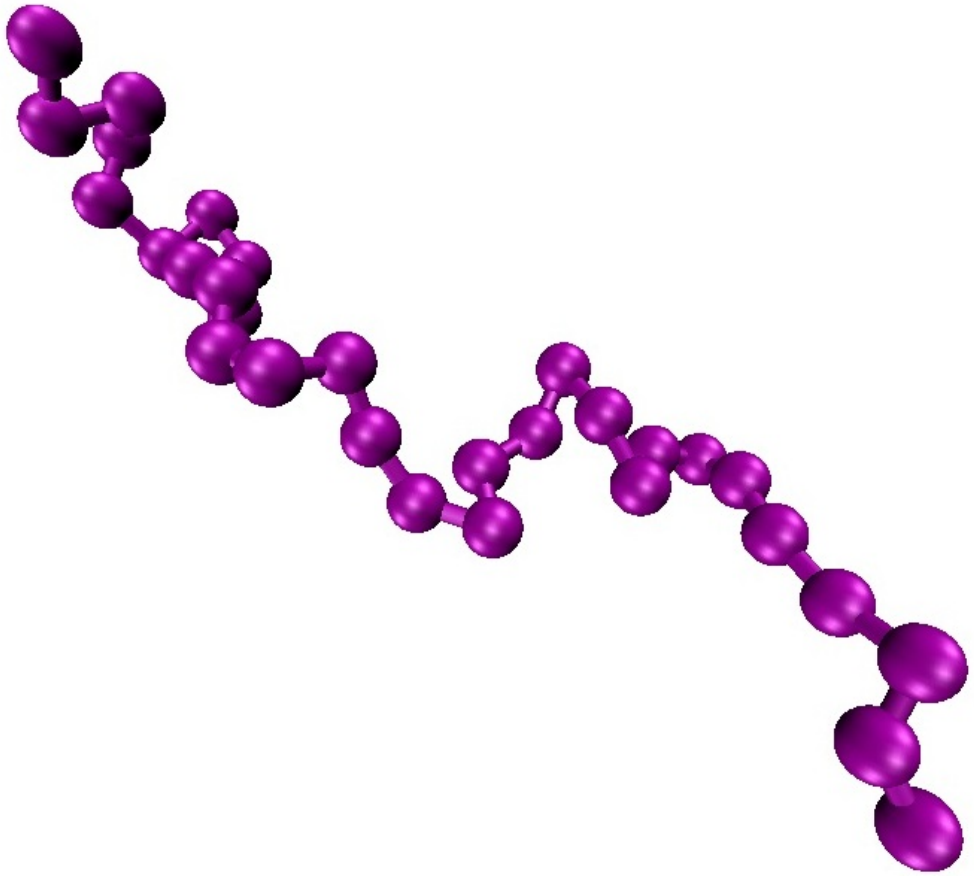
$$\mathbf{F}_i = \sum_{\mathbf{n}}^{\text{boxes}} \sum_{j=0}^{N} {}'\mathbf{F}(\mathbf{r}_i - [\mathbf{r}_j + \mathbf{n}L]).$$

In the above equation, the dash$'$ represents '$i \neq j$ when $\mathbf{n} = \mathbf{0}$', which essentially means that a particle in the central box can interact with any image of itself not in the central box. Taking $n = 1$ in (17) corresponds to 27 cubes; it follows that the number of boxes in a simulation will be $(2n + 1)^3$. If the simulation box has opposite corners at position $(0, 0, 0)$ and (L, L, L), (whilst results are equivalent to any box position, this is generally the most intuitive) the PBC implies the mapping

$$\text{PBC} : \mathbf{r}_i \longmapsto \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} \mod \begin{pmatrix} L \\ L \\ L \end{pmatrix}.$$

This is simple to implement computationally. To avoid a particle interacting with its own image from all directions at the same time, it is necessary to choose $L > r_c$. Hence a force with a shorter cutoff radius r_c requires less periodic boxes. Many boxes are required to model long range charged interactions, a method of calculating this efficiently is discussed in §5.3.

4 Modelling a single polymer in solution



In this section, the theories and methods introduced in §2 and §3 are used to model a single polymer in solution. Various quantities such as end-to-end distance, radius of gyration and average bond length are introduced and discussed within the context of the simulation results. The code for this simulation can be found in appendix A.

4.1 The model

We model the single polymer with no explicit solvent, under the potential energy

$$U = U^*_{LJ} + U_{Fene}$$

where the * represents a truncation in the LJ potential. We choose the LJ cut off $r_c = 2^{\frac{1}{6}}\sigma$, resulting in purely repulsive LJ. We take ϵ, σ and m to be the fundamental units of energy, distance and mass respectively. The use of reduced units is convenient for two reasons. Firstly, our reduced quantities are now comparable as they are of similar magnitude - this is far more convenient than having to deal with lengths on a molecular scale (for example, quantities of 10^{-9} meters). Secondly, for example by setting $\sigma = 1$, any calculation in which a number is multiplied by σ can now be ignored - reducing the overall computation time (Allen & Tildesley 1987)[4]. We do not use periodic boundary conditions, and any frictional or random forces are ignored. The parameters κ and R_0 of U_{Fene} are assigned within the suggested boundaries of Rubinstein & Colby (2003)[9] as follows:

$$1.5\sigma \leq R_0 \leq 2\sigma, \qquad 5\epsilon/\sigma^2 \leq \kappa \leq 30\epsilon/\sigma^2.$$

Specifically we choose parameters based on Kremer & Grest (1990)[7], which agrees with the above conditions. It is worth noting that all numerical results of the simulation will depend on these:

$$R_0 = 1.5, \qquad \kappa = 30,$$

we also choose a purely repulsive Lennard-Jones cutoff $r_c = 2^{1/6}$. The equation of motion for a single particle i becomes

$$\begin{aligned}
\mathbf{F}_i &= -\nabla_{\mathbf{r}_i}(U^*_{LJ} + U_{Fene}) \\
&= 30\left\{(\mathbf{r}_i - \mathbf{r}_{i+1})\frac{1}{1-(\frac{2r_{i,i+1}}{3})^2} + (\mathbf{r}_i - \mathbf{r}_{i-1})\frac{1}{1-(\frac{2r_{i,i-1}}{3})^2}\right\} \\
&\quad +24\sum_{\substack{j=1 \\ j\neq i}}^{N}(\mathbf{r}_i - \mathbf{r}_j)\left[2\frac{1}{r_{ij}^{14}} - \frac{1}{r_{ij}^{8}}\right]^*.
\end{aligned}$$

Note that the above equation is the most general that can be written; the FENE force must drop the left hand term when $i = N$, and the right hand term when $i = 1$. Similarly the * represents the condition that the entire LJ term must be dropped if $r_c > 2^{1/6}$. Most simulations will model a chain of $N = 30$ monomers.

4.2 Initialization

Before forces can be calculated, initial positions and velocities must be assigned to all particles in the system. The initial velocities will typically follow a Gaussian distribution, and can be scaled in accordance with equation (4). There are many ways to assign positions, but for such a simple system, it is sufficient to adopt the following procedure:

- Set $\mathbf{r}_1 = (0, 0, 0)^T$
- Generate successive position vectors for particle 2 up to particle N, using the simple relationship

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \frac{l_0}{|\mathbf{R}|}\mathbf{R}$$

where $\mathbf{R}$ is a random 3 dimensional vector and l_0 is the initial bond length which must satisfy $l_0 < R_0$. To initialize under excluded volume we choose $l_0 = 1$; after initialization we can rely on the repulsive LJ to prevent significant particle overlaps and maintain this excluded volume. After each monomer ($i \geq 3$) is generated, the condition

$$\min_{j \in 0,1 \ldots N}(r_{ij}) \geq l_0$$

must hold, otherwise it is re-generated. This ensures that no particle is closer to particle i than its bonded neighbours. However, this could lead to an infinite loop, (described informally by figure 6). This is compensated for by the code; should there be more than 50 regenerations the entire chain is generated again from scratch. This initialization method is only appropriate for smaller systems such as this; in §5.1 a method for larger systems is introduced. The random vector $\mathbf{R}$ and the initial Gaussian velocity are generated using `ran1` and `gasdev` respectively, both taken from Press et al. (1992)[11].

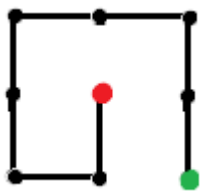


Figure 6: A 2-D representation of how a self-avoiding random walk generated using the method above could go wrong, where the green node is the first step in the walk and the red node is the last. There is no possible next step.

4.3 End-to-end distance distribution

A useful quantity is the end-to-end vector of the polymer, defined as

$$\mathbf{r}_{ee} := \sum_{i=1}^{N-1} (\mathbf{r}_{i+1} - \mathbf{r}_i) = \mathbf{r}_N - \mathbf{r}_1,$$

it follows that the predicted value of $\mathbf{r}_{ee}$ depends on the assumptions we make, and the model we use. Rubinstein & Colby (2003)[9] explains a variety of ideal chain models, which ignore excluded volume. A general assumption made for these static ideal chains is that

$$\langle \mathbf{r}_{ee} \rangle \cong \mathbf{0}, \tag{18}$$

based on the assumption that the position of the next monomer in the chain is just as likely to be in one direction as it is in the opposite direction. However, the above assumption should still apply under excluded volume - the monomers do not need to overlap for their average separation to be $\mathbf{0}$ (apart from a 1D case). Over a sufficient interval $[T_0, T_f]$ we can similarly assume that

$$\int_{T_0}^{T_f} (\mathbf{r}_{ee}) \mathrm{d}t \cong \mathbf{0}. \tag{19}$$

It is a more interesting result to measure $r_{ee} = \sqrt{\mathbf{r}_{ee} \cdot \mathbf{r}_{ee}}$, the end to end *distance*. During simulation, r_{ee} was measured over $10^8 \Delta t$; it was found that for a chain of $N = 30$ monomers $\langle r_{ee} \rangle \approx 8.426$. Rubinstein & Colby (2003)[9] predict the distribution of r_{ee} to take the form

$$P(r_{ee}) \cong 0.278 \left(\frac{r_{ee}}{\sqrt{\langle r_{ee}^2 \rangle}} \right)^{0.28} \exp\left(-1.206 \left(\frac{r_{ee}}{\sqrt{\langle r_{ee}^2 \rangle}} \right)^{2.43} \right),$$

although this is not based on rigorous theory. To obtain useful predictions from this we must consider the probability of r_{ee} lying in some given area under the curve. We convert into spherical coordinates, by selecting a spherical interval $[r_{ee}, r_{ee} + \mathrm{d}r]$. The predicted distribution now represents the probability of $\mathbf{r}_N$ lying between a sphere of radius r_{ee} and another of radius $r_{ee} + \mathrm{d}r$, with both spheres centered at $\mathbf{r}_1$

$$P_\circ(r_{ee}) \cong 4\pi 0.278 \left(\frac{r_{ee}}{\sqrt{\langle r_{ee}^2 \rangle}} \right)^{0.28} \exp\left(-1.206 \left(\frac{r_{ee}}{\sqrt{\langle r_{ee}^2 \rangle}} \right)^{2.43} \right) {r_{ee}}^2 \mathrm{d}r.$$

Figure 7 compares the prediction of Rubinstein & Colby (2003)[9] with the results of our simulation - which closely agrees with the prediction.

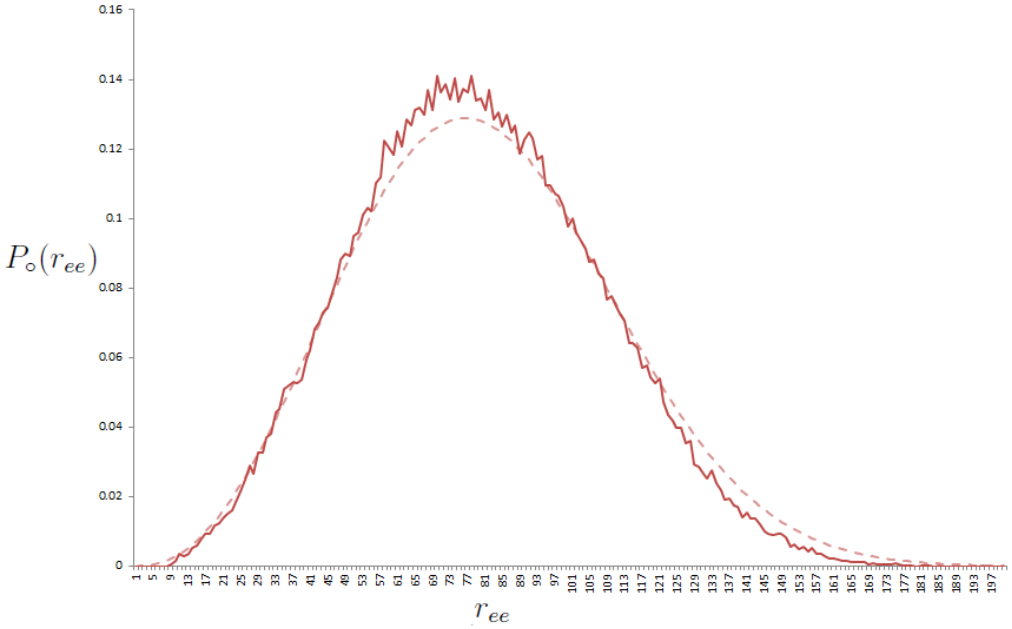


Figure 7: The dashed curve is the predicted r_{ee} from Rubinstein & Colby (2003)[9], the full red curve is from a sample of data taken over 10^8 time steps.

4.4 Flory's characteristic ratio

Flory's characteristic ratio is an effective measurement of the stiffness of the chain. Instead of considering the end-to-end distance, we consider the quantity $\langle r_n^2 \rangle$, which is the average squared distance between two monomers n bonds apart in the chain. For convenience, we switch notation to deal with bond vectors, as opposed to position vectors. We define $\mathbf{r}_{i,i+1} = \overline{\mathbf{r}}_i$ representing a bond vector, and $n = N - 1$ representing the number of bonds. $\langle r_n^2 \rangle$ can now be written as

$$\begin{aligned} \left\langle {r_n}^2 \right\rangle &= \langle \mathbf{r}_n \cdot \mathbf{r}_n \rangle \\ &= \left\langle \left(\sum_{i=1}^{n} \mathbf{r}_i \right) \cdot \left(\sum_{j=1}^{n} \mathbf{r}_j \right) \right\rangle \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} \langle \mathbf{r}_i \cdot \mathbf{r}_j \rangle \,. \end{aligned}$$

By the definition of the dot product, we can write

$$\begin{aligned} \left\langle r_n^2 \right\rangle &= \sum_{i=1}^{n}\sum_{j=1}^{n} \left\langle l^2 \cos(\theta_{ij}) \right\rangle \\ &= \left\langle l^2 \right\rangle \sum_{i=1}^{n}\sum_{j=1}^{n} \left\langle \cos(\theta_{ij}) \right\rangle \\ &= C_n n \left\langle l^2 \right\rangle \end{aligned} \tag{20}$$

where θ_{ij} is the angle between bonds i and j, and $\langle l^2 \rangle$ is the average bond length. This assumes there is no correlation between the bond length and the bond angle. Rubinstein & Colby (2003)[9] defines C_n as **Flory's characteristic ratio**, where

$$C_n := \frac{1}{n}\sum_{i=1}^{n} \cos(\theta_{ij}).$$

As C_n considers all of the bond angles of the system, it is an effective measurement of the stiffness of the polymer. As $n \to \infty$, C_n converges to a limit C_∞. In our simulation, C_n is calculated via the relationship

$$C_n = \frac{\left\langle r_n^2 \right\rangle}{n \left\langle l^2 \right\rangle}$$

where $\langle l^2 \rangle$ is necessary, because the bond length during simulation is not constant and gives the result $\langle l^2 \rangle \approx 0.934$. Remembering that the fundamental unit of length is taken in terms of the particle diameter $\sigma = 1$, this average length actually represents a small amount of overlap. The chain modeled ($N = 30$ monomers) is too small to calculate C_∞ in this case. As a couple of examples, the values of C_∞ for chains of polyethylene and atactic polystyrene are 7.4 and 9.5 respectively (Rubinstein & Colby 2003)[9].

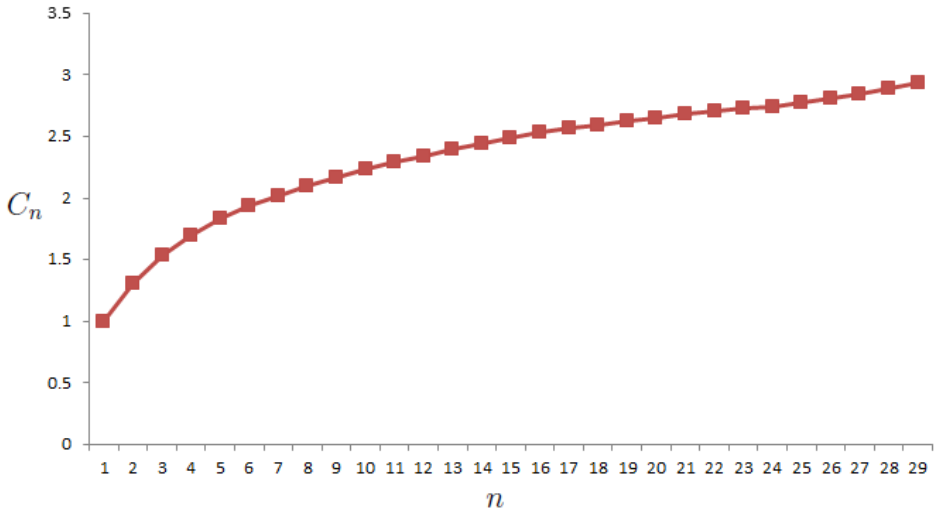


Figure 8: Graph of Flory's characteristic ratio for N=30 monomers (with n=29 bonds)

4.5 Radius of gyration

Another characteristic of a polymer chain is the radius of gyration r_g, which represents the average distance from the center of mass. In practice (due to computational convenience) it is more common to consider the square radius of gyration

$$r_g^2 = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{r}_i - \mathbf{r}_{cm})^2, \tag{21}$$

where $\mathbf{r}_{cm}$ is the center of mass (Rubinstein & Colby 2003)[9]. Theoretically $\mathbf{r}_{cm}$ should be written as

$$\mathbf{r}_{cm} = \frac{\sum_{i=1}^{N} m_i \mathbf{r}_i}{\sum_{i=1}^{N} m_i},$$

but for the chain we are modeling all masses are assumed to be equal, so the above formula simplifies to

$$\mathbf{r}_{cm} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{r}_i.$$

The above is combined with (21) to compute $\langle r_g^2 \rangle$. The radius of gyration is a good measure of the freedom of the monomers in the chain; a large r_g

corresponds to monomers being further away from the center of mass, suggesting a greater amount of freedom, allowing the chain to expand. Polymers exhibiting this are said to be in 'good solvent'; meaning the polymer is more attracted to the solvent than itself.[5] A smaller value of r_g corresponds to a chain that is contracting - when the polymer is in 'poor solvent' it contracts, forming globules. The case where a polymer is equally attracted to its surroundings and itself is described as θ-point. In §6 we model a polymer melt, in which the polymer is surrounded by other polymers identical to it and thus is at θ-point. Flory (1958)[12] gives an interesting result; when at θ-point, excluded volume can be ignored in theoretical models - and the polymers can be treated as ideal chains[6].

4.6 Summary of results

The following table is a summary of the quantities discussed when modeling a single polymer chain, consisting of 30 monomers. Under the reduced units already discussed, considering that κ=30 and ϵ=1, we record data over $10^8\Delta t$, where the time step is taken to be $\Delta t = 0.012$. All the values are in terms of the particle diameter σ.

Quantity:	$\langle r_g^2 \rangle$	$\langle r_{ee}^2 \rangle$	$\langle l^2 \rangle$
Value (3.d.p):	11.0173	71.004	0.934

As the model is very basic, we cannot draw any meaningful conclusions from these figures - other than that the numbers agree with Kremer & Grest (1990)[7], verifying our model as correct. To gain some relative perspective, the fully extended chain is given by $N\langle l \rangle \approx 29.1$, when compared with the average end-to-end distance $\langle r_{ee} \rangle \approx 8.4$, this represents the average extension of the chain. These quantities are also computable in the more complex systems modeled in §6, where we are primarily interested in quantities related to the reversible network formation of the polymers. Furthermore, there are two types of polymer present, so two separate sets of results are required - despite the fact that the results for one polymer chain most probably depend on the parameters of the other. Before we are equipped to model and analyze these self-healing polymer systems, further methods must be introduced.

[5] An example of a polymer in good solvent is sodium polyacrylate - the gel used in absorbent nappies. This expands by over a hundred times when exposed to water - a polymer than likes its surroundings much more than itself (Helmenstine 2007)[13].

[6] In the ideal case, this can be further simplified to show the interesting *Deybe* result; $\langle r_g^2 \rangle = \frac{1}{6}\langle r_{ee}^2 \rangle$, (Rubinstein & Colby 2003)[9].

5 Advanced methods

In this section we introduce the advanced methods required for more complex MD simulations, which were not required for the comparatively simple simulation of a single polymer. Mostly, the content of these methods does not overlap and so their description is in no particular order - although it is worth noting that the fast Fourier transform and the multiple-τ autocorrelation function have applications ranging far beyond MD simulations.

5.1 Initialization

For large dense systems, it is very unlikely that randomly positioned particles will not overlap. If this is the case, a **warm up** period as described by Auhl et al. (2003)[14] might be used to initialize the system. At the start of this warm up, a random walk is used to position particles of the polymers, and the polymers are randomly positioned inside the simulation box. This results in some overlap initially. At this stage a **force-capped Lennard-Jones** potential U_{LJ}^{fc} is introduced to slowly separate the particles, without the explosive properties that regular U_{LJ} would impose. Similar to the concept of the cutoff radius discussed in §3.2 a force cutoff r_{fc} is used, which corresponds to the maximum force allowed by this capped potential[7]. Hence force-capped Lennard-Jones is defined as

$$U_{LJ,ij}^{\text{fc}} := \begin{cases} (r_{ij} - r_{fc})U'_{LJ,ij}(r_{fc}) + U_{LJ,ij}(r_{ij}), & (r_{ij} < r_{fc}), \\ U_{LJ,ij}(r_{ij}), & (r_{fc} \leq r_{ij}). \end{cases}$$

This potential is linear at $r_{ij} < r_{fc}$, and bounded at $r_{ij} = 0$, whereas normal U_{LJ} explodes as $r_{ij} \to 0^+$. The corresponding force at any separation less than r_{fc} is equal to the force at r_{fc}. Physically U_{LJ}^{fc} will cause a gentle repulsion between overlapping particles, allowing them to gradually reposition until a realistic setting for initialization (without overlap) is reached. In the case of Auhl et al. (2003)[14], at the start of the warm up period r_{fc}=r_c but gradually decreases, hence $U_{LJ}^{\text{fc}} \to U_{LJ}$ typically until $r_{fc} = 0.8\sigma$. By this stage, there should be no particles overlapping by this amount, so U_{LJ}^{fc} will have the same effect on our system as U_{LJ} anyway. The main simulation loop can now begin, although if there are other forces which can only be introduced after warming up, it is worth allowing time for these forces to settle before worthwhile data can be recorded.

[7] Note that it is most likely a truncated LJ potential $U_{LJ,ij}^{*}$ will also be used in place of $U_{LJ,ij}$.

5.2 The Verlet & Cell lists

Whilst deploying a cutoff radius r_c (as explained in §3.2) reduces the number of necessary pairwise force calculations from $\frac{N(N-1)}{2}$, we must still *check* this number of particles to decide whether or not the force between them should be calculated. Realistically, if two particles are far apart, say $r_{ij} = 2r_c$ at a time t, we can judiciously assume (without checking) that $r_{ij} > r_c$ at $t + \Delta t$, for some reasonable Δt. The Verlet list and The Cell list algorithms described by Frenkel & Smit (1996)[5] are based on this assumption - that there is often no need to even check r_{ij}.

5.2.1 Verlet list

In Verlet list, a second radius $r_v > r_c$ is used, encapsulating all of the potential candidates for force calculation in the near future. This annulus centered at particle i

$$\{\mathbf{r}_j : r_c < r_{ij} < r_v\},$$

consists of all the particles which are *not* currently contributing to the force, but may at the next time step. All particles inside this set will be *checked*. For every particle i, the particles inside the sphere

$$\{\mathbf{r}_j : r_{ij} < r_v\}$$

are noted as possible contributors to the force $\mathbf{F}_i$ in the near future. For a system of N particles, this information can be represented as a symmetric $N \times N$ binary matrix A with entries a_{ij}=1 if $r_{ij} < r_v$, and $a_{ij} = 0$ otherwise. As self interactions are not considered, the diagonal entries $a_{ii} = 0$ also. Since A is symmetric, and typically pairwise forces f_{ij} and f_{ji} are computed at the same time, A can be re-written as an upper triangular matrix to save space. Hence, during simulation, $r_{ij} \leq r_c$ is only *checked* if $a_{ij} = 1$, otherwise we can safely assume that $r_{ij} > r_c$ and (in the case of LJ) $F^*_{LJ} = 0$. This list need not be updated every time step (and would not be worthwhile if it was), it is more typical to update the list every $M\Delta t$, where typically $M = 10$ or 100. Clearly based on this, r_v must be picked such that

$$r_v - r_c > v_{max} M \Delta t$$

where v_{max} is the maximum possible velocity of any particle in the system, to avoid any particle j not included in the list becoming a distance $< r_c$ from a particle i, resulting in the corresponding force between them being (wrongly) ignored. The Verlet list scales as $O(N^2)$, which is actually the same as *not* using it, but is worthwhile due to its key advantage that it does not need to be updated every time step.

5.2.2 Verlet list for a periodic system

The Verlet list can be effectively extended into PBC with little hassle. Considering the pairwise forces $\mathbf{F}_{ij}(\mathbf{r}_{ij})$ acting on a particle i in a periodic system

$$\mathbf{F}_i = \sum_{j=1}^{N} \sum_{\mathbf{n}}^{\text{boxes}} {}'\mathbf{F}_{ij}(\mathbf{r}_i - [\mathbf{r}_j + \mathbf{n}L]). \tag{22}$$

After a small amount of thought, the symmetry of the periodicity gives the relationship

$$\mathbf{F}_{ij}(\mathbf{r}_i - [\mathbf{r}_j + \mathbf{n}L]) = \mathbf{F}_{ij}([\mathbf{r}_i - \mathbf{n}L] - \mathbf{r}_j) = -\mathbf{F}_{ij}([\mathbf{r}_j + \mathbf{n}L] - \mathbf{r}_i) = -\mathbf{F}_{ij}(\mathbf{r}_j - [\mathbf{r}_i - \mathbf{n}L]),$$

using this result we can re-write (22) as

$$\mathbf{F}_i = \sum_{j=1}^{N} \sum_{\mathbf{n}}^{\text{boxes}} {}'\mathbf{F}_{ij}([\mathbf{r}_i + \mathbf{n}L] - \mathbf{r}_j).$$

This means that the summation has been changed - instead of computing r_{ij} for each j in every neighboring box $\mathbf{n}$, we now compare each j in the central box with its nearest image of i. Hence we can construct a different Verlet list for each particle i, and will only need to consider all particles j that interact with i, but more importantly, these will only be particles in the central box - so the list will be much smaller. This makes this seemingly minor adjustment to (22) worthwhile.

5.2.3 Cell list and comparison with Verlet list

Similar in principle to Verlet list, the **cell list** method divides the central box into sub-boxes of length $L_{sub-box} > r_c$. For a particle i in a sub-box, r_{ij} is only tested for particles j in the neighboring sub-boxes. This operation scales as $O(N)$. Frenkel & Smit (1996)[5] compares the two methods, for a system of number density ρ the number of particles for which r_{ij} must be calculated in the Verlet list is given by

$$N_{Verlet} = \frac{4}{3}\pi\rho r_v^3$$

where the equation for a sphere has been used. Similarly the number of particles considered for the cell list is given by

$$N_{Cell} = 27\rho r_c^3.$$

So the most effective method will depend on which parameters are chosen. Many computer simulations use a combination of both Verlet and cell lists; Auerbach et al. (1987)[15] used the Verlet list to avoid large numbers of particle considerations, whilst using the cell list method to construct this Verlet list.

5.3 Ewald summation

For a system in which periodic boundary conditions are used, problems may arise where the cutoff radius r_c of a long range potential such as coulomb potential U_c, is large enough such that the number of periodic boxes must be very large. In these situations, the **Ewald** method can be used to reduce the computation cost. The basic idea behind it is to replace the point charges of the system with a set of Gaussian distributions. The following derivation is adapted from that of Frenkel & Smit (1996)[5]. Considering the charged interactions of a system under PBC,

$$U_c = \frac{1}{4\pi\varepsilon_0\varepsilon_r}\sum_{\mathbf{n}}\sum_{i=1}^{N-1}\sum_{j=i+1}^{N}{}' \frac{q_i q_j}{|\mathbf{r}_i - [\mathbf{r}_j + \mathbf{n}L]|}. \tag{23}$$

The Ewald summation will convert the formulation of U_c into one which converges faster, and absolutely. A particle i in the periodic system has an **electric field**, given by

$$\phi_i(\mathbf{r}) = \frac{1}{4\pi\varepsilon_0\varepsilon_r}\sum_{\mathbf{n}}\frac{q_i}{|\mathbf{r} - [\mathbf{r}_i + \mathbf{n}L]|}. \tag{24}$$

For convenience we now drop the constant term preceding the summation, as this will not effect the result. The Ewald summation collects these fields as a collection of point charges. We start by considering a compensating Gaussian charge cloud surrounding individual terms, which in general is given by

$$\rho_G(\mathbf{r}) = -q_i\left(\frac{\alpha}{\pi}\right)^{3/2} e^{-\alpha r^2}$$

where r is the particle separation, α will be determined later on, and must be selected carefully to split real space and reciprocal space. Firstly the continuous background charge is computed, by considering the sum of these Gaussians;

$$\rho_L(\mathbf{r}) = \sum_{\mathbf{n}}\sum_{j=1}^{N} q_j\left(\frac{\alpha}{\pi}\right)^{3/2} e^{-\alpha|\mathbf{r}-(\mathbf{r}_j+\mathbf{n}L)|^2}.$$

We now seek the solution to the following Poisson's equation in order to compute the corresponding total electrostatic potential

$$-\nabla^2\phi_L(\mathbf{r}) = 4\pi\rho_L(\mathbf{r}). \tag{25}$$

Next we take the Fourier transform of $\rho_1(\mathbf{r})$, where $\mathbf{k}$ is the corresponding reciprocal space vector and $V = L^3$ is the volume, giving

$$\begin{aligned}\hat{\rho_L}(\mathbf{k}) &= \frac{1}{V}\int_V e^{-i\mathbf{k}\cdot\mathbf{r}}\sum_{\mathbf{n}}\sum_{j=1}^{N} q_j\left(\frac{\alpha}{\pi}\right)^{3/2} e^{-\alpha|\mathbf{r}-(\mathbf{r}_j+\mathbf{n}L)|^2}d\mathbf{r} \\ &= \frac{1}{V}\sum_{j=1}^{N} q_j e^{-i\mathbf{k}\cdot\mathbf{r}} e^{-k^2/4\alpha},\end{aligned}$$

which can be subbed into the Fourier form of (25), where $k^2 = \mathbf{k}\cdot\mathbf{k}$, to yield

$$\hat{\phi_L}(\mathbf{k}) = \frac{4\pi}{Vk^2}\sum_{j=1}^{N} q_j e^{-i\mathbf{k}\cdot\mathbf{r}} e^{k^2/4\alpha}.$$

It conveniently follows that in computation the $\mathbf{k} = \mathbf{0}$ term can be ignored. Taking the inverse Fourier transform, we now convert back to real space,

$$\phi_L(\mathbf{r}) = \frac{1}{V}\sum_{\mathbf{k}\neq\mathbf{0}}\sum_{j=1}^{N}\frac{4\pi q_j}{k^2} e^{i\mathbf{k}\cdot(\mathbf{r}-\mathbf{r}_j)} e^{-k^2/4\alpha}$$

and the long range potential becomes[8]

$$U_L = \sum_{\mathbf{k}\neq\mathbf{0}}\sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\frac{4\pi q_i q_j}{k^2} e^{i\mathbf{k}\cdot(\mathbf{r}_i-\mathbf{r}_j)} e^{-k^2/4\alpha}. \tag{26}$$

The current energy formulation U_L includes a *self-interaction* term, due to the point charge q_i and its own corresponding Gaussian charge cloud

$$\rho_G(\mathbf{r}) = q_i\left(\frac{\alpha}{\pi}\right)^{3/2} e^{-\alpha r^2},$$

which must be calculated and subtracted. Frenkel & Smit (1996)[5] shows this term to be

$$U_{self} = \left(\frac{\alpha}{\pi}\right)^{1/2}\sum_{i=1}^{N} {q_i}^2. \tag{27}$$

[8]Note the slight abuse of notation which starts in equation (26), where we take $i = \sqrt{-1}$ but when used as a subscript or index in a sum, i still represents the index of the particle we are currently summing.

The above does not contain any $\mathbf{r}_i$ terms, making it conveniently constant throughout simulation. Hence, (27) only needs to be calculated once. Finally, we can compute the comparatively simple short range interactions, where we make use of the error function $\text{erf}(x)$ and its complementary error function $\text{erfc}(x)$, defined as

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-s^2} ds, \qquad \text{erfc}(x) = 1 - \text{erf}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-s^2} ds.$$

It follows that the solution to Poisson's equation for $\rho_G(\mathbf{r})$ becomes

$$\phi_G(\mathbf{r}) = \frac{q_i}{r} \text{erf}(\sqrt{\alpha} r)$$

and using this, $\phi_s(\mathbf{r})$ can be simply defined (compensating for its error term) as

$$\begin{aligned} \phi_s(\mathbf{r}) &= \frac{q_i}{r} - \frac{q_i}{r} \text{erf}\left(\sqrt{\alpha} r\right) \\ &= \frac{q_i}{r} \text{erfc}\left(\sqrt{\alpha} r\right). \end{aligned}$$

The total short range contribution is therefore given by

$$U_s = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} q_i q_j \text{erfc}(\sqrt{\alpha} r_{ij}). \tag{28}$$

Finally, combining the real space (28) and reciprocal space (26) terms, and subtracting the error and self interaction terms (27) gives the result

$$\begin{aligned} U_c &= U_s + U_L - U_{self} \\ &= \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} q_i q_j \text{erfc}(\sqrt{\alpha} r_{ij}) + \sum_{\mathbf{k} \neq \mathbf{0}} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{4\pi q_i q_j}{k^2} e^{i\mathbf{k} \cdot (\mathbf{r}_i - \mathbf{r}_j)} e^{-k^2/4\alpha} \\ &\quad - \left(\frac{\alpha}{\pi}\right)^{1/2} \sum_{i=1}^{N} {q_i}^2. \end{aligned} \tag{29}$$

The finer details of this derivation are omitted, but can be found in Frenkel & Smit (1996)[5]. A faster but less intuitive derivation is available from Le & Cai (2009)[16]. After all, it is not the derivation, but the result that we are interested in. The charged version of the self-healing model discussed in §6 actually uses 'Particle-particle/particle-mesh' (P3M), a more advanced version of the above method. This method computes short range interactions directly whilst long range interactions are computed by solving the discrete Poisson equation on a mesh. It scales at $O(N \log N)$ which for large systems is less expensive than the regular Ewald, which has cost $O(N^{3/2})$.

5.4 Fast Fourier transform

To compute the Fourier transform in the Ewald method, we use a version of the fast Fourier transform. The following description is based on pages from Benson (2008)[17], where the explanation is within the context of digitizing audio in modern music software[9], but the process is essentially the same. This description is only in one dimension, but can simply be extended to the 3 dimensional real and reciprocal space by replacing the product of the real and reciprocal variables rk in the exponential term by the dot product $\mathbf{r} \cdot \mathbf{k}$. A function $f(r)$ is discretized after firstly multiplying by a series of Dirac delta functions

$$\delta_s(r) = \sum_{n=-\infty}^{\infty} \delta(r - n\Delta r),$$

yielding

$$\begin{aligned} \delta_s(r)f(r) &= \sum_{n=-\infty}^{\infty} \delta(r - n\Delta r)f(r) \\ &= \sum_{n=-\infty}^{\infty} \delta(r - n\Delta r)f(n\Delta r), \end{aligned}$$

where the simplification is a result of each delta function equal to zero at every point other than $n\Delta r$. By definition, after integrating over $(-\infty, \infty)$ the Fourier transform of $\delta_s(r)f(r)$ becomes

$$\hat{\delta_s f}(\upsilon) = \sum_{n=-\infty}^{\infty} f(n\Delta r)e^{-2\pi i \upsilon n \Delta r}.$$

We take the Fourier transform to be over a sufficiently wide range of length M and index such that $f(r) = 0$ unless $0 \leq n < M$, rewriting as

$$\hat{\delta_s f}(\upsilon) = \sum_{n=0}^{M-1} f(n\Delta r)e^{-2\pi i \upsilon n \Delta r}.$$

There are M pieces of information here, which must be evaluated at points $\upsilon = \frac{k}{M\Delta r}$, for $k = 0, \ldots M$. The discrete Fourier transform is therefore

[9] In the digital Fourier transform of audio data, the frequency represents the number of sounds-per-second (the samplerate) that we hear. Typically for an audio CD this value is 44.1kHz - this has been the case since this it was agreed on by Sony and Phillips in 1980. This corresponds to 74 minutes worth of audio; which was extended from 60 minutes after both companies agreed that the CD should be large enough to encompass a full (74 minute) version of Beethovens 9th Symphony (*Password* 2001)[18]

defined as

$$F(k) = \hat{\delta_s f}(\frac{k}{M\Delta r}) = \sum_{n=0}^{M-1} f(n\Delta r)e^{-2\pi ikn/M}.$$

However, it follows that during calculation many values are calculated twice. The fast Fourier transform takes advantage of this. Suppose that M is even, we can split the sum of F into even (left) and odd (right) components

$$\begin{aligned} F(k) &= F_{even}(k) + F_{odd}(k) \\ &= \sum_{n=0}^{\frac{M}{2}-1} f((2n)\Delta r)e^{-2\pi i(2n)k/M} + \sum_{n=0}^{\frac{M}{2}-1} f((2n+1)\Delta r)e^{-2\pi i(2n+1)k/M}. \end{aligned}$$

By inspection it is clear that the contributions to $F(k)$ and $F(k+\frac{M}{2})$ are closely related. Comparing with the above,

$$F(k+\frac{M}{2}) = \sum_{n=0}^{\frac{M}{2}-1} f((2n)\Delta r)e^{-2\pi i(2n)(k+\frac{M}{2})/M} + \sum_{n=0}^{\frac{M}{2}-1} f((2n+1)\Delta r)e^{-2\pi i(2n+1)(k+\frac{M}{2})/M}$$

$$\begin{aligned} F(k+\frac{M}{2}) &= \sum_{n=0}^{\frac{M}{2}-1} f((2n)\Delta r)e^{-2\pi i(2n)k/M}e^{-\pi i(2n)} \\ &\quad + \sum_{n=0}^{\frac{M}{2}-1} f((2n+1)\Delta r)e^{-2\pi i(2n+1)(k)/M}e^{-\pi i(2n+1)} \\ F(k+\frac{M}{2}) &= \sum_{n=0}^{\frac{M}{2}-1} f((2n)\Delta r)e^{-2\pi i(2n)k/M}\cdot 1 + \sum_{n=0}^{\frac{M}{2}-1} f((2n+1)\Delta r)e^{-2\pi i(2n+1)(k)/M}\cdot -1 \\ &= F_{even}(k) - F_{odd}(k). \end{aligned}$$

Hence, the 'trick' with the fast Fourier transform is taking advantage of the fact that

$$F_{even}(k+\frac{M}{2}) + F_{odd}(k+\frac{M}{2}) = F_{even}(k) - F_{odd}(k).$$

If M is a power of 2, FFT can compute the above in $2M\log_2 M$ operations, instead of the less efficient M^2 through systematic computation. The model discussed in §6 actually uses FFTW (*Fastest Fourier Transform in the West*) which considers a wider range of options (such as the above) to reduce computation time.

5.5 Multiple-tau auto-correlation function

In the self-healing simulations, we will aim to calculate the stress autocorrelation function $G(t)$, in particular the component of the form $\langle\sigma(t_0)\sigma(t-t_0)\rangle$. This component is incredibly noisy, due to fluctuations caused by bond vibrations. For

now we will not concern ourselves with what this component represents, instead the focus lies on finding an efficient method to compute it. The Multiple-τ is a very effective method for computing average correlations over large ranges by making use of the averages already taken from shorter ranges, significantly improving prediction accuracy and allowing the calculation of quantities such as the stress modulus $G(t)$ to be performed on-the-fly. Aside from stress calculations, autocorrelation functions have a wide range of applications in modern technology, for finding hidden harmonic frequencies in a signal or repeated patterns in a large sample of data[10].

As described by Ramírez et al. (2010)[19], the data for the correlation must be stored in a hierarchy of S levels, each consisting of three data arrays of size p plus an accumulator. Each level correlates over a larger interval than the one before it, using information from the level below. Due to this, there will be less samples over the wider intervals, so the resolution of the correlation will decrease as $(t-t_0)$ increases. The three arrays on each level consist of the following:

- D_{ij}, which stores the sampled data at lag j on level i
- C_{ij}, which accumulates values of $D_{i0}D_{ij}$ (equivalent to $\langle\sigma(t_0)\sigma(t-t_0)\rangle$) for each lag j.
- N_{ij} is the counter for this correlation and increases by 1 each time a value is added to the accumulation of C_{ij}.

This process starts at level $i = 1$, and continues until $i = S$, and on each level the lag starts at level $j = 1$, until $j = p$. Each time a new data value ω_i is added to the accumulator level i, an accumulator counter M_i for that level is also increased by 1. Once the accumulation is complete, $\omega_{i+1} = A_i/M_i$ is sent to the level $i + 1$, and the process repeats at a higher level, correlating over a wider range. Data will continue to accumulate at level i and sent to $i + 1$ until all of the data has been successfully auto-correlated, when the average at level $i = S$ has been computed.

[10] Auto-correlation is a method made use of by oil companies - in short the method used is to send seismic waves through the ground and record the reflections, which can then be auto-correlated to form an underground map to help uncover potential drilling sites. Andy Hildebrand was able to retire at 40 due to this money saving technique, and then went on to use auto-correlation to create the controversial auto-tuning algorithm, which is now being used more and more frequently in live popular music performances (Tyrangiel 2009)[20].

6 Modelling self-healing polymer networks

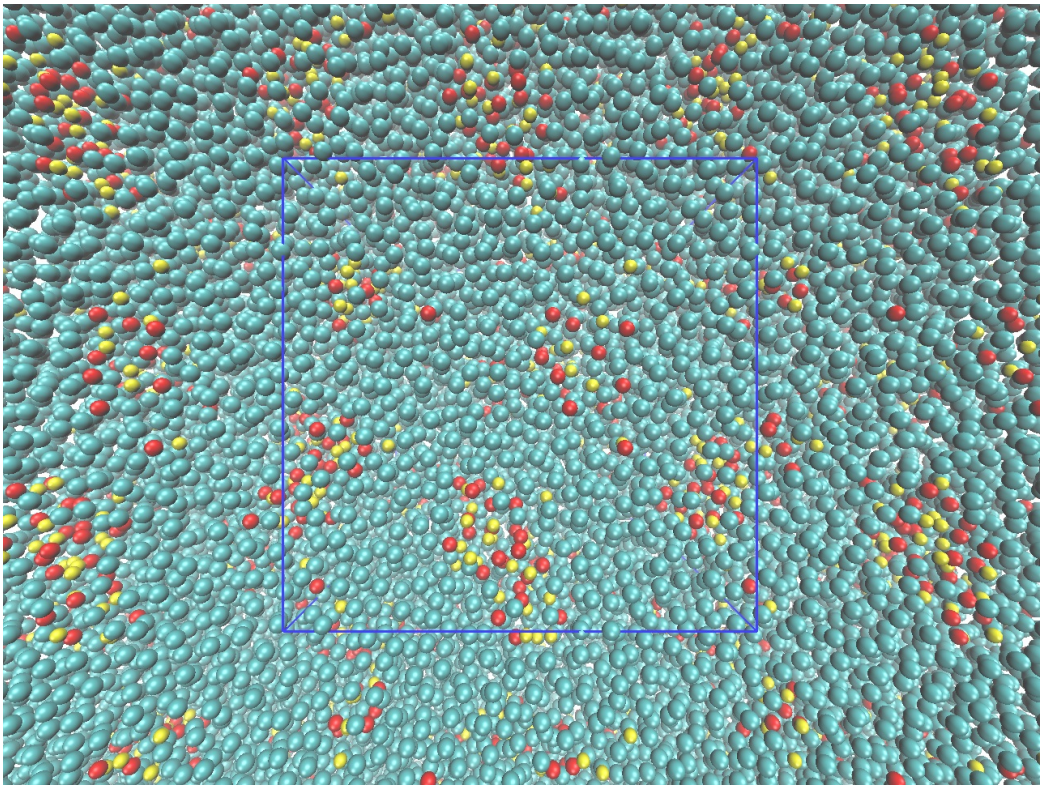


Figure 9: Snapshot of the infinite network formed by Cordier's polymers. The central image (inside the blue box) has been repeated periodically.

Sufficient methods have now been introduced to model the self-healing polymer networks - combining almost everything that has been discussed so far. This section will explain the model and the theory used to obtain and analyze the simulation results, specifically the stress modulus and the cluster analysis. Finally these characteristics and others are compared between different simulation results. Parts of the code for these simulations can be found in appendix B.

6.1 The model

Our simulations are initialized as a polymer melt, where the linear polymers are in liquid state with no additional solvent. If the polymers are of sufficient length, this melt will demonstrate a visco-elastic property. This behaves as an elastic solid on a short time scale, whilst behaving as a viscous liquid at larger time scales. With the added association of attractive or charged monomers, the polymers can form a reversible network - similar to rubber. The base code for the simulation is designed to model two types of polymer, which we denote type-X and type-Y. These can each have varying charges or attractive monomers distributed along their backbone. Lennard-Jones cutoff r_c and potential depth ϵ vary depending on the monomer; for certain interactions we desire an attractive tail. This tail is set such that it only attracts a certain type of monomer. Typically we have a long range LJ attraction ($r_c = 2.5$, $\epsilon = 3$) between type X-Y attractive monomer interactions, but all X-X and Y-Y interactions remain purely repulsive ($r_c = 2^{1/6}$, $\epsilon = 1$). We have

$$N = N_X P_X + N_Y P_Y,$$

where N is the total number of particles, N_X and N_Y are the number of monomers in the X-type and Y-type chains, whilst P_X and P_Y are the number of X-type and Y-type polymer chains respectively. We typically choose these polymer parameters specifically with an attractive (this attraction could be electrostatic or LJ) monomer at either end, and an even distribution of attractive monomers along it. Hence the equation

$$\frac{N_p - 1}{N_{p,attractive} - 1} = d_p + 1 \qquad i = X \text{ or } Y$$

must be satisfied, where $d_p \in \mathbb{N}$ is the number of monomers in the chain type p, between each pair of attractive monomers. On top of this, the number of positively and negatively charged monomers is chosen carefully to ensure that the net charge of the system is zero;

$$\sum_{i=1}^{N} q_i = 0.$$

Similarly, we choose $N_{X,attractive} P_X = N_{Y,attractive} P_Y$. The model automatically determines the simulation box volume as $V = N/\rho$, where we choose the number density of each system to be $\rho = 0.85$ - representing a polymer melt. As in §4, we take our fundamental units of mass, energy and length to be $m = \epsilon = \sigma = 1$ respectively, similarly we choose our FENE constants $\kappa = 30, R_0 = 1.5$. We choose a warm up period (§5.1) until $r_{fc} = 0.8\sigma$, and allow the simulation to run for certain length of time before data is recorded. This will allow time for the forces not considered during warm up to be introduced to the system. P3M (§5.3) is used to calculate charged forces, whilst all non-bonded forces are considered under periodic boundary conditions (§3.3). For LJ cutoff we use a combination of Verlet

list and cell list (§5.2.1) where we ensure our sub-box length $l > r_c$. Considering certain simulations will use multiple values for r_c - it is important that the largest of these is compared with sub-box length. Hence the general form of our equation of motion, including frictional and noise terms (§2.4) can be written as

$$\begin{aligned}\mathbf{F}_i = &- 30\left[(\mathbf{r}_i-\mathbf{r}_{i+1})\frac{1}{1-(\frac{r_{i,i+1}}{R_0})^2}+(\mathbf{r}_i-\mathbf{r}_{i-1})\frac{1}{1-(\frac{r_{i,i-1}}{R_0})^2}\right]\\ &+ 24\sum_{\mathbf{n}}\sum_{j=1}^{N}{}'\epsilon_{ij}(\mathbf{r}_i-[\mathbf{r}_j+\mathbf{n}L])\left[\frac{2}{|\mathbf{r}_i-[\mathbf{r}_j+\mathbf{n}L]|^{14}}-\frac{1}{|\mathbf{r}_i-[\mathbf{r}_j+\mathbf{n}L]|^{8}}\right]^*\\ &- \frac{q_i}{4\pi\varepsilon_0\varepsilon_r}\sum_{\mathbf{n}}\sum_{j=1}^{N}{}'(\mathbf{r}_i-[\mathbf{r}_j+\mathbf{n}L])\frac{q_j}{|\mathbf{r}_i-[\mathbf{r}_j+\mathbf{n}L]|^3}-\gamma\dot{\mathbf{r}}_i+W_i.\end{aligned}$$

Again, this equation does not tell the whole story - but it is the most mathematical way of writing the equation of motion. The first line only applies to particles in the same chain as i, and if i is the last or first term in the chain then the left hand or right hand term must be dropped. On top of this, the LJ truncation in the second line (represented by *) will vary depending on the particles considered, as will ϵ_{ij}. The two quantities that will contribute mostly to the strength of the network formation are ε_r and ϵ, the latter of which will only contribute to association for particles with $r_c > 2^{\frac{1}{6}}$. These associative monomers are often described as 'sticky' monomers. The instantaneous stress and average functionality of each polymer in the resulting polymer network is computed on-the-fly; the theory behind these is discussed next.

6.2 The stress modulus

Typically in mechanics, stress is defined as the change in force responding to a deformation of the system. In our molecular dynamics simulations, we compute the stress autocorrelation function, defined as

$$G(t) := \frac{V}{k_B\mathcal{T}}\left\langle\varphi_{\alpha\beta}(t+t_0)\varphi_{\alpha\beta}(t_0)\right\rangle \tag{30}$$

(Zhou & Larson 2006)[21]. Here, α and β are taken to be each of x, y, z directions for the average. Interestingly it is not necessary to deform the system to measure this relaxation G(t); by the fluctuation dissipation theorem we can link fluctuations in equilibrium with forced perturbations from equilibrium - hence we are able to plot the relaxation $G(t)$ based on data taken whilst the system is in equilibrium. Every possible t_0 is considered for the average, to gain the smoothest possible correlation. $\varphi_{\alpha\beta}$ represents the shear stress, which is defined as

$$\varphi_{\alpha\beta} = \frac{1}{V}\left[-\sum_{i=0}^{N}m_i v_{\alpha i}v_{\beta i}-\sum_{i=0}^{N-1}\sum_{j=i+1}^{N}\frac{r_{\alpha ij}r_{\beta ij}F_{ij}}{|r_{ij}|}\right], \tag{31}$$

which can be physically interpreted as the α directional component of the force over a cross sectional area normal to β. Winkler (2002)[22] derives (31) after dividing the force by the volume and multiplying by one direction (hence we are instead dividing by a single plane), and then summing over all particles, in each plane. The resulting stress moduli are stored on-the-fly in the form of the stress tensor

$$\varphi = \begin{pmatrix} \varphi_{xx} & \varphi_{xy} & \varphi_{xz} \\ \varphi_{yx} & \varphi_{yy} & \varphi_{yz} \\ \varphi_{zx} & \varphi_{zy} & \varphi_{zz} \end{pmatrix}. \tag{32}$$

The diagonal entries correspond to the pressure calculation, which are typically much larger than the non-diagonal entries. Theoretically, the matrix should be symmetric, as one excepts that $\varphi_{\alpha\beta} = \varphi_{\beta\alpha}$. The average $\varphi_{\alpha\beta}$ are taken for each x, y, z, not including the diagonal (pressure) entries. The curve of the resulting stress modulus over a single interval $[t_0, t]$ is very noisy, so unlike most quantities which only need to be recorded every $100\Delta t$ or so, it is necessary to record the values of φ at every time step. Once the simulation is complete, the results of the stress tensor (32) are correlated using the multiple Tau correlator (discussed in §5.5) and then used to effectively compute the stress modulus $G(t)$, which will give a physical insight into the relaxation of the polymer system after deformation.

6.2.1 The Ewald problem

For a charged system using the Ewald method, a problem arises due to the form of the Ewald equation (29); the long range interactions are not calculated pairwise. Hence, it is impossible to find F_{ij}, used in (31) to calculate the stress modulus from the expression for $G(t)$ given in (30). It would seem that such an equation should exist, as it is purely the form of equation (29) which is preventing the calculation - in a small closed system the charges could all be calculated pairwise, and the stress could be calculated. In fact, a method has been derived by Winkler (2002)[22] to calculate the diagonal elements (pressure) of the stress tensor in conjunction with the Ewald method, but it seems there exists no method for the remaining elements of the matrix - at least for our P3M method. Furthermore, as discussed by Cao & Likhtman (2010)[23], in certain cases the stress tensor is proportional to the orientation tensor - where the orientation tensor of a polymer j is defined as

$$O_{j,\alpha\beta} = \sum_{i=0,\text{ on j}}^{n} \bar{r}_{i\alpha}\bar{r}_{i\beta},$$

where the notation defined in §4.4 is used. However, the reasoning behind this proportionality is still somewhat unclear, and further experimentation would be required before the theory could be applied to a charged system. Rather than attempting to derive a formula which will calculate the stress of Burattini's π-π system, we simply adopt a different model - one more similar to that used to model Cordier's polymers - consisting of only U^*_{LJ} and U_{Fene}. Hence, the charged

monomers are replaced by monomers with an attractive LJ tail. Coincidently, the University of Reading chemistry department now suggest that the association of Burattini's self-healing polymers is driven by short-ranged, non-electrical interactions; it may follow that this model is more effective after all. The model we have created *is* still capable of modelling charged systems - regrettably we do not make full use of this in the simulations that follow.

6.3 Network analysis

To analyze the network formed by the associativity of the polymers, we compute the $M \times M$ binary adjacency matrix A, corresponding to the system of $M = P_X + P_Y$ polymers. We introduce the cluster cutoff r_{nc} (typically 1.5) and the elements a_{mn} of the matrix are formed from the condition

$$a_{mn} = \begin{cases} 1, & m \neq n \quad \& \quad \exists\, i, j, \mathbf{n} : |\mathbf{r}_i - [\mathbf{r}_j + \mathbf{n}L]| < r_{nc},\ i \text{ and } j \text{ 'associating'}, \\ 1, & m = n, \\ 0, & \text{otherwise}, \end{cases}$$

where i and j are monomers on polymer m and n respectively. Hence two linked polymers are represented by a '1' in A. Naturally, if polymers m and n are linked, it will follow that n and m are linked - so A must be symmetric[11]. To effectively analyze the network, every polymer is considered to be linked with itself. To determine the strength of our network, we will compute the **functionality** f of the associative monomers - which is the average number of other monomers that each associative monomer is associating with. Another quantity of interest is the cluster size, which is the number of polymers in each disjoint network. To compute the size of each disjoint cluster (the sum of these must equal M), we reduce A by merging columns n and j that share a 1 (the choice of rows or columns is arbitrary by the symmetry of A), written mathematically as

$$a^*_{mn} = \begin{cases} \max(a_{mn}, a_{mj}), & \exists\, i : a_{in} = a_{ij} = 1, \\ a_{mn}, & \text{otherwise}. \end{cases}$$

Physically this means that if the polymers represented by columns j and n are both linked to a polymer i, then j and n are at least linked indirectly, via i - it follows that they are in the same cluster. Therefore, any polymer linked to j is also linked to n, the columns are then merged; they now share connections. In computation, the algorithm tries each possible j for each n, and the column n is then removed after merging with column j. Continuing this process will result in an $M \times C$ matrix A^* where C is the number of disjoint clusters in the system.

[11] In fact, polymers in the same cluster network can be treated as an equivalence relation. Even if they are not related directly, they are linked by some number of polymers between them (*transitivity*). Furthermore, each polymer is related to itself (*reflexivity*) and A is symmetric (*symmetry*).

Furthermore, the sum of the entries in each column represents the number of polymers in each cluster. In computation this data is recorded on-the-fly to an array of size M. After simulation, this can be plotted as a representation of the probability of the distribution of the polymers amongst clusters of the system. When written mathematically the process may seem more complicated than it actually is, the following example demonstrates the idea on a basic level.

6.3.1 Example: Adjacency matrix of a simple system

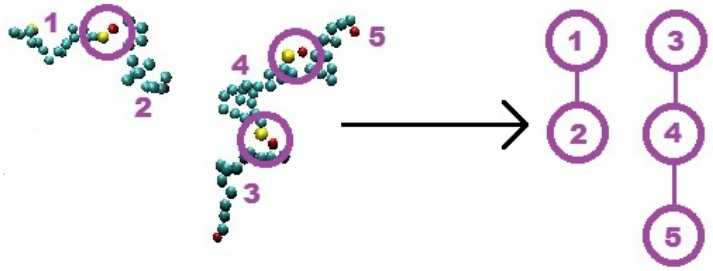


Figure 10: Mapping the topology of a polymer system to a network.

Consider the system of 5 polymers represented by figure 6.3.1. The polymers are indexed from 1 to 5, and the circles represent attractive monomers whose separation $r_{ij} < r_{nc}$ and are considered to be associating. The graph represents the network formed between the associating polymers. The corresponding adjacency matrix A is binary and symmetric, and is given by

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

where each entry '1' represents an association. This can be reduced to represent individual clusters using the method described in §6.3, yielding

$$A^* = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}.$$

Notice how polymer 3 and 5 are now 'connected', based on their indirect connection via polymer 4. The columns of the matrix can be summed, yielding the result (which was actually fairly clear from the diagram) of 2 disjoint clusters, one consisting of 2 polymers and the other consisting of 3.

6.4 Self-healing polymer networks

We analyze the results of 3 simulations, under the following parameters:

- Cordier's polymers are simulated with $N_X = N_Y = 17, P_X = P_Y = 100$, with associative monomers at the end of each chain ($d_X = d_Y = 15$), to represent the hydrogen bonding. Notice the symmetry of this simulation, any result applied to an X-type polymer should also apply to a Y-type polymer.
- Burattini's rubber is simulated with $N_X = 36, N_Y = 9, P_X = 20, P_Y = 80$, with $d_X = 4, d_Y = 7$ - modelling the π-π stacking between diimide groups and pyrenyl units. Relatively, there are more associative monomers in this simulation than Cordier's, whilst the total number of monomers N in the central box is halved.
- The neutral polymer melt consists of $N_X = N_Y = 17, P_X = P_Y = 100$, with no associative monomers. It is worth noting that this is the same as Cordier's polymers but without association and is therefore a useful comparison.

We aim to compare the topology and the strength of the networks formed in these simulations.

Using the methods described in §6.3 and taking $r_{nc} = 1.5$, we produce the cluster network distribution formed by the 2 self-healing polymer simulations. Clearly the polymer melt will have no associating monomers, so it will not form clusters. Data is taken after $10^4\Delta t$ to allow time for equilibrium, and over a period of $10^5\Delta t$ - recording relevant information every $100\Delta t$.

6.4.1 Cordier's network

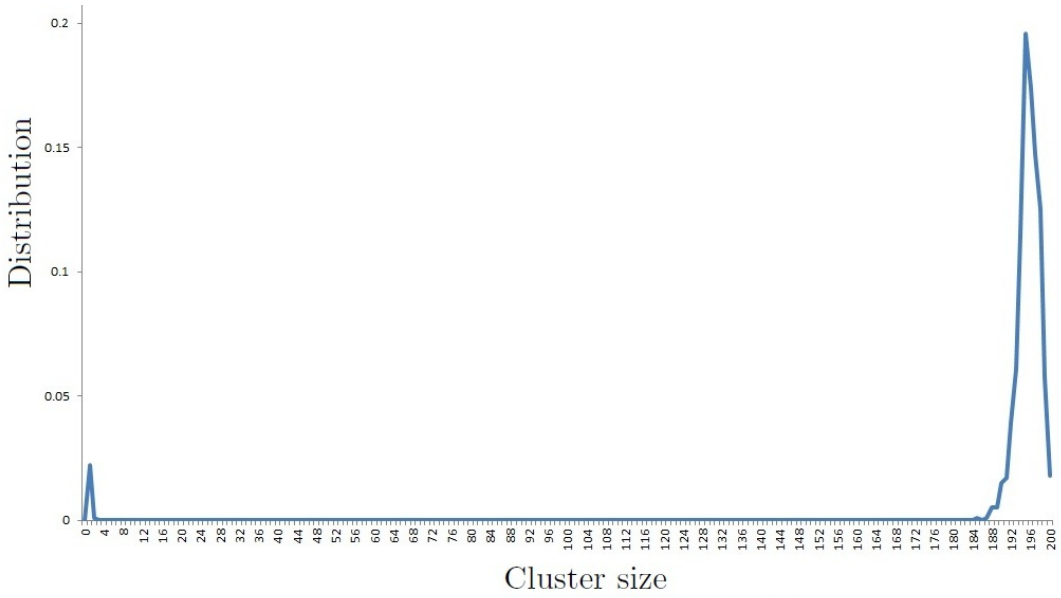


Figure 11: Normalized distribution of clusters in Cordier's network.

Figure 11 strongly suggests that of the 200 polymers modeled in the central box, the most likely network consists of 195 polymers in one cluster, and 5 disjoint clusters consisting of one polymer each. As the peaks on the graph are very thin, it would seem that this is often the case. Something that must be considered; this data suggests that the most likely cluster size is 195 polymers, based on a system of 200 polymers. However, in a system of 400 polymers we would not expect 2 disjoint clusters of 195 polymers each - instead it seems more likely that roughly 97.5% of the polymers in the system will be in the same network. Qualitative evidence to suggest this can be gained from figure 9 - clearly the cluster is connected to itself via the periodic boundary conditions; the network passes through opposite faces of the simulation box, forming a percolated network. Therefore, if we consider any number of repeated images of the central box, this would still represent a network in which 97.5% of the polymers are connected.

6.4.2 Burattini's network

Under exactly the same conditions as Cordier's network, we compute the cluster distribution for Burattini's network.

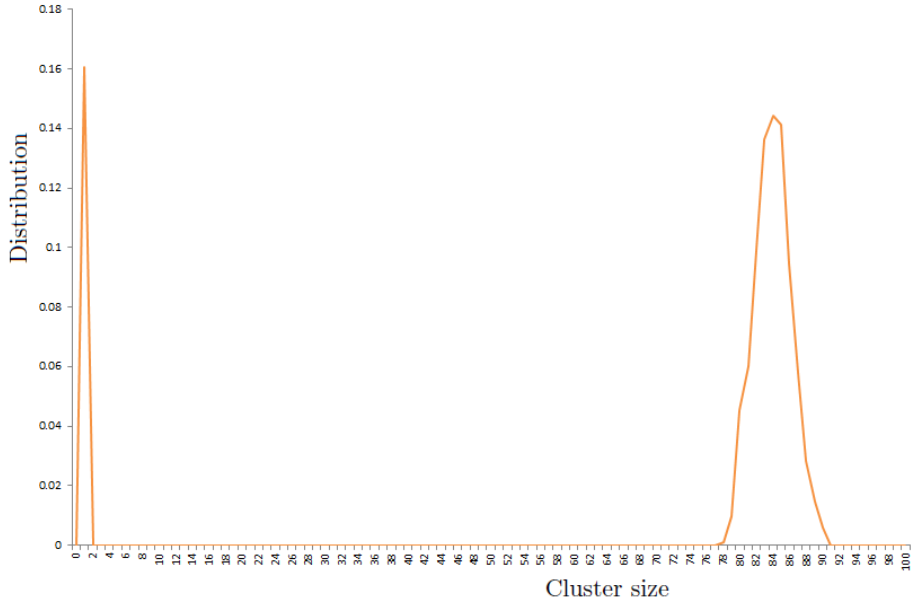


Figure 12: Cluster distribution for Burattini's network

Figure 12 seems to exhibit a larger distribution of polymers that are not associating with any others. One reason for this may arise from the nature of the compact clusters resulting from associations. This will make it harder for smaller clusters to associate with each other and form large clusters, as there will be a greater separation between clusters than in Cordier's case. It is still reasonable to assume that an infinite network is formed, as clearly most of the polymers lie in one large cluster. Notice that the right hand peak is wider than exhibited by Cordier's rubber in figure 11, suggesting more regular fluctuations in cluster size (although the horizontal axis scale is only half as long).

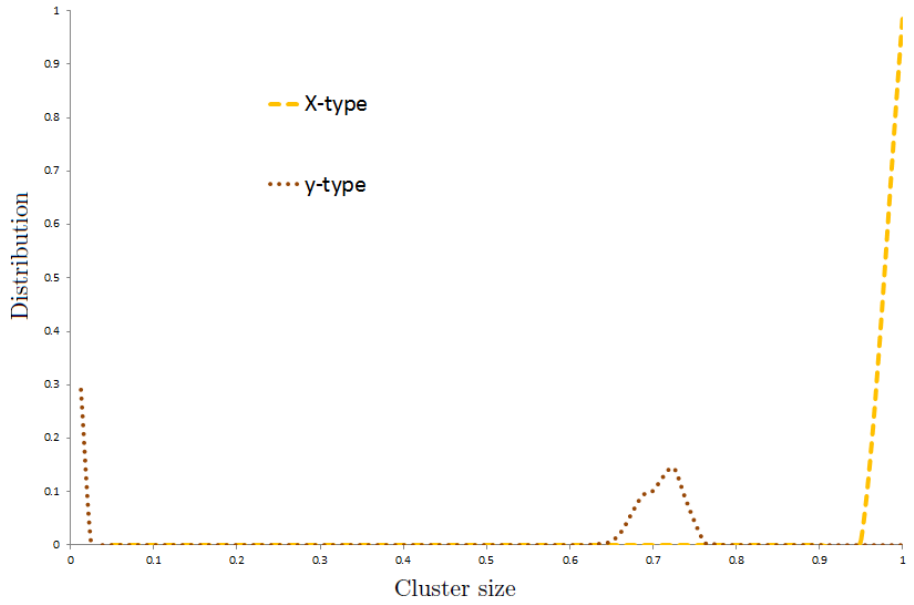


Figure 13: Cluster size of both types of Burattini's polymers. The result has been normalized for clarity, as $P_X \neq P_Y$.

To determine the contributions of each polymer type to the left hand peak of figure 12, we compute the individual cluster contributions over the same sample. To do so, when summing the columns of the adjacency matrix A^*, we sum the first P_X rows into one array, and the the remaining P_Y rows into a second. From figure 13 it is clear that all of the X-type polymers lie in one large cluster, whilst many of the Y-type polymers are not associating with this cluster. It would be beneficial to test this network formation over a larger sample of data, as this may give different results, allowing the Y-type chains longer to associate with the X-type chains in the cluster.

6.5 Functionality results

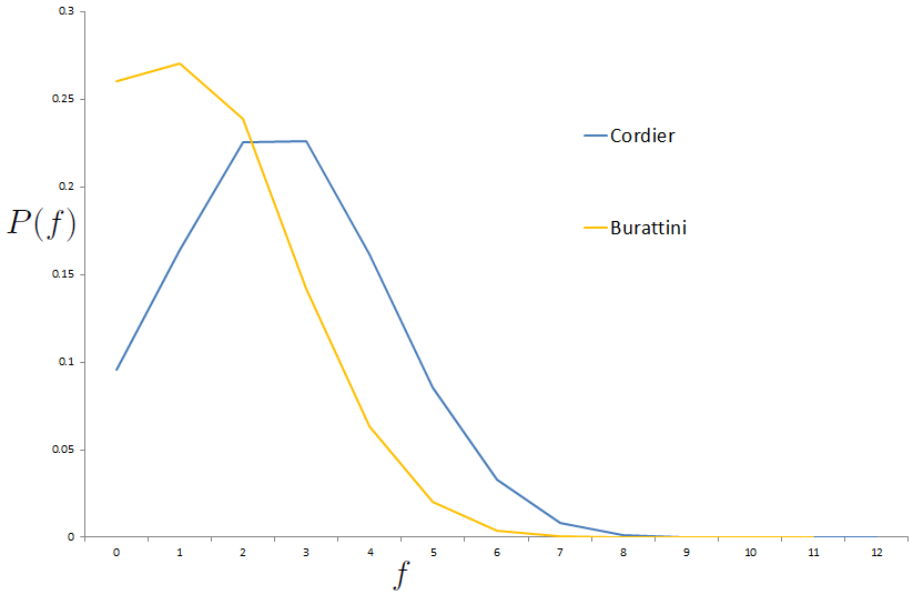


Figure 14: Functionality distribution f of both polymer networks

We consider the functionality of both systems displayed in figure 14, taken over the same sample of data as the cluster network distributions. This distribution suggests that in Cordier's network, most sticky monomers are associating with 2 or 3 others, whilst in Burattini's case, most sticky monomers are only associating with 1 other. Burattini's network exhibits a higher number of sticky monomers which are not associating, which seems logical based on the cluster distribution - many Y-type polymers are disjoint from the main cluster. As Cordier's network exhibits a higher average functionality, this suggests a stronger network than Burattini's. This result is compared with the stress modulus (an alternative measure of network strength) in §6.7.

6.6 Diffusion results

Using the coordinate data written out during simulation, we compute the mean squared displacement of the individual monomers. The mean squared displacement of the entire chain follows the relationship

$$6(t - t_0)D = \left\langle |\mathbf{r}(t - t_0) - \mathbf{r}(t_0)|^2 \right\rangle,$$

where D is the diffusion coefficient of the chain (Frenkel & Smit 1996)[5].

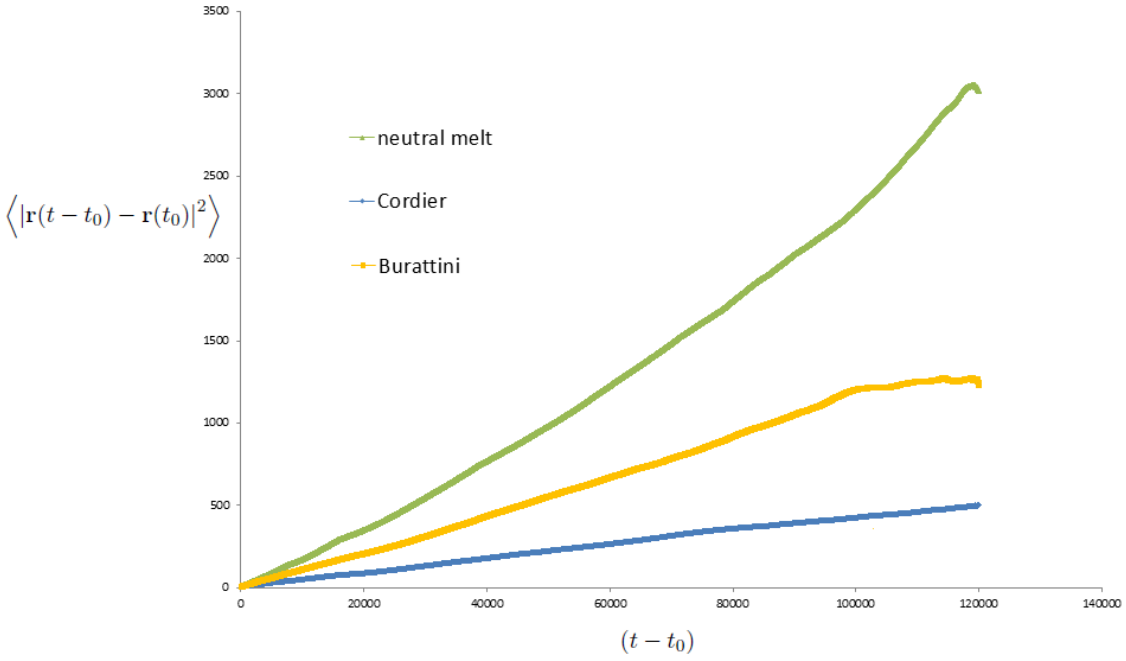


Figure 15: Graph of the mean squared displacement of both self-healing polymer networks compared with a neutral polymer melt.

The results in figure 15 suggest a more rapid diffusion in the polymer melt case, which is to be expected as the attractive monomers will restrict diffusion in the self-healing cases. Burattini's polymers are shown to diffuse faster than Corder's. This is most likely due to the larger number of polymers in Burattini's system which are not associated with the main cluster - these are free to diffuse faster, which pulls up the average value. Whilst there is not a sufficient sign of a plateau in the self-healing case, there is perhaps some evidence of a limit in Burattini's polymers. It could be that this limit will become clear at a later stage, but the diffusion has clearly not reached its limit in the sample taken; a larger sample must be taken before conclusions can be drawn. This limit would represent a restriction on the space in which individual monomers are able to move, imposed by the

connection of certain monomers in the chain to other polymers inside the main cluster. Alternatively we could restrict our calculations to only the largest cluster, thus avoiding contributions from these smaller, fast-moving polymers which have not associated with this main cluster.

6.7 Stress modulus results

In each simulation, we compute the stress auto-correlation function using the multiple-τ correlator. Due to time and hardware constraints it is not always possible to take sufficient statistical data to obtain a smooth correlation - in this case the data was taken over $10^7 \Delta t$ of each simulation.

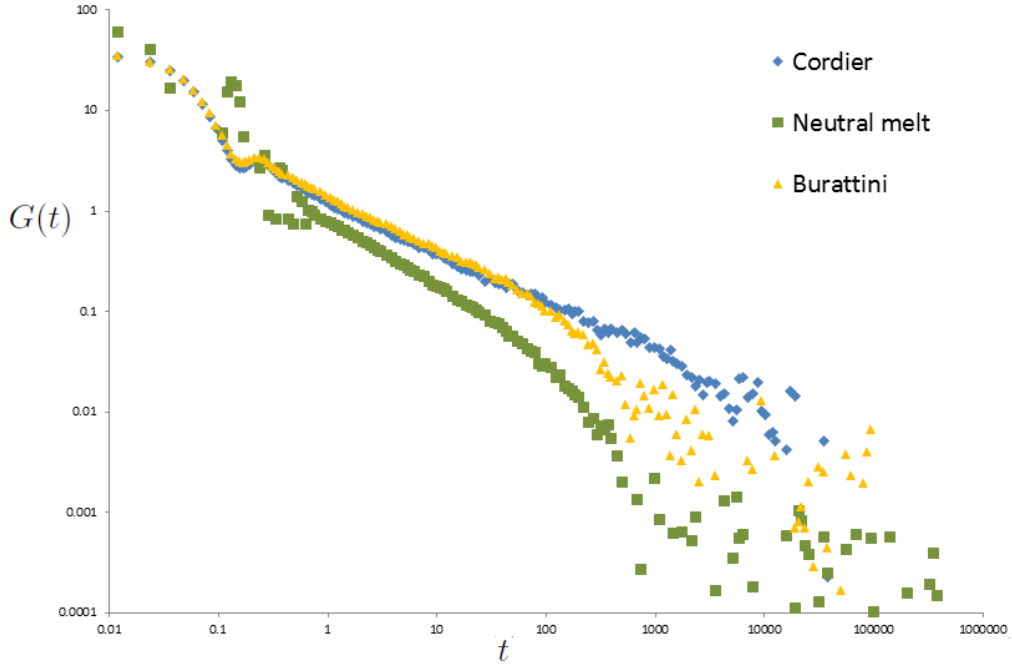


Figure 16: $G(t)$ of each simulation.

Figure 16 suggests that the self-healing polymers have a slower relaxation than the polymer melt case, as would be expected for a more complicated network. Note specifically that the only difference between the polymer melt and Cordier's rubber is the attractive LJ-tail of the end monomers on the 17 monomer chain. Furthermore, the polymers of both Burattini and Cordier relax similarly, despite Cordier's network having a higher functionality. This contradicts the theoretical predictions of the ideal case - Rubinstein & Colby (2003)[9] describes a 'phantom model' of ideal chains, predicted to satisfy

$$G = \frac{Nk_BT(f-2)}{fV}.$$

The relationship between the functionality f and the stress modulus G of our real chains is clearly more complex, due to the additional constraint of excluded volume. Considering the data for $G(t)$ was taken over a much wider range than for functionality f, Burattini's network may become stronger over this time, resulting in this behavior similar to Cordier's. As $G(t)$ represents the time taken to reach equilibrium from a deformation, it would be interesting to investigate the relationship between $G(t)$ and the healing time after a (macroscopic) cut in the material. To verify this would require more complex simulations, it would again be beneficial to restrict our calculations to the largest cluster.

6.8 Summary

Using the MD simulation model created, we have results that show a polymer network formed by both types of self-healing polymer - with an infinite network formed in both cases. The relaxation time for these self-healing polymers is significantly longer than that of the polymer melt. We have tested the strength of self-healing networks of non-ideal chains through comparison between the stress modulus and functionality. Further experimentation could involve simulating a cut in the system, and recalculating the stress modulus - as Burattini et al. (2010)[3] claim their elastic modulus decreases to 95% after the cut. It is also necessary to run these simulations over a much wider range to obtain more reliable results, in particular to compare the functionality with the stress modulus more effectively. It would be interesting to see if the diffusion rate would also reach a limit in this larger interval, as the network should restrict movement of the monomers within it. The model developed has many applications, and is capable of modelling a far wider range of polymers than those discussed here. However, as is always the case with computational science - for these results to be considered reliable, it is necessary to run the simulation over a period of months, even years.

References

[1] White, S.R et al. Autonomic healing of polymer composites *Nature*, vol. 409, 2001 pp. 794-797

[2] Cordier et al. Self-healing and thermorevsersible rubber from supramolecular assembly, *Nature,* vol. 451, 2008, pp. 977

[3] Burattini et al, 'A healable supramolecular polymer blend based on aromatic π-π stacking and hydrogen-bonding interactions', *Journal of the American chemical society*, Vol. 132, 2010, pp.12051-12058.

[4] Allen, M.P & Tildesley, D.J 1987, *Computer Simulation of Liquids,* Oxford University Press, New York

[5] Frenkel, D & Smit, B 1996,*Understanding Molecular Simulation,* Academic Press, United States

[6] *Atomistic Computer Modeling of Materials, Lecture 15: Molecular Dynamics* video lecture recording, MIT Nicola Marzari, viewed 17th September, 2010 <http://deimos3.apple.com/WebObjects/Core.woa/Feed/mit.edu-dz.2820710563.02820710565> Massachusetts institute of technology

[7] Kremer, K & Grest, G.S 'Dynamics of entangled linear polymer melts: A molecular-dynamics simulation', *Journal of chemical physics*, Vol. 92, 1990 pp.5057-5086

[8] Felderhof, B.H 1978 'On the derivation of the fluctuation-dissipation theorem', *Journal of Physics A: Mathematical and General*, vol. 11, No. 5, pp. 921-927

[9] Rubinstein, M & Colby, R.H 2003, *Polymer Physics,* Oxford University Press, New York

[10] Jackson, J.D 1975 *Classical Electrodynamics (Second Edition)*, John Wiley & Sons, Inc.

[11] Press et al. *Numerical Recipes in C, Second Edition*, Cambridge University Press

[12] Flory, P 1953 *Principles of Polymer Chemistry*, Cornell University Press

[13] Helmenstine, A.M 2007 *How Diapers Work & Why They Leak*, viewed 20th Marth 2011, <http://chemistry.about.com/b/2007/02/06/how-diapers-work-why-they-leak.htm>

[14] Auhl et al. 'Equilibrium of long chain polymer melts', *The Journal of chemical physics,* vol. 119, No. 24, pp. 12723-12723

[15] Auerbach et al. 1987 'Research Article A special purpose parallel computer for molecular dynamics: motivation, design, implementation, and application', *Journal of physical chemistry*, vol. 91, No. 19, pp. 48814890

[16] Le, H & Cai, W 2009 *Ewald Summation for Coulomb Interactions in a Periodic Supercell*, viewed 07th February 2011, Standford University, 'http://micro.stanford.edu/mediawiki/images/4/46/Ewald_notes.pdf'

[17] Benson, D 2008, *Music: a Mathematical Offering*, Cambridge University Press, Available online via University of Edinborough

[18] *Password* 2007, '25 years of the compact disc', viewed 18th march 2011, p.28
<http://www.research.philips.com/downloads/
password/download/password_30.pdf>

[19] Ramírez et al. 2010, 'Efficient on the fly calculation of time correlation functions in computer simulations'*Journal of chemical physics*, vol. 133, pp. 154103-1 - 154103-12

[20] Tyrangiel, J 2009, *Auto-Tune: Why Pop Music Sounds Perfect*, viewed 10th March 2011,
<http://www.time.com/time/magazine/article/0,9171,1877372,00.html>

[21] Zhou, Q & Larson, R.G 2006 'Direct Calculation of the Tube Potential Confining Entangled Polymers', *Macromolecules*, vol. 39, No. 19, pp. 67376743

[22] Winkler, R.G 2002 'Viral pressure of periodic systems with long range forces', *The Journal of chemical physics,* vol. 117, No. 5, pp.2449

[23] Cao, J Likhtman, A.E 2010, 'Time-Dependent Orientation Coupling in Equilibrium Polymer Melts', *American Physical Society*, vol. 104, No. 20, pp. 207801(1-4)

APPENDICES

A Single polymer in solution code

This code was written in the C. This is the code accompanying most of §4. Many different versions of it were used, but this is the most general. This code is certainly not as efficient as it could be, but this lack of elegance makes it easier to follow. The code uses 3 random number generation functions taken from 'Numerical Methods in C'. It produces up to 5 data files, including a .pdb and .xyz file, which can be loaded into VMD software, available from www.ks.uiuc.edu/Research/vmd/ to produce a visualization of the simulation.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <stdlib.h>
#include <stddef.h>// WinApi header
//                      SINGLE POLYMER IN SOLUTION MOLECULAR DYNAMICS SIMULATION
#include "random.h"
//define functions//
void LJ(int pi, int pj);
void Spr(int pi, int pj);
void verlet();
double endtoend();
double radgy();
double rmsaccel();
double bondav();
void KE(int dj);
double cn(int nn);
int flag_spring;
double bondav2();
void Uspr(int pi, int pj, int dj);
void Ulj(int pi, int pj, int dj);
//void vmd_pdb(int pi);

long    idum;            /* Seed for the random number generator */
long    *idumPtr = &idum;  /* Pointer for the seed */

/*define global variables*/
int N;//number of particles//
double r[1000][3];
double v[1000][3];
double a[1000][3];
double kspr;                    //spring constant//
double depth;
double t;                //the current time in the simulation
int steps;       //total number of  steps that the program will run for
double dt;               //length of each time step//
double rij;              //particle separation between Pi and Pj
double rc;
double L;                 //length of simulation box
double r0;
double sprcutoff;
double KEt;      //correct kinetic energy based on temperature
double U[10002][3];    //bin for the potential energy every 100 time steps
int d;/*the entry in the potential array that is currently being calculated*/

int main()
{
    int i;
    int j;
    int k=0;
    double KEt;          //kinetic energy from temperature
    double KEv;
    FILE *file_ptr;
    FILE *file_ptr2;
    FILE *file_ptr3;
    FILE *file_ptr4;
    FILE *file_ptr5;
    double bond;      //separation of neighbouring particles at separation
    double r2;
    double modr;
    int fr;
    int rt;
    long seed;
```

```
    L=20.;              //length of simulation box
    KEv = 0.;            //kinetic energy from velocities
    N=30;
    t=0.;
    dt = 0.0012;
    steps=1000000;   /*total number of time steps that the
    program will run for,    typically 1 million*/
    rc=pow(2, 1/6);      /*use 2.5 as standard with depth/60 error,
                                                    2^{1/6} for purely repulsive*/
    depth=1;           //potential depth for LJ
    kspr=30;
    bond=1.;           //initial separation of neighbouring particles
    r0=1.5;             //maximum legnth of bonds between particles
    sprcutoff=3*sqrt(2)/2;
    KEt=1;                  //kinetic energy of system
    flag_spring=1;    /*when flag =1, entropic spring force is used
                                      - when flag=0,
                                         linear spring force is used*/
    //movie makin' data: framerate and runtime
    fr=10;
    rt=steps;
    d=0;


    //set random positions//


    printf("\nProgram start - assigning random positions to particles\n\n");
    for(i=0; i<N; i++)//set all initial particle accerlerations to zero
    {
        a[i][0]=0;
        a[i][1]=0;
        a[i][2]=0;
    }

    printf("\n!\n");
   for (i=0; i<N; i++)
    {
        v[i][0] = gasdev(idumPtr);
        v[i][1] = gasdev(idumPtr);
        v[i][2] = gasdev(idumPtr);
/*assigns a small random gaussian velocity to particle i in each direction*/
    }


        r[0][0] = 0;
        r[0][1] = 0;
        r[0][2] = 0;
    printf("\n!\n");
    for (i=1; i<N; i++)
        {
          k=0;

          r[i][0] = ran2(idumPtr)-0.5;
          r[i][1] = ran2(idumPtr)-0.5;
          r[i][2] = ran2(idumPtr)-0.5;
          //chooses a random number on [-0.5,0.5]

          printf("Particle %d:%f\t%f\t%f\t\n", i+1,
           r[i][0], r[i][1], r[i][2]);

          modr=sqrt((r[i][0]*r[i][0])+(r[i][1]*r[i][1])
          +(r[i][2]*r[i][2]));
          if(modr==0){modr=1;}

          r[i][0] = r[i-1][0] + ((r[i][0]*bond) / modr);
          r[i][1] = r[i-1][1] + ((r[i][1]*bond) / modr);
          r[i][2] = r[i-1][2] + ((r[i][2]*bond) / modr);

          for(j=0; j<i; j++)
              {
              rij=((r[i][0]-r[j][0])*(r[i][0]-r[j][0]))
              +((r[i][1]-r[j][1])*(r[i][1]-r[j][1]))
              +((r[i][2]-r[j][2])*(r[i][2]-r[j][2]));
              if(rij<1)
                  {printf("Particle %d repositioned\n", i+1);
                  j=i;
                  i=i-1;
                  k=k+1;
                  }
              if(k==100)
                  {
```

```
                printf("\nRandom generation failed, try again");
                return 0;
                }
            }
        }

      /*Write out initial Pdb file*/
      file_ptr5 = fopen("Confinit.pdb", "w");
      for(i=0; i<N; i++)
      {fprintf((file_ptr5),"%-6s%5d%5s%4s%6d
      %8.3f%8.3f%8.3f%6.2f%6.2f\n", "ATOM  ",
       i+1, "CAA", "POL", 1, r[i][0], r[i][1], r[i][2], 1.0, 20.000);}

      for(i=1; i<N; i++)
      {fprintf((file_ptr5),"%-6s%5d%5d\n", "CONECT", i, i+1);}

      fprintf(file_ptr5, "END\n\n");


      fclose(file_ptr5);

  for(i=0; i<N-1; i++)
  /*calculate acceleration at first time step based on particle positions*/
      {for(j=i+1; j<N; j++)
          {
          LJ(i, j);  //calculates lennard jones force between particle i and j.
          }

      j=i+1;
      //only neighbouring particles are considered for bond energy
      Spr(i, j);
      }

    for(i=0; i<N-1; i++)
                {
                for(k=i+1; k<N; k++)
                {
                        Ulj(i, k, d);
                      //initial LJ potential is calculated
                }

     k=i+1;//only neighbouring particles are considered for bond energy

            Uspr(i, k, d);
            }
            KE(d);//initial kinetic energy

    printf("\n\nInitialization complete, simulation now in progress");


    file_ptr = fopen("positiondata.txt", "w");
    //opens file to record results
    file_ptr2 = fopen("U(t)_excelldata.txt", "w");
    file_ptr3 = fopen("Ree_exceldata.txt", "w");
    file_ptr4 = fopen("Conf.xyz", "w");

    fprintf((file_ptr3), "For N=%d\n\n", N);

    for(j=0; j<=steps; j++)//main program loop//
    {

            if(j==steps/4)
            printf("\n25%% complete");
            if(j==steps/2)
            printf("\n50%% complete");
            if(j==steps*3/4)
            printf("\n75%% complete");
//selects frames for movie:
        if(j%fr==0 && j<rt)
        {   fprintf(file_ptr4, "%d\n", N);
            fprintf(file_ptr4, "%d\n", j);
            for(i=0; i<N; i++)
            {
            fprintf(file_ptr4, "%10d%15.6f%15.6f%15.6f\n", 1, r[i][0],
            r[i][1], r[i][2]);
                        }
        }
            if(j%20000==0)//info is only recorded every 2000 time steps
            {   t=dt*j;//calculates time elapsed
                fprintf(file_ptr, "\n\nt = %.1f = %ddt\n", t, j);
                //records time to file

                for(i=0; i<N; i++)
```

```
                    {
                    fprintf(file_ptr, "\t%.2f\t%.2f\t%.2f\n", r[i][0],
                    r[i][1], r[i][2]);
  /*records the position of each particle i to the
  data string every 2000 time steps*/
                    }
            }


            if( j%(steps/100)==0)
            {
                fprintf((file_ptr3), "%f\n", endtoend());

            }

                verlet();
                //calculates coordinates of particles for next loop//

            if( j<=9998)
            {
                d++;
                for(i=0; i<N-1; i++)
                {
                    for(k=i+1; k<N; k++)
                    {
                            Ulj(i, k, d);
                    }

          k=i+1;
          //only neighbouring particles are considered for bond energy

                Uspr(i, k, d);
                }
                KE(d);
            }
        }
   fprintf(file_ptr2, "N=%d\tUlj\t\tUspr\t\tU\t\tK\n\n", N);

    for(i=0; i<=d+2; i++)
        {
        fprintf(file_ptr2, "%d\t%f\t%f\t%f\t%f\n",
        i, U[i][0], U[i][1], (U[i][0]+U[i][1]), U[i][2]);
        }

    fclose(file_ptr);
    fclose(file_ptr2);
    fclose(file_ptr3);
    fclose(file_ptr4);
    printf( "\n\nSimulation complete, results stored\n\n" );
    Beep(440,1000);
    Beep(880,1000);
    // Beeps to signal the end of the simulation

    return 0;
}

void LJ(int pi, int pj)//calculates lennard jones force
{
    double rij[3];
    double r2;
    double modr;

    rij[0]=r[pi][0]-r[pj][0];
    rij[1]=r[pi][1]-r[pj][1];
    rij[2]=r[pi][2]-r[pj][2];

    r2=(rij[0]*rij[0])+(rij[1]*rij[1])+(rij[2]*rij[2]);
    /*square of distance of particles i and j*/

if(r2<=rc*rc)
{
  a[pi][0] = a[pi][0] + (24*depth*rij[0]*((2/pow(r2, 7))-(1/pow(r2, 4))));
  /*updates acceleration for m=1, particle i*/
  a[pi][1] = a[pi][1] + (24*depth*rij[1]*((2/pow(r2, 7))-(1/pow(r2, 4))));
  a[pi][2] = a[pi][2] + (24*depth*rij[2]*((2/pow(r2, 7))-(1/pow(r2, 4))));

      if(pj<(2*N))
          {
    a[pj][0] = a[pj][0] - (24*depth*rij[0]*((2/pow(r2, 7))-(1/pow(r2, 4))));
 //updates acceleration for m=1, particle j
    a[pj][1] = a[pj][1] - (24*depth*rij[1]*((2/pow(r2, 7))-(1/pow(r2, 4))));
    a[pj][2] = a[pj][2] - (24*depth*rij[2]*((2/pow(r2, 7))-(1/pow(r2, 4))));
//no square rooting :)
```

```
            }
        }
}

void Spr(int pi, int pj)
{
    double rij[3];
    double fene;
    double modr;

    rij[0]=r[pi][0]-r[pj][0];
    rij[1]=r[pi][1]-r[pj][1];
    rij[2]=r[pi][2]-r[pj][2];

    if(flag_spring==1)
    {modr=sqrt((rij[0]*rij[0])+(rij[1]*rij[1])+(rij[2]*rij[2]));

    fene = (kspr*modr)/(1-((modr/r0)*(modr/r0)));

    a[pi][0] = a[pi][0] - fene*rij[0];
    //calculates accel due to spring potential for each i and j, m=1
    a[pi][1] = a[pi][1] - fene*rij[1];
    a[pi][2] = a[pi][2] - fene*rij[2];

    a[pj][0] = a[pj][0] + fene*rij[0];
    a[pj][1] = a[pj][1] + fene*rij[1];
    a[pj][2] = a[pj][2] + fene*rij[2];
    }
    else
    {
    a[pi][0] = a[pi][0] - kspr*rij[0];      //old style spring potential
    a[pi][1] = a[pi][1] - kspr*rij[1];
    a[pi][2] = a[pi][2] - kspr*rij[2];

    a[pj][0] = a[pj][0] + kspr*rij[0];
    a[pj][1] = a[pj][1] + kspr*rij[1];
    a[pj][2] = a[pj][2] + kspr*rij[2];
    }
}

void verlet()
{
double rij;
int i;
int j;

for(i=0; i<N; i++)
{
    v[i][0] = v[i][0] + (dt*0.5*a[i][0]);
    v[i][1] = v[i][1] + (dt*0.5*a[i][1]);
    v[i][2] = v[i][2] + (dt*0.5*a[i][2]);
    //stage 1: half step velocity is calculated for particle i//


    r[i][0] = r[i][0] + (v[i][0]*dt);
    r[i][1] = r[i][1] + (v[i][1]*dt);
    r[i][2] = r[i][2] + (v[i][2]*dt);
    //stage 2: new position is calculated using half step velocity//
}


   for(i=0; i<(2*N); i++)
   /*set all particle accerlerations to zero,
   before acceleration is recalculated*/
        {
        a[i][0]=0;
        a[i][1]=0;
        a[i][2]=0;
        }


    for(i=0; i<N-1; i++)
    /*calculate acceleration at current time step
    based on particle positions*/
        {
            for(j=i+1; j<N; j++)
            {
                    LJ(i, j);
            }

       j=i+1;//only neighbouring particles are considered for bond energy

                Spr(i, j);
```

```
        }
/*stage 3: acceleration has been derived (again)
based on new positions of particles*/

    for(i=0; i<N; i++)
    {
        v[i][0] = v[i][0] + (0.5*dt*a[i][0]);
    }//stage 4: new velocity is calculated, function is complete.
    }


double radgy()
    {
    int i=0;
    double com[3]={0,0,0};     //centre of mass
    double sum=0;

    for(i=0; i<N; i++)
        {
        com[0] = com[0] + r[i][0];
        com[1] = com[1] + r[i][1];
        com[2] = com[2] + r[i][2];
        }

    com[0] = com[0]/N;
    com[1] = com[1]/N;
    com[2] = com[2]/N;     //computes centre of mass 'com'//


    for(i=0; i<N; i++)
    {
        sum = sum + ((com[0]-r[i][0])*(com[0]-r[i][0]))
        + ((com[1]-r[i][1])*(com[1]-r[i][1]))
        + ((com[2]-r[i][2])*(com[2]-r[i][2]));
        //adds distance of particle from centre of mass to sum
    }
    sum=sum/N;
    return (double) (sum);  //returns Rg^2
}


double endtoend()
{
    return ((r[0][0]-r[N-1][0])*(r[0][0]-r[N-1][0]))
    +((r[0][1]-r[N-1][1])*(r[0][1]-r[N-1][1]))
    +((r[0][2]-r[N-1][2])*(r[0][2]-r[N-1][2]));
}

void KE(int dj)
{
    int i;
    int j;
    double KEv=0;     //actual kinetic energy based on velocities

    for(i=0; i<N; i++)
    //calculate current kinetic energy, based on random velocities above//
        {
         KEv = KEv + (0.5*( (v[i][0]*v[i][0]) +
         (v[i][1]*v[i][1]) + (v[i][2]*v[i][2]) ) );
        }
U[dj][2]=KEv;
}


double bondav()
    {
    double sum=0;
    int i;

    for(i=1; i<N; i++)
        {
        sum=sum+sqrt(((r[i][0]-r[i-1][0])*(r[i][0]-r[i-1][0]))
        + ((r[i][1]-r[i-1][1])*(r[i][1]-r[i-1][1]))
        + ((r[i][2]-r[i-1][2])*(r[i][2]-r[i-1][2])));
        }
    return sum/(N-1);
    }

double bondav2()
    {
    double sum=0;
    int i;
```

```
    for(i=1; i<N; i++)
        {
        sum=sum+(((r[i][0]-r[i-1][0])*(r[i][0]-r[i-1][0]))
        + ((r[i][1]-r[i-1][1])*(r[i][1]-r[i-1][1]))
        + ((r[i][2]-r[i-1][2])*(r[i][2]-r[i-1][2])));
        }
    return sum/(N-1);
    }

double cn(int nn)
/*nn is the number of particles away in the
 chain the two monomers are from each other*/
    {
    double rij2=0;
    int i;
    int m=0;  //number of bonds being averaged over
    double sum=0;

    for(i=0; i<N; i++)
        {
        if(i+nn<N)
        /*if the two particles are not too far
        apart to both lie in the N monomer chain*/
            {
            sum = sum + ((r[i][0]-r[i+nn][0])*(r[i][0]-r[i+nn][0]))
            + ((r[i][1]-r[i+nn][1])*(r[i][1]-r[i+nn][1]))
            + ((r[i][2]-r[i+nn][2])*(r[i][2]-r[i+nn][2]));
            /*sums distance between all pairs of particles that are
            nn particles apart in chain*/
            m++;
            }
        }
    rij2 = (double) sum/(nn*m);
    return rij2;
    }


double ran1(long *idum)
{
  int j;
  long k;
  static long iy=0;
  static long iv[NTAB];
  double temp;

  if (*idum <= 0 || !iy) {
    if (-(*idum) < 1) *idum=1;
    else *idum = -(*idum);
    for (j=NTAB+7;j>=0;j--) {
      k=(*idum)/IQ;
      *idum=IA*(*idum-k*IQ)-IR*k;
      if (*idum < 0) *idum += IM;
      if (j < NTAB) iv[j] = *idum;
    }
    iy=iv[0];
  }
  k=(*idum)/IQ;
  *idum=IA*(*idum-k*IQ)-IR*k;
  if (*idum < 0) *idum += IM;
  j=iy/NDIV;
  iy=iv[j];
  iv[j] = *idum;
  if ((temp=AM*iy) > RNMX) return RNMX;
  else return temp;
}

double gasdev(long *idum)
{
  //double ran1(long *idum);
  static int iset=0;
  static double gset;
  double fac,rsq,v1,v2;

  if  (iset == 0)
  {
    do {
      v1=2.0*ran2(idum)-1.0;
      v2=2.0*ran2(idum)-1.0;
      rsq=v1*v1+v2*v2;
    } while (rsq >= 1.0 || rsq == 0.0);
    fac=sqrt(-2.0*log(rsq)/rsq);
    gset=v1*fac;
    iset=1;
```

```
    return v2*fac;
  }
  else
  {
    iset=0;
    return gset;
  }
}

double ran2(long *idum)
{
    int j;
    long k;
    static long idum2=123456789;
    static long iy=0;
    static long iv[NTAB];
    double temp;

    if (*idum <= 0) {
        if (-(*idum) < 1) *idum=1;
        else *idum = -(*idum);
        idum2=(*idum);
        for (j=NTAB+7;j>=0;j--) {
            k=(*idum)/IQ1;
            *idum=IA1*(*idum-k*IQ1)-k*IR1;
            if (*idum < 0) *idum += IM1;
            if (j < NTAB) iv[j] = *idum;
        }
        iy=iv[0];
    }
    k=(*idum)/IQ1;
    *idum=IA1*(*idum-k*IQ1)-k*IR1;
    if (*idum < 0) *idum += IM1;
    k=idum2/IQ2;
    idum2=IA2*(idum2-k*IQ2)-k*IR2;
    if (idum2 < 0) idum2 +=IM2;
    j=iy/NDIV1;
    iy=iv[j]-idum2;
    iv[j] = *idum;
    if (iy < 1) iy += IMM1;
    if ((temp=AM1*iy) > RNMX) return RNMX;
    else return temp;
}


void Ulj(int pi, int pj, int dj)
{
double Pot=0;
double r2;
    r2=((r[pi][0]-r[pj][0])*(r[pi][0]-r[pj][0]))
    +((r[pi][1]-r[pj][1])*(r[pi][1]-r[pj][1]))
    +((r[pi][2]-r[pj][2])*(r[pi][2]-r[pj][2]));
    if(r2<(rc*rc))
        {

        Pot=4*depth*((pow(1/r2, 6))-(pow(1/r2, 3))) + 1;
        //shifted for continuous curve

       U[dj][0]=U[dj][0]+Pot;
        }
}

void Uspr(int pi, int pj, int dj)
{
double Pot=0;
double r2;
    r2=((r[pi][0]-r[pj][0])*(r[pi][0]-r[pj][0]))
    +((r[pi][1]-r[pj][1])*(r[pi][1]-r[pj][1]))
    +((r[pi][2]-r[pj][2])*(r[pi][2]-r[pj][2]));

    Pot=-kspr*r0*r0*0.5*log(1-(r2/(r0*r0)));
    //Pot= 0.5*kspr*r2;

    U[dj][1]=U[dj][1]+Pot;
}
```

A.1 Verlet test code

The code written to test the Velocity Verlet method in §3.1.1 is displayed below:

```
#include <stdio.h>
#include <math.h>

double t;    /*time*/
double dt;   /*timestep*/
double r;    /*position*/
double v;    /*velocity*/

void verlet();

int main()
{
r=1.;
dt=0.01;
v=2.;

FILE *file_ptr;
file_ptr = fopen("verlettest.txt", "w");
for(t=0.; t<=150.; t+=dt)
{
/*record time, exact solution, verlet solution*/
fprintf(file_ptr, "\n%f\t%f\t%f", t, cos(t)+(2*sin(t)), r);
verlet();
}
fclose(file_ptr);
return 0;
}

void verlet()
{
double a;
a=-r;
v=v+(0.5*dt*a);
r=r+(v*dt);
a=-r;
v=v+(0.5*dt*a);
}
```

B Polymer network code

This code was originally written by Dr Zuowei Wang (in C) to model multiple charged diblock polymers (of one type). For the benefit of this project it was adapted to model two types of polymer in the same simulation. The code consists of several C files and would be a tremendous waste of paper if it were included here. Instead, a few examples of the types of changes made are included below. This code also uses random number generators taken from 'Numerical methods in C', alongside the FFTW (fastest Fourier transform in the west) code for the ewald summation, where the P3M method is used. The multiple-tau method used to correlate the stress was written by Dr Zuowei Wang in Fortran.

B.1 Analysis functions

Below is the list of the majoriy of analysis functions added to Dr Wang's code to perform on-the-fly analysis. These also produces the pictures and frames required to visualize the simulation in VMD.

```
void write_out_pdb(FILE *FilePtr, FILE *FilePtr2)
{
int i, j, atom_nmb, res_nmb;
char *atom_name, *res_name;
double atomccp, B_value;
char *record_ID="ATOM  ";

atomccp = 1.0;
B_value=20.000;


fprintf(FilePtr2, "CRYST1%9.3f%9.3f%9.3f%7.2f%7.2f%7.2f P 1           1\n",
 10*box_l[0], 10*box_l[1], 10*box_l[2], 90., 90., 90.);/*records PBC data*/

    /*record to pdb file details of x-type polymers*/
    for(i=0; i<polyX.N; i++)
    {
         res_name="PLX";
         res_nmb=i+1;

         /*this section has been modified from the base code,
         all commented atom_names are the only ones that there'll be*/
         /*organic atoms are  S, O, N, */
         for(j=0; j<polyX.MPC; j++)
         {
              atom_nmb=i*polyX.MPC + j;

              if(part[atom_nmb].type==0)
              {atom_name="NXA";}          //  :)  neutrals
              else
              {atom_name="OXB";}          //      stickys

              /*scale up for video*/
         fprintf(FilePtr2, "%-6s%5d%5s%4s%6d    %8.3f%8.3f%8.3f%6.2f%6.2f\n",
         record_ID, atom_nmb+1, atom_name, res_name, res_nmb,
         10*fmod2(part[atom_nmb].r[0], box_l[0]), 10*fmod2(part[atom_nmb].r[1], box_l[1]),
         10*fmod2(part[atom_nmb].r[2], box_l[2]), atomccp, B_value);
         fprintf(FilePtr, "%-6s%5d%5s%4s%6d    %8.3f%8.3f%8.3f%6.2f%6.2f\n",
         record_ID, atom_nmb+1, atom_name, res_name, res_nmb, part[atom_nmb].r[0],
         part[atom_nmb].r[1], part[atom_nmb].r[2], atomccp, B_value);


         }
    }

    for(i=0; i<polyY.N; i++)
```

```
    {
        res_name="PLY";
        res_nmb=polyX.N + i + 1;

        for(j=0; j<polyY.MPC; j++)
        {   atom_nmb= polyX.N*polyX.MPC + i*polyY.MPC + j;

            if(part[atom_nmb].type==0)
            {atom_name="CYA";}          //  :)  neutrals
            else
            {atom_name="SYB";}      //stickys

            fprintf(FilePtr2, "%-6s%5d%5s%4s%6d    %8.3f%8.3f%8.3f%6.2f%6.2f\n",
             record_ID, atom_nmb+1, atom_name, res_name, res_nmb,
              10*fmod2(part[atom_nmb].r[0], box_l[0]), 10*fmod2(part[atom_nmb].r[1],
               box_l[1]), 10*fmod2(part[atom_nmb].r[2], box_l[2]), atomccp, B_value);
            fprintf(FilePtr, "%-6s%5d%5s%4s%6d    %8.3f%8.3f%8.3f%6.2f%6.2f\n",
            record_ID, atom_nmb+1, atom_name, res_name, res_nmb, part[atom_nmb].r[0],
            part[atom_nmb].r[1],part[atom_nmb].r[2], atomccp, B_value);
        }
    }




    for(i=0; i<cout.N; i++) /* NO PBC UPDATE FOR COUNTERIONS*/
    {
        res_name="PCN";
        atom_name="PCN";
        res_nmb=polyX.N*polyX.MPC + polyY.N*polyY.MPC + i ;
        atom_nmb=res_nmb;

        fprintf(FilePtr, "%-6s%5d%5s%4s%6d    %8.3f%8.3f%8.3f%6.2f%6.2f\n",
        record_ID, atom_nmb+1, atom_name, res_name, res_nmb+1,
        fmod2(part[atom_nmb].r[0], box_l[0]), fmod2(part[atom_nmb].r[1], box_l[1]),
        fmod2(part[atom_nmb].r[2], box_l[2]), atomccp, B_value);
    }
    /*set up bond links*/
    record_ID="CONECT";

    for(i=0; i<polyX.N; i++)
    {
        for(j=0; j<polyX.MPC-1; j++)
        {
        atom_nmb=i*polyX.MPC + j + 1;
        fprintf(FilePtr, "%-6s%5d%5d\n", record_ID, atom_nmb, atom_nmb+1);
        fprintf(FilePtr2, "%-6s%5d%5d\n", record_ID, atom_nmb, atom_nmb+1);
        }
    }

    for(i=0; i<polyY.N; i++)
    {
        for(j=0; j<polyY.MPC-1; j++)
        {
        atom_nmb=polyX.MPC*polyX.N + i*polyY.MPC + j + 1;
        fprintf(FilePtr, "%-6s%5d%5d\n", record_ID, atom_nmb, atom_nmb+1);
        fprintf(FilePtr2, "%-6s%5d%5d\n", record_ID, atom_nmb, atom_nmb+1);
        }
    }
    fprintf(FilePtr, "END\n\n");
}


/* Generate frames for VMD */
void write_out_xyz(FILE *FilePtr, FILE *FilePtr2, int istep)
{
    int i,part_index;

    /*bits for xyz*/
    fprintf(FilePtr, "%d\n",  N_Particle);
    fprintf(FilePtr, "%d\n", istep);

    /*bits for xyzPBC*/
    fprintf(FilePtr2, "%d\n",  N_Particle);
    fprintf(FilePtr2, "%d\n", istep);

    for(i=0; i<N_Particle; i++)
    {/*write out coords*/
     part_index=part[i].type+((int)part[i].q+1)*10;
    fprintf(FilePtr, "%10d%15.6f%15.6f%15.6f\n",
    part_index , part[i].r[0], part[i].r[1], part[i].r[2]);
```

```
    /*write out coords with PBC*/
    part_index=part[i].type+((int)part[i].q+1)*10;
    fprintf(FilePtr2, "%10d%15.6f%15.6f%15.6f\n",
    part_index , 10*fmod2(part[i].r[0], box_l[0]),
     10*fmod2(part[i].r[1], box_l[1]), 10*fmod2(part[i].r[2], box_l[2]));
    }

}

/*function to calculate modulus*/
double fmod2(double position, double length)
{
double newpos;
newpos=fmod(position, length);
if(newpos<0)
{newpos=newpos+length;}

return newpos;
}



/* List of instructions to implement this into other files:
Function is called JUST AFTER velocity verlet.
rf_part[n]+=da*db*force_over_dist; must be added to each force calculation loop
A global 'rf_part[9]' array will need to be defined


*/


void stress_calc(FILE *FilePtr, double ttime)
{
int i, a, b, n;     /*a and b correspond to alpha and beta from definition*/
double V;

V=box_l[0]*box_l[1]*box_l[2];   /*volume of box*/

/*calculate velocity part here, and add to rf to find sigma*/

for(i=0; i<N_Particle; i++)
{   n=0;
    for(a=0; a<3; a++)
    {for(b=0; b<3; b++)
        {
        rf_part[n]=rf_part[n]-((part[i].v[a]*part[i].v[b])/SQR(dt));
        /*divide velocity by dt*/
        n++;
        }
    }
}

for(n=0; n<9; n++)
{rf_part[n]=rf_part[n]/V;}  /*divide by volume to complete the calculation of shear stress*/

//printf("%16.8f%16.8f%16.8f%16.8f%16.8f%16.8f%16.8f%16.8f%16.8f\n",
ttime, rf_part[0], rf_part[1], rf_part[2], rf_part[3], rf_part[4],
 rf_part[5], rf_part[6], rf_part[7], rf_part[8]);
fprintf(FilePtr, "%16.8f%16.8f%16.8f%16.8f%16.8f%16.8f%16.8f%16.8f%16.8f\n",
 ttime, rf_part[0], rf_part[1], rf_part[2], rf_part[3], rf_part[4], rf_part[5],
  rf_part[6], rf_part[7], rf_part[8]);
/*records xyz data to file*/

for(n=0; n<9; n++)
{rf_part[n]=0.;}                    /*milked rf_part[] dry now - set it back to zero for the next time step*
}


void rec_functionality()
{
    /*add a global double clust_cut & integer arrays network[] & links[]
    that are both size N_Poly. It would be possible to have a 2D array,
    where the second part is size 2, and tallys the number of X and Y polymers*/

int N_poly=polyX.N+polyY.N;     /*total number of polymers*/
int connect[N_poly][N_poly];        /*connectivity matrix*/
int i, j, ip, jp;
int k, col, sum;
int a, b, c;        /*directions for PBC*/
int f;      /*functionality counter*/
//int f_flag;
printf("1");
sum=0;
```

```
/*initialize matrix with zeros everywhere*/
for(i=0; i<N_poly; i++)
    {for(j=0; j<N_poly; j++)
        {if(i==j)
            {connect[i][j]=1;}  /*always 1 on diagonal*/
        else
            {connect[i][j]=0;}
    }
}
f=0;
for(i=0; i<N_Particle; i++)
    /*PBC loops...?*/
    {for(j=0; j<N_Particle; j++)/*can't go from j=i+1 due to functionaity calculations*/
        {for(a=-1; a<2; a++)    /*over PBC*/
        {for(b=-1; b<2; b++)
        {for(c=-1; c<2; c++)                /*if two particles satisfy conditions of connectivity*/
            {if(((part[j].type+part[i].type)==3) && (SQR(part[i].r[0]-part[j].r[0]+
            (a*box_l[0]))+SQR(part[i].r[1]-part[j].r[1]+
            (b*box_l[1]))+SQR(part[i].r[2]-part[j].r[2]+(c*box_l[2]))<=SQR(clust_cut)))
            /*if satisfied, the two monomers and their corresponding
            polymers are connected in the network*/
            /*they must now be unpicked and their corresponding polymers
             must be found and officially linked*/
                {
                f++;        /*There's a link - so we add one to the functionality counter*/
                if(i<polyX.MPC*polyX.N)     /*if 'i' is on an X type polymer*/
                    {ip=i/polyX.MPC;}       /*index of polymer*/
                else                        /*if i is a Y type polymer*/
                    {ip=((i-(polyX.MPC*polyX.N))/polyY.MPC)+polyX.N;}

                if(j<polyX.MPC*polyX.N)     /*if 'j' is on an X type polymer*/
                    {jp=j/polyX.MPC;}       /*index of polymer*/
                else                        /*if j is a Y type polymer*/
                    {jp=((j-(polyX.MPC*polyX.N))/polyY.MPC)+polyX.N;}
                connect[ip][jp]=1;
                connect[jp][ip]=1;      /*probably don't need this as both sums go
                 over all particles, symmetry should simply fall out naturally*/
                }       /*~outside if statement*/
            }       /*~inner for loop*/
        }}}         /*~over PBC*/
    /* f is now equal to the functionality of monomer i */
if(part[i].type!=0)         /*obviously most monomers will not be able to associate -
these should not contribute to the functionality calcualations*/
{functionality[f]++;        /*bin for functionality of asociating monomers*/
}
f=0;

}       /*~outer for loop*/
/*sum up the number of connections of each polymer*/
sum=0;
for(i=0; i<N_poly; i++)             /*over all columns*/
{   for(j=0; j<N_poly; j++)         /*sum elements of each entry in that column*/
    {sum+=connect[i][j];}
sum--;                      /*subtract diagonal (self) term from links*/
links[sum]++;               /*records the total number of links*/
sum=0;
printf("2");
}



/*now reduce the matrix into minimum number of 'linked' columns*/

for(col=N_poly-1; col>=0; col--)    /*col is index of (current)
column of matrix being evaluated*/
    {for(j=0; j<N_poly; j++)            /*'col' is compared against this columns*/
        {for(i=0; i<N_poly; i++)        /*rows*/
            {if(connect[i][j]==1 && connect[i][col]==1 && col!=j)
            /*if the two polymers are 'linked'...*/
                {
                for(k=0; k<N_poly; k++)
                {if(connect[k][col]==1)
                /*...then link the first column with every polymer
                that the final polymer is linked with*/
                    {
                    connect[k][j]=1;
                    connect[k][col]=0;
                    }
                }
                i=N_poly;
/*breaks loop over rows to save time, as the two columns are identical now
 and this column won't need any more updating for the moment*/
```

```
                }
            }
        }
    }
printf("3");
for(j=0; j<N_poly; j++) /*columns, most will be zero by now*/
    {for(i=0; i<N_poly; i++)    /*rows*/
        {
        sum+=connect[i][j];     /*add up the number of polymers in the network
        represented by this column*/
        }
    network[sum]++;     /*array keeps tally of varying sizes of networks.
    the total contribution at each time step should equal N_Poly, so
     we can divide by (N_Poly*number of samples) when writing out*/
    //most sums will be zero. these can be ignored later on.
    sum=0;              /*initialize sum for to tally up the next column */
    }
}

void print_out_network()
{
    int n, i;

  FILE *clust_FilePtr;                                  /*File for cluster probability*/
  clust_FilePtr = fopen("cluster_probability.txt", "w");
  FILE *connect_FilePtr;                                /*File for storing connection probability*/
  connect_FilePtr = fopen("connection_probability.txt", "w");
  FILE *funct_FilePtr;                                  /*File for functionality probability*/
  funct_FilePtr = fopen("functionality_probability.txt", "w");

    n=polyX.N+polyY.N;

    fprintf(connect_FilePtr, "links\tfreq\t");
    fprintf(clust_FilePtr, "clustersize\tfreq");
    fprintf(funct_FilePtr, "Functionality\tfreq");

    for(i=0; i<=n; i++)
        {
        fprintf(connect_FilePtr, "\n%d\t%d", i, links[i]);
        fprintf(clust_FilePtr, "\n%d\t%d", i, network[i]);
        fprintf(funct_FilePtr, "\n%d\t%d", i, functionality[i]);
        /*links[i]=0;
        network[i]=0;*/
        }
        fclose(connect_FilePtr);
        fclose(clust_FilePtr);
        fclose(funct_FilePtr);
}

void init_arrays()
{int n, i;
 n=polyX.N+polyY.N;
 for(i=0; i<=n; i++)
 {links[i]=0;
 network[i]=0;
 functionality[i]=0;}
}
```

B.2 Mean squared displacement analysis code

The mean squared displacement was not calculated on-the-fly, instead the coordinates of the .xyz file recorded during simulation are read into the following code and analyzed separately. In the code we make use of the following relationship, if

$$\mathbf{R}(t) = (\mathbf{r}_1(t), \ldots, \mathbf{r}_N(t))^T$$

then it follows that

$$\left\langle |\mathbf{r}_i(t) - \mathbf{r}_i(t_0)|^2 \right\rangle = \frac{1}{N} \sum_{i=1}^{N} (\mathbf{R}(t) - \mathbf{R}(t_0)) \cdot (\mathbf{R}(t) - \mathbf{R}(t_0))$$

```
#include <stdio.h>

float data[10000][10200];/*[steps][3N]*/
float correlation[10000];   /* entry j = <t_j-t_0>, must be size 'steps' */

int N, steps;

int main()
{
    int i, j, k, k1, k2, N, T;  /*remember all steps are multiplied by the
     frequency at which the .xyz data was recorded*/
    int te=10;              /*time allowed for equilibrium*/
    int steps=10000;   /*total number of coordinate entries in the file*/
    //double sum;
    N=3400;//3400;
    float d1, d2, d3, corrl;
/*total number of frames is j=10001*/
    FILE *xyz_ptr;
    xyz_ptr=fopen("frames.xyz", "r");
    FILE *rmd_ptr;
    rmd_ptr=fopen("sqrmn_displacement.txt", "w");

    /*scan over first two integer entries...these won't be needed though*/

/*sllow time for equilibration*/
        for(j=0; j<(te); j++)
            {fscanf(xyz_ptr, "%d\n%d\n", &k1, &k2);
        //printf("%d\t%d", k1, k2);
        /*scans first set of coordinates*/
            for(i=0; i<N; i++)
                {
                fscanf(xyz_ptr, "%d%f%f%f", &k1, &d1, &d2, &d3);
                }
        }

//while(!feof(xyz_ptr))
    {
    for(j=0; j<steps; j++)
            {
            fscanf(xyz_ptr, "%d\n%d\n", &k1, &k2);
            /*scans first set of coordinates*/
            for(i=0; i<N; i++)
                {
                fscanf(xyz_ptr, "%d%f%f%f", &k1, &data[j][3*i],
                 &data[j][3*i+1], &data[j][3*i+2]);
                }
               // printf("%d\n", j);
            }
    }

/*data was only taken every 1000dt in simulation anyway.
This is how many time steps of data we actually have.*/

fclose(xyz_ptr);

printf("XYZ scanned\n");

for(T=1; T<steps; T++)
    {
    corrl=0;
   // printf("!!!");
        for(j=T; j<steps; j++)
            {for(i=0; i<3*N; i++)
                {corrl+=(data[j][i]-
                data[j-T][i])*(data[j][i]-data[j-T][i]);
                }
            }
    correlation[T-1]=corrl/(N*(steps-T);
    /*There will have been steps-T samples of data taken,
     averaged over N particles also*/
    //printf("%f\n", correlation[T-1]);
    fprintf(rmd_ptr, "%f\n", correlation[T-1]);
    }

    fclose(rmd_ptr);
    printf("\nDONE");
    return 0;
}
```