EMSIPON

an Engine for Mesoscopic Simulations of Polymers

Contents

Chapter 1

Introduction

1.1 Model

We have developed a Field Theory - inspired Monte Carlo approach for equilibrating long-chain melts based on a coarse-grained Hamiltonian that includes bonded, excluded volume, and cohesive interactions [1]. Inputs to this method are the Kuhn length b, the equilibrium melt density ρ_0 and the isothermal compressibility of the melt κ_T under the conditions of interest. Chains, which are represented as freely jointed sequences of Kuhn segments, adopt unperturbed conformations at equilibrium. This scheme can be coarse-grained further by lumping sequences of $n_{\mathrm{Kuhns/bead}}$ Kuhn segments into one bead. The representation described above relies on soft intermolecular interactions between the beads of different chains. These molecules can therefore cross each other, and do not exhibit any signature of entanglement. Literature attempts to describe entanglements have largely relied on so-called tube or network models. Our description of entanglements is based on the concept of slip-springs [2] . These slip-springs restrict the lateral motion of the beads with respect to the axis of the backbone, and favor a reptating motion along the backbone.

1.2 Interactions

At this coarse-grained level of representation, the polymeric system is assumed to be governed by a Helmholtz free energy function A depending on its spatial extent, on the positions of coarse-grained segments, on the connectivity, and on temperature. A is written as a sum of five terms reflecting contributions from the conformational entropy of strands between beads, slip-springs and nonbonded (polymer, polymer-particle and particle-particle) interactions

$$A = A_\mathrm{b} + A_\mathrm{ss} + A_\mathrm{nb} + A_\mathrm{ps} + A_\mathrm{pp}$$

1.2.1 Intramolecular Interactions

The intramolecular interactions acting along the chain's backbone, between two beads i and j located at positions $\mathbf{R}_i$ and $\mathbf{R}_j$, respectively, are given by

$$A_\mathrm{b}\left(r_{ij}^2\right) = \varepsilon_\mathrm{b} T \frac{r_{ij}^2}{\sigma_\mathrm{b}^2} = \frac{3}{2} k_\mathrm{B} T \frac{\mathbf{R}_{ij} \cdot \mathbf{R}_{ij}}{n_{\mathrm{Kuhns/bead}} b^2}$$

with the parameters ε_b and σ_b read from the data file. In our approach $\varepsilon_\mathrm{b} = 0.012471$ kJ/mol/K and $\sigma_\mathrm{b}^2 = 810$ Å^2 for polyisoprene melt.

1.2.2 Slip-spring Interactions

The slip-spring forcefield can be either modeled by an harmonic potential, similar to the one used for the chain backbone, or by a Finitely Extensible Non-linear Elastic (FENE) potential of the form

$$\mathscr{V}_{\text{ss}}\left(r_{ij}^2\right) = -\varepsilon_{\text{ss}} T \sigma_{\text{ss}}^2 \ln\left[1 - \frac{r_{ij}^2}{\sigma_{\text{ss}}^2}\right]$$

with the parameters ε_{ss} measured in kJ/mol/K/ Å^2 and σ_{ss} in Å^2. Chappa et al. [2], give the following values to the above coefficients. The distance term can be connected to our harmonic springs as

$$\sigma_{\text{ss}}^2 = 5.76\left(n_{\text{Kuhns/bead}} b^2\right) = 4665.6\ \text{Å}^2$$

which is commensurate with the entanglement tube diameter of the polyisoprene. The energetic contribution, following the same authors, can be given as

$$\varepsilon_{\text{ss}} T \sigma_{\text{ss}}^2 = \frac{k_{\text{ss}} r_{\text{ss}}^2}{2} = 17.28 k_{\text{B}} T$$

$$\varepsilon_{\text{ss}} = \frac{17.28 k_{\text{B}}}{\sigma_{\text{ss}}^2} = 3.0794 \times 10^{-5}\ \frac{\text{kJ}}{\text{mol K Å}^2}$$

Alternatively, the strength of the slip-springs can be considered comparable to the strength of the harmonic bonds along the chain backbone:

$$\varepsilon_{\text{ss}} T \sigma_{\text{ss}}^2 = \frac{3}{2} k_{\text{B}} T$$

1.2.3 Polymer Non-bonded Interactions

To deal with nonbonded (excluded volume and van der Waals attractive) interactions in the network representation, we introduce a network free energy:

$$A_{\text{nb}} = \int d^3\mathbf{r}\, f\left[\rho\left(\mathbf{r}\right)\right]$$

In the above equation, $\rho\left(\mathbf{r}\right)$ is the local density (number of Kuhn segments per unit volume) at position $\mathbf{r}$ and $f\left(\rho\right)$ is a free energy density (free energy per unit volume). Expressions for $f\left(\rho\right)$ may be extracted from an equation of state. Here the plan is to invoke a simple expression for $f\left(\rho\right)$, in the form of a Taylor expansion,

$$f\left(\rho\right) = C\rho + B\rho^2$$

with C, B fitted such that the volumetric properties (pressure and compressibility at mean density of interest) are reproduced. Local density will be resolved only at the level of entire cells, defined by passing an orthogonal grid through the entire system. The free energy of the system is approximated by

$$A_{\text{nb}} = \sum_{\text{cells}} V_{\text{cell}}^{\text{acc}} f\left(\rho_{\text{cell}}\right)$$

where $V_{\text{cell}}^{\text{acc}}$, the accessible of a cell, is the volume of the rectangular parallelepiped defining the cell minus the volume of any parts of nanoparticles that may find themselves in the cell.

The cell density ρ_{cell} must be defined based on the nodal points in and around the cell, each nodal point contributing a mass equal to the node's mass. Each nodal point j has mass n_j (in Kuhn segments) and a characteristic size R_j. We will discuss below how these quantities depend on the node's molecular characteristics. We denote the position vector of node j by $\mathbf{r}_j = \left(x_j, y_j, z_j\right)$. The cell dimensions along the x, y, z directions will be denoted as L_x, L_y, L_z, respectively.

We will focus on a cell extending between $x_{\text{cell}} - L_x$ and x_{cell} along the x-direction, between $y_{\text{cell}} - L_y$ and y_{cell} along the y-direction, and between $z_{\text{cell}} - L_z$ and z_{cell} along the z-direction. In the regular grid considered, if $(0,0,0)$ is taken as one of the grid points x_{cell}, y_{cell}, and z_{cell} will be integer multiples of L_x, L_y and L_z, respectively.

In the following we will assume that

$$R_j < \min\left(L_x, L_y, L_z\right)$$

The simplest option for relating the positions and masses of the node to ρ_{cell} is to envision each node j as a cube containing n_j Kuhn segments, of edge length R_j, centered at $\mathbf{r}_j$. Node j will contribute to a cell if its cube (cube j) overlaps with the cell. Note that, for this to happen, it is not necessary that the nodal position of the center, $\mathbf{r}_j$, lie in the cell. The mass (number of Kuhn segments) contributed by the node to the cell is:

$$n_{j,\text{cell}} = n_j \frac{V_{\text{cube } j \cap \text{cell}}}{V_{\text{cube } j}}$$

with $V_{\text{cube } j \cap \text{cell}}$ being the volume of the intersection of cube j, associated with node j, and the considered cell, while $V_{\text{cube } j} = R_j^3$ is the volume of cube j.

Under the condition $R_j < \min(L_x, L_y, L_z)$, $V_{\text{cube } j \cap \text{cell}}$ is obtainable as:

$$\begin{aligned} V_{\text{cube } j \cap \text{cell}} &= \max\left\{\left[\min\left(x_j + \frac{R_j}{2}, x_{\text{cell}}\right) - \max\left(x_j - \frac{R_j}{2}, x_{\text{cell}} - L_x\right)\right], 0\right\} \\ &\times \max\left\{\left[\min\left(y_j + \frac{R_j}{2}, y_{\text{cell}}\right) - \max\left(y_j - \frac{R_j}{2}, y_{\text{cell}} - L_y\right)\right], 0\right\} \\ &\times \max\left\{\left[\min\left(z_j + \frac{R_j}{2}, z_{\text{cell}}\right) - \max\left(z_j - \frac{R_j}{2}, z_{\text{cell}} - L_z\right)\right], 0\right\} \end{aligned}$$

As defined by the above equation, $V_{\text{cube } j \cap \text{cell}}$ is a linear function of the node coordinates. Clearly, if cube j lies entirely within the cell, $V_{\text{cube } j \cap \text{cell}} = V_{\text{cube } j}$ and, consequently, $n_{j,\text{cell}} = n_j$. If however, the borders of cube j intersect the borders of the considered cell, then node j will contribute a mass $n_{j,\text{cell}} < n_j$ to the cell. The total mass contributed by bead j to all cells in which it participates will always be n_j.

The density ρ_{cell} in the considered cell is estimated as:

$$\rho_{\text{cell}} = \frac{1}{V_{\text{cell}}^{\text{acc}}} \sum_j n_{j,\text{cell}}$$

Clearly, only nodal points j whose cubes have a nonzero overlap with the considered cell will contribute to the above summation. The positions vectors $\mathbf{r}_j$ of these beads will necessarily lie within the considered cell or its immediate neighbors.

The precise conditions for cube j to have common points with the considered cell are:

$$x_{\text{cell}} - L_x < x_j + \frac{R_j}{2} < x_{\text{cell}} + R_j$$

$$y_{\text{cell}} - L_y < y_j + \frac{R_j}{2} < y_{\text{cell}} + R_j$$

$$z_{\text{cell}} - L_z < z_j + \frac{R_j}{2} < z_{\text{cell}} + R_j$$

According to the above approach, the force on node j due to nonbonded interactions is:

$$\mathbf{F}_j = -\nabla_{\mathbf{r}_j} A_{\text{nb}} = - \sum_{\substack{\text{cells having common} \\ \text{points with cube } j}} V_{\text{cell}}^{\text{acc}} \left. \frac{df}{d\rho} \right|_{\rho=\rho\text{cell}} \nabla_{\mathbf{r}_j} \rho_{\text{cell}}$$

1.3 Coarse-Grained Dynamics

Dynamics at this level of representation will be tracked as two types of processes occuring in paralell: (a) Brownian motion of the beads (including crosslinks and end-points) in continuous three-dimensional space; hops of the slip-springs between adjacent segments along a chain, destruction and creation of slip-springs. Both types of processes are governed by the free energy of the system.

1.3.1 Brownian Dynamics

We adopt a Brownian Dynamics approach, in which the time evolution of the configuration $\{R_j\}$ is governed by [3] :

$$R_j(t_n+\Delta t) = R_j(t_n) + \frac{\Delta t}{k_\mathrm{B}T} D_\mathrm{t} F_j(t_n) + \frac{1}{2}\frac{\Delta t^2}{k_\mathrm{B}T} D_\mathrm{t} \dot{F}_j(t_n) + X_j^\mathrm{t}(\Delta t)$$

where F_j is the total force acting on the j-th degree of freedom, Δt the integration timestep and D_t the translational diffusion coefficient of the beads. X_j^t represents the stochastic displacement due to the influence of the coarse-grained microscopic degrees of freedom and satisfies the fluctuation-dissipation relation. Random displacements X_j^t are sampled from a Gaussian distribution with zero mean and width:

$$\left\langle (X_j^\mathrm{t})^2 \right\rangle = 2D_\mathrm{t}\Delta t = 2\frac{k_\mathrm{B}T}{m_\mathrm{bead}\gamma}\Delta t$$

with γ being the friction coefficient.

For large values of $\gamma\Delta t$ in the diffusive regime, the friction is so strong that the velocities relax within Δt. For polyisoprene $\zeta_0(500\,\mathrm{K}) = m_\mathrm{monomer}\gamma = 2.63\times10^{-12}\ \mathrm{kg/s}$, $\gamma(500\,\mathrm{K}) = 2.325\times10^{13}\ \mathrm{s}^{-1}$ and thus $\gamma\Delta t = 23.25$ for $\Delta t = 10^{-12}\ \mathrm{s}$.

1.3.2 Rotational Brownian Dynamics

As originally formulated, the Brownian Dynamics algorithm of Ermak and McGammon [4] deals only with the translational motion of the particles. In reality, however, dispersed particles or molecules also execute rotational Brownian motion arising from the fluctuating torque exerted on them by the surrounding solvent molecules.

In complete analogy to the Brownian Dynamics equation of motion written for the translational degrees of freedom, a similar time evolution equation can be written for the rotational degrees of freedom [5] . Thus, angular displacements ϕ_j are given by:

$$\phi_j(t_n+\Delta t) = \phi(t_n) + \frac{\Delta t}{k_\mathrm{B}T} D_\mathrm{r} T_j(t_n) + \frac{1}{2}\frac{\Delta t^2}{k_\mathrm{B}T} D_\mathrm{r} \dot{T}_j + X_j^\mathrm{r}(t_n)$$

where now D_r stands for the rotational diffusion coefficient of the particles, measured in $\mathrm{rad}^2\,\mathrm{s}^{-1}$, and T_j is the sum of external and interparticle torques acted in direction j. The timestep Δt should be sufficiently large for angular velocities correlations to vanish out during it.

X_j^r represents the stochastic rotation due to the influence of the coarse-grained microscopic degrees of freedom and satisfies the fluctuation-dissipation relation. Random rotations X_j^r are sampled from a Gaussian distribution with zero mean and width:

$$\left\langle (X_j^\mathrm{r})^2 \right\rangle = 2D_\mathrm{r}\Delta t$$

The rotational diffusion coefficient can be related to the rotational frictional coefficient, f_r, by the Einstein - Smoluchowski equation:

$$D_\mathrm{r} = \frac{k_\mathrm{B}T}{f_\mathrm{r}}$$

The rotational frictional drag coefficient for a sphere of radius α is:

$$f_\mathrm{r,sphere} = 8\pi\eta\alpha^3$$

with η being the dynamic (or shear) viscosity of the medium.

1.3.3 Kinetic Monte Carlo Hopping

In order to develop a formalism of elementary events of slip-spring hopping, creation or destruction, we need expressions for the rate of slippage along the chain backbone. In order to extract the diffusivity of the slip-springs, we will proceed along the lines of Terzis and Theodorou's work. [6] We describe self-diffusion along the chain contour with the Rouse model. The Rouse model addresses the dynamics of polymers in unentangled melts. A polymer chain is represented by a set of beads connected by harmonic springs. The dynamics, as in our simulations, are modeled as a Brownian motion of these tethered beads, the environment of a chain being represented as a continuum (viscous medium), ignoring all excluded volume and hydrodynamic interactions.

In this model the self-diffusion of the center of the mass of the polymer is related to the friction coefficient, ζ on a bead by:

$$D_{\mathrm{Rouse}} = \frac{k_{\mathrm{B}}T}{N\zeta}$$

with N being the number of beads per chain. In the picture we invoke in our network model, the center of mass diffusivity along the contour is related to the rate of slip-spring jumps across beads (by distance $\left(n_{\mathrm{Kuhns/bead}}b^2\right)^{1/2}$ in each direction by (see below for the definition of ν_{diff})

$$D_{\mathrm{Rouse}} = k_{\mathrm{diff}}\frac{n_{\mathrm{Kuhns/bead}}b^2}{N} = \nu_{\mathrm{diff}}\frac{n_{\mathrm{Kuhns/bead}}b^2}{N}\exp\left(-\frac{A_0}{k_{\mathrm{B}}T}\right)$$

Hence, one must have:

$$\nu_{\mathrm{diff}} = \frac{k_{\mathrm{B}}T}{n_{\mathrm{Kuhns/bead}}b^2\zeta}\exp\left(-\frac{A_0}{k_{\mathrm{B}}T}\right)$$

where A_0 is a free energy per slip-spring in the equilibrium melt, which establishes a baseline for measuring free energies.

An individual jump of the one end of a slip-spring along the chain backbone takes place with rate:

$$k_{\mathrm{hopping}} = \nu_0\exp\left(-\frac{A^{\dagger}_{1\to2} - A_{\mathrm{a_0-b_0}}}{k_{\mathrm{B}}T}\right) = \nu_{\mathrm{diff}}\exp\left(-\frac{A_{\mathrm{a_0-b_0}}}{k_{\mathrm{B}}T}\right)$$

conforming to a transition state theory (TST) picture of the slippage along the backbone as an infrequent event, which involves a transition from state "1" to state "2" over a free energy barrier. In the final expression, the rate of hopping, k_{hopping}, depends directly on the energy of the initial state of the slip-spring, $A_{\mathrm{a_0-b_0}}$, while the dependence on the height of the free energy barrier (i.e. $A^{\dagger}_{1\to2}$) has been absorbed into the pre-exponential factor, ν_{diff}.

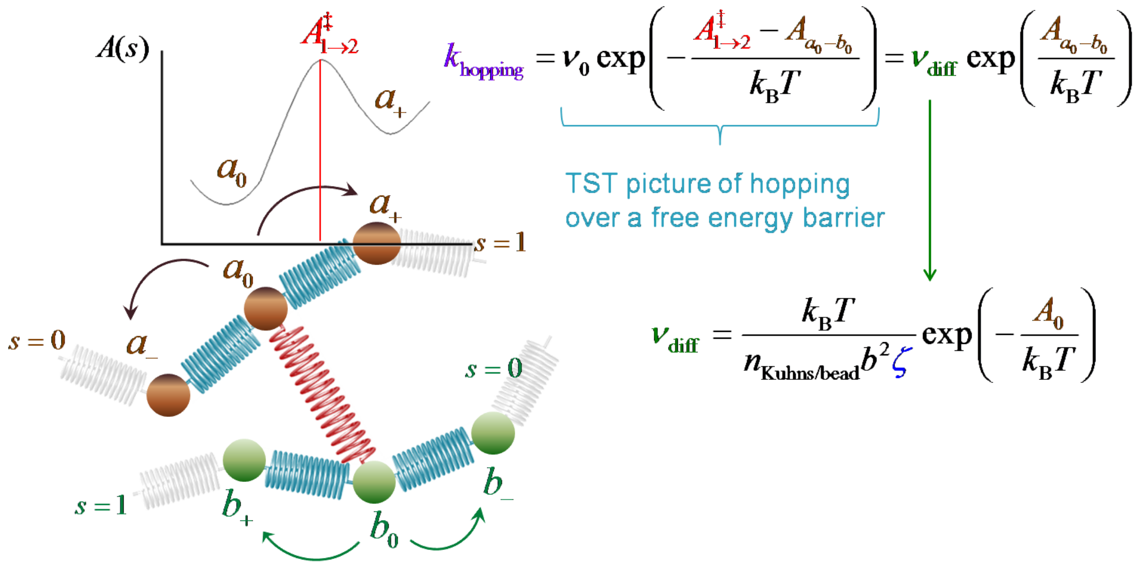

$A(s)$
$A^{\ddagger}_{1\to2}$
a_+
a_0
a_+
$s=1$
a_0
$s=0$
a_-
$s=0$
$s=1$
b_+
b_0
b_-
$k_{\mathrm{hopping}} = \nu_0\exp\left(-\frac{A^{\ddagger}_{1\to2} - A_{a_0-b_0}}{k_{\mathrm{B}}T}\right) = \nu_{\mathrm{diff}}\exp\left(\frac{A_{a_0-b_0}}{k_{\mathrm{B}}T}\right)$
TST picture of hopping over a free energy barrier
$\nu_{\mathrm{diff}} = \frac{k_{\mathrm{B}}T}{n_{\mathrm{Kuhns/bead}}b^2\zeta}\exp\left(-\frac{A_0}{k_{\mathrm{B}}T}\right)$

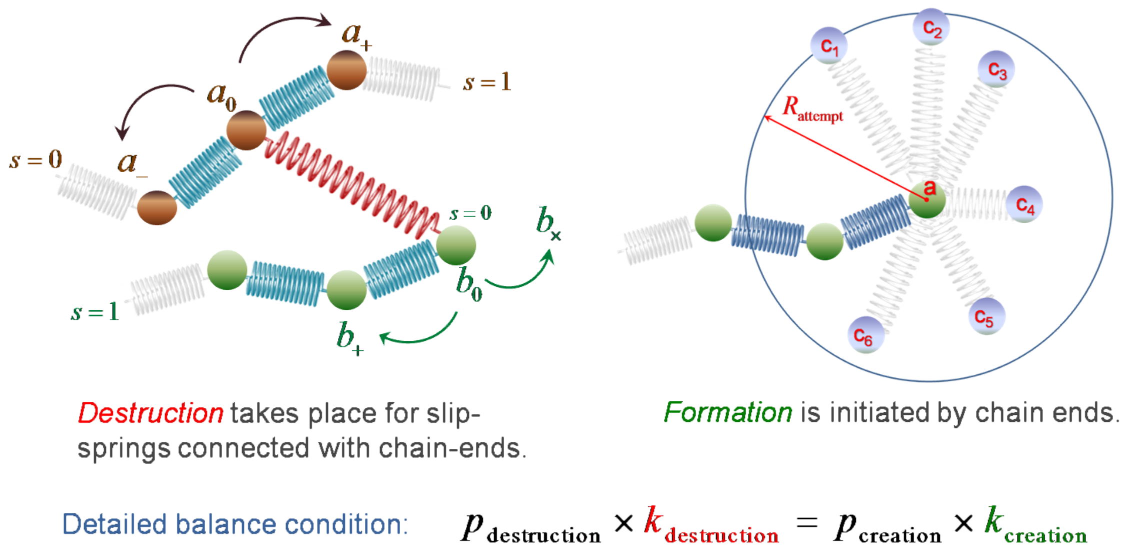
Figure 1.1: Slip-spring hopping schematic

The destruction of a slip-spring can be envisioned as an infrequent event during which a slip-spring is pushed to the end of the chain (by traveling a distance $\left(n_{\mathrm{Kuhns/bead}}b^2\right)^{1/2}$, corresponding to the last bead of the chain) with rate ν_{diff} and there faces a transition which can take place with overall rate:

$$k_{\mathrm{destruction}} = \nu_{\mathrm{diff}}\exp\left(-\frac{A_{\mathrm{a_0-b_0}}}{k_{\mathrm{B}}T}\right)$$

where $A_{\mathrm{a}_0-\mathrm{b}_0}$ is the free energy the slip-spring (harmonic spring) contributes to the total free energy of the system. Again, v_{diff} is calculated a la Rouse.

At the timestep of the 3D Brownian Dynamics simulation, when hopping kinetic Monte Carlo has to take place, every free end of the system can randomly create a new slip-spring with an internal bead of a neighboring chain. This may be accomplished by a rate constant k_{creation}. The rate for the creation of a slip-spring is closely related to the probability of pairing the end "a" with one of its candidate mates which lie inside a sphere of prescribed radius R_{attempt}. The definition of the probability implies that the more crowded chain ends are the more probable to create a slip-spring. The number of neighbors around a chain end can be tuned via the radius of the sphere within which the search takes place, R_{attempt}. A good estimate of R_{attempt} for polyisoprene (either pure or crosslinked) can be given by the tube diameter of the polymer. A computational study of the tube diameter of the polyisoprene as a function of the molecular weight has been done by the Li et al. [7] The rate constant k_{creation} can be treated as an adjustable parameter of our model, which will be used to ensure that the average number of slip-spring present in the system is conserved throughout the simulation.



Destruction takes place for slip-springs connected with chain-ends.

Formation is initiated by chain ends.

Detailed balance condition: $p_{\mathrm{destruction}} \times k_{\mathrm{destruction}} = p_{\mathrm{creation}} \times k_{\mathrm{creation}}$

Figure 1.2: Slip-spring destruction and creation

1.4 Stress Tensor Calculation

Given the free energy functional described in the "Model" subsection, the stress tensor of the system can be derived as [8] :

$$\boldsymbol{\sigma} = \rho \boldsymbol{F} \left(\frac{\partial A}{\partial \boldsymbol{F}} \right)^{\mathrm{T}}$$

where $\boldsymbol{F}$ denotes the deformation gradient tensor and may be considered as a mapping of an infinitesimal vector $\mathrm{d}\boldsymbol{x}$ of the initial configuration onto the infinitesimal vector $\mathrm{d}\boldsymbol{x}$ of the distorted configuration.

1.4.1 Bonded contributions to the stress tensor

The contribution of the bonds to the stress tensor can be easily calcualated due to the fact that invokes central forces between the beads. The stress tensor of the atom i, $\sigma_{i,\alpha\beta}$ is given by the following formula, where α and β take on values x, y, z to generate the six components of the symmetric tensor:

$$\sigma_{i,\alpha\beta} = -\frac{1}{2} \sum_{j=1}^{N_{\mathrm{b}}(i)} \left(r_{i,\alpha} - r_{j,\alpha} \right)^{\mathrm{min.im.}} F_{ij,\beta}^{\mathrm{min.im.}}$$

where $N_b(i)$ stands for the number of bonds atom i participates to and F_{ij} is the force between atoms i and j calculated by the partial derivative of the free energy, $\partial A_b/\partial \mathbf{R}_j$.

1.4.2 Polymer Non-bonded interaction contributions to the stress tensor

As already pointed out, the non-bonded intearactions are computed by passing an orthogonal grid in the simulation box and therefore the derivatives present in the definition of the stress tensor are written as a sum over the grid cells:

$$\frac{\partial A_{\text{nb}}}{\partial \boldsymbol{F}} = \frac{\partial A_{\text{nb}}}{\partial \rho_{\text{cell}}^{(1)}} \frac{\partial \rho_{\text{cell}}^{(1)}}{\partial \mathbf{F}} + ... + \frac{\partial A_{\text{nb}}}{\partial \rho_{\text{cell}}^{(N_{\text{cells}})}} \frac{\partial \rho_{\text{cell}}^{(N_{\text{cells}})}}{\partial \mathbf{F}}$$

The derivative of the determinant of $\boldsymbol{F}$ with respect to the tensor $\boldsymbol{F}$ itself is calculated by the following equation:

$$\frac{\partial\left(\det(\mathbf{F})\right)}{\partial \mathbf{F}} = \det(\mathbf{F})\mathbf{F}^{-1} = \det(\mathbf{F})\left(\mathbf{F}^{-1}\right)^{\mathrm{T}}$$

Besides, the determinant of the deformation gradient tensor is written as the ratio of volumes or densities of the distorted and initial configurations:

$$\det(\mathbf{F}) = \frac{V'}{V} = \frac{\rho}{\rho'}$$

The symbol " $'$" indicates the initial (undistorted) configuration. Thus, we have:

$$\frac{\partial \rho_{\text{cell}}^{(1)}}{\partial \mathbf{F}} = \rho_{\text{cell}}^{\prime(1)} = \rho_{\text{cell}}^{\prime(1)} \det(\mathbf{F})\left(\mathbf{F}^{-1}\right)^{\mathrm{T}}$$

and the derivative of the free energy with respect to the local density of a cell becomes:

$$\left(\frac{\partial A_{\text{nb}}}{\partial \rho_{\text{cell}}^{(1)}} \frac{\partial \rho_{\text{cell}}^{(1)}}{\partial \mathbf{F}}\right)^{\mathrm{T}} = \rho_{\text{cell}}^{(1)} \frac{\partial A_{\text{nb}}}{\partial \rho_{\text{cell}}^{(1)}} \mathbf{F}^{-1}$$

Similar expressions hold for the rest derivatives and thus, the final form of the stress tensor, due to non-bonded interactions is:

$$\left(\frac{\partial A_{\text{nb}}}{\partial \mathbf{F}}\right)^{\mathrm{T}} = \left(\rho_{\text{cell}}^{(1)} \frac{\partial A_{\text{nb}}}{\partial \rho_{\text{cell}}^{(1)}} + ... + \rho_{\text{cell}}^{(N_{\text{cell}})} \frac{\partial A_{\text{nb}}}{\partial \rho_{\text{cell}}^{N_{\text{cells}}}}\right)\mathbf{F}^{-1}$$

and

$$\boldsymbol{\sigma}_{\text{nb}} = \left[\rho_{\text{cell}}^{(1)} \frac{\partial A_{\text{nb}}}{\partial \rho_{\text{cell}}^{(1)}} + ... + \rho_{\text{cell}}^{(N_{\text{cell}})} \frac{\partial A_{\text{nb}}}{\partial \rho_{\text{cell}}^{N_{\text{cells}}}}\right]\mathbf{I}_{3\times 3}$$

It is concluded from the last equation that all off-diagonal elements are equal to zero. Thus, non-bonded interaction do not contribute to the shear elements of the stress tensor of our model.

1.4.3 Polymer-particle and Particle-particle interaction contributuions to the stress tensor

Since polymer-particle and particle-particle interactions yield central forces, the stress tensor contribution can readily be written as:

$$\sigma_{i,\text{ps},\alpha\beta} = -\frac{1}{2}\sum_{j=1}^{N_{\text{particles}}} \left(r_{i,\alpha} - r_{j,\alpha}\right)^{\text{min.im.}} F_{ij,\beta}^{\text{min.im.}}$$

where $N_{\text{particles}}$ stands for the number of particles present in the system, and F_{ij} is the force acted between the i-th bead and the j-th particle. The same can also apply for the estimation of the contribution to the stress tensor due to particle-particle interactions.

1.4.4 Estimation of Rheological Properties

The main reason for computing the stress tensor is the estimation of rheological properties such as zero-shear viscosity, complex modulus $G^*(\omega)$, storage modulus $G'(\omega)$ and loss modulus $G''(\omega)$. Zero-shear viscosity can be calculated as the limit of the off-diagonal stress components autocorrelation function:

$$\eta_0 = \frac{V}{k_\mathrm{B}T}\int_0^{+\infty} \left\langle \sigma_{\alpha\beta}(t)\,\sigma_{\alpha\beta}(0)\right\rangle \mathrm{d}t$$

where α, β should be two orthogonal axes. The complex modulus is given as:

$$G^*(\omega) = G'(\omega) + iG''(\omega) = i\omega\frac{V}{k_\mathrm{B}T}\int_0^{+\infty} \mathrm{e}^{-i\omega t}\left\langle \sigma_{\alpha\beta}(t)\,\sigma_{\alpha\beta}(0)\right\rangle \mathrm{d}t$$

1.5 Benchmark Simulations

To characterize the equilibrium dynamical behavior, we compute the beads' mean-squared displacement, $g_1(t)$, defined by:

$$g_1(t) = \langle[\mathbf{r}(t)\mathbf{r}(0)]\rangle$$

where the brackets indicate an average over all beads in the simulation box. We also compute the mean-squared center-of-mass displacements

$$g_3(t) = \left\langle[\mathbf{r}_\mathrm{CM}(t) - \mathbf{r}_\mathrm{CM}(0)]^2\right\rangle$$

We begin our discussion by examining the behavior of a simple melt. To a first approximation, the dynamics of short chains in a melt can be described by the Rouse model [9] . We have performed simulations for a polymerization index of $N = 80$ beads per chain, each bead representing 10 Kuhn segments of PI (or equivalently 21 PI monomers) having a mass of $m_\mathrm{bead} = 1460\,\mathrm{g/mol}$. The simulation box size was fixed at a volume of $(R_{\mathrm{e},0})^3$ with $R_{\mathrm{e},0}$ being the square root of the unperturbed mean-squared end-to-end distance of the chains and the number of chains present in the simulation box was set to $n = 27$ in order to match PI's density. The bonded interactions were parametrized based on the Kuhn length of the PI, $b = 0.9\,\mathrm{nm}$, while the non-bonded interactions were parametrized by using the Sanchez - Lacombe equation of \ state with PI parameters [10] .

1.5.1 Rouse Dynamics

The next figure shows results for the time dependence of the mean-squared displacements, $g_1(t)$ and $g_3(t)$ for an unentangled melt. As can be seen, the mean-squared center-of-mass displacement, $g_3(t)$, remains linear all times; this means the intermolecular forces between polymers are too weak to affect diffusive behavior and play a minor role compared to the bonded interactions. The beads mean-squared displacements, $g_1(t)$, exhibit a subdiffusive behavior that arises from chains' connectivity, and is characterized by a power law of the form $g_1(t) \sim t^{-1/2}$. After an initial relaxation time where a change in $g_1(t)$ occurs, a regular diffusive regime is entered, where $g_1(t) \sim t$. This sequence of scaling trends is predicted by the Rouse theory. The limiting behavior of the chains' center-of-mass displacement can yield an estimate of the diffusivity of the chains:

$$\lim_{t\to\infty} g_3(t) = 6Dt$$

which has been found in excellent agreement with the predicted diffusivity by the Rouse model:

$$D_\mathrm{Rouse} = \frac{k_B T}{N\zeta_\mathrm{bead}} = 1.56\times10^{-12}\,\frac{\mathrm{m}^2}{s}$$

where our bead's friction coefficient, ζ_bead is $5.52\times10^{-11}\,\mathrm{kg/s}$. The introduction of nonbonded interactions does not seem to affect the scaling laws of the unentangled melt.

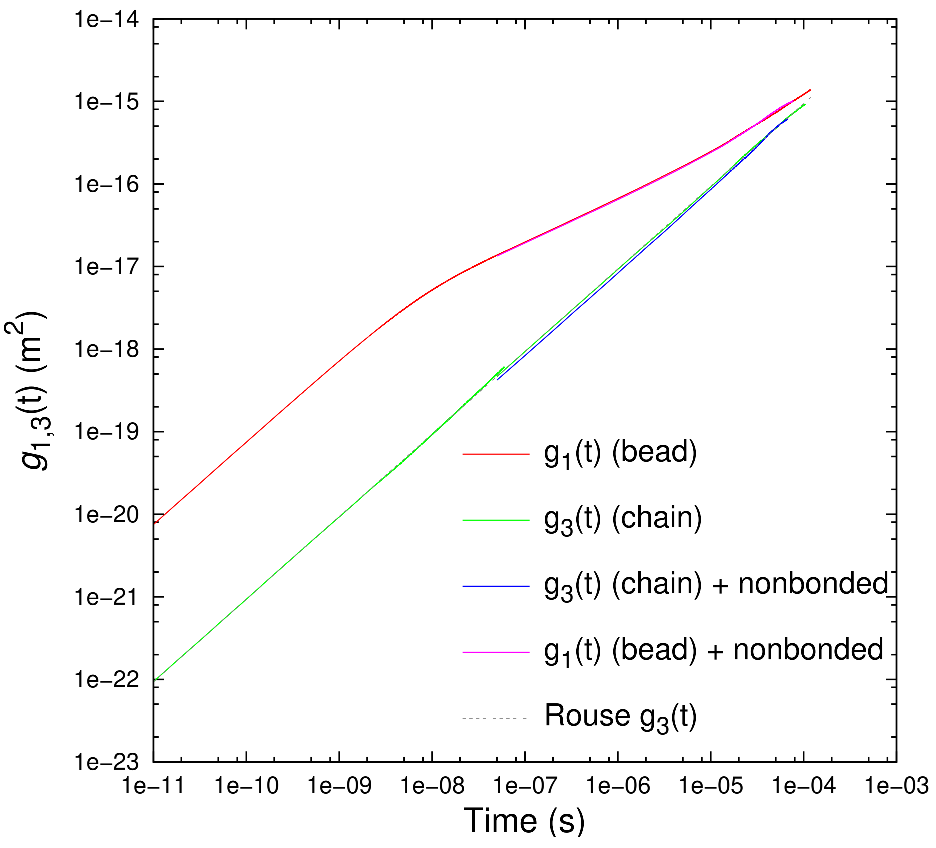


Figure 1.3: Time evolution of the mean-squared displacement of beads and center of mass of the chains for an unentangled PI melt.

1.5.2 Entangled Dynamics

For highly entangled polymer melts, the tube model offers concrete predictions regarding the scaling behavior of the mean-squared displacement of beads and of the center-of-mass of the chains. For a very short time the segment does not feel the constraints of the network formed by the neighboring chains, so that $g_1(t)$ is the same as that calcualted for the Rouse model in free space. Hence, $g_1(t)$ can be approximated as:

$$g_1(t) = \left(\frac{k_B T \left(n_{\text{Kuhns/bead}} b^2 \right)}{\zeta_{\text{bead}}} \right)^{\frac{1}{2}} t^{\frac{1}{2}}$$

This formula is correct when the average displacement is much less than the tube diameter. Let τ_e be the time at which the segmental displacement becomes comparable to the tube diameter α_{pp} :

$$\tau_e \simeq \frac{\alpha_{\text{pp}}^4 \zeta_{\text{bead}}}{k_B T \left(n_{\text{Kuhns/bead}} b^2 \right)}$$

The time τ_e denotes the onset of the effect of the tube constraints: for $t < \tau_e$, the chain behaves as a Rouse chain in free space, while for $t > \tau_e$ the chain feels the constraints imposed by the tube.

For $t > \tau_e$ the motion of the Rouse segment perpendicular to the primitive path is restricted, but the motion along the primitive path is free. The mean-squared displacement along the tube can be approximated as:

$$g_1(t) = \begin{cases} \left(\frac{\alpha_{\text{pp}}^4 k_B T \left(n_{\text{Kuhns/bead}} b^2 \right)}{\zeta_{\text{bead}}} \right)^{\frac{1}{4}} t^{\frac{1}{4}} & , \tau_e \lesssim t \lesssim \tau_R \\ \left(\frac{\alpha_{\text{pp}}^2 k_B T}{N \zeta_{\text{bead}}} \right)^{\frac{1}{2}} t^{\frac{1}{2}} & , \tau_R \lesssim t \lesssim \tau_d \end{cases}$$

where the characteristic times are:

$$\tau_{\mathrm{R}} = \frac{\zeta_{\mathrm{bead}} N^2 \left(n_{\mathrm{Kuhns/bead}} b^2\right)}{3\pi^2 k_{\mathrm{B}} T}$$

and:

$$\tau_{\mathrm{d}} = \frac{\zeta_{\mathrm{bead}} N^3 \left(n_{\mathrm{Kuhns/bead}} b^2\right)^2}{\pi^2 k_{\mathrm{B}} T \alpha_{\mathrm{pp}}^2}$$

For $t > \tau_{\mathrm{d}}$, the dynamics is governed by the reptation process. The mean-squared displacement of a bead is approximated by:

$$g_1(t) \simeq \frac{k_{\mathrm{B}} T \alpha_{\mathrm{pp}}^2}{N^2 \zeta_{\mathrm{bead}} \left(n_{\mathrm{Kuhns/bead}} b^2\right)} t$$

In order to reproduce the entangled dynamics of the PI melts we start from the unentangled melt of the previous section and we gradually introduce slip-springs between the chains by following the kinetic Monte Carlo algorithm described. We can tune the rate of slip-spring creation process, k_{creation} in a way that the system stabilizes at the experimental count of entanglements, as expected by the average entanglement molecular weight. This procedure is depicted in the following figure, where our simulation starts with a completely unentangled melt until it reaches a fully entangled system, representative of the experimental PI.

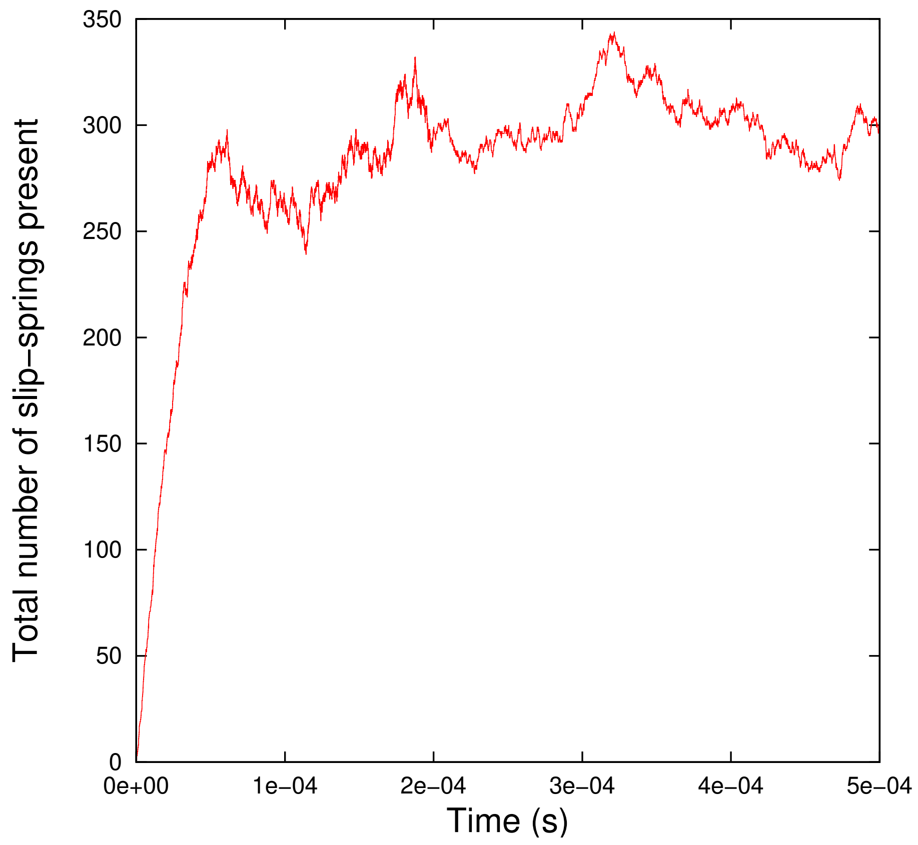


Figure 1.4: Number of slip-springs present in the system as a function of time.

Finally, we study the mean-squared displacements $g_1(t)$ and $g_3(t)$ as a function of time, by using the simulation trajectory after the point where the number of slip-sprins has stabilized at its mean value. It can be seen in the following figure that at short times the bead's mean-squared displacement, $g_1(t)$, shows a scaling regime with a power law $t^{1/2}$; at intermediate times, a regime with a power law $t^{1/4}$ appears and eventually we observe a crossover to regular diffusion at long times. Before the diffusive behavior appears, tube model predicts a crossover to $t^{1/2}$ which is completely absent from our model. Together with the simulation results, the experimental estimations of the the characteristic times of PI are presented in the figure. Our model seems capable of boldly reproducing the dynamics of entangled melts. However, careful parametrization is needed before rheological predictions can be extracted. The mean-squared displacement of the chains center-of-mass, $g_3(t)$, also exhibits subdiffusive behavior at intermediate times, with a scaling behavior of $t^{1/2}$, as predicted by the tube model; at long times regular diffusion is achieved.

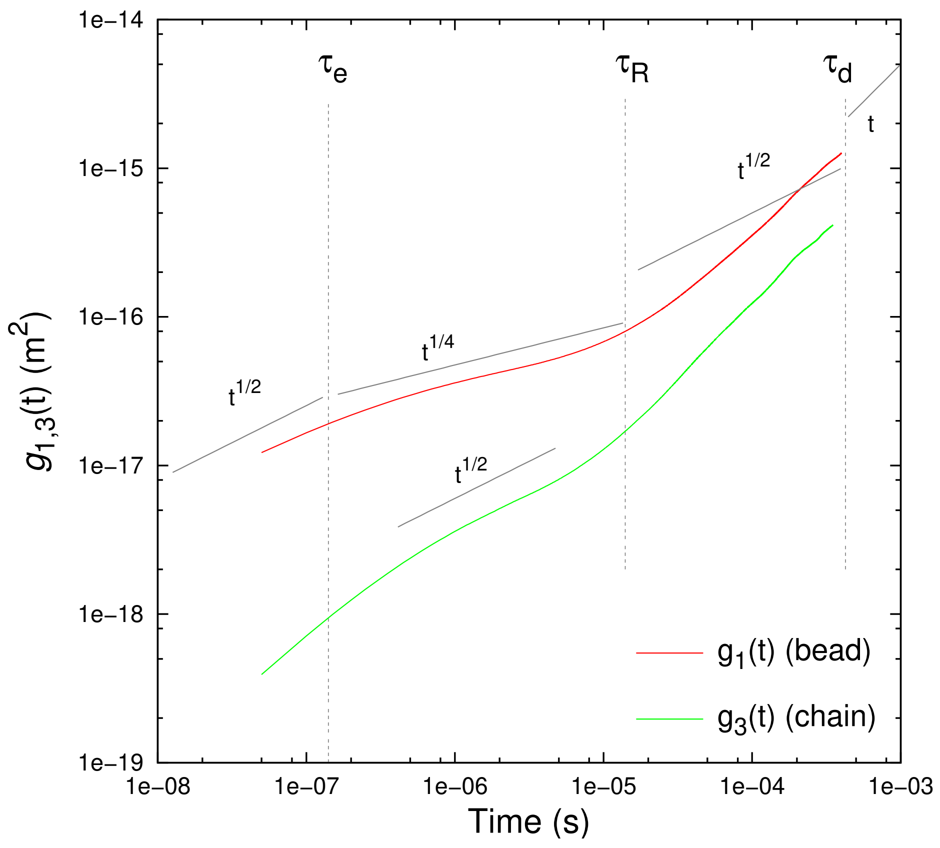


Figure 1.5: Time evolution of the mean-squared displacement of beads and center of mass of the chains for an entangled PI melt.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 NetworkNS::cb3D_integrator Class Reference

The class of the Brownian Dynamics integrator.

```
#include <b3D_integrator.h>
```

Public Member Functions

- cb3D_integrator (class NetwMin ∗, double, double)

  The constructor of the class.
- ∼cb3D_integrator ()

  The destructor of the class.
- void integrate (unsigned int nsteps, double dt, unsigned int nstout)

  The main function which performs the time integration, following Brownian Dynamics.
- void extract_positions (double ∗x)

  A function for extracting the positions from the integrator.
- void compute_stresses (void)

  A function for computing the per-atom stress.
- void report (unsigned int, double, double)
- double bonded_force_calculation (bool)

  The function for the calculation of bonded interactions.
- double simpler_scheme_non_bonded_force_calculation (void)

  A function implementing the nonbonded free energy estimation scheme.
- void calculate_pressure (double ∗)

Public Attributes

- double ∗ bd_gamma
- double ∗ bd_mass
- double ∗ bd_x
- double ∗∗ bd_stress

  The per-atom stress tensor.
- unsigned int bd_cur_step

  The current step of the integration.

Private Member Functions

- void from_bd_x_to_polymer_network (void)
- void **cell_density_nodal_points** (void)

Private Attributes

- class NetwMin ∗ cur_bd_net

  The network application that created the class.
- unsigned int dofs

  The count of particles simulated using the BD scheme.
- unsigned int dofs_3N

  The number of degrees of freedom for which BD takes place.
- double ∗ bd_f
- double ∗ bd_nb_f
- double ∗ bd_x_ps
- double ∗ bd_f_ps
- class Hopping ∗ my_hopping_scheme

  A pointer to a hopping kinetic Monte Carlo class.
- clock_t tbegin

  The intial timestep of the Brownian Dynamics simulation.
- double ∗ xshift
- double ∗ yshift
- double ∗ zshift
- int ∗ grid_cell

  The global tag of the cell the node belongs to.
- double bd_temp

  The temperature of the Brownian Dynamics simulation.
- bool gamma_mass_opt
- double ∗ density_cells

  The local density per cell.
- double ∗ den_dx
- double ∗ den_dy
- double ∗ den_dz
- FILE ∗ **p_cell_density**

4.1.1 Detailed Description

Definition at line 37 of file b3D_integrator.h.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 cb3D_integrator()

```
NetworkNS::cb3D_integrator::cb3D_integrator (
              class NetwMin ∗ init_net,
              double temperature,
              double slipspring_rate )
```

Parameters

`in`	*∗init_net*	A pointer to the application itself, From this pointer data concerning the network and the domain can be retrieved.
`in`	*temperature*	The temperature at whichi the Brownian Dynamics integrator will be initialized.

Klopffer et al. (*Polymer* **1998**, *39*, 3445 - 3449) have characterized the rheological behavior of a series of polybutadienes and polyisoprenes over a wide range of temperatures. The viscoelastic coefficients resulting from the time-temperature superposition principle were determined. The Rouse theory modified for undiluted polymers was used to calculate the monomeric friction coefficient, ζ_0 from the transition zone. It was concluded that, within experimental error, a single set of WLF parameters at $T_{\rm g}$ was adequate to characterize the relaxation dynamics irrespective of the vinyl content of the polybutadienes and polyisoprenes.

The monomeric friction coefficient, ζ_0, characterizes the resistance encountered by a monomer unit moving through its surroundings. It has been shown to follow the WLF law. The variation of the monomeric friction coefficient with temperature is:

$$\log \zeta_0 \left(T\right) = \log \zeta_\infty + \frac{C_1^{\rm g} C_2^{\rm g}}{T - T_{\rm g} + C_2^{\rm g}}$$

with the parameters $C_1^{\rm g} = 13.5 \pm 0.2$, $C_2^{\rm g} = 45 \pm 3$ K, $\log \zeta_\infty = -10.4$ dyn s cm^{-1} and $T_{\rm g} = 211.15$ K. At a temperature of 298 K, $\zeta_0(298\,{\rm K}) = 1.61 \times 10^{-6}$ dyn s cm^{-1}, while at a temperature of 500 K, $\zeta_0(500\,{\rm K}) = 2.63 \times 10^{-9}$ dyn s cm^{-1} $= 2.63 \times 10^{-12}$ kg/s.

The ζ parameter refers to a monomer of PI moving through its environment. Since, we are dealing with larger entities, we analyze it as:

$$\zeta_0 \left(T\right) = \gamma\left(T\right) \cdot m_{\rm monomer}$$

where $\gamma(T)$ is measured in s^{-1} and can be then multiplied by the mass of the Brownian bead which consists of several PI monomers.

Definition at line 47 of file b3D_integrator.cpp.

4.1.3 Member Function Documentation

4.1.3.1 integrate()

```
void NetworkNS::cb3D_integrator::integrate (
            unsigned int nsteps,
            double dt,
            unsigned int nstout )
```

Parameters

`in`	*nsteps*	The number of integration steps to be carried out.
`in`	*dt*	Integration timestep in ps.
`in`	*nstout*	Every how many steps a report is given to the user.

For large values of $\gamma\Delta t$ in the diffusive regime, when the friction is so strong that the velocities relax within Δt. For example for $\zeta_0(500\,{\rm K}) = 2.63 \times 10^{-12}$ kg/s, $\gamma = 2.325 \times 10^{13}$ s^{-1} and thus $\gamma\Delta t = 23.25$ for $\Delta t = 10^{-12}$ s. The Brownian Dynamics algorithm consists of the following equation of motion:

$$x(t_n + \Delta t) = x(t_n) + \frac{\Delta t}{m_{bead}\gamma} F\left(t_n\right) + X_n\left(\Delta t\right)$$

The coefficient $\Delta t / \left(\gamma(T) m_{\rm bead}\right)$ is measured in ${\rm s}^2/({\rm g/mol})$. The forces, are measured in ${\rm kJ}/({\rm mol\AA})$. Thus, there is a factor of 10^{26} which multiplies the force in order to make it compatible with the $\Delta t / (\gamma(T) m_{\text{bead}})$ product:

$$\frac{\text{kJ}}{\text{mol}\text{Å}} = 10^{26}\frac{\text{g Å}}{\text{mol s}^2}$$

Random displacements, $X_n(\Delta t)$ are sampled from a Gaussian distribution with zero mean and width:

$$\left\langle X_n^2(\Delta t) \right\rangle = 2k_{\text{B}}T\frac{\Delta t}{m_{\text{bead}}\gamma(T)}$$

Note that the full free energy of the system will include a contribution from the entropy elasticity of the strands between nodal points,

Definition at line 186 of file b3D_integrator.cpp.

4.1.3.2 report()

```
void NetworkNS::cb3D_integrator::report (
            unsigned int istep,
            double b_energy,
            double nb_energy )
```

The function taking care of reporting the current status of the simulation to both the standard output and the dump file.

Parameters

in	*istep*	The current step of the Brownian Dynamics integration.
in	*b_energy*	The bonded energy of the system at the current timestep.
in	*nb_energy*	The non-bonded energy of the system at the current timestep.

Definition at line 334 of file b3D_integrator.cpp.

4.1.3.3 bonded_force_calculation()

```
double NetworkNS::cb3D_integrator::bonded_force_calculation (
            bool stress_calc )
```

A function for evaluating the forces and virials due to the bonded interactions of the system. Both entropic springs along the chain backbone and slip-springs representing entanglements are taken into account.

Parameters

in	*stress_calc*	Boolean variable controlling whether stress calculation will take place.

The stress tensor of the atom i, $\sigma_{i,\text{a},\text{b}}$ is given by the following formula, where a and b take on values x, y, z to generate the six components of the symmetric tensor:

$$\sigma_{i,\text{a},\text{b}} = -\frac{1}{2}\sum_{j=1}^{N_{\text{b}}(i)} (r_{i,a} - r_{j,a})^{\text{min.im.}} F_{ij,b}^{\text{min.im.}}$$

where $N_{\text{b}}(i)$ stands for the number of bonds atom i participates to.

Definition at line 404 of file b3D_integrator.cpp.

4.1.3.4 simpler_scheme_non_bonded_force_calculation()

```
double NetworkNS::cb3D_integrator::simpler_scheme_non_bonded_force_calculation (
            void  )
```

To deal with nonbonded (excluded volume and van der Waals attractive) interactions in the network representation, we introduce a network free energy:

$$A_{\mathrm{nb}} = \int d^3\mathbf{r} f\left[\rho\left(\mathbf{r}\right)\right]$$

In the above equation, $\rho\left(\mathbf{r}\right)$ is the local density (number of Kuhn segments per unit volume) at position $\mathbf{r}$ and $f\left(\rho\right)$ is a free energy density (free energy per unit volume). Expressions for $f\left(\rho\right)$ may be extracted from an equation of state. Here the plan is to invoke a simple expression for $f\left(\rho\right)$, in the form of a Taylor expansion,

$$f\left(\rho\right) = C\rho + B\rho^2$$

with C, B fitted such that the volumetric properties (pressure and compressibility at mean density of interest) are reproduced.

Local density will be resolved only at the level of entire cells, defined by passing an orthogonal grid through the entire system. The free energy of the system is approximated by

$$A_{\mathrm{nb}} = \sum_{\mathrm{cells}} V_{\mathrm{cell}}^{\mathrm{acc}} f\left(\rho_{\mathrm{cell}}\right)$$

where $V_{\mathrm{cell}}^{\mathrm{acc}}$, the accessible of a cell, is the volume of the rectangular parallelepiped defining the cell minus the volume of any parts of nanoparticles that may find themselves in the cell.

The cell density ρ_{cell} must be defined based on the nodal points in and around the cell, each nodal point contributing a mass equal to the node's mass. Each nodal point j has mass n_j (in Kuhn segments) and a characteristic size R_j. We will discuss below how these quantities depend on the node's molecular characteristics. We denote the position vector of node j by $\mathbf{r}_j = \left(x_j, y_j, z_j\right)$. The cell dimensions along the x, y, z directions will be denoted as L_x, L_y, L_z, respectively.

We will focus on a cell extending between $x_{\mathrm{cell}} - L_x$ and x_{cell} along the x-direction, between $y_{\mathrm{cell}} - L_y$ and y_{cell} along the y-direction, and between $z_{\mathrm{cell}} - L_z$ and z_{cell} along the z-direction. In the regular grid considered, if $(0,0,0)$ is taken as one of the grid points x_{cell}, y_{cell}, and z_{cell} will be integer multiples of L_x, L_y and L_z, respectively.

In the following we will assume that

$$R_j < \min\left(L_x, L_y, L_z\right)$$

The simplest option for relating the positions and masses of the node to ρ_{cell} is to envision each node j as a cube containing n_j Kuhn segments, of edge length R_j, centered at $\mathbf{r}_j$. Node j will contribute to a cell if its cube (cube j) overlaps with the cell. Note that, for this to happen, it is not necessary that the nodal position of the center, $\mathbf{r}_j$, lie in the cell. The mass (number of Kuhn segments) contributed by the node to the cell is:

$$n_{j,\mathrm{cell}} = n_j \frac{V_{\mathrm{cube}\ j\cap\mathrm{cell}}}{V_{\mathrm{cube}\ j}}$$

with $V_{\mathrm{cube}\ j\cap\mathrm{cell}}$ being the volume of the intersection of cube j, associated with node j, and the considered cell, while $V_{\mathrm{cube}\ j} = R_j^3$ is the volume of cube j.

Under the condition $R_j < \min\left(L_x, L_y, L_z\right)$, $V_{\mathrm{cube}\ j\cap\mathrm{cell}}$ is obtainable as:

$$\begin{aligned} V_{\mathrm{cube}\ j\cap\mathrm{cell}} &= \max\left\{\left[\min\left(x_j + \frac{R_j}{2}, x_{\mathrm{cell}}\right) - \max\left(x_j - \frac{R_j}{2}, x_{\mathrm{cell}} - L_x\right)\right], 0\right\} \\ &\times \max\left\{\left[\min\left(y_j + \frac{R_j}{2}, y_{\mathrm{cell}}\right) - \max\left(y_j - \frac{R_j}{2}, y_{\mathrm{cell}} - L_y\right)\right], 0\right\} \\ &\times \max\left\{\left[\min\left(z_j + \frac{R_j}{2}, z_{\mathrm{cell}}\right) - \max\left(z_j - \frac{R_j}{2}, z_{\mathrm{cell}} - L_z\right)\right], 0\right\} \end{aligned}$$

As defined by the above equation, $V_{\mathrm{cube}\ j\cap\mathrm{cell}}$ is a linear function of the node coordinates. Clearly, if cube j lies entirely within the cell, $V_{\mathrm{cube}\ j\cap\mathrm{cell}} = V_{\mathrm{cube}\ j}$ and, consequently, $n_{j,\mathrm{cell}} = n_j$. If however, the borders of cube j intersect the borders of the considered cell, then node j will contribute a mass $n_{j,\mathrm{cell}} < n_j$ to the cell. The total mass contributed by bead j to all cells in which it participates will always be n_j.

The density ρ_{cell} in the considered cell is estimated as:

$$\rho_{\mathrm{cell}} = \frac{1}{V_{\mathrm{cell}}^{\mathrm{acc}}} \sum_j n_{j,\mathrm{cell}}$$

Clearly, only nodal points j whose cubes have a nonzero overlap with the considered cell will contribute to the above summation. The positions vectors $\mathbf{r}_j$ of these beads will necessarily lie within the considered cell or its immediate neighbors.

The non-bonded energy is considered to be a quadratic function of the density, i.e.,

$$A_{\mathrm{nb}} = \sum_{i\in\mathrm{cells}} V_{\mathrm{cell},i}\left(C_1\rho_i + C_2\rho_i^2\right)$$

The precise conditions for cube j to have common points with the considered cell are:

$$x_{\mathrm{cell}} - L_x < x_j + \frac{R_j}{2} < x_{\mathrm{cell}} + R_j$$

$$y_{\mathrm{cell}} - L_y < y_j + \frac{R_j}{2} < y_{\mathrm{cell}} + R_j$$

$$z_{\mathrm{cell}} - L_z < z_j + \frac{R_j}{2} < z_{\mathrm{cell}} + R_j$$

According to the above approach, the force on node j due to nonbonded interactions is:

$$\mathbf{F}_j = -\nabla_{\mathbf{r}_j} A_{\mathrm{nb}} = -\sum_{\substack{\text{cells having common}\\ \text{points with cube } j}} V_{\mathrm{cell}}^{\mathrm{acc}} \left.\frac{df}{d\rho}\right|_{\rho=\rho\mathrm{cell}} \nabla_{\mathbf{r}_j}\rho_{\mathrm{cell}}$$

Definition at line 513 of file b3D_integrator.cpp.

4.1.3.5 calculate_pressure()

```
void NetworkNS::cb3D_integrator::calculate_pressure (
            double * press_tens )
```

Sum the per-atom stresses to calculate the pressure of the simulation box.

Parameters

out	*press_tens	An array containing the six components of the box pressure tensor.

A function which accumulates the per-atom stresses in order to calculate the pressure of the simulation box.

The per-atom stress is the negative of the per-atom pressure tensor. It is also really a stress∗volume formulation, meaning the computed quantity is in units of pressure∗volume:

$$\frac{\mathrm{kJ}}{\mathrm{mol}} = \frac{10^{33}}{6.022\times 10^{23}} \frac{\mathrm{kg}}{\mathrm{m\,s^2}}$$

Thus, if the diagonal components of the per-atom stress tensor are summed for all beads in the system and the sum is divided by $3V$, where V is the volume of the system, the result should be $-p$, where p is the total pressure of the system.

Definition at line 363 of file b3D_integrator.cpp.

4.1.3.6 from_bd_x_to_polymer_network()

```
void NetworkNS::cb3D_integrator::from_bd_x_to_polymer_network (
             void  )  [private]
```

The positions of the bead are updated.

Definition at line 495 of file b3D_integrator.cpp.

4.1.4 Member Data Documentation

4.1.4.1 bd_gamma

```
double* NetworkNS::cb3D_integrator::bd_gamma
```

The friction coefficient, γ, of every degree of freedom, measured in s^{-1}.

Definition at line 60 of file b3D_integrator.h.

4.1.4.2 bd_mass

```
double* NetworkNS::cb3D_integrator::bd_mass
```

The mass, m_i, of every degree of freedom, measured in g/mol.

Definition at line 63 of file b3D_integrator.h.

4.1.4.3 bd_x

```
double* NetworkNS::cb3D_integrator::bd_x
```

The current position of the beads during the Brownian Dynamics integration, at the current timestep, i.e. $\mathbf{r}_i(t)$. The size of the vector is three times the degrees of freedom of the BD simulation.

Definition at line 66 of file b3D_integrator.h.

4.1.4.4 bd_f

```
double* NetworkNS::cb3D_integrator::bd_f  [private]
```

The forces due to bonded interactions during the Brownian Dynamics simulation, at the current timestep, i.e. $\mathbf{F}_i(t)$.

Definition at line 83 of file b3D_integrator.h.

4.1.4.5 bd_nb_f

```
double* NetworkNS::cb3D_integrator::bd_nb_f  [private]
```

The forces due to nonbonded interactions duringn the BD simulations, at the current timestep. The size of the vector is three times the degrees of freedom of the BD simulation.

Definition at line 87 of file b3D_integrator.h.

4.1.4.6 bd_x_ps

```
double* NetworkNS::cb3D_integrator::bd_x_ps  [private]
```

The positions of the beads of the BD simulation at the previous timestep. The size of the vector is

Definition at line 91 of file b3D_integrator.h.

4.1.4.7 bd_f_ps

```
double* NetworkNS::cb3D_integrator::bd_f_ps  [private]
```

The forces acted on the beads durign the previous timestep of the BD simulation.

Definition at line 94 of file b3D_integrator.h.

4.1.4.8 xshift

```
double* NetworkNS::cb3D_integrator::xshift  [private]
```

The vector for keeping the x coordinates of the beads with respect to the nonbonded free energy estimation grid, xshift $\in [0, L_{\mathrm{x}})$ with L_{x} being the x edge length of the simulation box.

Definition at line 101 of file b3D_integrator.h.

4.1.4.9 yshift

```
double* NetworkNS::cb3D_integrator::yshift  [private]
```

The vector for keeping the y coordinates of the beads with respect to the nonbonded free energy estimation grid, yshift $\in [0, L_{\mathrm{y}})$ with L_{y} being the y edge length of the simulation box.

Definition at line 105 of file b3D_integrator.h.

4.1.4.10 zshift

```
double* NetworkNS::cb3D_integrator::zshift  [private]
```

The vector for keeping the z coordinates of the beads with respect to the nonbonded free energy estimation grid, zshift $\in [0, L_{\mathrm{z}})$ with L_{z} being the z edge length of the simulation box.

Definition at line 109 of file b3D_integrator.h.

4.1.4.11 gamma_mass_opt

```
bool NetworkNS::cb3D_integrator::gamma_mass_opt  [private]
```

A boolean variable indicating whether or no the BD integrator run in an optimized way. If true, all beads have the same mass and friction coefficient, so the integrator does not use arrays in order to speed-up the execution.

Definition at line 117 of file b3D_integrator.h.

4.1.4.12 den_dx

```
double* NetworkNS::cb3D_integrator::den_dx  [private]
```

The partial derivative of the local cell density with respect to the x-coordinate of a bead, $\nabla_{\mathbf{r}_j}\rho_{\mathrm{cell}}$.

Definition at line 123 of file b3D_integrator.h.

4.1.4.13 den_dy

```
double* NetworkNS::cb3D_integrator::den_dy  [private]
```

The partial derivative of the local cell density with respect to the y-coordinate of a bead, $\nabla_{\mathbf{r}_j}\rho_{\mathrm{cell}}$.

Definition at line 126 of file b3D_integrator.h.

4.1.4.14 den_dz

```
double* NetworkNS::cb3D_integrator::den_dz  [private]
```

The partial derivative of the local cell density with respect to the z-coordinate of a bead, $\nabla_{\mathbf{r}_j}\rho_{\mathrm{cell}}$.

Definition at line 129 of file b3D_integrator.h.

The documentation for this class was generated from the following files:

- b3D_integrator.h
- b3D_integrator.cpp

4.2 D3DVector< T > Class Template Reference

A class describing a vector with three components of type T.

```
#include <Auxiliary.h>
```

Public Member Functions

- D3DVector (T a, T b, T c)

  The constructor of the D3DVector class, assigning value to each one of the three components.
- D3DVector ()

  Constructor which initializes all of the three components of the vector to zero.
- void zero ()

  Set all components of the vector to zero.
- D3DVector operator= (const D3DVector &rhs)

  The assignment operator defined for the D3DVector vector.
- D3DVector operator- (const D3DVector &rhs)

  The substraction operator defined for two D3DVector vectors.

- D3DVector operator+ (const D3DVector &rhs)

  The addition operator defined for two D3DVector vectors.
- D3DVector operator! (void)

  The operator "!" is used for calculating the norm of a D3DVector vector.
- T norm ()

  A function calculating the norm of a D3DVector vector.
- T dotproduct (const D3DVector &rhs)

  The dot (inner) product defined for two D3DVector vectors.
- T operator∗ (const D3DVector &rhs)

  The operator "∗" is used for the dot product between two D3DVector vectors.
- D3DVector operator∗ (T mult)
- D3DVector operator/ (T mult)

  The operator "/" is used for the scalar division between a D3DVector vector and a number.
- D3DVector crossproduct (const D3DVector &rhs)

  The function caclculating the cross product between two vectors.
- D3DVector operator^ (const D3DVector &rhs)

  The operator "^" is used for the cross product between D3DVector vectors.
- D3DVector triplevec (D3DVector &a, D3DVector &b)

  The triple vector product between two D3DVector vectors.
- T triplescal (D3DVector &a, D3DVector &b)

  The mixed product.

Public Attributes

- T x

  The x-component of the D3DVector vector.
- T y

  The y-component of the D3DVector vector.
- T z

  The z-component of the D3DVector vector.

4.2.1 Detailed Description

template<class T>
class D3DVector< T >

Adapted from: `http://rosettacode.org/`

Definition at line 77 of file Auxiliary.h.

4.2.2 Member Function Documentation

4.2.2.1 operator∗()

```
template<class T >
D3DVector D3DVector< T >::operator∗ (
             T mult )  [inline]
```

The operator "∗" defined for the scalar product between a single number and a D3DVector vector.

Definition at line 139 of file Auxiliary.h.

The documentation for this class was generated from the following file:

- Auxiliary.h

4.3 NetworkNS::Domain Class Reference

The class of the simulation domain.

```
#include <domain.h>
```

Public Member Functions

- Domain (std::string)
- virtual ∼Domain ()

  The destructor of the Domain class.
- void minimum_image (double &x, double &y, double &z)

  Apply minimum image convention along the three spatial directions.
- void zero_to_length_minimum_image (double &, double &, double &)

  Put the coordinates inside the primary simulation box.
- void put_in_primary_box (double $*$, int $*$)

  Put the coordinates inside the primary simulation box.

Public Attributes

- int BoxExists
- int NonPeriodic
- int Xperiodic

  Periodicity along x direction, 0 = non-periodic, 1 = periodic.
- int Yperiodic

  Periodicity along y direction, 0 = non-periodic, 1 = periodic.
- int Zperiodic

  Periodicity along z direction, 0 = non-periodic, 1 = periodic.
- int Periodicity [3]

  xyz periodicity as an array.
- int Boundary [3][2]
- double BoxLow [3]

  Orthogonal box global lower bounds along all three directions.
- double BoxHigh [3]

  Orthogonal box global lower bounds along all three directions.
- double XBoxLen

  Simulation box edge length along x direction, L_x.
- double YBoxLen

  Simulation box edge length along y direction, L_y.
- double ZBoxLen

  Simulation box edge length along z direction, L_z.
- double iXBoxLen

  Inverse simulation box edge length along x direction, $1/L_x$.
- double iYBoxLen

  Inverse simulation box edge length along y direction, $1/L_y$.
- double iZBoxLen

  Inverse simulation box edge length along z direction, $1/L_z$.

4.3.1 Detailed Description

Definition at line 26 of file domain.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Domain()

```
NetworkNS::Domain::Domain (
             std::string filename )
```

The constructor of the Domain class, where only the name of the data file has to be provided.

Definition at line 29 of file domain.cpp.

4.3.3 Member Data Documentation

4.3.3.1 BoxExists

```
int NetworkNS::Domain::BoxExists
```

An integer variable denoting whether the simulation box exists or no. 0 = not yet created, 1 = exists

Definition at line 28 of file domain.h.

4.3.3.2 NonPeriodic

```
int NetworkNS::Domain::NonPeriodic
```

An integer variable denoting whether the simulation box is periodic or no. 0 = periodic in all 3 dims 1 = periodic or fixed in all 6 2 = shrink-wrap in any of 6

Definition at line 30 of file domain.h.

4.3.3.3 Boundary

```
int NetworkNS::Domain::Boundary[3][2]
```

Settings for 6 boundaries 0 = periodic, 1 = fixed non-periodic, 2 = shrink-wrap non-periodic 3 = shrink-wrap.

Definition at line 41 of file domain.h.

The documentation for this class was generated from the following files:

- domain.h
- domain.cpp

4.4 NetworkNS::dump Class Reference

The class which dumps a snapshot of atom quantities (positions, atomic-level stresses) to one or more files every a predefined number of timesteps.

`#include <dump.h>`

Public Member Functions

- dump (std::string)

  The constructor of the dump class where the name of the dump file is specified.
- dump (const dump &orig)

  The destructor of the class.
- void add_snapshot_to_dump (const class NetwMin ∗, const class cb3D_integrator ∗, unsigned int)

  Add the current snapshot to the dump file.

Private Attributes

- std::ofstream my_file

  The output file stream corresponding to the dump file.

4.4.1 Detailed Description

Definition at line 30 of file dump.h.

4.4.2 Member Function Documentation

4.4.2.1 add_snapshot_to_dump()

```
void NetworkNS::dump::add_snapshot_to_dump (
             const class NetwMin * netw_app,
             const class cb3D_integrator * b3D,
             unsigned int timestep )
```

Parameters

in	*netw_app*	A pointer to a network application, which contains all relevant information concerning the topology of the system under investigation.
in	*b3D*	A pointer to a Brownian Dynamics integrator containing the current positions of the bead in the course of the simulation.
in	*timestep*	The current timestep of the simulation.

Definition at line 40 of file dump.cpp.

The documentation for this class was generated from the following files:

- dump.h
- dump.cpp

4.5 NetworkNS::Grid Class Reference

The nonboned free energy estimation grid class.

`#include <grid.h>`

Public Member Functions

- Grid (double, double, double, int, int, int)

  The constructor of the Grid class.
- virtual ∼Grid ()

  The destructor of the Grid class.
- int find_grid_cell (const double &xnode, const double &ynode, const double &znode)

  A routine for finding the cell to which a bead belongs to.

Public Attributes

- double dlx

  The grid spacing along the x direction, ΔL_x.
- double dly

  The grid spacing along the y direction, ΔL_y.
- double dlz

  The grid spacing along the z direction, ΔL_z.
- double idlx

  The inverse of the grid spacing along the x direction, $1/(\Delta L_x)$.
- double idly

  The inverse of the grid spacing along the y direction, $1/(\Delta L_y)$.
- double idlz

  The inverse of the grid spacing along the z direction, $1/(\Delta L_z)$.
- double vcell
- double ivcell

  The inverse of the volume of the grid cell, i.e. $1/V_{\text{cell}}$.
- int ncellx

  Number of cells along x direction.
- int ncelly

  Number of cells along y direction.
- int ncellz

  Number of cells along z direction.
- int ncells

  Total number of cells.
- grid_cell $*$ cells

  A vector containing the cells of the computation grid.

4.5.1 Detailed Description

Definition at line 33 of file grid.h.

4.5.2 Member Data Documentation

4.5.2.1 vcell

```
double NetworkNS::Grid::vcell
```

The volume of the grid cell, $V_{\text{cell}} = \Delta L_x \times \Delta L_y \times \Delta L_z$.

Definition at line 52 of file grid.h.

The documentation for this class was generated from the following files:

- grid.h
- grid.cpp

4.6 NetworkNS::Hopping Class Reference

The class of the hopping kinetic Monte Carlo scheme. It can be called by a Brownian Dynamics class, get all the necessary information from it and alter the connectivity of the system, based on the rates described in hopping.cpp .

```
#include <hopping.h>
```

Public Member Functions

- Hopping (double)

  slipsprings is created.
- ∼Hopping ()

  The destructor of the class.
- void hopping_step (class NetwMin ∗netapp, const class cb3D_integrator ∗b3D, double ∗pos_array, double temperature, double elapsed_time)

  The routine which executes a single kinetic Monte Carlo slipspring hopping step.

Private Attributes

- std::ofstream lifetimes_file

  A file for writing the lifetimes of the slipsprings to.
- std::ofstream events_file

  A file for writing the hopping events taking place.
- double **nu_hopping_times_exp_of_barrier**

4.6.1 Detailed Description

Definition at line 28 of file hopping.h.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 Hopping()

```
NetworkNS::Hopping::Hopping (
            double hopping_rate_constant )
```

The constructor of the class, where a file for writing the lifetime of the

The constructor takes care of opening a file to write the life-time of slip-spring to.

Definition at line 31 of file hopping.cpp.

4.6.2.2 ∼Hopping()

```
NetworkNS::Hopping::∼Hopping ( )
```

The destructor which takes care of closing the file of the slip-springs lifetimes.

Definition at line 48 of file hopping.cpp.

4.6.3 Member Function Documentation

4.6.3.1 hopping_step()

```
void NetworkNS::Hopping::hopping_step (
            class NetwMin * netapp,
            const class cb3D_integrator * b3D,
            double * pos_array,
            double temperature,
            double elapsed_time )
```

Parameters

in	*netapp*	A pointer to the original application.
in	*b3D*	A pointer to a Brownian Dynamics simulation scheme, in order to extract the current positions of the beads.
in	*pos_array*	The array containing the positions of the beads.
in	*temperature*	The temperature at which the rates will be calculated.
in	*elapsed_time*	The time in ps elapsed from the previous call to the hopping routine.

In order to develop a formalism of elementary events of slip-spring hopping, creation or destruction, we need expressions for the rate of slippage along the chain backbone. In order to extract the diffusivity of the slip-springs, we will proceed along the lines of Terzis and Theodorou work. [6] We describe self-diffusion along the chain contour with the Rouse model. The Rouse model addresses the dynamics of polymers in unentangled melts. A polymer chain is represented by a set of beads connected by harmonic springs. The dynamics, as in our simulations, are modeled as a Brownian motion of these tethered beads, the environment of a chain being represented as a continuum (viscous medium), ignoring all excluded volume and hydrodynamic interactions.

In this model the self-diffusion of the center of the mass of the polymer is related to the friction coefficient, ζ on a bead by:

$$D_{\mathrm{Rouse}} = \frac{k_{\mathrm{B}}T}{N\zeta}$$

with N being the number of beads per chain. In the picture we invoke in our network model, the center of mass diffusivity along the contour is related to the rate of slip-spring jumps across beads (by distance $\left(n_{\mathrm{Kuhns/bead}}b^2\right)^{1/2}$ in each direction by (see below for the definition of v_{diff})

$$D_{\mathrm{Rouse}} = k_{\mathrm{diff}}\frac{n_{\mathrm{Kuhns/bead}}b^2}{N} = v_{\mathrm{diff}}\frac{n_{\mathrm{Kuhns/bead}}b^2}{N}\exp\left(-\frac{A_0}{k_{\mathrm{B}}T}\right)$$

Hence, one must have:

$$v_{\mathrm{diff}} = \frac{k_{\mathrm{B}}T}{n_{\mathrm{Kuhns/bead}}b^2\zeta}\exp\left(-\frac{A_0}{k_{\mathrm{B}}T}\right)$$

where A_0 is a free energy per slip-spring in the equilibrium melt, which establishes a baseline for measuring free energies.

At every step of the 3D Brownian Dynamics simulation, where hopping kinetic Monte Carlo takes place, every free end of the system can randomly create a new slip-spring with an internal bead of a neighboring chain. This may be accomplished by a rate constant k_{creation}. The rate for the creation of a slip-spring is closely related to the probability of pairing the end “a" with one of its candidate mates which lie inside a sphere of prescribed radius R_{attempt}. The definition of the probability implies that the more crowded chain ends are the more probable to create a slip-spring. The number of neighbors around a chain end can be tuned via the radius of the sphere within which the search takes place, R_{attempt}. A good estimate of R_{attempt} for polyisoprene (either pure or crosslinked) can be given by the tube diameter of the polymer. A computational study of the tube diameter of the polyisoprene as a function of the molecular weight has been done by the Li et al. [7] The rate constant k_{creation} can be treated as an adjustable parameter of our model, which will be used to ensure that the average number of slip-spring present in the system is conserved throughout the simulation.

Definition at line 65 of file hopping.cpp.

The documentation for this class was generated from the following files:

- hopping.h
- hopping.cpp

4.7 NetworkNS::NetwMin Class Reference

The class of the host application itself. It contains pointers to all constituents, e.g. the simulation domain, random number generator, etc.

```
#include <netmin.h>
```

Public Member Functions

- NetwMin (std::string)
- void write_network_to_lammps_data_file ()

Public Attributes

- class Network ∗ network
- class Domain ∗ domain
- class Grid ∗ grid
- class RanMars ∗ my_rnd_gen

  A pointer to the class of the random number generator.
- class dump ∗ my_traj_file

4.7.1 Detailed Description

Definition at line 36 of file netmin.h.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 NetwMin()

```
NetworkNS::NetwMin::NetwMin (
             std::string filename )
```

The constructor of the application. The only input is the name of the data file to read the initial configuration from.

Parameters

in	*filename*	The name of the data file to be read in order to initialize the simulation.

Definition at line 23 of file netmin.cpp.

4.7.3 Member Function Documentation

4.7.3.1 write_network_to_lammps_data_file()

```
void NetworkNS::NetwMin::write_network_to_lammps_data_file ( )
```

A function for writing the current configuration of the system in LAMMPS-like format. It can be used for restarting a simulation.

Definition at line 43 of file netmin.cpp.

4.7.4 Member Data Documentation

4.7.4.1 network

```
class Network* NetworkNS::NetwMin::network
```

A pointer to a network class hosting the connectivity of the system.

Definition at line 38 of file netmin.h.

4.7.4.2 domain

```
class Domain* NetworkNS::NetwMin::domain
```

A pointer to a domain class hosting the dimensions of the simulation domain.

Definition at line 41 of file netmin.h.

4.7.4.3 grid

```
class Grid* NetworkNS::NetwMin::grid
```

A pointer to a grid class hosting the information about the nonbonded free energy estimation grid.

Definition at line 44 of file netmin.h.

4.7.4.4 my_traj_file

```
class dump* NetworkNS::NetwMin::my_traj_file
```

A pointer to the class which takes care of writing the trajectory file of the simulation.

Definition at line 49 of file netmin.h.

The documentation for this class was generated from the following files:

- netmin.h
- netmin.cpp

4.8 NetworkNS::Network Class Reference

The class which stores all information concerning the polymeric network.

```
#include <network.h>
```

Public Member Functions

- Network (class NetwMin $*$, std::string)

    The constructor of the class.
- virtual ∼Network ()

    The destructor of the class.

Public Attributes

- std::list< tNode > nodes
- std::list< tStrand > strands
- std::list< tSubCh > subchains
- std::list< tStrand $*$ > pslip_springs
- std::vector< tBead_type > node_types
- std::vector< tBond_type > bond_types
- std::vector< std::list< tStrand $*$ > > sorted_chains

4.8.1 Detailed Description

Definition at line 25 of file network.h.

4.8.2 Constructor & Destructor Documentation

4.8.2.1 Network()

```
NetworkNS::Network::Network (
            class NetwMin * netw_min,
            std::string filename )
```

Parameters

in	*netw_min*	A pointer to the application which initializes the constructor. The application is a class NetwMin.

Parameters

`in`	*filename*	A standard C++ string containing the name of the data file to open in order to read the polymeric network from.

Definition at line 46 of file network.cpp.

4.8.3 Member Data Documentation

4.8.3.1 nodes

`std::list<tNode> NetworkNS::Network::nodes`

A list of the nodes the network consists of. The nodes are described by using the sNode stucture.

Definition at line 31 of file network.h.

4.8.3.2 strands

`std::list<tStrand> NetworkNS::Network::strands`

A list of all strands present in the network, described by using the sStrand structure.

Definition at line 33 of file network.h.

4.8.3.3 subchains

`std::list<tSubCh> NetworkNS::Network::subchains`

A list of subchains present in the network, described by using the tSubCh structure.

Definition at line 35 of file network.h.

4.8.3.4 pslip_springs

`std::list<tStrand *> NetworkNS::Network::pslip_springs`

A list of slip-springs present in the network. In order to avoid duplicated occurences of the slip-spring strands, we use pointers to strands stored in the nodes list defined above.

Definition at line 37 of file network.h.

4.8.3.5 node_types

`std::vector<tBead_type> NetworkNS::Network::node_types`

A vector which stores the desctiption of bead types present in the system.

Definition at line 40 of file network.h.

4.8.3.6 bond_types

`std::vector<tBond_type> NetworkNS::Network::bond_types`

A vector which stores the desctiption of bond types present in the system.

Definition at line 42 of file network.h.

4.8.3.7 sorted_chains

`std::vector<std::list<tStrand *> > NetworkNS::Network::sorted_chains`

Each element of the sorted_chains vector consists of a list of pointers to the internal strands a polymeric chain consists of. The pointers refer to the array strands defined above.

Definition at line 45 of file network.h.

The documentation for this class was generated from the following files:

- network.h
- network.cpp

4.9 NetworkNS::RanMars Class Reference

The class of the pseudorandom number generator. It is based on Marsaglia's KISS design.

`#include <rng.h>`

Public Member Functions

- RanMars (int)

  The constructor of the random number generator class.
- ~RanMars ()

  The destructor of the random number generator class.
- double uniform ()

  Function generating unifomly distributed random numbers in [0,1).
- double gaussian ()
- double modified_gaussian (double mean, double stdev)
- double **rand_gauss** (void)
- unsigned int **devrand** ()
- unsigned int **uint_rand** ()

Private Attributes

- int seed

  The seed used for the pseudorandom number generator.
- int **save**
- double **second**
- double ∗ **u**
- int **i97**
- int **j97**
- unsigned int **x**
- unsigned int **y**
- unsigned int **z**
- unsigned int **c**

4.9.1 Detailed Description

Definition at line 23 of file rng.h.

4.9.2 Member Function Documentation

4.9.2.1 uniform()

```
double NetworkNS::RanMars::uniform ( )
```

A random number generator returning number in the interval [0,1).

Definition at line 97 of file rng.cpp.

4.9.2.2 gaussian()

```
double NetworkNS::RanMars::gaussian ( )
```

Function generating random numbers distributed according to a Gaussian distribution centered at zero with unit standard deviation.

Definition at line 112 of file rng.cpp.

4.9.2.3 modified_gaussian()

```
double NetworkNS::RanMars::modified_gaussian (
             double mean,
             double stdev )
```

A function returning a modified Gaussian function, centered at a specified mean and having a pre-specified deviation.

Definition at line 134 of file rng.cpp.

The documentation for this class was generated from the following files:

- rng.h
- rng.cpp

4.10 sBead_type Struct Reference

An elementary data type for reading in the information concerning a bead.

```
#include <net_types.h>
```

Public Attributes

- double mass

  The mass of the bead type in g/mol.
- double n_mass

  The mass of the bead type in number of Kuhn segments.

- double r_node

  The edge length of the bead, if its mass is smeared into a cube.

4.10.1 Detailed Description

Definition at line 100 of file net_types.h.

The documentation for this struct was generated from the following file:

- net_types.h

4.11 sBond_type Struct Reference

An elementary data type for reading in the information concerning a strand.

```
#include <net_types.h>
```

Public Attributes

- double spring_coeff
- double sq_ete

  The equilibrium squared end-to-end distance of the strand.
- double kuhnl

  The Kuhn length of the underlying Kuhn segments of the strand, i.e. b.

4.11.1 Detailed Description

Definition at line 108 of file net_types.h.

4.11.2 Member Data Documentation

4.11.2.1 spring_coeff

```
double sBond_type::spring_coeff
```

The spring coefficient of the strand. This quantity depends on the nature of the potential used to describe the specific strand. More can be found in distributions.cpp file documentation.

Definition at line 110 of file net_types.h.

The documentation for this struct was generated from the following file:

- net_types.h

4.12 NetworkNS::sgrid_cell Struct Reference

```
#include <grid.h>
```

Public Attributes

- int Id

  The ID of the cell.
- double Vec [3]

  The position of the center of the cell.
- int neigh [27]

  An array of the neighboring cells. The first record is the ID of the cell itself.

4.12.1 Detailed Description

The sgrid_cell is the elementary struct for storing the information concerning a cell of the free energy estimation grid.

Definition at line 23 of file grid.h.

The documentation for this struct was generated from the following file:

- grid.h

4.13 sNode Struct Reference

The sNode is the basic struct keeping all information relevant to a bead or a network node. Once it is defined, it is converted to type tNode, which is used throughout the application.

```
#include <net_types.h>
```

Public Attributes

- int Id

  The identity tag of the nodal point.
- int Type
- double Pos [3]

  The position of the node.
- std::vector< int > OrChains

  The chain or chains to which the node belongs to.
- std::vector< sStrand ∗ > pStrands

  A vector of pointers to the strands the node is connected to.
- std::vector< int > SubCh

  A vector of the IDs of the subchains the node is part of.
- std::vector< sChain ∗ > pChains

  A vector of pointers to the chains the node belongs to.
- int node_cell
- double n_mass

  Mass of the node in Kuhn segments.
- double mass

  Mass of the node in g/mol (molecular weight).
- double r_node
- double r_star

4.13.1 Detailed Description

Definition at line 33 of file net_types.h.

4.13.2 Member Data Documentation

4.13.2.1 Type

`int sNode::Type`

The type of the nodal point:

- "1" corresponds to chain ends,
- "2" corresponds to internal beads, and
- "3" corresponds to crosslinks.

Definition at line 35 of file net_types.h.

4.13.2.2 node_cell

`int sNode::node_cell`

The cell of the density estimation grid the node belongs to. This feature seems obsolete. It may be removed in future version.

Definition at line 50 of file net_types.h.

4.13.2.3 r_node

`double sNode::r_node`

Edge length of a cube formed around the node for the estimation of nonbonded interactions.

Definition at line 57 of file net_types.h.

4.13.2.4 r_star

`double sNode::r_star`

Edge length of a cube formed around the node for the estimation of nonbonded interactions, computed by employing a star polymer approximation. This feature is obsolete. It will be removed in a future version.

Definition at line 60 of file net_types.h.

The documentation for this struct was generated from the following file:

- net_types.h

4.14 sStrand Struct Reference

The sStrand is the basic abstract data type keeping all relevant information concerning a strand of a chain.

`#include <net_types.h>`

Public Attributes

- int Id

  The identity tag of the strand.
- int Type
- int OrChain

  The chain to which this strand belongs (applicable only if it is an internal strand)/.
- bool slip_spring

  A boolean variable desribing whether the strand is a slip-spring or not.
- unsigned int tcreation

  The time when the strand was created. (It is useful for calculating its lifetime.)
- double spring_coeff
- double sq_end_to_end
- double kuhn_length

  The Kuhn length of the underlying Kuhn segments of the strand, i.e. b.
- std::vector$<$ tNode $* >$ pEnds

  Pointers to the nodal points the strands connects.
- double $*$ pChain

  A pointer to the chain the strand belongs to. Obsolete feature.

4.14.1 Detailed Description

Definition at line 67 of file net_types.h.

4.14.2 Member Data Documentation

4.14.2.1 Type

```
int sStrand::Type
```

The type of the strand.

- "1" stands for internal chain strands
- "2" stands for slip-springs

Definition at line 71 of file net_types.h.

4.14.2.2 spring_coeff

```
double sStrand::spring_coeff
```

The spring coefficient of the strand. This quantity depends on the nature of the potential used to describe the specific strand. More can be found in distributions.cpp file documentation.

Definition at line 81 of file net_types.h.

4.14.2.3 sq_end_to_end

```
double sStrand::sq_end_to_end
```

The equilibrium squared end-to-end distance of the strand, i.e. $\left\langle R_{\mathrm{e}}^{2} \right\rangle$.

Definition at line 85 of file net_types.h.

The documentation for this struct was generated from the following file:

- net_types.h

Chapter 5

File Documentation

5.1 Auxiliary.cpp File Reference

The C++ source code file containing some auxiliary functions.

```
#include <string>
#include <sstream>
#include <sys/time.h>
#include <vector>
#include <Auxiliary.h>
```

Functions

- std::vector< string > tokenize (std::string input_string)
- void **create_dir** (string name_of_dir)
- double **get_wall_time** ()
- double **get_cpu_time** ()

5.1.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

1.0 (January 24, 2014)

5.1.2 LICENSE

Definition in file Auxiliary.cpp.

5.1.3 Function Documentation

5.1.3.1 tokenize()

```
std::vector<string> tokenize (
            std::string input_string )
```

A function for tokenizing a string into substrings.

Definition at line 23 of file Auxiliary.cpp.

5.2 Auxiliary.cpp

```
00001
00014 #include<string>
00015 #include<sstream>
00016 #include<sys/time.h>
00017 #include<vector>
00018
00019 #include<Auxiliary.h>
00020
00021 using namespace std;
00022
00023 std::vector<string> tokenize(std::string input_string) {
00024
00025    string buf;
00026    stringstream ss(input_string);
00027    std::vector<string> tokens;
00028
00029    while (ss >> buf)
00030       tokens.push_back(buf);
00031
00032    return tokens;
00033 }
00034
00035 void create_dir(string name_of_dir){
00036    struct stat st;
00037    if (stat(name_of_dir.c_str(), &st) != 0)
00038       mkdir(name_of_dir.c_str(), 0750);
00039 }
00040
00041
00042 double get_wall_time(){
00043     struct timeval time;
00044     if (gettimeofday(&time,NULL)){
00045         //  Handle error
00046         return 0;
00047     }
00048     return (double)time.tv_sec + (double)time.tv_usec * .000001;
00049 }
00050
00051 double get_cpu_time(){
00052     return (double)clock() / CLOCKS_PER_SEC;
00053 }
00054
00055
```

5.3 Auxiliary.h File Reference

A C++ header file containing auxiliary type definitions and functions.

```
#include <string>
#include <sys/stat.h>
#include <sstream>
#include <vector>
```

Classes

- class D3DVector< T >

  A class describing a vector with three components of type T.

Functions

- std::vector< string > tokenize (std::string input_string)
- double **get_wall_time** ()
- double **get_cpu_time** ()
- template<typename T >
  T **StringToNumber** (const string &Text)
- template<typename T >
  string **NumberToString** (T Number)
- void **create_dir** (string name_of_dir)

5.3.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

1.0 (January 7, 2014)

5.3.2 LICENSE

Definition in file Auxiliary.h.

5.3.3 Function Documentation

5.3.3.1 tokenize()

```
std::vector<string> tokenize (
              std::string input_string )
```

A function for tokenizing a string into substrings.

Definition at line 23 of file Auxiliary.cpp.

5.4 Auxiliary.h

```

#ifndef AUXILIARY_H
#define  AUXILIARY_H

#include<string>
#include<sys/stat.h>
#include<sstream>
#include<string>
#include<vector>


using namespace std;

std::vector<string> tokenize(std::string input_string);
double get_wall_time();
double get_cpu_time();

template <typename T>
T StringToNumber ( const string &Text )//Text not by const reference so that the function can be used with a
{                               //character array as argument
   stringstream ss(Text);
   T result;
   return ss >> result ? result : 0;
}

template <typename T>
string NumberToString ( T Number )
{
   stringstream ss;
   ss << Number;
   return ss.str();
}


void create_dir(string name_of_dir);



static inline double powint(const double &x, const int n) {
   double yy, ww;

   if (x == 0.0) return 0.0;
   int nn = (n > 0) ? n : -n;
   ww = x;

   for (yy = 1.0; nn != 0; nn >>= 1, ww *= ww)
      if (nn & 1) yy *= ww;

   return (n > 0) ? yy : 1.0 / yy;
}



template< class T >
class D3DVector {

public :
   T x;
   T y;
   T z ;

   D3DVector( T a , T b , T c ) {
      x = a ;
      y = b ;
      z = c ;
   }

   D3DVector(){
      x = 0.0;
      y = 0.0;
      z = 0.0;
   }

   void zero (){
      x = (T)0.0;
      y = (T)0.0;
      z = (T)0.0;
   }

   D3DVector operator=(const D3DVector & rhs){
      return (D3DVector(rhs.x, rhs.y, rhs.z));
   }

   D3DVector operator-(const D3DVector & rhs){
      return (D3DVector(x-rhs.x, y-rhs.y, z-rhs.z));
   }

   D3DVector operator+(const D3DVector & rhs){
      return (D3DVector(x+rhs.x, y+rhs.y, z+rhs.z));
   }

   D3DVector operator!(void){
      T inorm = 1.0 / sqrt(x*x + y*y + z*z);
      T nx = inorm * x;
      T ny = inorm * y;
      T nz = inorm * z;
      return (D3DVector(nx,ny,nz));
   }

   T norm( ){
      return (sqrt(x*x + y*y + z*z));
```

```
    }

    T dotproduct (const D3DVector & rhs ) {
       T scalar = x * rhs.x + y * rhs.y + z * rhs.z ;
       return scalar ;
    }


    /* gvog: DOT PRODUCT operator is defined: */
    T operator*(const D3DVector & rhs){
       T scalar = x * rhs.x + y * rhs.y + z * rhs.z ;
       return scalar ;
    }

    /* gvog: SCALAR PRODUCT operator is defined: */
    D3DVector operator*(T mult){
       return (D3DVector(x*mult, y*mult, z*mult));
    }

    /* gvog: SCALAR DIVISION operator is defined: */
    D3DVector operator/(T mult){
          return (D3DVector(x/mult, y/mult, z/mult));
    }

    D3DVector crossproduct ( const D3DVector & rhs ) {
       T a = y * rhs.z - z * rhs.y ;
       T b = z * rhs.x - x * rhs.z ;
       T c = x * rhs.y - y * rhs.x ;
       D3DVector product( a , b , c ) ;
       return product ;
    }

    D3DVector operator^ (const D3DVector & rhs){
       return crossproduct(rhs);
    }

    D3DVector triplevec( D3DVector & a , D3DVector & b ) {
       return crossproduct ( a.crossproduct( b ) ) ;
    }

    T triplescal( D3DVector & a, D3DVector & b ) {
       return dotproduct( a.crossproduct( b ) ) ;
    }

} ;

#endif   /* AUXILIARY_H */

```

5.5 b3D_integrator.cpp File Reference

C++ source file implementing the Brownian Dynamics simulation of PI.

```
#include <time.h>
#include "b3D_integrator.h"
#include "constants.h"
#include "distributions.h"
#include "hopping.h"
#include "netmin.h"
```

5.5.1 Detailed Description

Author

Grigorios Megariotis (gmegariotis@yahoo.gr)
Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

1.0

5.5.2 LICENSE

Definition in file b3D_integrator.cpp.

5.6 b3D_integrator.cpp

```

#include <time.h>

#include "b3D_integrator.h"
#include "constants.h"
#include "distributions.h"
#include "hopping.h"
#include "netmin.h"



using namespace std;

namespace NetworkNS{

   cb3D_integrator::~cb3D_integrator() {
      free(bd_x);
      free(bd_x_ps);
      free(bd_mass);
      free(bd_f);
      free(bd_nb_f);
      free(bd_gamma);

      return;
   }

   cb3D_integrator::cb3D_integrator(class NetwMin *init_net, double temperature, double
slipspring_rate) {


      cur_bd_net = init_net;
      bd_cur_step = 0;
      dofs = cur_bd_net->network->nodes.size();
      dofs_3N = 3 * dofs;

      xshift = (double*)malloc(dofs*sizeof(double));
      yshift = (double*)malloc(dofs*sizeof(double));
      zshift = (double*)malloc(dofs*sizeof(double));
      grid_cell = (int*)malloc(dofs*sizeof(int));

      bd_x = (double*) malloc(dofs_3N * sizeof (double));
      bd_x_ps = (double*) malloc(dofs_3N * sizeof (double));
      bd_mass = (double*) malloc(dofs_3N * sizeof (double));
      bd_f = (double*) malloc(dofs_3N * sizeof (double));
      bd_f_ps = (double*) malloc(dofs_3N * sizeof (double));
      bd_nb_f = (double*) malloc(dofs_3N * sizeof (double));
      bd_gamma = (double*) malloc(dofs_3N * sizeof (double));


      bd_stress = (double**)malloc(dofs * sizeof(double*));
      for(unsigned int i = 0; i < dofs; i++)
         bd_stress[i] = (double*)malloc(6*sizeof(double));


      // Set the temperature:
      bd_temp = temperature;

      double gamma;
      unsigned int inode = 0;


      // Monomeric friction coefficient in  kg/s
      double monomeric_friction = 1.e-3 * pow(10.0, ((13.5 * 45.0)/(bd_temp-211.15+45)-10.4));

      // convert monomeric friction coefficient (kg/s) to g/mol/s and then divide it with the mass
      // of a PI monomer
      gamma = monomeric_friction / (pi_monomer_mass * amu_to_kg); // s^{-1}

      for (std::list<tNode>::iterator it = cur_bd_net->network->nodes.begin();
              it != cur_bd_net->network->nodes.end(); ++it) {
```

```

      // positions in A
      bd_x[3 * inode + 0] = (*it).Pos[0];
      bd_x[3 * inode + 1] = (*it).Pos[1];
      bd_x[3 * inode + 2] = (*it).Pos[2];

      // positions in A
      bd_x_ps[3 * inode + 0] = (*it).Pos[0];
      bd_x_ps[3 * inode + 1] = (*it).Pos[1];
      bd_x_ps[3 * inode + 2] = (*it).Pos[2];

      // mass of nodal points in g/mol
      bd_mass[3 * inode + 0] = (*it).mass;
      bd_mass[3 * inode + 1] = (*it).mass;
      bd_mass[3 * inode + 2] = (*it).mass;

      // All gamma are measured in s^{-1}
      bd_gamma[3 * inode + 0] = gamma;
      bd_gamma[3 * inode + 1] = gamma;
      bd_gamma[3 * inode + 2] = gamma;

      inode++;
   }


   /* Check whether all beads have the same mass and the same friction coefficient. */
   double prev_val = bd_mass[0] * bd_gamma[0];
   gamma_mass_opt = true;
   for (inode = 1; inode < dofs_3N; inode++){
      if ( ((bd_mass[inode]*bd_gamma[inode])-prev_val)
          *((bd_mass[inode]*bd_gamma[inode])-prev_val) >= tol){
         gamma_mass_opt = false;
         break;
      }
      else
         prev_val = bd_mass[inode]*bd_gamma[inode];
   }

   if (gamma_mass_opt)
      cout << "#:\n#: Brownian Dynamics integrator will run in the optimized way.\n"
           << "#: --- All beads have the same mass and friction coefficient.\n"
           << "#: --- The monomeric friction coefficient is " << monomeric_friction << " kg/s.\n"
           << "#: --- The friction coefficient is " << bd_gamma[0] << " s^{-1}.\n"
           << "#: --- The bead friction coefficient is " << bd_mass[0]*bd_gamma[0]*
amu_to_kg
           << " kg/s.\n"
           << "#: --- The expected Rouse diffusivity multiplied by N would be: "
           << boltz_const_Joule_K * bd_temp / bd_mass[0] / bd_gamma[0] /
amu_to_kg
           << " m^2/s.\n#:"<< endl;

   // Initialize the coupled hopping scheme:
   // Here we can input the zeta parameter...
   if (cur_bd_net->network->pslip_springs.size() > 0)
      my_hopping_scheme = new Hopping(slipspring_rate);

   return;
}


void cb3D_integrator::integrate(unsigned int nsteps, double dt, unsigned int nstout) {

   double current_energy = 0.0, current_nb_energy = 0.0;

   density_cells = (double*) malloc(cur_bd_net->grid->ncells*sizeof(double));
   den_dx = (double*) malloc(27 * cur_bd_net->network->nodes.size() * sizeof (double));
   den_dy = (double*) malloc(27 * cur_bd_net->network->nodes.size() * sizeof (double));
   den_dz = (double*) malloc(27 * cur_bd_net->network->nodes.size() * sizeof (double));

   // Keep the time:
   tbegin = clock();
   double dt_times_inv_mass_gamma, std;
   for (unsigned int idof = 0; idof < dofs_3N; idof++)
         bd_f_ps[idof] = 0.0;

   // Ensure that there are slip-springs present in the network
   bool slipspring_hopping = false;
   if (cur_bd_net->network->pslip_springs.size() > 0) {
      slipspring_hopping = true;
      // and inform the user about the slip-spring hopping
      cout << "#: Slip-spring hopping has been enabled.\n";
   }
   else
      cout << "#: Slip-spring hopping has been disabled.\n";


   bool out_step;
```

```
    for (unsigned int istep = 0; istep < nsteps; istep++) {
       /* Update the step counter: */
       bd_cur_step ++;

       /* Check whether it is time to report the statistics.*/
       out_step = (istep % nstout == 0);

       /* Calculate bonded and non-bonded interactions.*/
       current_energy = bonded_force_calculation(out_step);
       /* Calculate non-bonded interactions every 5 timesteps.*/
       if (istep % 5 == 0)
          current_nb_energy = simpler_scheme_non_bonded_force_calculation();

       /* If it's time to report, do it: */
       if (out_step)
          report(istep, current_energy, current_nb_energy);

       /* Keep a restart file: */
       if (istep % 20000 == 0){
          from_bd_x_to_polymer_network();
          cur_bd_net->write_network_to_lammps_data_file();
          //if (cur_bd_net->network->pslip_springs.size() > 1400)
          //   break;
       }

       // Hopping starts here.
       if ((istep % 1000 == 0) && slipspring_hopping)
          my_hopping_scheme->hopping_step(cur_bd_net, this, bd_x, bd_temp, 1.e3*dt);


       /* Optimized integration scheme, in case every bead of the system has the same mass and
        * friction coefficient. */
       if (gamma_mass_opt){
          dt_times_inv_mass_gamma = (1.e-12*dt)/(bd_mass[0]*amu_to_kg*bd_gamma[0]); // s^2/kg
          std = sqrt(2.e20 * bd_temp * boltz_const_Joule_K * dt_times_inv_mass_gamma);
 // A
          dt_times_inv_mass_gamma *= 1.e23 / avogadro_constant; // s^2 mol / kg

          for (unsigned int i = 0; i < dofs_3N; i++) {
             bd_x[i] = bd_x_ps[i]
                     + (bd_f[i] + bd_nb_f[i]) * dt_times_inv_mass_gamma
                     + 0.5 * (bd_f[i] + bd_nb_f[i] - bd_f_ps[i]) * dt_times_inv_mass_gamma * (1.e-12*dt)
                     + cur_bd_net->my_rnd_gen->gaussian()*std;

             bd_x_ps[i] = bd_x[i];
             bd_f_ps[i] = bd_f[i] + bd_nb_f[i];
          }
       }

       else {

          for (unsigned int i = 0; i < dofs_3N; i++) {
             dt_times_inv_mass_gamma = (1.e-12 * dt) / (bd_mass[i]*amu_to_kg*bd_gamma[i]);//
s^2/kg
             std = sqrt(2.e20*bd_temp*boltz_const_Joule_molK*
dt_times_inv_mass_gamma); // A
             dt_times_inv_mass_gamma *= 1.e23 / avogadro_constant; // s^2 mol / kg

             bd_x[i] = bd_x_ps[i]
                     + (bd_f[i] + bd_nb_f[i]) * dt_times_inv_mass_gamma
                     + 0.5 * (bd_f[i] + bd_nb_f[i] - bd_f_ps[i]) * dt_times_inv_mass_gamma
                     + cur_bd_net->my_rnd_gen->gaussian()*std;

             bd_x_ps[i] = bd_x[i];
             bd_f_ps[i] = bd_f[i] + bd_nb_f[i];

          }
       }

    }


    // Output the final statistics.
    current_energy = bonded_force_calculation(true);
    current_nb_energy = simpler_scheme_non_bonded_force_calculation();

    report(nsteps, current_energy, current_nb_energy);

    // Deallocate the arrays:
    free(den_dx);
    free(den_dy);
    free(den_dz);
    free(density_cells);

    // Update the network with the new positions of the beads.
    from_bd_x_to_polymer_network();
    cur_bd_net->write_network_to_lammps_data_file();
    return;
 }


 void cb3D_integrator::report(unsigned int istep, double b_energy, double nb_energy){

    double press_tens[6];

    // gvog: Ask for the current time:
    clock_t tend = clock();
    calculate_pressure(press_tens);

    double inv_vol = 1.0 / (  cur_bd_net->domain->XBoxLen
                            * cur_bd_net->domain->YBoxLen
                            * cur_bd_net->domain->ZBoxLen);

    cout << istep << "\t" << b_energy << "\t" << nb_energy << "\t"
            << cur_bd_net->network->pslip_springs.size() << "\t"
            //<< pressure      * inv_vol << "\t"
            << press_tens[0] * inv_vol << "\t"
            << press_tens[1] * inv_vol << "\t"
            << press_tens[2] * inv_vol << "\t"
            << press_tens[3] * inv_vol << "\t"
            << press_tens[4] * inv_vol << "\t"
            << press_tens[5] * inv_vol << "\t"
            << " # (" << (double) (tend - tbegin) / CLOCKS_PER_SEC << "s )" << endl;

    //cur_bd_net->my_traj_file->add_snapshot_to_dump(cur_bd_net, this, istep);

    return;
 }

 void cb3D_integrator::calculate_pressure(double *press_tens){
    // Initialize the pressure tensor to zero.
    for (unsigned int j = 0; j < 6; j++)
       press_tens[j] = 0.0;

    // Accumulated the per-atom pressures to the global tensor:
    for (unsigned int i = 0; i < dofs; i++)
       for (unsigned int j = 0; j < 6; j++)
          press_tens[j] += bd_stress[i][j];

    // Convert per-atom pressure*vol in atm*Angstrom
    // An interesting thread concerning loop unrolling in C++:
    // http://stackoverflow.com/questions/15275023/clang-force-loop-unroll-for-specific-loop
    for (unsigned int j = 0; j < 6; j++)
       press_tens[j] *= 1.e33 / avogadro_constant / 101.325e3;

    return;
 }



 double cb3D_integrator::bonded_force_calculation(bool stress_calc) {

    double fenergy = 0.0;

    // Initialize the forces to zero.
    for (unsigned int i = 0; i < dofs_3N; i++)
       bd_f[i] = 0.0;

    // Initialize the stresses to zero, if we have been asked for stress calculation:
    if (stress_calc)
       for (unsigned int i = 0; i < dofs; i++)
          for (unsigned int j = 0; j < 6; j++)
             bd_stress[i][j] = 0.0;


    int taga, taga3, tagb, tagb3;
    //double *sep_vec = (double*)malloc(3*sizeof(double));
    //double *grada   = (double*)malloc(3*sizeof(double));
    //double *gradb   = (double*)malloc(3*sizeof(double));

    double sep_vec[3], grada[3], gradb[3];

    for (std::list<tStrand>::iterator it = cur_bd_net->network->strands.begin();
            it != cur_bd_net->network->strands.end(); ++it) {

       /* Ask for the tags of the nodes connected to the current strand. */
       taga  = (*it).pEnds[0]->Id - 1;
       taga3 = 3*taga;
       tagb =  (*it).pEnds[1]->Id - 1;
       tagb3 = 3*tagb;

       /* Form the "strand" vector, based on the x vector coming from
        * the minimizer. */
```

```
         sep_vec[0] = bd_x[tagb3 + 0] - bd_x[taga3 + 0];
         sep_vec[1] = bd_x[tagb3 + 1] - bd_x[taga3 + 1];
         sep_vec[2] = bd_x[tagb3 + 2] - bd_x[taga3 + 2];

         /* Apply minimum image convention. */
         cur_bd_net->domain->minimum_image(sep_vec[0], sep_vec[1], sep_vec[2]);

#ifdef FENE_SLS
         if ((*it).slip_spring)
            fenergy += f_fene( sep_vec, (*it).spring_coeff,
                               (*it).sq_end_to_end, bd_temp, grada, gradb);
         else
#endif
         /* Calculate the spring's contribution to the free energy of the system.*/
         fenergy += f_gaussian( sep_vec, (*it).spring_coeff,
                                (*it).sq_end_to_end, bd_temp, grada, gradb);

         // Accumulate the forces of the first atom:
         bd_f[taga3 + 0] += grada[0];
         bd_f[taga3 + 1] += grada[1];
         bd_f[taga3 + 2] += grada[2];
         // Accumulate the forces of the second atom:
         bd_f[tagb3 + 0] += gradb[0];
         bd_f[tagb3 + 1] += gradb[1];
         bd_f[tagb3 + 2] += gradb[2];

         if (stress_calc) {
            bd_stress[taga][0] += 0.5 * sep_vec[0] * grada[0]; // xx
            bd_stress[taga][1] += 0.5 * sep_vec[1] * grada[1]; // yy
            bd_stress[taga][2] += 0.5 * sep_vec[2] * grada[2]; // zz
            bd_stress[taga][3] += 0.5 * sep_vec[0] * grada[1]; // xy
            bd_stress[taga][4] += 0.5 * sep_vec[0] * grada[2]; // xz
            bd_stress[taga][5] += 0.5 * sep_vec[1] * grada[2]; // yz

            bd_stress[tagb][0] += 0.5 * sep_vec[0] * grada[0];
            bd_stress[tagb][1] += 0.5 * sep_vec[1] * grada[1];
            bd_stress[tagb][2] += 0.5 * sep_vec[2] * grada[2];
            bd_stress[tagb][3] += 0.5 * sep_vec[0] * grada[1];
            bd_stress[tagb][4] += 0.5 * sep_vec[0] * grada[2];
            bd_stress[tagb][5] += 0.5 * sep_vec[1] * grada[2];
         }
      }




      return (fenergy);
   }


   void cb3D_integrator::from_bd_x_to_polymer_network(void) {

      unsigned int inode = 0;

      for (std::list<tNode>::iterator it = cur_bd_net->network->nodes.begin();
              it != cur_bd_net->network->nodes.end(); ++it) {
         (*it).Pos[0] = bd_x[3 * inode + 0];
         (*it).Pos[1] = bd_x[3 * inode + 1];
         (*it).Pos[2] = bd_x[3 * inode + 2];
         inode ++;
      }

      return;
   }



   double cb3D_integrator::simpler_scheme_non_bonded_force_calculation(void) {

      // Variables holding the volume of a cell.
      double vcube_cell, vx, vy, vz;

      // The nonbonded contribution to the free energy of the network.
      double f_nb_energy = 0.0;

      //at this point we may need to call a subroutine that will convert bd_x elements to positions

      int l; // indices used to find the parent cell and its first neighbours

      for (int i = 0; i < cur_bd_net->grid->ncells; i++)
         density_cells[i] = 0.0;

      int cur_node = 0, i, j, cur_elem, max_node = cur_bd_net->network->nodes.size();
      double dx, dy, dz, xl, yl, zl, half_rnode, mass_over_rnode3;

      half_rnode = 0.5 * cur_bd_net->network->nodes.front().r_node;
      mass_over_rnode3 = cur_bd_net->network->nodes.front().n_mass
```

```
              / (cur_bd_net->network->nodes.front().r_node
              * cur_bd_net->network->nodes.front().r_node
              * cur_bd_net->network->nodes.front().r_node);


      for (cur_node = 0; cur_node < max_node; cur_node++) {
         /*loop over the (*it).node_cell itself and its first neighbors (it is always equal
          * to 27, or 26 starting the numbering from zero*/
         /*expressed in Angstrom^3. node coordinates have to be shifted
          * boxl/2.0 so as to be embedded into a grid extended from
          * zero to cur_bd_net->domain->XBoxLen */

         //half_rnode = 0.5 * (*it).r_node;
         //mass_over_rnode3 = (*it).n_mass / (*it).r_node / (*it).r_node / (*it).r_node;

         xshift[cur_node] = bd_x[3 * cur_node];
         yshift[cur_node] = bd_x[3 * cur_node + 1];
         zshift[cur_node] = bd_x[3 * cur_node + 2];

         //return the nodes back into the primary box
         cur_bd_net->domain->minimum_image(xshift[cur_node], yshift[cur_node], zshift[cur_node]);

         //shift the node position to a shifted simulation box that contains the  grid
         xshift[cur_node] += 0.5 * cur_bd_net->domain->XBoxLen;
         yshift[cur_node] += 0.5 * cur_bd_net->domain->YBoxLen;
         zshift[cur_node] += 0.5 * cur_bd_net->domain->ZBoxLen;

         grid_cell[cur_node] = cur_bd_net->grid->find_grid_cell(xshift[cur_node], yshift[cur_node]
,
                                          zshift[cur_node]);


         for (j = 0; j < 27; j++) {
            // find the neighbours of (*it).node_cell, zero corresponds to the cell itself

            l = cur_bd_net->grid->cells[grid_cell[cur_node]].neigh[j];

            /*find the intersection of cube formed by node (*it).Id, how to define
             * cell_vec[l][0:2]: vector of cell l, is used for the calculation of
             * vcube_cell,*/

            // if statements for vx
            // for the computation of minimum images of xshift, yshift, zshift with respect
            // to xcell, ycell and zcell respectively
            dx = xshift[cur_node] - cur_bd_net->grid->cells[l].Vec[0];
            dy = yshift[cur_node] - cur_bd_net->grid->cells[l].Vec[1];
            dz = zshift[cur_node] - cur_bd_net->grid->cells[l].Vec[2];

            // minimum images in a box from zero to box_l
            //where xl, yl and zl updated values, due to minimum image, of xshift, yshift and zshift
            cur_bd_net->domain->zero_to_length_minimum_image(dx, dy, dz);

            xl = cur_bd_net->grid->cells[l].Vec[0] + dx;
            yl = cur_bd_net->grid->cells[l].Vec[1] + dy;
            zl = cur_bd_net->grid->cells[l].Vec[2] + dz;


            vx = max(min(xl + half_rnode, cur_bd_net->grid->cells[l].Vec[0])
               - max(xl-half_rnode, cur_bd_net->grid->cells[l].Vec[0]-cur_bd_net->grid->dlx), 0.0);

            vy = max(min(yl + half_rnode, cur_bd_net->grid->cells[l].Vec[1])
               - max(yl-half_rnode, cur_bd_net->grid->cells[l].Vec[1]-cur_bd_net->grid->dly), 0.0);

            vz = max(min(zl + half_rnode, cur_bd_net->grid->cells[l].Vec[2])
               - max(zl-half_rnode, cur_bd_net->grid->cells[l].Vec[2]-cur_bd_net->grid->dlz), 0.0);

            vcube_cell = vx * vy * vz;

            cur_elem = 27 * cur_node + j;

            if ((xl > cur_bd_net->grid->cells[l].Vec[0] - cur_bd_net->grid->dlx - half_rnode) &&
                    (xl < cur_bd_net->grid->cells[l].Vec[0] - cur_bd_net->grid->dlx + half_rnode))
               den_dx[cur_elem] = mass_over_rnode3 * vy * vz / cur_bd_net->grid->vcell;
            else if ((xl > cur_bd_net->grid->cells[l].Vec[0] - half_rnode) &&
                    (xl < cur_bd_net->grid->cells[l].Vec[0] + half_rnode))
               den_dx[cur_elem] = -mass_over_rnode3 * vy * vz / cur_bd_net->grid->vcell;
            else
               den_dx[cur_elem] = 0.0;

            if ((yl > cur_bd_net->grid->cells[l].Vec[1] - cur_bd_net->grid->dly - half_rnode) &&
                    (yl < cur_bd_net->grid->cells[l].Vec[1] - cur_bd_net->grid->dly + half_rnode))
               den_dy[cur_elem] = mass_over_rnode3 * vx * vz / cur_bd_net->grid->vcell;
            else if ((yl > cur_bd_net->grid->cells[l].Vec[1] - half_rnode) &&
                    (yl < cur_bd_net->grid->cells[l].Vec[1] + half_rnode))
               den_dy[cur_elem] = -mass_over_rnode3 * vx * vz / cur_bd_net->grid->vcell;
            else
               den_dy[cur_elem] = 0.0;

            if ((zl > cur_bd_net->grid->cells[l].Vec[2] - cur_bd_net->grid->dlz - half_rnode) &&
                  (zl < cur_bd_net->grid->cells[l].Vec[2] - cur_bd_net->grid->dlz + half_rnode))
               den_dz[cur_elem] = mass_over_rnode3 * vx * vy / cur_bd_net->grid->vcell;
            else if ((zl > cur_bd_net->grid->cells[l].Vec[2] - half_rnode) &&
                  (zl < cur_bd_net->grid->cells[l].Vec[2] + half_rnode))
               den_dz[cur_elem] = -mass_over_rnode3 * vx * vy / cur_bd_net->grid->vcell;
            else
               den_dz[cur_elem] = 0.0;

            density_cells[l] += mass_over_rnode3*vcube_cell;
         }
      }

      for (i = 0; i < cur_bd_net->grid->ncells; i++) {
         density_cells[i] *= cur_bd_net->grid->ivcell; //expressed in kuhn segments per Angstom^3
         f_nb_energy += cur_bd_net->grid->vcell
                     * (c1 * density_cells[i] + c2 * density_cells[i] * density_cells[i]);
      }


      // new nested for-loops for updating the 3N vector of derivatives
      double fx, fy, fz; // force components on a nod due to non-bonded interactions
      // used for updating the array of derivatives[3N]
      for (cur_node = 0; cur_node < max_node; cur_node++){
         fx = 0.0;
         fy = 0.0;
         fz = 0.0;

         for (j = 0; j < 27; j++) {

            cur_elem = 27 * cur_node + j;

            l = cur_bd_net->grid->cells[grid_cell[cur_node]].neigh[j];

            fx -= cur_bd_net->grid->vcell * (c1 + 2.0 * c2 * density_cells[l]) * den_dx[cur_elem];
            fy -= cur_bd_net->grid->vcell * (c1 + 2.0 * c2 * density_cells[l]) * den_dy[cur_elem];
            fz -= cur_bd_net->grid->vcell * (c1 + 2.0 * c2 * density_cells[l]) * den_dz[cur_elem];
         }

         bd_nb_f[3 * cur_node]     = fx;
         bd_nb_f[3 * cur_node + 1] = fy;
         bd_nb_f[3 * cur_node + 2] = fz;

      }

      return (f_nb_energy);
   }



   void cb3D_integrator::cell_density_nodal_points() {

      double xnew, ynew, znew;
      int Id;
      int *hist_grid = (int*) malloc(cur_bd_net->grid->ncells * sizeof (int));

      for (int i = 0; i < cur_bd_net->grid->ncells; i++)
         hist_grid[i] = 0;

      for (unsigned int inode = 0; inode < dofs; inode++) {

         xnew = bd_x[3 * inode + 0];
         ynew = bd_x[3 * inode + 1];
         znew = bd_x[3 * inode + 2];
         cur_bd_net->domain->minimum_image(xnew, ynew, znew);
         xnew = xnew + cur_bd_net->domain->XBoxLen / 2.0;
         ynew = ynew + cur_bd_net->domain->YBoxLen / 2.0;
         znew = znew + cur_bd_net->domain->ZBoxLen / 2.0;
         Id = cur_bd_net->grid->find_grid_cell(xnew, ynew, znew);
         hist_grid[Id] = hist_grid[Id] + 1;

      }

      for (int i = 0; i < cur_bd_net->grid->ncells; i++)
         fprintf(p_cell_density, "%d " " %d\n", i, hist_grid[i]);

      return;
   }

}
```

5.7 b3D_integrator.h File Reference

Header file accompanying the "b3D_integrator.cpp" C++ source file.

```
#include <cmath>
#include <cstdio>
#include <list>
#include <stdlib.h>
#include <vector>
#include "hopping.h"
#include "net_types.h"
#include "network.h"
#include "domain.h"
#include "rng.h"
```

Classes

- class NetworkNS::cb3D_integrator

  The class of the Brownian Dynamics integrator.

5.7.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)
Grigorios Megariotis (gmegariotis@yahoo.gr)

Version

1.0 (January 15, 2013)

5.7.2 LICENSE

Copyright (c) 2013 Georgios Vogiatzis and Grigorios Megariotis. All rights reserved.

This work is licensed under the terms of the MIT license. For a copy, see https://opensource.↩org/licenses/MIT.

Definition in file b3D_integrator.h.

5.8 b3D_integrator.h

```

#ifndef _B3D_INTEGRATOR_H
#define _B3D_INTEGRATOR_H

#include <cmath>
#include <cstdio>
#include <list>
#include <stdlib.h>
#include <vector>

#include "hopping.h"
#include "net_types.h"
#include "network.h"
#include "domain.h"
#include "rng.h"

namespace NetworkNS {

  class cb3D_integrator{
  public:

     cb3D_integrator(class NetwMin *, double, double);
     ~cb3D_integrator();

     void integrate(unsigned int nsteps, double dt, unsigned int nstout);

     void extract_positions(double *x);

     void compute_stresses (void);

     void report(unsigned int, double, double);

     double bonded_force_calculation(bool);

     double simpler_scheme_non_bonded_force_calculation(void);


     void calculate_pressure(double *);

     double *bd_gamma;

     double *bd_mass;

     double *bd_x;
     double **bd_stress;

     unsigned int bd_cur_step;

  private:

     class NetwMin *cur_bd_net;

     unsigned int dofs;
     unsigned int dofs_3N;



     double *bd_f;

     double *bd_nb_f;

     double *bd_x_ps;

     double *bd_f_ps;

     class Hopping *my_hopping_scheme;

     clock_t tbegin;

     double *xshift;

     double *yshift;

     double *zshift;

     int *grid_cell;

     double bd_temp;

     bool gamma_mass_opt;

     double *density_cells;

     double *den_dx;

     double *den_dy;

     double *den_dz;


     FILE * p_cell_density;

     void from_bd_x_to_polymer_network(void);     // Convert elements of array bd_x to positions of the polymer network

     void cell_density_nodal_points(void);        // compute the density in each cell of the orthogonal grid

  };

}

#endif   /* NETWORK_H */

```

```
00146
00147
```

5.9 constants.h File Reference

Header file containing the definitions of physical constants.

Variables

- double const avogadro_constant = 6.02214129e23
- double const kg_to_amus = 6.022141129e26
- double const amu_to_kg = 1.660538921e-27
- double const pi_monomer_mass = 68.12

  The mass of an isoprene monomer in g/mol.
- double const boltz_const_Joule_molK = 8.3144621

  The Boltzmann constant in J/mol/K.
- double const boltz_const_kJoule_molK = 8.3144621e-3

  The Boltzmann constant in kJ/mol/K.
- double const boltz_const_Joule_K = 1.3806488e-23

  The Boltzmann constant in J/K.
- double const pi_tube_diameter = 80.39
- const double c1 =-5.4e02
- const double c2 =7.0e04
- const double tol = 1.e-4

  Tolerance for numerical comparisons.
- const double PI = 3.14159265358979323846264338

  The well-known π constant.
- double const **hopping_attempt_radius** = 60.0

5.9.1 Detailed Description

Author

Grigorios Megariotis (gmegariotis@yahoo.gr)
Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

5.9.2 LICENSE

Definition in file constants.h.

5.9.3 Variable Documentation

5.9.3.1 avogadro_constant

```
double const avogadro_constant = 6.02214129e23
```

The Avogadro constant measured in mol^{-1}

Definition at line 21 of file constants.h.

5.9.3.2 kg_to_amus

```
double const kg_to_amus = 6.022141129e26
```

Conversion from kg to g/mol, $1\mathrm{kg} =$

Definition at line 24 of file constants.h.

5.9.3.3 amu_to_kg

```
double const amu_to_kg = 1.660538921e-27
```

Conversion from g/mol to kg. $1\,\mathrm{g/mol} = 1.660538921 < \times 10^{-27} kg$

Definition at line 27 of file constants.h.

5.9.3.4 pi_tube_diameter

```
double const pi_tube_diameter = 80.39
```

The tube diameter of polyisoprene in Å. The estimation is based on the work of Li et al.[7].

Definition at line 36 of file constants.h.

5.9.3.5 c1

```
const double c1 =-5.4e02
```

Constant c_1 of the functional of the nonbonded free energy, which is of the form: $\mathscr{V}_{\mathrm{nb}} < (\rho) = c_1\rho + c_2\rho^2$.

Definition at line 41 of file constants.h.

5.9.3.6 c2

```
const double c2 =7.0e04
```

Constant c_2 of the functional of the nonbonded free energy, which is of the form: $\mathscr{V}_{\mathrm{nb}}(\rho) =< c_1\rho + c_2\rho^2$.

Definition at line 46 of file constants.h.

5.10 constants.h

```
00001
00018 #ifndef CONSTANTS_H
```

```
#define  CONSTANTS_H

double const avogadro_constant = 6.02214129e23;

double const kg_to_amus = 6.022141129e26;

double const amu_to_kg = 1.660538921e-27;

double const pi_monomer_mass = 68.12;

double const boltz_const_Joule_molK = 8.3144621;
double const boltz_const_kJoule_molK = 8.3144621e-3;
double const boltz_const_Joule_K = 1.3806488e-23;
double const pi_tube_diameter = 80.39;

const double c1=-5.4e02;

const double c2=7.0e04;

const double tol = 1.e-4;

const double PI = 3.14159265358979323846264338;

double const hopping_attempt_radius = 60.0;

#endif   /* CONSTANTS_H */

```

5.11 distributions.cpp File Reference

C++ source file containing the implementation of bonded interactions.

```
#include <cmath>
#include <iostream>
#include "distributions.h"
```

Functions

- double f_gaussian (const double ∗rij, const double &coeff, const double &sq_ete, const double &temp, double ∗gradi, double ∗gradj)
- double e_gaussian (const double ∗rij, const double &coeff, const double &sq_ete, const double &temp)

5.11.1 Detailed Description

Author

Grigorios Megariotis (gmegariotis@yahoo.gr)
Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

5.11.2 LICENSE

Definition in file distributions.cpp.

5.11.3 Function Documentation

5.11.3.1 f_gaussian()

```
double f_gaussian (
            const double * rij,
            const double & coeff,
            const double & sq_ete,
            const double & temp,
            double * gradi,
            double * gradj )
```

Parameters

in	*rij*	the separation vector between beads i and j, i.e. $\mathbf{R}_{ij}$.
in	*coeff*	the strength of the entropic springs $\varepsilon_\mathrm{b} = 3/2\,k_\mathrm{B}$, measured in kJ/mol/K.
in	*sq_ete*	the equilibrium mean-squared end-to-end length of the strand, i.e. $\sigma_\mathrm{b} = n_{\mathrm{Kuhns/bead}} b^2$, measured in Å^2.
in	*temp*	the temperature of the simulation, T, in K.
out	*gradi*	the force acted on the bead i, due to its bond with bead j
out	*gradj*	the force acted on the bead j, due to its bond with bead i

This routine applies a Gaussian free energy potential of the form:

$$\mathcal{V}_\mathrm{b}\left(r_{ij}^2\right) = \varepsilon_\mathrm{b} T \frac{r_{ij}^2}{\sigma_\mathrm{b}^2} = \frac{3}{2} k_\mathrm{B} T \frac{\mathbf{R}_{ij} \cdot \mathbf{R}_{ij}}{n_{\mathrm{Kuhns/bead}} b^2}$$

with the parameters ε_b and σ_b read from the data file. In our approach $\varepsilon_\mathrm{b} = 0.012471$ kJ/mol/K and $\sigma_\mathrm{b} = 810$ Å^2 for polyisoprene melt.

Definition at line 31 of file distributions.cpp.

5.11.3.2 e_gaussian()

```
double e_gaussian (
            const double * rij,
            const double & coeff,
            const double & sq_ete,
            const double & temp )
```

Parameters

in	*rij*	the separation vector between beads i and j, i.e. $\mathbf{R}_{ij}$.
in	*coeff*	the strength of the entropic springs $\varepsilon_\mathrm{b} = 3/2\,k_\mathrm{B}$, measured in kJ/mol/K.
in	*sq_ete*	the equilibrium mean-squared end-to-end length of the strand, i.e. $\sigma_\mathrm{b} = n_{\mathrm{Kuhns/bead}} b^2$, measured in Å^2.
in	*temp*	the temperature of the simulation, T, in K.

Definition at line 71 of file distributions.cpp.

5.12 distributions.cpp

```
00001
```

```
#include <cmath>
#include <iostream>

#include "distributions.h"

using namespace std;

double f_gaussian(const double *rij, const double & coeff, const double & sq_ete,
                  const double & temp, double *gradi, double *gradj) {

   // Compute the distance between positional vectors ri and rj
   double rsq = rij[0]*rij[0] + rij[1]*rij[1] + rij[2]*rij[2];

   // Compute the free energy of the Gaussian spring.
   double p = coeff * temp / sq_ete;

   // Compute the forces as gradients of the Gaussian distribution along ri and rj direction.
   gradi[0] = 2.0 * p * rij[0];
   gradi[1] = 2.0 * p * rij[1];
   gradi[2] = 2.0 * p * rij[2];

   gradj[0] = -gradi[0];
   gradj[1] = -gradi[1];
   gradj[2] = -gradi[2];

   return (p*rsq);
}


double e_gaussian(const double *rij, const double & coeff, const double & sq_ete,
                      const double & temp) {
   //compute the distance between positional vectors ri and rj
   double rsq = rij[0]*rij[0] + rij[1]*rij[1] + rij[2]*rij[2];

   // Compute only the free energy of the Gaussian spring.
   return (coeff * temp * rsq / sq_ete);
}

```

5.13 distributions.h File Reference

Header file accompanying the "distributions.cpp" C++ source file.

Functions

- double f_gaussian (const double ∗, const double &, const double &, const double &, double ∗, double ∗)
- double e_gaussian (const double ∗, const double &, const double &, const double &)

5.13.1 Detailed Description

Author

Grigorios Megariotis
Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

5.13.2 LICENSE

Definition in file distributions.h.

5.13.3 Function Documentation

5.13.3.1 f_gaussian()

```
double f_gaussian (
            const double * rij,
            const double & coeff,
            const double & sq_ete,
            const double & temp,
            double * gradi,
            double * gradj )
```

Parameters

in	*rij*	the separation vector between beads i and j, i.e. $\mathbf{R}_{ij}$.
in	*coeff*	the strength of the entropic springs $\varepsilon_{\mathrm{b}} = 3/2\,k_{\mathrm{B}}$, measured in kJ/mol/K.
in	*sq_ete*	the equilibrium mean-squared end-to-end length of the strand, i.e. $\sigma_{\mathrm{b}} = n_{\mathrm{Kuhns/bead}} b^2$, measured in Å^2.
in	*temp*	the temperature of the simulation, T, in K.
out	*gradi*	the force acted on the bead i, due to its bond with bead j
out	*gradj*	the force acted on the bead j, due to its bond with bead i

This routine applies a Gaussian free energy potential of the form:

$$\mathcal{V}_{\mathrm{b}}\left(r_{ij}^2\right) = \varepsilon_{\mathrm{b}} T \frac{r_{ij}^2}{\sigma_{\mathrm{b}}^2} = \frac{3}{2} k_{\mathrm{B}} T \frac{\mathbf{R}_{ij} \cdot \mathbf{R}_{ij}}{n_{\mathrm{Kuhns/bead}} b^2}$$

with the parameters ε_{b} and σ_{b} read from the data file. In our approach $\varepsilon_{\mathrm{b}} = 0.012471$ kJ/mol/K and $\sigma_{\mathrm{b}} = 810$ Å^2 for polyisoprene melt.

Definition at line 31 of file distributions.cpp.

5.13.3.2 e_gaussian()

```
double e_gaussian (
            const double * rij,
            const double & coeff,
            const double & sq_ete,
            const double & temp )
```

Parameters

in	*rij*	the separation vector between beads i and j, i.e. $\mathbf{R}_{ij}$.
in	*coeff*	the strength of the entropic springs $\varepsilon_{\mathrm{b}} = 3/2\,k_{\mathrm{B}}$, measured in kJ/mol/K.
in	*sq_ete*	the equilibrium mean-squared end-to-end length of the strand, i.e. $\sigma_{\mathrm{b}} = n_{\mathrm{Kuhns/bead}} b^2$, measured in Å^2.
in	*temp*	the temperature of the simulation, T, in K.

Definition at line 71 of file distributions.cpp.

5.14 distributions.h

```
00001
```

```
#ifndef _DISTRIBUTIONS_H_
#define _DISTRIBUTIONS_H_

double f_gaussian(const double *, const double &, const double &, const double &, double *,
  double *);
double e_gaussian(const double *, const double &, const double &, const double &);

#endif
```

5.15 domain.cpp File Reference

C++ source file containing the necessary functions for the manipulation of the simulation domain.

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdlib.h>
#include <stdio.h>
#include <vector>
#include "domain.h"
```

5.15.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)
Grigorios Megariotis (gmegariotis@yahoo.gr)

5.15.2 LICENSE

Definition in file domain.cpp.

5.16 domain.cpp

```

#include <cmath>
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdlib.h>
#include <stdio.h>
#include <vector>

#include "domain.h"

using namespace std;

namespace NetworkNS {

   Domain::Domain(std::string filename) {

      ifstream data_file(filename.c_str(), ifstream::in);
      /* Define an array of strings to hold the contents of the file. */
      std::vector <string> lines_of_file;

      for (int i = 0; i < 11; i++) {
         /* A temporary string for the current line of the file. */
         std::string current_line;
         getline(data_file, current_line);
```

```
            /* Add current line to file's array of lines. */
            lines_of_file.push_back(current_line);
         }

         /* x box dimension is stored in the 9th line: */
         string buf;
         stringstream ss;
         vector<string> tokens;

         // http://www.cplusplus.com/faq/sequences/strings/split/#boost-split

         for (int i = 0; i < 3; i++) {
            ss.flush();
            tokens.clear();
            ss << lines_of_file[8 + i];
            while (ss >> buf)
               tokens.push_back(buf);
            BoxLow[i] = atof(tokens[0].c_str());
            BoxHigh[i] = atof(tokens[1].c_str());
         }


         XBoxLen = BoxHigh[0] - BoxLow[0];
         iXBoxLen = 1.0 / XBoxLen;

         YBoxLen = BoxHigh[1] - BoxLow[1];
         iYBoxLen = 1.0 / YBoxLen;

         ZBoxLen = BoxHigh[2] - BoxLow[2];
         iZBoxLen = 1.0 / ZBoxLen;

         BoxExists = 1;
         NonPeriodic = 0;
         Xperiodic = Yperiodic = Zperiodic = 1;
         Periodicity[0] = Xperiodic;
         Periodicity[1] = Yperiodic;
         Periodicity[2] = Zperiodic;

         Boundary[0][0] = Boundary[0][1] = 0;
         Boundary[1][0] = Boundary[1][1] = 0;
         Boundary[2][0] = Boundary[2][1] = 0;

         return;
      }

      Domain::~Domain() {
         return;
      }

      void Domain::minimum_image(double &x, double &y, double &z) {
         x = x - XBoxLen * round(x * iXBoxLen);
         y = y - YBoxLen * round(y * iYBoxLen);
         z = z - ZBoxLen * round(z * iZBoxLen);
         return;
      }

      void Domain::zero_to_length_minimum_image(double &x, double&y, double &z)
    {
         x = x - XBoxLen * round(x * iXBoxLen);
         y = y - YBoxLen * round(y * iYBoxLen);
         z = z - ZBoxLen * round(z * iZBoxLen);

      }

      void Domain::put_in_primary_box(double *coords, int *pcoeffs){

         // Calculate the distance from the center of the simulation box.
         double distx = coords[0] - (BoxLow[0] + 0.5*XBoxLen);
         double disty = coords[1] - (BoxLow[1] + 0.5*YBoxLen);
         double distz = coords[2] - (BoxLow[2] + 0.5*ZBoxLen);

         // Calculate the periodic continuation coefficients:
         pcoeffs[0] = round(distx * iXBoxLen);
         pcoeffs[1] = round(disty * iYBoxLen);
         pcoeffs[2] = round(distz * iZBoxLen);

         coords[0] += (double)pcoeffs[0] * XBoxLen + BoxLow[0];
         coords[1] += (double)pcoeffs[1] * YBoxLen + BoxLow[1];
         coords[2] += (double)pcoeffs[2] * ZBoxLen + BoxLow[2];

         return;
      }

   }
```

5.17 domain.h File Reference

Header file containing the definitions of Domain class.

`#include <string>`

Classes

- class NetworkNS::Domain

  The class of the simulation domain.

5.17.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

1.0 (May 16, 2012)

5.17.2 LICENSE

Definition in file domain.h.

5.18 domain.h

```

#ifndef DOMAIN_H
#define  DOMAIN_H

#include <string>

namespace NetworkNS {

   class Domain{
      public:
         int BoxExists;
         int NonPeriodic;

         int Xperiodic;
         int Yperiodic;
         int Zperiodic;

         int Periodicity[3];

         int Boundary[3][2];

         double BoxLow[3];
         double BoxHigh[3];


         double XBoxLen;
         double YBoxLen;
         double ZBoxLen;

         double iXBoxLen;
         double iYBoxLen;
         double iZBoxLen;

```

```

        Domain(std::string);

        virtual ~Domain();

        void minimum_image(double &x, double &y, double &z);

        void zero_to_length_minimum_image(double &, double&, double &);

        void put_in_primary_box(double *, int *);

    };
}

#endif  /* DOMAIN_H */

```

5.19 dump.cpp File Reference

This file contains all routines necessary to write a trajectory file of the simulation. It uses a pretty standard LAMMPS trajectory format.

```
#include "dump.h"
#include "network.h"
#include "netmin.h"
#include "b3D_integrator.h"
```

5.19.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

1.0 (created on October 28, 2013)

5.19.2 LICENSE

Definition in file dump.cpp.

5.20 dump.cpp

```

#include "dump.h"
#include "network.h"
#include "netmin.h"
#include "b3D_integrator.h"

using namespace std;

namespace NetworkNS {

   dump::dump(std::string filename_to_open) {

      my_file.open(filename_to_open.c_str(), ofstream::out);
      if (not my_file.good())
         cout << "Specified dump file does not exist!" << endl;

   }
   void dump::add_snapshot_to_dump(const class NetwMin *netw_app, const class
cb3D_integrator *b3D,
                                   unsigned int timestep){

      my_file << "ITEM: TIMESTEP\n";
      my_file << timestep << "\n";
      my_file << "ITEM: NUMBER OF ATOMS\n";
      my_file << netw_app->network->nodes.size() << "\n";
      my_file << "ITEM: BOX BOUNDS pp pp pp\n";
      my_file << netw_app->domain->BoxLow[0] << " " << netw_app->
domain->BoxHigh[0] << "\n";
      my_file << netw_app->domain->BoxLow[1] << " " << netw_app->
domain->BoxHigh[1] << "\n";
      my_file << netw_app->domain->BoxLow[2] << " " << netw_app->
domain->BoxHigh[2] << "\n";
      my_file << "ITEM: ATOMS id x y z ix iy iz Vor VorNeigh xx yy zz xy xz yz\n";

      // TODO: Here we can ask for a Voronoi tessellation of the simulation box in order to
      //       calculate atomic volumes.

      my_file.precision(6);
      for (std::list <tNode>::iterator it = netw_app->network->nodes.begin();
           it != netw_app->network->nodes.end(); ++it){

         unsigned int ibead = (*it).Id - 1;

         my_file << (*it).Id << " "
                 << b3D->bd_x[3*ibead + 0] << " "
                 << b3D->bd_x[3*ibead + 1] << " "
                 << b3D->bd_x[3*ibead + 2] << " "
                 << " 0 0 0 1.0 0 "
                 << b3D->bd_stress[ibead][0] << " "
                 << b3D->bd_stress[ibead][1] << " "
                 << b3D->bd_stress[ibead][2] << " "
                 << b3D->bd_stress[ibead][3] << " "
                 << b3D->bd_stress[ibead][4] << " "
                 << b3D->bd_stress[ibead][5] << endl;
      }
   }

   dump::dump(const dump& orig) {

   }

   dump::~dump() {
      my_file.close();
   }

}
```

5.21 dump.h File Reference

The header file for trajectory file keeping.

```
#include <fstream>
#include <iostream>
#include <string>
#include "b3D_integrator.h"
```

Classes

- class NetworkNS::dump

  The class which dumps a snapshot of atom quantities (positions, atomic-level stresses) to one or more files every a predefined number of timesteps.

5.21.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

1.0 (Created on October 28, 2013)

5.21.2 LICENSE

Definition in file dump.h.

5.22 dump.h

```
#ifndef DUMP_H
#define  DUMP_H

#include <fstream>
#include <iostream>
#include <string>

#include "b3D_integrator.h"


namespace NetworkNS {

   class dump {

   public:

      dump(std::string);
      dump(const dump& orig);
      void add_snapshot_to_dump(const class NetwMin *, const class
cb3D_integrator *, unsigned int);
      virtual ~dump();


   private:
      std::ofstream my_file;


   };
}
#endif   /* DUMP_H */

```

5.23 grid.cpp File Reference

The source file containing the routines of the grid class used for the estimation of non-bonded interactions.

```
#include <cstdlib>
#include "grid.h"
```

5.23.1 Detailed Description

Author

Grigorios Megariotis (gmegariotis@yahoo.gr)
Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

5.23.2 LICENSE

Definition in file grid.cpp.

5.24 grid.cpp

```
#include <cstdlib>
#include "grid.h"

namespace NetworkNS {

   Grid::Grid(double Lx, double Ly, double Lz, int nx, int ny, int nz) {

      /* Initialize the arrays with the cells: */
      ncellx = nx;
      ncelly = ny;
      ncellz = nz;
      ncells = ncellx * ncelly * ncellz;
      cells = (grid_cell*)malloc(ncells * sizeof(grid_cell));

      /*define dlx, dly, dlz and vecell: all of these are common parameters for each subcell
       *  and thus they are defined only once*/

      dlx = Lx / double(ncellx);
      idlx = 1.0 / dlx;
      dly = Ly / double(ncelly);
      idly = 1.0 / dly;
      dlz = Lz / double(ncellz);
      idlz = 1.0 / dlz;
      vcell = dlx * dly*dlz;
      ivcell = 1.0 / vcell;

      int counter = 0;
      //define Id, and the vector of each cell

      for (int k = 0; k < ncellz; k++)

         for (int j = 0; j < ncelly; j++)

            for (int i = 0; i < ncellx; i++) {


               cells[counter].Id = k * ncelly * ncellx + j * ncellx + i;
               cells[counter].Vec[0] = double(i + 1) * dlx;
               cells[counter].Vec[1] = double(j + 1) * dly;
               cells[counter].Vec[2] = double(k + 1) * dlz;



               counter = counter + 1;
            }






      //define the first neighbours of all nodes


      int icell, jcell, kcell, klower, kupper, jlower, jupper, ilower, iupper, c_id;

      for (kcell = 0; kcell < ncellz; kcell++) {

         kupper = kcell + 1;
         if ((kcell + 1)>(ncellz - 1)) kupper = 0;
         klower = kcell - 1;
         if ((kcell - 1) < 0) klower = ncellz - 1;

         for (jcell = 0; jcell < ncelly; jcell++) {

            jupper = jcell + 1;
            if ((jcell + 1)>(ncelly - 1)) jupper = 0;
            jlower = jcell - 1;
            if ((jcell - 1) < 0) jlower = ncelly - 1;


            for (icell = 0; icell < ncellx; icell++) {

               iupper = icell + 1;
               if ((icell + 1)>(ncellx - 1)) iupper = 0;
               ilower = icell - 1;
               if ((icell - 1) < 0) ilower = ncellx - 1;

               c_id = (kcell) * ncelly * ncellx + (jcell) * ncellx + icell;

               // 1: z, y , x:
               //neigh[cell_id][0] = (kcell)*ncelly*ncellx + (jcell)*ncellx + icell;
               cells[c_id].neigh[0] = (kcell) * ncelly * ncellx + (jcell) * ncellx + icell;
               //2: same z, same y, x+
               //neigh[cell_id][1] = (kcell)*ncelly*ncellx + (jcell)*ncellx + iupper;
               cells[c_id].neigh[1] = (kcell) * ncelly * ncellx + (jcell) * ncellx + iupper;
               // 3: same z, same y, x-
               //neigh[cell_id][2] = (kcell)*ncelly*ncellx + (jcell)*ncellx + ilower;
               cells[c_id].neigh[2] = (kcell) * ncelly * ncellx + (jcell) * ncellx + ilower;
               //4: same z, y+ , same x
               //neigh[cell_id][3] = (kcell)*ncelly*ncellx + (jupper)*ncellx + icell;
               cells[c_id].neigh[3] = (kcell) * ncelly * ncellx + (jupper) * ncellx + icell;
               //5: same z, y- , same x
               //neigh[cell_id][4] = (kcell)*ncelly*ncellx + (jlower)*ncellx + icell;
               cells[c_id].neigh[4] = (kcell) * ncelly * ncellx + (jlower) * ncellx + icell;
               //6: same z, y+ , x+
               //neigh[cell_id][5] = (kcell)*ncelly*ncellx + (jupper)*ncellx + iupper;
               cells[c_id].neigh[5] = (kcell) * ncelly * ncellx + (jupper) * ncellx + iupper;
               //7: same z, y+ , x-
               //neigh[cell_id][6] = (kcell)*ncelly*ncellx + (jupper)*ncellx + ilower;
               cells[c_id].neigh[6] = (kcell) * ncelly * ncellx + (jupper) * ncellx + ilower;
               //8: same z, y-, x+
               //neigh[cell_id][7] = (kcell)*ncelly*ncellx + (jlower)*ncellx + iupper;
               cells[c_id].neigh[7] = (kcell) * ncelly * ncellx + (jlower) * ncellx + iupper;
               //9: same z, y-, x-
               //neigh[cell_id][8] = (kcell)*ncelly*ncellx + (jlower)*ncellx + ilower;
               cells[c_id].neigh[8] = (kcell) * ncelly * ncellx + (jlower) * ncellx + ilower;

               //10: z+1, y, x :
               //neigh[cell_id][9] = (kupper)*ncelly*ncellx + (jcell)*ncellx + icell;
               cells[c_id].neigh[9] = (kupper) * ncelly * ncellx + (jcell) * ncellx + icell;
               //11: z+1, same y, x+
               //neigh[cell_id][10] = (kupper)*ncelly*ncellx + (jcell)*ncellx + iupper;
               cells[c_id].neigh[10] = (kupper) * ncelly * ncellx + (jcell) * ncellx + iupper;
               //12: z+1, same y, x-
               //neigh[cell_id][11] = (kupper)*ncelly*ncellx + (jcell)*ncellx + ilower;
               cells[c_id].neigh[11] = (kupper) * ncelly * ncellx + (jcell) * ncellx + ilower;
               //13: z+1, y+ , same x
               //neigh[cell_id][12] = (kupper)*ncelly*ncellx + (jupper)*ncellx + icell;
               cells[c_id].neigh[12] = (kupper) * ncelly * ncellx + (jupper) * ncellx + icell;
               //14: z+1, y- , same x
               //neigh[cell_id][13] = (kupper)*ncelly*ncellx + (jlower)*ncellx + icell;
               cells[c_id].neigh[13] = (kupper) * ncelly * ncellx + (jlower) * ncellx + icell;
               //15: z+1, y+ , x+
               //neigh[cell_id][14] = (kupper)*ncelly*ncellx + (jupper)*ncellx + iupper;
               cells[c_id].neigh[14] = (kupper) * ncelly * ncellx + (jupper) * ncellx + iupper;
               //16: z+1, y+ , x-
               //neigh[cell_id][15] = (kupper)*ncelly*ncellx + (jupper)*ncellx + ilower;
               cells[c_id].neigh[15] = (kupper) * ncelly * ncellx + (jupper) * ncellx + ilower;
               //17: z+1, y-, x+
               //neigh[cell_id][16] = (kupper)*ncelly*ncellx + (jlower)*ncellx + iupper;
               cells[c_id].neigh[16] = (kupper) * ncelly * ncellx + (jlower) * ncellx + iupper;
               //18: z+1, y-, x-
               //neigh[cell_id][17] = (kupper)*ncelly*ncellx + (jlower)*ncellx + ilower;
               cells[c_id].neigh[17] = (kupper) * ncelly * ncellx + (jlower) * ncellx + ilower;

               //19: z-1, y, x :
               //neigh[cell_id][18] = (klower)*ncelly*ncellx + (jcell)*ncellx + icell;
               cells[c_id].neigh[18] = (klower) * ncelly * ncellx + (jcell) * ncellx + icell;
               //20: z-1, same y, x+
               //neigh[cell_id][19] = (klower)*ncelly*ncellx + (jcell)*ncellx + iupper;
               cells[c_id].neigh[19] = (klower) * ncelly * ncellx + (jcell) * ncellx + iupper;
               //21: z-1, same y, x-
               //neigh[cell_id][20] = (klower)*ncelly*ncellx + (jcell)*ncellx + ilower;
               cells[c_id].neigh[20] = (klower) * ncelly * ncellx + (jcell) * ncellx + ilower;
               //22: z-1, y+ , same x
               //neigh[cell_id][21] = (klower)*ncelly*ncellx + (jupper)*ncellx + icell;
               cells[c_id].neigh[21] = (klower) * ncelly * ncellx + (jupper) * ncellx + icell;
               //23: z-1, y- , same x
               //neigh[cell_id][22] = (klower)*ncelly*ncellx + (jlower)*ncellx + icell;
               cells[c_id].neigh[22] = (klower) * ncelly * ncellx + (jlower) * ncellx + icell;
               //24: z-1, y+ , x+
               //neigh[cell_id][23] = (klower)*ncelly*ncellx + (jupper)*ncellx + iupper;
               cells[c_id].neigh[23] = (klower) * ncelly * ncellx + (jupper) * ncellx + iupper;
               //25: z-1, y+ , x-
               //neigh[cell_id][24] = (klower)*ncelly*ncellx + (jupper)*ncellx + ilower;
               cells[c_id].neigh[24] = (klower) * ncelly * ncellx + (jupper) * ncellx + ilower;
```

```
                //26: z-1, y-, x+
                //neigh[cell_id][25] = (klower)*ncelly*ncellx + (jlower)*ncellx + iupper;
                cells[c_id].neigh[25] = (klower) * ncelly * ncellx + (jlower) * ncellx + iupper;
                //27: z-1, y-, x-
                //neigh[cell_id][26] = (klower)*ncelly*ncellx + (jlower)*ncellx + ilower;
                cells[c_id].neigh[26] = (klower) * ncelly * ncellx + (jlower) * ncellx + ilower;

              }

            }

        }



    }

    int Grid::find_grid_cell(const double &xnode, const double &ynode,
                             const double &znode) {

        double xid = xnode * idlx;
        double yid = ynode * idly;
        double zid = znode * idlz;

        return(int(zid)*(ncelly)*(ncellx) + int(yid) * ncellx + int(xid));

    }

    Grid::~Grid(){
        free(cells);

        return;
    }
 }
```

5.25 grid.h File Reference

The class of the non-bonded energy estimation grid.

Classes

- struct NetworkNS::sgrid_cell
- class NetworkNS::Grid

  The nonboned free energy estimation grid class.

Typedefs

- typedef struct NetworkNS::sgrid_cell NetworkNS::grid_cell

5.25.1 Detailed Description

Author

Grigorios G. Megariotis (gmegariotis@yahoo.gr)

Version

1.0 (July 21, 2012)

5.25.2 LICENSE

Copyright (c) 2012 Grigorios Megariotis. All rights reserved.

This work is licensed under the terms of the MIT license. For a copy, see https://opensource.↩ org/licenses/MIT.

Definition in file grid.h.

5.25.3 Typedef Documentation

5.25.3.1 grid_cell

```
typedef struct NetworkNS::sgrid_cell NetworkNS::grid_cell
```

The sgrid_cell is the elementary struct for storing the information concerning a cell of the free energy estimation grid.

5.26 grid.h

```

#ifndef GRID_H
#define  GRID_H


namespace NetworkNS {


   typedef struct sgrid_cell {
      int Id;
      double Vec[3];
      int neigh[27];
   } grid_cell;

   class Grid {
   public:

      Grid(double, double, double, int, int, int);
      virtual ~Grid();

      double dlx;
      double dly;
      double dlz;

      double idlx;

      double idly;

      double idlz;

      double vcell;
      double ivcell;

      int find_grid_cell(const double &xnode, const double &ynode, const double &znode);

      int ncellx;
      int ncelly;
      int ncellz;
      int ncells;

      grid_cell *cells;
   };
}




#endif   /* GRID_H */
```

5.27 hopping.cpp File Reference

This file contains the necessary routine for carrying out the slip-spring kinetic MC simulation.

```
#include <iostream>
#include <list>
```

```
#include <vector>
#include <cmath>
#include "constants.h"
#include "distributions.h"
#include "b3D_integrator.h"
#include "network.h"
#include "netmin.h"
#include "hopping.h"
#include "stdlib.h"
```

5.27.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

3.0

Warning

This class assumes that a network has been initialized and positions of the nodes can be found at the corresponding array.

Definition in file hopping.cpp.

5.28 hopping.cpp

```
#include <iostream>
#include <list>
#include <vector>
#include <cmath>
#include "constants.h"
#include "distributions.h"
#include "b3D_integrator.h"
#include "network.h"
#include "netmin.h"
#include "hopping.h"

#include "stdlib.h"

using namespace std;
using namespace NetworkNS;


namespace NetworkNS {

   Hopping::Hopping(double hopping_rate_constant) {

      /* Open a file to write the life-time of slip-springs. */
      lifetimes_file.open("ss_lifetimes.txt", ofstream::out);
      if (not lifetimes_file.good())
         cout << "Specified slip-springs lifetime file does not exist!" << endl;

      events_file.open("events.txt", ofstream::out);
      if (not events_file.good())
         cout << "Specified events file does not exist!" << endl;

      nu_hopping_times_exp_of_barrier = hopping_rate_constant; // s^{-1}

      return;
   }

   Hopping::~Hopping(){
      /* Close the lifetimes file. */
      lifetimes_file.close();
      // gvog: Close the events file:
      events_file.close();

      return;
   }


   void Hopping::hopping_step(NetwMin *netapp, const cb3D_integrator *b3D,
           double *pos_array, double temperature, double elapsed_time) {

      double dr[3], rsq;
      unsigned int transitions_performed = 0; // transitions counter
      double time_in_seconds = elapsed_time * 1.e-12; // simulation time in s
      double feng_old = 0.0;

#ifndef CONST_SL_SCHEME
      unsigned int new_slipsprings = 0;
#endif

      // gvog: A list of strands to be deleted at the current time step.
#ifdef CONST_SL_SCHEME
      std::list<std::pair<tStrand *, unsigned int > > to_be_deleted;
#else
      std::list<tStrand *> to_be_deleted;
#endif


      /* gvog: Loop over all slip-springs. */
      for (std::list<tStrand *>::iterator it = netapp->network->
pslip_springs.begin();
              it != netapp->network->pslip_springs.end(); ++it) {

         //compute the initial free-energy of the current slip-spring
         dr[0] = pos_array[3 * ((*it)->pEnds[0]->Id - 1) + 0]
               - pos_array[3 * ((*it)->pEnds[1]->Id - 1) + 0];

         dr[1] = pos_array[3 * ((*it)->pEnds[0]->Id - 1) + 1]
               - pos_array[3 * ((*it)->pEnds[1]->Id - 1) + 1];

         dr[2] = pos_array[3 * ((*it)->pEnds[0]->Id - 1) + 2]
               - pos_array[3 * ((*it)->pEnds[1]->Id - 1) + 2];

         // gvog: Ask for the shortest distance between the two beads:
         netapp->domain->minimum_image(dr[0], dr[1], dr[2]);

         // Calculate the free energy the spring contributes to the total free energy of the system.
#ifdef FENE_SLS
         feng_old = e_fene    (dr, (*it)->spring_coeff, (*it)->sq_end_to_end, temperature);
#else
         feng_old = e_gaussian(dr, (*it)->spring_coeff, (*it)->sq_end_to_end, temperature);
#endif

         double cur_slipspring_sq_distance = dr[0]*dr[0] + dr[1]*dr[1] + dr[2]*dr[2];

         // gvog: Calculate the exponential of the slip-spring free energy.
         double spring_boltz_factor = exp(feng_old/boltz_const_kJoule_molK/
temperature);
         // gvog: Calculate the probability in a timestep of Delta t
         double hopping_prob = nu_hopping_times_exp_of_barrier * spring_boltz_factor * time_in_seconds;

         /* REMINDER:
          *  Type = 1 for chain ends.
          *  Type = 2 for internal beads.
          *  Type = 3 for crosslinks.
          */

         // gvog: Allocate a list of possible candidates to jump on for every end of the slip-spring:
         vector<list<pair<tNode*, double> > > sls_attchmnts(2);

         /* Loop over both ends of the slip-spring. */
         for (unsigned int iend = 0; iend < 2; iend++) {

            tNode* cur_sls_end = (*it)->pEnds[iend];

            /* Check whether the slip-spring has any of its ends connected to a chain end. */
            if (cur_sls_end->Type == 1)
               /* The one possible point of attachment is the vacuum, so set it accordingly. */
               sls_attchmnts[iend].push_back(pair<tNode*,double>(static_cast<tNode *>(0), hopping_prob));

            if (cur_sls_end->Type != 3) {
               // Loop over all strands connected to the other end of the slip-spring
               for (vector<tStrand *>::iterator end_inc_strand = cur_sls_end->
pStrands.begin();
                    end_inc_strand != cur_sls_end->pStrands.end(); ++end_inc_strand){

                  // check which end of the incident strand we should consider:
                  if ((*end_inc_strand)->pEnds[0] == cur_sls_end) {
                     if ((*end_inc_strand)->pEnds[1]->Type != 3)
                        sls_attchmnts[iend].push_back(pair<tNode*, double>((*end_inc_strand)->pEnds[1],
hopping_prob));
                  }
                  else {
                     if ((*end_inc_strand)->pEnds[0]->Type != 3)
                        sls_attchmnts[iend].push_back(pair<tNode*, double>((*end_inc_strand)->pEnds[0],
hopping_prob));
                  }
               }
            }
            else /* (*it)->pEnds[0]->Type == 3 */
               cout << "#: hopping.cpp: Slip-spring attached to crosslink detected!.\n";
         }

//#define DEBUG_HOPPING
#ifdef DEBUG_HOPPING
         cout << " ---- ---- ---- ----" << endl;
         cout << "Slip - spring: " << (*it)->Id << endl;
         cout << "possible transitions for left end: " ;
         for (list<pair<tNode*, double> >::iterator isls = sls_attchmnts[0].begin();
              isls != sls_attchmnts[0].end(); ++isls){
            if ((*isls).first != 0)
               cout << "( " << (*isls).first->Id << ", " << (*isls).second << "), ";
            else
               cout << "( vacuum, " << (*isls).second << "), ";
         }
         cout << endl;

         cout << "possible transitions for right end: ";
         for (list<pair<tNode*, double> >::iterator isls = sls_attchmnts[1].begin();
              isls != sls_attchmnts[1].end(); ++isls){
            if ((*isls).first != 0)
               cout << "( " << (*isls).first->Id << ", " << (*isls).second << "), ";
            else
               cout << "( vacuum, " << (*isls).second << "), ";
         }
         cout << endl << " ---- ---- ---- ----" << endl << endl;
#endif

         // gvog: Check whether the sum of the transition probabilities is greater than 1.0.
         double summer = 0.0;
         for (vector<list<pair<tNode*,double > > >::iterator iend = sls_attchmnts.begin(); iend !=
sls_attchmnts.end(); ++iend)
            for (list<pair<tNode*, double > >::iterator end_ineigh = (*iend).begin(); end_ineigh != (*iend)
.end(); ++end_ineigh)
               summer += (*end_ineigh).second;

         // gvog: If the sum of the probabilities exceeded 1.0, normalize it to unity.
         if (summer > 1.0) {
            // TODO: Replace the following quick fix with something smarter:

            cout << "#: warning: Please change the transition probabilities. Sum = " << summer << " > 1.0\n
";

            for (vector<list<pair<tNode*,double > > >::iterator iend = sls_attchmnts.begin(); iend !=
sls_attchmnts.end(); ++iend)
               for (list<pair<tNode*, double > >::iterator end_ineigh = (*iend).begin(); end_ineigh !=
(*iend).end(); ++end_ineigh)
                  (*end_ineigh).second /= summer;
         }


         bool perform_transition = false;
         double cum_prob = 0.0;
         double ran_num = netapp->my_rnd_gen->uniform();

         unsigned int end_to_jump = 0;
         tNode* target_node = 0;

         for (unsigned int iend = 0; iend < 2; iend++)
               for (list<pair<tNode*, double > >::iterator end_ineigh = sls_attchmnts[iend].begin();
                    end_ineigh != sls_attchmnts[iend].end(); ++end_ineigh)

               if (!perform_transition) {
                  // gvog: Add to the cumulative probability:
                  cum_prob += (*end_ineigh).second;

                  if (cum_prob > ran_num) {
                     // Here I have to store the pair (itr,jtr) and break.
                     end_to_jump = iend;
                     target_node = (*end_ineigh).first;

                     perform_transition = true;
                  }
               }
```

```

         // gvog: In case the probability is too small, try another loop iteration.
         // gvog: pre 2016/02/02 - bug pointed by Aris Sgouros:
         // if ((cum_prob < 1.e-12) || (end_to_jump == 0 && target_node == 0))
         //    continue;
         // gvog: post 2016/02/02: no boolean condition

         bool destroy_slpsprng = false;

         if (perform_transition) {

            // gvog: Increase the counter of transitions performed.
            transitions_performed++;

            // gvog: Write the ID of the slip-spring to the file:
            events_file << (*it)->Id << " " << ((double)b3D->bd_cur_step) << endl;

#ifdef DEBUG_HOPPING
            if (target_node)
               cout << "end " << end_to_jump << " has hopped to " << target_node->
Id << endl;
            else
               cout << "end " << end_to_jump << " has gone to vacuum" << endl;
            cout << "random number = " << ran_num << endl;

            exit(0);
#endif

            /* In case the attachment point exists: */
            if (target_node)
               (*it)->pEnds[end_to_jump] = target_node;
            else
               destroy_slpsprng = true;

            if (destroy_slpsprng)
               // gvog: The slip-spring can be deleted only if its distance is smaller than the attempt
 radius:
               if (cur_slipspring_sq_distance < (hopping_attempt_radius * hopping_attempt_radius))
#ifdef CONST_SL_SCHEME
                  to_be_deleted.push_back(std::pair<tStrand *, unsigned int>((*it), end_to_jump));
#else
                  to_be_deleted.push_back((*it));
#endif
         }
      }


#ifndef CONST_SL_SCHEME
      for (std::list<tStrand *>::iterator it = to_be_deleted.begin();
              it != to_be_deleted.end(); ++it) {

         // Find the element to be deleted:
         for (std::list<tStrand>::iterator jt = netapp->network->strands.begin();
                 jt != netapp->network->strands.end(); ++jt)
            if (&(*jt) == (*it)) {
               netapp->network->strands.erase(jt);
               break;
            }

         /* Write the lifetime of the slip-spring to the file: */
         lifetimes_file << (int) (b3D->bd_cur_step - (*it)->tcreation) << endl;

         // gvog: Finally, we can delete the slip-spring from the pointer array:
         netapp->network->pslip_springs.remove((*it));
      }
#else
      for (std::list<std::pair<tStrand *, unsigned int> >::iterator it = to_be_deleted.begin();
              it != to_be_deleted.end(); ++it) {

         /*
          * gvog: In the constant number of slip-springs scheme, we have to check whether the
          *       end-to-end distance of the spring to be destroyed is smaller than the capture
          *       radius, for the detailed balance to hold.
          */
         dr[0] = pos_array[3 * ((*it).first->pEnds[1]->Id - 1) + 0]
               - pos_array[3 * ((*it).first->pEnds[0]->Id - 1) + 0];

         dr[1] = pos_array[3 * ((*it).first->pEnds[1]->Id - 1) + 1]
               - pos_array[3 * ((*it).first->pEnds[0]->Id - 1) + 1];

         dr[2] = pos_array[3 * ((*it).first->pEnds[1]->Id - 1) + 2]
               - pos_array[3 * ((*it).first->pEnds[0]->Id - 1) + 2];

         /* Apply minimum image convention to the separation vector. */
         netapp->domain->minimum_image(dr[0], dr[1], dr[2]);
         rsq = dr[0] * dr[0] + dr[1] * dr[1] + dr[2] * dr[2];

         /*
          * gvog: Only if the distance of the strand to be deleted is smaller than the tube diameter the
          *       whole deletion/creation procedure can proceed.
          */
         if (rsq <= hopping_attempt_radius * hopping_attempt_radius) {

            // gvog: Initialize Rosenbluth weights to zero.
            double rosen_old = 0.0, rosen_new = 0.0;

            // gvog: Calculate the Rosenbluth weight in the old configuration:
            for (std::list <tNode>::iterator jat = netapp->network->nodes.begin();
                    jat != netapp->network->nodes.end(); ++jat) {

               /*
                * gvog: We allow internal and end-end connections. Only crosslinks (Type = 3) are excluded
                *       from possible candidates. Moreover, connections along the same chain are allowed,
                *       even with the same bead.
                */
#ifdef ALLOW_INTRAMOLECULAR
               // gvog: Here we exclude sites belonging to the same chain:
               if ((*jat).Type == 3)
#else
               if (((*jat).Type == 3) || ((*jat).OrChains[0] == (*it).first->pEnds[(*it).second]->OrChains[0]))
#endif
                  continue;

               /* check the distance from the end to the bead: */
               dr[0] = pos_array[3 * ((*jat).Id - 1) + 0]
                     - pos_array[3 * ((*it).first->pEnds[(*it).second]->Id - 1) + 0];

               dr[1] = pos_array[3 * ((*jat).Id - 1) + 1]
                     - pos_array[3 * ((*it).first->pEnds[(*it).second]->Id - 1) + 1];

               dr[2] = pos_array[3 * ((*jat).Id - 1) + 2]
                     - pos_array[3 * ((*it).first->pEnds[(*it).second]->Id - 1) + 2];
               /* Apply minimum image convention to the separation vector. */
               netapp->domain->minimum_image(dr[0], dr[1], dr[2]);
               rsq = dr[0] * dr[0] + dr[1] * dr[1] + dr[2] * dr[2];

               // gvog: Add the contribution of the strand to the Rosenbluth weight
               rosen_old += exp(-e_gaussian(dr, (*it).first->spring_coeff, (*it).first->sq_end_to_end, temperature)
                          / boltz_const_kJoule_molK / temperature);
            }

            // gvog: NEW CONFIGURATION
            // gvog: Select randomly one of the chain ends
            unsigned int nends = netapp->network->sorted_chains.size() * 2;
            unsigned int iend = (unsigned int) (netapp->my_rnd_gen->uniform() * double(nends));
            unsigned int ichain = (unsigned int) (iend / 2);

            tNode* new_end = 0;

            // gvog: Bug fix proposed by Aris Sgouros on January, 29
            if (iend % 2 == 0)
               // gvog: Select the end of the chain:
               new_end = netapp->network->sorted_chains[ichain].back()->pEnds[1];
            else
               // gvog: Select the start of the chain:
               new_end = netapp->network->sorted_chains[ichain].front()->pEnds[0];

            /* Create a vector to store the candidates for slip-spring bridging. */
            std::vector<std::pair<tNode *, double> > candidates;
            std::pair<tNode *, double> cur_cand;

            /* Loop over all beads: */
            for (std::list <tNode>::iterator jat = netapp->network->nodes.begin();
                    jat != netapp->network->nodes.end(); ++jat) {

               // gvog: check whether the candidate is a crosslink, or belongs to the same chain.

#ifdef ALLOW_INTRAMOLECULAR
               if ((*jat).Type == 3)
#else
               if (((*jat).Type == 3) || ((*jat).OrChains[0] == (new_end->OrChains[0])))
#endif
                  continue;

               /* check the distance from the end to the bead: */
               dr[0] = pos_array[3 * ((*jat).Id - 1) + 0]
                     - pos_array[3 * (new_end->Id - 1) + 0];

               dr[1] = pos_array[3 * ((*jat).Id - 1) + 1]
                     - pos_array[3 * (new_end->Id - 1) + 1];

               dr[2] = pos_array[3 * ((*jat).Id - 1) + 2]
                     - pos_array[3 * (new_end->Id - 1) + 2];

               /* Apply minimum image convention to the separation vector. */
               netapp->domain->minimum_image(dr[0], dr[1], dr[2]);
               rsq = dr[0] * dr[0] + dr[1] * dr[1] + dr[2] * dr[2];

               /* If the distance is smaller than the tube diameter of the polyisoprene,
                * append this neighbor to the candidates list. */
               if (rsq <= hopping_attempt_radius * hopping_attempt_radius) {
                  cur_cand.first = &(*jat);
                  cur_cand.second = e_gaussian(dr, (*it).first->spring_coeff, (*it).first->
sq_end_to_end, temperature);

                  rosen_new += exp(-cur_cand.second / boltz_const_kJoule_molK /
temperature);

                  candidates.push_back(cur_cand);
               }
            }

            /* Calculate the cumulative probability of each candidate: */
            double cum_prob = 0.0, ran_num = netapp->my_rnd_gen->
uniform();
            tNode *sel_candidate = 0;
            for (std::vector<std::pair<tNode *, double> >::iterator icand = candidates.begin();
                  icand != candidates.end(); ++icand) {
               cum_prob += exp(-(*icand).second / boltz_const_kJoule_molK /
temperature) / rosen_new;
               if (cum_prob > ran_num) {
                  sel_candidate = (*icand).first;
                  feng_new = (*icand).second;
                  break;
               }
            }

            // gvog: rosen_old -> old rosenbluth weight
            // gvog: rosen_new -> new rosenbluth weight
            // gvog: feng_old -> old energy
            // gvog: feng_new -> new energy
            double criterion =  exp((feng_new-feng_old)/boltz_const_kJoule_molK/
temperature)*rosen_new/rosen_old;

            // gvog: Ask for a random number:
            ran_num = netapp->my_rnd_gen->uniform();
            if (criterion >= ran_num) {
               // gvog: The move has been accepted, update the connectivity of the slip-spring.
               (*it).first->pEnds[0] = new_end;
               (*it).first->pEnds[1] = sel_candidate;
            }

         }
         else
            cout << "# warning: increase capture radius for the hopping scheme." << endl;

      }
#endif

      // cout << "#: hopping.cpp: slip-springs deleted = " << to_be_deleted.size() << endl;


#ifndef CONST_SL_SCHEME
      /* gvog:
       * Slip-spring creation event:
       * 1. loop over all chain ends of the system.
       * 2. for every chain end search in a sphere of prescribed radius for other beads
       * 3. calculate the probability of creating a new slip-spring and select one of the candidates at
 random
       */
      std::vector<std::list<tStrand *> >::iterator ich;
      std::list <tNode>::iterator jat;
      std::vector<tNode *> ich_ends(2);


      for (ich = netapp->network->sorted_chains.begin();
              ich != netapp->network->sorted_chains.end(); ++ich) {

         /* Extract the first and the last bead of chain ich: */
         ich_ends[0] = (*ich).front()->pEnds[0];
         ich_ends[1] = (*ich).back()->pEnds[1];


         /* Loop over both ends of the chain: */
         for (std::vector<tNode *>::iterator iend = ich_ends.begin();
                iend != ich_ends.end(); ++iend) {

```

```
            /* Create a vector to store the candidates for slip-spring bridging. */
            std::list<tNode *> candidates;

            /* Loop over all beads of the system: */
            for (jat = netapp->network->nodes.begin();
                    jat != netapp->network->nodes.end(); ++jat) {

               // gvog: Do not construct a slip-spring with a crosslink:
#ifdef ALLOW_INTRAMOLECULAR
               if ((*jat).Type == 3)
#else
               if (((*jat).Type == 3) || ((*jat).OrChains[0] == (*iend)->OrChains[0]))
#endif
                  continue;

               /* check the distance from the end to the bead: */
               dr[0] = pos_array[3 * ((*jat).Id - 1) + 0]
                     - pos_array[3 * ((*iend)->Id - 1) + 0];

               dr[1] = pos_array[3 * ((*jat).Id - 1) + 1]
                     - pos_array[3 * ((*iend)->Id - 1) + 1];

               dr[2] = pos_array[3 * ((*jat).Id - 1) + 2]
                     - pos_array[3 * ((*iend)->Id - 1) + 2];


               /* Apply minimum image convention to the separation vector. */
               netapp->domain->minimum_image(dr[0], dr[1], dr[2]);
               rsq = dr[0]*dr[0] + dr[1]*dr[1] + dr[2]*dr[2];

               /* If the distance is smaller than the tube diameter of the polyisoprene,
                * append this neighbor to the candidates list. */
               if (rsq <= hopping_attempt_radius * hopping_attempt_radius)
                  candidates.push_back(&(*jat));
            }

            // gvog: Calculate the creation probability for the current chain end:
            double creation_prob = nu_hopping_times_exp_of_barrier * (double)candidates.size() *
      time_in_seconds;

            // gvog: and check that it is lower than 1.0
            if (creation_prob > 1.0) {
               cout << "#: hopping.cpp: Please change the rate prefactor. Creation probability = " <<
      creation_prob << " > 1.0\n";
               //exit(EXIT_FAILURE);
            }

            // gvog: ask for a random number to decide whether the creation should be attempted:
            double ran_num = netapp->my_rnd_gen->uniform();
            tNode *sel_candidate = 0;
            unsigned int isel_cand = 0;

            if (creation_prob > ran_num){
               // gvog: we have to select one of the candidates to bridge our end with:
               ran_num = netapp->my_rnd_gen->uniform();
               isel_cand = (unsigned int)(ran_num*(double)candidates.size());

               unsigned int icand = 0;
               for (list<tNode *>::iterator it = candidates.begin(); it!=candidates.end(); ++it){
                  sel_candidate = (*it);
                  icand ++;
                  if (icand > isel_cand)
                     break;
               }
            }


            /* Create a new strand and append it the appropriate lists. */
            if (sel_candidate) {
               /* Here I have to create a new strand, and update the corresponding arrays.*/
               tStrand new_slip_spring;
               new_slip_spring.Id = netapp->network->strands.back().Id + new_slipsprings +
      1;
               new_slip_spring.pEnds.resize(2);
               new_slip_spring.pEnds[0] = *iend;
               new_slip_spring.pEnds[1] = sel_candidate;
               new_slip_spring.slip_spring = true;
               new_slip_spring.OrChain = 0;
               new_slip_spring.tcreation = b3D->bd_cur_step;

               if (netapp->network->pslip_springs.size() > 0){
                  new_slip_spring.spring_coeff = netapp->network->
      pslip_springs.back()->spring_coeff;
                  new_slip_spring.sq_end_to_end = netapp->network->
      pslip_springs.back()->sq_end_to_end;
                  new_slip_spring.kuhn_length = netapp->network->
      pslip_springs.back()->kuhn_length;
```

```
               }
               else {
                  cout << "#: (warning): The first slip-spring of the system has been initialized with "
                       << "hard-coded coefficients.\n";
                  new_slip_spring.spring_coeff =  netapp->network->strands.front().spring_coeff;
                  new_slip_spring.kuhn_length = 9.58;
                  new_slip_spring.sq_end_to_end = pi_tube_diameter * pi_tube_diameter;

                  /* Also, create a new bond type: */
                  tBond_type new_bond_type;
                  new_bond_type.spring_coeff = new_slip_spring.spring_coeff;
                  new_bond_type.kuhnl = new_slip_spring.kuhn_length;
                  new_bond_type.sq_ete = new_slip_spring.sq_end_to_end;

                  netapp->network->bond_types.push_back(new_bond_type);
               }


               // gvog: and add it to the list of slip-springs
               netapp->network->strands.push_back(new_slip_spring);
               netapp->network->pslip_springs.push_back(&(netapp->network->strands.back()));
               new_slipsprings++;
            }
         }
      }

      //cout << "# hopping.cpp: new slip-springs created: " << new_slipsprings << endl;


      /* Re-count all springs in order to achieve continuous enumeration.*/
      unsigned int start_tag = netapp->network->strands.size() - netapp->network->pslip_springs.size();

      for (std::list<tStrand *>::iterator it = netapp->network->pslip_springs.begin();
              it != netapp->network->pslip_springs.end(); ++it)
         (*it)->Id = start_tag++;
#endif

      //cout << "#: hopping.cpp: transitions performed = " << transitions_performed - to_be_deleted.size() << endl;

      return;

   }

}

```

5.29 hopping.h File Reference

The header file of the slip-spring hopping kinetic Monte Carlo scheme.

```
#include <fstream>
#include <iostream>
#include <string>
#include "b3D_integrator.h"
#include "net_types.h"
#include "network.h"
```

Classes

- class NetworkNS::Hopping

  The class of the hopping kinetic Monte Carlo scheme. It can be called by a Brownian Dynamics class, get all the necessary information from it and alter the connectivity of the system, based on the rates described in hopping.cpp .

5.29.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

2.0 (Created on May 14, 2013)

Definition in file hopping.h.

5.30 hopping.h

```

#ifndef HOPPING_H
#define  HOPPING_H

#include <fstream>
#include <iostream>
#include <string>

#include "b3D_integrator.h"
#include "net_types.h"
#include "network.h"

using namespace std;
using namespace NetworkNS;

namespace NetworkNS{

   class Hopping {

   public:
      Hopping(double);
      ~Hopping();

      void hopping_step(class NetwMin *netapp, const class
cb3D_integrator *b3D, double *pos_array,
                        double temperature, double elapsed_time);

   private:
      std::ofstream lifetimes_file;

      std::ofstream events_file;

      double nu_hopping_times_exp_of_barrier;

   };


}




#endif   /* HOPPING_H */

```

5.31 main.cpp File Reference

The main C++ source file driving the execution of the code.

```
#include "b3D_integrator.h"
#include "Auxiliary.h"
#include "netmin.h"
```

Functions

- int main (int argc, char **argv)

5.31.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)
Grigorios Megariotis (gmegariotis@yahoo.gr)

Version

1.0 (Created on April 30, 2012)

5.31.2 LICENSE

Definition in file main.cpp.

5.31.3 Function Documentation

5.31.3.1 main()

```
int main (
            int argc,
            char ** argv )
```

Parameters

in	*argc*	The count of arguments provided through the command line.
in	*argv*	A 2D-array of characters containing the arguments provided through the command line.

Definition at line 30 of file main.cpp.

5.32 main.cpp

```
00001
00017 // Keep in mind the following web-page concerning Doxygen documentation system:
00018 // https://modelingguru.nasa.gov/docs/DOC-1811
00019
00020 #include "b3D_integrator.h"
00021 #include "Auxiliary.h"
00022 #include "netmin.h"
00023
00024 using namespace NetworkNS;
00025
00030 int main(int argc, char** argv) {
00031
00032
00033    unsigned int bd3D_nsteps = 0, bd3D_write_every = 0;
00034
00035    cout.precision(12);
00036
00037    cout << "#: Use of the code: "<< endl;
00038    cout << "#: ./netmin.x [data_file] [slip-spring creation rate] [nsteps] [nevery_steps]\n" << endl;
00039
00040    std::string filename;
00041    if (argc > 1)
00042       filename = std::string(argv[1]);
```

```
   else{
      cout << "#: Data file has not been specified from command line.\n";
      filename = "emsipon.data";
      cout << "#: --- '" << filename << "' will be used as data file.\n";
   }

   double creation_rate;
   if (argc > 2)
      creation_rate = StringToNumber<double>(argv[2]);
   else
      return(1);

   if (argc > 3)
      bd3D_nsteps = atol(argv[3]);
   else
      bd3D_nsteps = 2000000000;

   if (argc > 4)
      bd3D_write_every = atol(argv[4]);
   else
      bd3D_write_every = 1000;


#ifdef FENE_SLS
   cout << "#: (warning): FENE potential will be used for slip-springs.\n";
#endif

   // Define a new pointer to the class:
   NetwMin netw_min(filename);
   cb3D_integrator b3D_integrator(&netw_min, 500.0, creation_rate);

   cout << "#: Integrator has been initialized. Simulation will start for "<<bd3D_nsteps<< " steps.\n";
   cout << "#: The rate for the slip-spring creation is: " << creation_rate << endl;

   // Simulation starts here:
   double timestep = 1.0; // integration timestep in ps.
   cout << "#: The timestep of the integrator will be: " << timestep << " ps.\n";
   cout << "#: Report will be written every: " << bd3D_write_every << " steps.\n";


   b3D_integrator.integrate(bd3D_nsteps, timestep, bd3D_write_every);

   // Close trajectory file:
   netw_min.my_traj_file->~dump();


   return (EXIT_SUCCESS);
}

```

5.33 net_types.h File Reference

The header file containing elementary data types of bead and strand.

```
#include <list>
#include <vector>
```

Classes

- struct sNode

  The sNode is the basic struct keeping all information relevant to a bead or a network node. Once it is defined, it is converted to type tNode, which is used throughout the application.

- struct sStrand

  The sStrand is the basic abstract data type keeping all relevant information concerning a strand of a chain.

- struct sBead_type

  An elementary data type for reading in the information concerning a bead.

- struct sBond_type

  An elementary data type for reading in the information concerning a strand.

Typedefs

- typedef struct sNode tNode

  The sNode is the basic struct keeping all information relevant to a bead or a network node. Once it is defined, it is converted to type tNode, which is used throughout the application.
- typedef struct sStrand tStrand

  The sStrand is the basic abstract data type keeping all relevant information concerning a strand of a chain.
- typedef std::list< tStrand > tSubCh

  A subchain of the network, treated as a vector of internal strands.
- typedef struct sBead_type tBead_type

  An elementary data type for reading in the information concerning a bead.
- typedef struct sBond_type tBond_type

  An elementary data type for reading in the information concerning a strand.

5.33.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

3.0 (May 12, 2012)

5.33.2 LICENSE

Copyright (c) 2012 Georgios Vogiatzis. All rights reserved.

This work is licensed under the terms of the MIT license. For a copy, see https://opensource.org/licenses/MIT.

Definition in file net_types.h.

5.34 net_types.h

```
#ifndef _NET_TYPES_H
#define _NET_TYPES_H

#include <list>
#include <vector>

// Forward declaration, in order to create a pointer...
using namespace std;


/* Forward declaration of structure (object) strand in order to
 * create a pointer to this kind inside structure (object) node. */
struct sStrand;
struct sChain;                    //new addition, November 2012


typedef struct sNode{
   int Id;
   int Type;

   double Pos[3];

   std:: vector <int>        OrChains;

   std:: vector <sStrand *> pStrands;

   std:: vector <int>        SubCh;

   std:: vector <sChain *>   pChains;
```

```

    int  node_cell;

    double n_mass;

    double mass;

    double r_node;

    double r_star;
}tNode;


typedef struct sStrand{

    int Id;

    int Type;

    int OrChain;

    bool slip_spring;

    unsigned int tcreation;

    double spring_coeff;

    double sq_end_to_end;

    double kuhn_length;

    std:: vector <tNode *> pEnds;

    double * pChain;

} tStrand;

typedef std::list <tStrand> tSubCh;

typedef struct sBead_type{

    double mass;
    double n_mass;
    double r_node;
} tBead_type;

typedef struct sBond_type{

    double spring_coeff;
    double sq_ete;
    double kuhnl;
} tBond_type;

#endif   /* _NET_TYPES_H */

```

5.35 netmin.cpp File Reference

The routines of the "NetwMin" application class are defined here.

```
#include <string>
#include "netmin.h"
```

5.35.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

1.1 (November 28, 2013)

5.35.2 LICENSE

Definition in file netmin.cpp.

5.36 netmin.cpp

```
#include <string>
#include "netmin.h"

using namespace std;

namespace NetworkNS{

   NetwMin::NetwMin(std::string filename){

      domain = new Domain(filename);
      cout << "#: Domain characteristics have been read.\n";
      network = new Network(this,filename);
      cout << "#: Network has been read.\n";
      my_rnd_gen = new RanMars(11111);

      cout << "#: Particles have been read.\n";

      my_traj_file = new dump("dump_b3D.lammpstrj");
      cout << "#: 'dump_b3D.lammpstrj' will be used as trajectory file.\n";

      grid = new Grid(domain->XBoxLen, domain->YBoxLen, domain->ZBoxLen, 5, 5, 5);
      write_network_to_lammps_data_file();

      return;
   }


   void NetwMin::write_network_to_lammps_data_file() {

      double wrapped_coords[3];
      int  pcoeffs[3];

      pcoeffs[0] = 0;
      pcoeffs[1] = 0;
      pcoeffs[2] = 0;

      FILE *lammps_file = fopen("restart.data", "wt");


      /* Write header to LAMMPS data file: */

      fprintf(lammps_file, "LAMMPS data file of crosslinked coarse-grained network.\n\n");
      fprintf(lammps_file, "%ld atoms\n", network->nodes.size());
      fprintf(lammps_file, "%ld bonds\n", network->strands.size());
      fprintf(lammps_file, "\n");

      fprintf(lammps_file, "%lu atom types\n", network->node_types.size());

      if (network->pslip_springs.size() > 0)
         fprintf(lammps_file, "2 bond types\n");
      else
         fprintf(lammps_file, "1 bond types\n");
      fprintf(lammps_file, "\n");

      fprintf(lammps_file, "%lf %lf xlo xhi\n",
              domain->BoxLow[0], domain->BoxHigh[0]);
      fprintf(lammps_file, "%lf %lf ylo yhi\n",
              domain->BoxLow[1], domain->BoxHigh[1]);
      fprintf(lammps_file, "%lf %lf zlo zhi\n",
              domain->BoxLow[2], domain->BoxHigh[2]);


      fprintf(lammps_file, "\nMasses\n\n");
      for (unsigned int i = 0; i < network->node_types.size(); i++)
         fprintf(lammps_file, "%d \t %lf \t %-1.4e \t # n_Kuhns/bead - g/mol\n",
                 i+1, network->node_types[i].n_mass, network->node_types[i].mass);


      fprintf(lammps_file, "\nBond Coeffs\n\n");
      for (unsigned int i = 0; i < network->bond_types.size(); i++)
         fprintf(lammps_file,
                 "%d \t %-1.4e \t %-1.4e \t %-1.4e \t # 3/2*k_B (kJ/mol/K) - <R_e^2> (A^2) - b (A)\n",
                 i+1, network->bond_types[i].spring_coeff,
                 network->bond_types[i].sq_ete, network->bond_types[i].kuhnl);


      fprintf(lammps_file, "\nPair Coeffs\n\n");
      for (unsigned int i = 0; i < network->node_types.size(); i++)
         fprintf(lammps_file, "%d \t %lf \t # r_node = b*n_j^(1/2)\n",i+1,
                 network->node_types[i].r_node);


      fprintf(lammps_file, "\nAtoms\n\n");
      int inode = 0;
      for (std::list<tNode> ::iterator it = network->nodes.begin();
           it != network->nodes.end(); ++it) {
         inode++;
         (*it).Id = inode;

         wrapped_coords[0] = (*it).Pos[0];
         wrapped_coords[1] = (*it).Pos[1];
         wrapped_coords[2] = (*it).Pos[2];
         //domain->put_in_primary_box(wrapped_coords, pcoeffs);

         fprintf(lammps_file, "%d  %d  %d  0.000  %.10f  %.10f  %.10f  %d  %d  %d\n",
                 (*it).Id, (*it).OrChains[0] ,(*it).Type,
                 wrapped_coords[0], wrapped_coords[1], wrapped_coords[2],
                 pcoeffs[0], pcoeffs[1], pcoeffs[2]);
      }

      fprintf(lammps_file, "\nBonds\n\n");

      int istrand = 0;
      for (std::list<tStrand> ::iterator it = network->strands.begin();
           it != network->strands.end(); ++it) {
         istrand++;

         if ((*it).slip_spring)
            fprintf(lammps_file, "%d  2  %d  %d # %d\n",
                 istrand, (*it).pEnds[0]->Id, (*it).pEnds[1]->Id, (*it).OrChain);
         else
            fprintf(lammps_file, "%d  1  %d  %d # %d\n",
                 istrand, (*it).pEnds[0]->Id, (*it).pEnds[1]->Id, (*it).OrChain);
      }
      fclose(lammps_file);

      return;
   }

}
```

5.37 netmin.h File Reference

The class of the network application itself.

```
#include <vector>
#include "domain.h"
#include "dump.h"
#include "grid.h"
#include "hopping.h"
#include "rng.h"
#include "network.h"
```

Classes

- class NetworkNS::NetwMin

  The class of the host application itself. It contains pointers to all constituents, e.g. the simulation domain, random number generator, etc.

5.37.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

4.0 (November 28, 2013)

5.37.2 LICENSE

Definition in file netmin.h.

5.38 netmin.h

```

#ifndef NETWMIN_H
#define  NETWMIN_H

#include <vector>

#include "domain.h"
#include "dump.h"
#include "grid.h"
#include "hopping.h"
#include "rng.h"
#include "network.h"


namespace NetworkNS {


   class NetwMin {
   public:
      class Network *network;

      class Domain *domain;

      class Grid *grid;

      class RanMars *my_rnd_gen;

      class dump *my_traj_file;

      NetwMin(std::string);

      void write_network_to_lammps_data_file();

   };
}

#endif   /* NETWMIN_H */

```

5.39 network.cpp File Reference

The C++ source file containing all functions relevant to the class Network.

```
#include <algorithm>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <iterator>
```

```
#include <list>
#include <sstream>
#include <vector>
#include <cmath>
#include "constants.h"
#include "netmin.h"
#include "Auxiliary.h"
```

Functions

- bool NetworkNS::pred_strand_is_end (tStrand ∗)

    A boolean function for judging whether a strand is a chain end.

5.39.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

5.39.2 LICENSE

Definition in file network.cpp.

5.39.3 Function Documentation

5.39.3.1 pred_strand_is_end()

```
bool NetworkNS::pred_strand_is_end (
            tStrand ∗ current )
```

Parameters

in	*current*	A pointer to a type tStrand. The strand to check whether is located at a chain end or no.

Definition at line 315 of file network.cpp.

5.40 network.cpp

```

#include <algorithm>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <iterator>
#include <list>
#include <sstream>
#include <vector>
#include <cmath>

```

```
#include "constants.h"
#include "netmin.h"
#include "Auxiliary.h"


using namespace std;


namespace NetworkNS {

   Network::~Network() {
      nodes.clear();
      strands.clear();
   }


   Network::Network(class NetwMin *netw_min, std::string filename) {

      int nnodes = 0, nstrands = 0, atoms_line_start = 0, bonds_line_start = 0,
              bond_coeffs_line_start = 0, masses_line_start = 0, pair_coeffs_line_start = 0,
              atom_types_line = 0, bond_types_line = 0, ihelp;

      nstrands = 0;
      cout << "#: Initializing network.\n";
      /* Define and open the desired file. */
      ifstream data_file(filename.c_str(), ifstream::in);
      /* Define an array of strings to hold the contents of the file. */
      std::vector <string> lines_of_file;
      std::vector <string> tokens;

      /* Read and search file. */
      int iline = 0;

      while (data_file.good()) {
         /* A temporary string for the current line of the file. */
         std::string current_line;
         getline(data_file, current_line);
         /* Add current line to file's array of lines. */
         lines_of_file.push_back(current_line);

         if (current_line.find("atoms") != string::npos)
            nnodes = atoi(current_line.c_str());
         if (current_line.find("bonds") != string::npos)
            nstrands = atoi(current_line.c_str());
         if (current_line.find("Atoms") != string::npos)
            atoms_line_start = iline;
         if (current_line.find("Bonds") != string::npos)
            bonds_line_start = iline;
         if (current_line.find("Masses") != string::npos)
            masses_line_start = iline;
         if ((current_line.find("Bond") != string::npos)
                 && (current_line.find("Coeffs") != string::npos))
            bond_coeffs_line_start = iline;
         if ((current_line.find("Pair") != string::npos)
                 && (current_line.find("Coeffs") != string::npos))
            pair_coeffs_line_start = iline;
         if ((current_line.find("atom") != string::npos)
                 && (current_line.find("types") != string::npos))
            atom_types_line = iline;
         if ((current_line.find("bond") != string::npos)
                 && (current_line.find("types") != string::npos))
            bond_types_line = iline;

         iline++;
      }
      data_file.close();

      /* Read the masses of the nodes from the corresponding session of the data file. */
      tokens = tokenize(lines_of_file[atom_types_line]);
      ihelp = atoi(tokens[0].c_str());
      cout << "#: --- Number of bead types: " << ihelp << " .\n";
      node_types.resize(ihelp);
      for (int i = 0; i < ihelp; i++) {
         tokens = tokenize(lines_of_file[masses_line_start + i + 2]);
         node_types[i].n_mass = atof(tokens[1].c_str());
         node_types[i].mass = atof(tokens[2].c_str());

         tokens = tokenize(lines_of_file[pair_coeffs_line_start+i+2]);
         node_types[i].r_node = atof(tokens[1].c_str());
      }

      /* Read the bond coefficients from the corresponding section of the data file. */
      tokens = tokenize(lines_of_file[bond_types_line]);
      ihelp = atoi(tokens[0].c_str());
      cout << "#: --- Number of bond types: " << ihelp << " .\n";
      bond_types.resize(ihelp);
      for (int i = 0; i < ihelp; i++){
         tokens = tokenize(lines_of_file[bond_coeffs_line_start + i + 2]);
         bond_types[i].spring_coeff = atof(tokens[1].c_str());
         bond_types[i].sq_ete = atof(tokens[2].c_str());
         bond_types[i].kuhnl = atof(tokens[3].c_str());
      }

      /* Read atom sections which corresponds to nodal points: */
      for (int i = 0; i < nnodes; i++) {
         /* Read and split the line from the input file: */
         tokens = tokenize(lines_of_file[atoms_line_start + 2 + i]);


         //int pbcx = atoi(tokens[7].c_str());
         //int pbcy = atoi(tokens[8].c_str());
         //int pbcz = atoi(tokens[9].c_str());

         /* Set the attributes of the current node object. */
         tNode current_node;
         current_node.Id = atoi(tokens[0].c_str());
         current_node.Type = atoi(tokens[2].c_str());
         current_node.Pos[0] = atof(tokens[4].c_str()); //+ (double)pbcx*netw_min->domain->XBoxLen;
         current_node.Pos[1] = atof(tokens[5].c_str()); //+ (double)pbcy*netw_min->domain->YBoxLen;
         current_node.Pos[2] = atof(tokens[6].c_str()); //+ (double)pbcz*netw_min->domain->ZBoxLen;
         current_node.node_cell = 0;

         // Read the mass of the nodal point from the "Masses" section of the file.
         current_node.n_mass = node_types[current_node.Type-1].n_mass;
         current_node.mass = node_types[current_node.Type-1].mass;

         /* Pair coeffs come from the data file in Angstrom. */
         current_node.r_node = node_types[current_node.Type-1].r_node;
         current_node.r_star = current_node.r_node;

         nodes.push_back(current_node);
      }
      cout << "#: --- " << nnodes << " nodes have been added to the network structure.\n";


      int nchains = 0;

      for (int i = 0; i < nstrands; i++) {
         /* Read and split the line from the input file: */
         tokens = tokenize(lines_of_file[bonds_line_start + 2 + i]);

         /* Create a temporary object: */
         tStrand current_strand;
         current_strand.Id = atoi(tokens[0].c_str());
         current_strand.Type = atoi(tokens[1].c_str());
         current_strand.OrChain = atoi(tokens[5].c_str());
         if (current_strand.OrChain > nchains)
            nchains = current_strand.OrChain;

         if (current_strand.OrChain == 0)
            current_strand.slip_spring = true;
         else
            current_strand.slip_spring = false;

         /* Read the tags of the start and the end of the strand. */
         int snode = atoi(tokens[2].c_str());
         int enode = atoi(tokens[3].c_str());

         if (snode > (nnodes) || enode > (nnodes)){
            cout << ": --- There is a problem at line: " << (bonds_line_start + 2 + i + 1)
                 << " of the data file.\n" << lines_of_file[bonds_line_start + 2 + i] << "\n";
            exit(0);
         }

         /* We have to find the nodal point whose tag is equal to snode: */
         int found = 0;
         for (std::list <tNode>::iterator it = nodes.begin(); it != nodes.end(); ++it) {
            if (found > 1)
               break;
            if ((*it).Id == snode || (*it).Id == enode) {
               found++;
               (*it).OrChains.push_back(current_strand.OrChain);
               current_strand.pEnds.push_back(&(*it));
            }
         }

         /* Spring coeffs comes from the data file in kJ/mol/K/A^2 */
         current_strand.spring_coeff = bond_types[current_strand.
Type-1].spring_coeff;
         /* Length of the strand (n*b) comes from the data file in Angstroms. */
         current_strand.sq_end_to_end = bond_types[current_strand.
Type-1].sq_ete;
         current_strand.kuhn_length = bond_types[current_strand.Type-1].kuhnl;

```

```
          /* Set the creation time of the slip-spring: */
          if (current_strand.slip_spring)
             current_strand.tcreation = 0;

          /* and push it back to the list: */
          strands.push_back(current_strand);

          /* 2013/04/09: Also, push back a pointer to the last element of "strands" vector, if
           *             the inserted strand is a slip spring. */
          if (current_strand.slip_spring)
             pslip_springs.push_back(&(strands.back()));
       }



       for (std::list<tStrand> ::iterator it = strands.begin(); it != strands.end(); ++it) {
          /* 2013/04/09: We exclude slip strings from the local registry of strands connected to
           *             a specific node. Array "pStrands" contains only chain strands,
           *             not slip-springs!!. */
          if (!(*it).slip_spring) {
             (*it).pEnds[0]->pStrands.push_back(&(*it));
             (*it).pEnds[1]->pStrands.push_back(&(*it));
          }
       }

       for (std::list<tStrand> ::iterator it = strands.begin(); it != strands.end(); ++it) {
          if ((*it).slip_spring)
             continue;

          double dxr = (*it).pEnds[1]->Pos[0] - (*it).pEnds[0]->Pos[0];
          double dyr = (*it).pEnds[1]->Pos[1] - (*it).pEnds[0]->Pos[1];
          double dzr = (*it).pEnds[1]->Pos[2] - (*it).pEnds[0]->Pos[2];
          netw_min->domain->minimum_image(dxr, dyr, dzr);

          double dist = sqrt(dxr * dxr + dyr * dyr + dzr * dzr);
          if (dist - (*it).sq_end_to_end / (*it).kuhn_length > tol)
             cout << "Strand " << (*it).Id << " has length " << (*it).sq_end_to_end
                   / (*it).kuhn_length << " but real dist " << dist << endl;

       }

       /* Categorize strands into chains: */
       std::vector<std::list<tStrand *> > chains(nchains);
       for (std::list<tStrand> ::iterator it = strands.begin(); it != strands.end(); ++it)
          if (!(*it).slip_spring)
             chains[(*it).OrChain - 1].push_back(&(*it));


       /* Loop over all chains: */
       sorted_chains.resize(nchains);

       for (unsigned int ich = 0; ich < chains.size(); ich++) {
          /* Search to find an end of the chain: */
          std::list <tStrand *>::iterator it, jt;
          jt = find_if(chains[ich].begin(), chains[ich].end(), pred_strand_is_end);
          if (jt == chains[ich].end()) {
             cout << "Chain with no ends was found: " << ich << " .\n";
             cout << "Number of strands: " << chains[ich].size() << " .\n";
             return;
          }

          if ((*jt)->pEnds[0]->Type != 1) {
             tNode *temp;
             temp = (*jt)->pEnds[0];
             (*jt)->pEnds[0] = (*jt)->pEnds[1];
             (*jt)->pEnds[1] = temp;
          }
          sorted_chains[ich].push_back((*jt));
          if (chains[ich].size() > 1)
             chains[ich].erase(jt);
          else
             continue;

          int nelems = chains[ich].size();
          for (int i = 0; i < nelems; i++) {
             /* Set the iterator to the end of the sorted list: */
             it = sorted_chains[ich].end();
             --it;

             /* Find the next element: */
             for (jt = chains[ich].begin(); jt != chains[ich].end(); ++jt) {
                /* check whether is connected to the preceeding strand: */
                if ((*jt)->pEnds[0] == (*it)->pEnds[1]) {
                   /* We can append the found strand as is: */
                   sorted_chains[ich].push_back((*jt));
                   /* an delete it from the previous list */
                   jt = chains[ich].erase(jt);
```

```
                  break;
               }

               if ((*jt)->pEnds[1] == (*it)->pEnds[1]) {
                  /* We have to change the order of the pointers. */
                  tNode *temp;
                  temp = (*jt)->pEnds[0];
                  (*jt)->pEnds[0] = (*jt)->pEnds[1];
                  (*jt)->pEnds[1] = temp;

                  /* And add it to the list: */
                  sorted_chains[ich].push_back((*jt));
                  jt = chains[ich].erase(jt);
                  break;
               }
            }
         }
      }


      return;
   }


   bool pred_strand_is_end(tStrand * current) {
      bool outcome;
      if (current->pEnds[0]->Type == 1 || current->pEnds[1]->Type == 1)
         outcome = true;
      else
         outcome = false;

      return outcome;
   }

}
```

5.41 network.h File Reference

The header file of the Network class itself, which describes the topology of the system under consideration.

`#include "net_types.h"`

Classes

- class NetworkNS::Network

  The class which stores all information concerning the polymeric network.

Functions

- bool NetworkNS::pred_strand_is_end (tStrand ∗)

  A boolean function for judging whether a strand is a chain end.

5.41.1 Detailed Description

Author

Georgios G. Vogiatzis (gvog@chemeng.ntua.gr)

Version

2.1 (May 12, 2012)

5.41.2 LICENSE

Definition in file network.h.

5.41.3 Function Documentation

5.41.3.1 pred_strand_is_end()

```
bool NetworkNS::pred_strand_is_end (
            tStrand * current )
```

Parameters

in	*current*	A pointer to a type tStrand. The strand to check whether is located at a chain end or no.

Definition at line 315 of file network.cpp.

5.42 network.h

```

#ifndef _NETWORK_H
#define _NETWORK_H

#include "net_types.h"

namespace NetworkNS {

   class Network {
   public:

      Network(class NetwMin *, std::string);
      virtual ~Network();

      std::list <tNode> nodes;
      std::list <tStrand> strands;
      std::list <tSubCh> subchains;
      std::list <tStrand *> pslip_springs;
      std::vector <tBead_type> node_types;
      std::vector <tBond_type> bond_types;

      std::vector<std::list<tStrand *> > sorted_chains;



   };
   bool pred_strand_is_end(tStrand *);
}

#endif   /* NETWORK_H */

```

5.43 non_bonded_scheme_routines.cpp File Reference

The C++ source file containing the functions of the smoothed non-bonded free energy estimation scheme.

```
#include <iostream>
#include <cmath>
```

Functions

- void **phi_function** (double x, double xj, double Rj, double delta, double &phi_value)
- void **integral_minus_infinity_to_x** (double xasterisk, double xj, double Rj, double delta, double &integral↩_value)
- void **integral_x_to_plus_infinity** (double xasterisk, double xj, double Rj, double delta, double &integral_↩value)

5.43.1 Detailed Description

Author

Grigorios Megariotis (gmegariotis@yahoo.gr)

5.43.2 LICENSE

Definition in file non_bonded_scheme_routines.cpp.

5.44 non_bonded_scheme_routines.cpp

```
#include <iostream>
#include <cmath>

void phi_function(double x, double xj, double Rj, double delta, double &phi_value) {

   double inv_delta = 1.0 / delta;

   if (x <= xj - 0.5 * Rj - 0.5 * delta)
      phi_value = 0.0;

   else if (x > xj - 0.5 * Rj - 0.5 * delta && x <= xj - 0.5 * Rj + 0.5 * delta) {
      double temp = 2.0 * (x - xj) * inv_delta + Rj*inv_delta;
      double tempsq = temp*temp;

      phi_value = (8.0 + 15.0 * temp
                - 10.0 * tempsq * temp + 3.0 * tempsq * tempsq * temp) / 16.0 / Rj;
   } else if (x > xj - 0.5 * Rj + 0.5 * delta && x <= xj + 0.5 * Rj - 0.5 * delta)
      phi_value = 1.0 / Rj;

   else if (x > xj + 0.5 * Rj - 0.5 * delta && x <= xj + 0.5 * Rj + 0.5 * delta) {
      double temp = 2.0 * (xj - x) * inv_delta + Rj*inv_delta;
      double tempsq = temp*temp;

      phi_value = (8.0 + 15.0 * temp
                  - 10.0 * tempsq * temp + 3.0 * tempsq * tempsq * temp) / 16.0 / Rj;
   }
   else if (x > xj + 0.5 * Rj + 0.5 * delta)
      phi_value = 0.0;
}

void integral_minus_infinity_to_x(double xasterisk, double xj, double Rj, double delta,
        double &integral_value) {

   if (xasterisk <= xj - Rj / 2.0 - delta / 2.0)
      integral_value = 0.0;

   else if (xasterisk > xj - Rj / 2.0 - delta / 2.0 && xasterisk <= xj - Rj / 2.0 + delta / 2.0)
      integral_value = delta * (5.0 / 4.0 + 4.0 * (2.0 * (xasterisk - xj) / delta + Rj / delta)
           + 15.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 2)\
                  - 5.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 4)
           + 1.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 6)) / 16.0 / Rj;

   else if (xasterisk > xj - Rj / 2.0 + delta / 2.0 && xasterisk <= xj + Rj / 2.0 - delta / 2.0)
      integral_value = (xasterisk - xj) / Rj + 1.0 / 2.0;

   else if (xasterisk > xj + Rj / 2.0 - delta / 2.0 && xasterisk <= xj + Rj / 2.0 + delta / 2.0)
```

```
      integral_value = 1.0 - delta * (5.0 / 4.0 + 4.0 * (2.0 * (xj - xasterisk) / delta + Rj / delta)
                     + 15.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 2)
                     - 5.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 4)
                     + 1.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 6)) / 16.0 / Rj;

   else if (xasterisk > xj + Rj / 2.0 + delta / 2.0)
      integral_value = 1.0;

}

void integral_x_to_plus_infinity(double xasterisk, double xj, double Rj,
                                 double delta, double &integral_value) {

   if (xasterisk <= xj - Rj / 2.0 - delta / 2.0)
      integral_value = 1.0;

   else if (xasterisk > xj - Rj / 2.0 - delta / 2.0 && xasterisk <= xj - Rj / 2.0 + delta / 2.0)
      integral_value = 1.0 - delta * (5.0 / 4.0 + 4.0 * (2.0 * (xasterisk - xj) / delta + Rj / delta)
                     + 15.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 2)
                     - 5.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 4)
                     + 1.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 6)) / 16.0 / Rj;

   else if (xasterisk > xj - Rj / 2.0 + delta / 2.0 && xasterisk <= xj + Rj / 2.0 - delta / 2.0)
      integral_value = (xj - xasterisk) / Rj + 1.0 / 2.0;

   else if (xasterisk > xj + Rj / 2.0 - delta / 2.0 && xasterisk <= xj + Rj / 2.0 + delta / 2.0)
      integral_value = delta * (5.0 / 4.0 + 4.0 * (2.0 * (xj - xasterisk) / delta + Rj / delta)
                     + 15.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 2)
                     - 5.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 4)
                     + 1.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 6)) / 16.0 / Rj;

   else if (xasterisk > xj + Rj / 2.0 + delta / 2.0)
      integral_value = 0.0;

}
```

5.45 non_bonded_scheme_routines.h File Reference

The header file containing the definitions of the functions of the smoothed non-bonded free energy estimation scheme.

Functions

- void **phi_function** (double x, double xj, double Rj, double delta, double &phi_value)
- void **integral_minus_infinity_to_x** (double xasterisk, double xj, double Rj, double delta, double &integral↩_value)
- void **integral_x_to_plus_infinity** (double xasterisk, double xj, double Rj, double delta, double &integral_↩value)

5.45.1 Detailed Description

Author

Grigorios Megariotis (gmegariotis@yahoo.gr)

Definition in file non_bonded_scheme_routines.h.

5.46 non_bonded_scheme_routines.h

```

#ifndef NEW_SCHEME_ROUTINES_H
#define  NEW_SCHEME_ROUTINES_H

void phi_function(double x, double xj, double Rj, double delta, double &phi_value);
void integral_minus_infinity_to_x(double xasterisk, double xj, double Rj, double delta, double &
integral_value);
void integral_x_to_plus_infinity(double xasterisk, double xj, double Rj, double delta, double &
integral_value);
```

```

#endif   /* NEW_SCHEME_ROUTINES_H */

```

5.47 rng.cpp File Reference

Random number generator based on Marsaglia's KISS (Keep it Simple and Stupid) algorithm.

```
#include <iostream>
#include <cmath>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#include "rng.h"
```

Macros

- #define **PHI** 0x9e3779b9

5.47.1 Detailed Description

Author

Grigorios Megariotis (gmegariotis@yahoo.gr)

5.47.2 LICENSE

Definition in file rng.cpp.

5.48 rng.cpp

```

#include <iostream>
#include <cmath>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#include <cmath>
#include "rng.h"
#define PHI 0x9e3779b9


using namespace std;

namespace NetworkNS {

    unsigned int RanMars::devrand()
    {

        int fn;
        unsigned int r;
        fn = open("/dev/urandom", O_RDONLY);
        if (fn == -1)
            exit(-1); /* Failed! */
        if (read(fn, &r, 4) != 4)
            exit(-1); /* Failed! */
        close(fn);
        return r;
    }

    unsigned int RanMars::uint_rand()
    {
        unsigned long long t;
        x = 314527869 * x + 1234567;
        y ^= y << 5;
        y ^= y >> 7;
        y ^= y << 22;
        t = 4294584393ULL * z + c;
        c = t >> 32;
        z = t;
        return x + y + z;
    }



   RanMars::RanMars(int seed) {
      /* Seed variables */
        x = 123456789;
        y = 987654321;
        z = 43219876;
        c = 6543217;

        x = devrand();
        while (!(y = devrand())); /* y must not be zero! */
        z = devrand();

        /* We don't really need to set c as well but let's anyway... */
        /* NOTE: offset c by 1 to avoid z=c=0 */
        c = devrand() % 698769068 + 1; /* Should be less than 698769069 */

        // gvog: And warm up the generator:
        /* Also, discard the first values... */
        int i, nelements = 1000000;
        //unsigned int temp;
        for (i = 0; i < nelements; i++)
            uint_rand();

        return;
   }

   /* ---------------------------------------------------------------------- */

   RanMars::~RanMars() {
      delete [] u;
   }

   /* ----------------------------------------------------------------------
      uniform RN
   ---------------------------------------------------------------------- */
   double RanMars::uniform() {

        double x;
        unsigned long long a;
        a = ((unsigned long long) uint_rand() << 32) + uint_rand();
        a = (a >> 12) | 0x3FF0000000000000ULL; /* Take upper 52 bits */
        *((unsigned long long *) &x) = a; /* Make a double from bits */

        return x - 1.0;
   }

   /* ----------------------------------------------------------------------
      gaussian RN
   ---------------------------------------------------------------------- */

   double RanMars::gaussian() {
      double first, v1, v2, rsq, fac;

      if (!save) {
         int again = 1;
         while (again) {
            v1 = 2.0 * uniform() - 1.0;
            v2 = 2.0 * uniform() - 1.0;
            rsq = v1 * v1 + v2*v2;
            if (rsq < 1.0 && rsq != 0.0) again = 0;
         }
```

```
         fac = sqrt(-2.0 * log(rsq) / rsq);
         second = v1*fac;
         first = v2*fac;
         save = 1;
      } else {
         first = second;
         save = 0;
      }
      return first;
   }

   double RanMars::modified_gaussian(double mean, double stdev) {
      double first, v1, v2, rsq, fac;

      if (!save) {
         int again = 1;
         while (again) {
            v1 = 2.0 * uniform() - 1.0;
            v2 = 2.0 * uniform() - 1.0;
            rsq = v1 * v1 + v2*v2;
            if (rsq < 1.0 && rsq != 0.0) again = 0;
         }
         fac = sqrt(-2.0 * log(rsq) / rsq);
         second = mean + stdev * v1*fac;
         first = mean + stdev * v2*fac;
         //cout << "in_marsaglia=" << mean << endl;
         save = 1;
      } else {
         first = second;
         save = 0;
      }
      return first;
   }

   double RanMars::rand_gauss(void) {
      double v1, v2, s;

      do {
         v1 = 2.0 * uniform() - 1;
         v2 = 2.0 * uniform() - 1;

         s = v1 * v1 + v2*v2;
      } while (s >= 1.0);

      if (s == 0.0)
         return 0.0;
      else
         return (v1 * sqrt(-2.0 * log(s) / s));
   }


}
```

5.49 rng.h File Reference

Random number generator based on Marsaglia's KISS (Keep it Simple and Stupid) algorithm.

Classes

- class NetworkNS::RanMars

    The class of the pseudorandom number generator. It is based on Marsaglia's KISS design.

5.49.1 Detailed Description

Author

Grigorios G. Megariotis (gmegariotis@yahoo.gr)

Version

1.0

5.49.2 LICENSE

Definition in file rng.h.

5.50 rng.h

```

#ifndef MARSAGLIA_H
#define  MARSAGLIA_H

namespace NetworkNS {

   class RanMars {
   public:

      RanMars(int);

      ~RanMars();

      double uniform();

      double gaussian();

      double modified_gaussian(double mean, double stdev);

      double rand_gauss(void);

      unsigned int devrand();

      unsigned int uint_rand();

   private:
      int seed;
      int save;
      double second;
      double *u;
      int i97, j97;

      unsigned int x;
      unsigned int y;
      unsigned int z;
      unsigned int c;

   };

}


#endif   /* MARSAGLIA_H */

```

Bibliography

[1] Vogiatzis, G. G.; Theodorou, D. N. *Macromolecules* **2013**, *46*, 4670–4683. 1

[2] Chappa, V. C.; Morse, D. C.; Zippelius, A.; Müller, M. *Phys. Rev. Lett.* **2012**, *109*, 148302. 1, 2

[3] van Gunsteren, W.; Berendsen, H. *Mol. Phys.* **1982**, *45*, 637–647. 4

[4] Ermak, D. L.; McCammon, J. A. *J. Chem. Phys.* **1978**, *69*, 1352–1360. 4

[5] Dickinson, E.; Allison, S. A.; McCammon, J. A. *J. Chem. Soc., Faraday Trans. 2* **1985**, *81*, 591–601. 4

[6] Terzis, A. F.; Theodorou, D. N.; Stroeks, A. *Macromolecules* **2002**, *35*, 508–521. 4, 32

[7] Li, Y.; Kröger, M.; Liu, W. K. *Polymer* **2011**, *52*, 5867 – 5878. 6, 33, 60

[8] Mavrantzas, V. G.; Theodorou, D. N. *Macromolecules* **1998**, *31*, 6310–6332. 6

[9] Rouse, P. E. *J. Chem. Phys.* **1953**, *21*, 1272–1280. 8

[10] Kojima, M.; Tosaka, M.; Funami, E.; Nitta, K.; Ohshima, M.; Kohjiya, S. *J. Supercrit. Fluids* **2005**, *35*, 175 – 181. 8

Index