
EMSIPON

an Engine for Mesoscopic Simulations of Polymers

Contents

1	Introduction	1
1.1	Model	1
1.2	Interactions	1
1.2.1	Intramolecular Interactions	1
1.2.2	Slip-spring Interactions	2
1.2.3	Polymer Non-bonded Interactions	2
1.3	Coarse-Grained Dynamics	3
1.3.1	Brownian Dynamics	4
1.3.2	Rotational Brownian Dynamics	4
1.3.3	Kinetic Monte Carlo Hopping	4
1.4	Stress Tensor Calculation	6
1.4.1	Bonded contributions to the stress tensor	6
1.4.2	Polymer Non-bonded interaction contributions to the stress tensor	7
1.4.3	Polymer-particle and Particle-particle interaction contributions to the stress tensor	7
1.4.4	Estimation of Rheological Properties	8
1.5	Benchmark Simulations	8
1.5.1	Rouse Dynamics	8
1.5.2	Entangled Dynamics	9
2	Class Index	13
2.1	Class List	13
3	File Index	15
3.1	File List	15
4	Class Documentation	17
4.1	NetworkNS::cb3D_integrator Class Reference	17
4.1.1	Detailed Description	18
4.1.2	Constructor & Destructor Documentation	18
4.1.2.1	cb3D_integrator()	18
4.1.3	Member Function Documentation	19
4.1.3.1	integrate()	19

4.1.3.2	report()	20
4.1.3.3	bonded_force_calculation()	20
4.1.3.4	simpler_scheme_non_bonded_force_calculation()	21
4.1.3.5	calculate_pressure()	22
4.1.3.6	from_bd_x_to_polymer_network()	23
4.1.4	Member Data Documentation	23
4.1.4.1	bd_gamma	23
4.1.4.2	bd_mass	23
4.1.4.3	bd_x	23
4.1.4.4	bd_f	23
4.1.4.5	bd_nb_f	23
4.1.4.6	bd_x_ps	24
4.1.4.7	bd_f_ps	24
4.1.4.8	xshift	24
4.1.4.9	yshift	24
4.1.4.10	zshift	24
4.1.4.11	gamma_mass_opt	24
4.1.4.12	den_dx	25
4.1.4.13	den_dy	25
4.1.4.14	den_dz	25
4.2	D3DVector< T > Class Template Reference	25
4.2.1	Detailed Description	26
4.2.2	Member Function Documentation	26
4.2.2.1	operator*()	26
4.3	NetworkNS::Domain Class Reference	27
4.3.1	Detailed Description	28
4.3.2	Constructor & Destructor Documentation	28
4.3.2.1	Domain()	28
4.3.3	Member Data Documentation	28
4.3.3.1	BoxExists	28
4.3.3.2	NonPeriodic	28
4.3.3.3	Boundary	28
4.4	NetworkNS::dump Class Reference	29
4.4.1	Detailed Description	29
4.4.2	Member Function Documentation	29
4.4.2.1	add_snapshot_to_dump()	29
4.5	NetworkNS::Grid Class Reference	30
4.5.1	Detailed Description	30
4.5.2	Member Data Documentation	30
4.5.2.1	vcell	31

4.6	NetworkNS::Hopping Class Reference	31
4.6.1	Detailed Description	31
4.6.2	Constructor & Destructor Documentation	31
4.6.2.1	Hopping()	31
4.6.2.2	~Hopping()	32
4.6.3	Member Function Documentation	32
4.6.3.1	hopping_step()	32
4.7	NetworkNS::NetwMin Class Reference	33
4.7.1	Detailed Description	33
4.7.2	Constructor & Destructor Documentation	33
4.7.2.1	NetwMin()	34
4.7.3	Member Function Documentation	34
4.7.3.1	write_network_to_lammps_data_file()	34
4.7.4	Member Data Documentation	34
4.7.4.1	network	34
4.7.4.2	domain	34
4.7.4.3	grid	34
4.7.4.4	my_traj_file	35
4.8	NetworkNS::Network Class Reference	35
4.8.1	Detailed Description	35
4.8.2	Constructor & Destructor Documentation	35
4.8.2.1	Network()	35
4.8.3	Member Data Documentation	36
4.8.3.1	nodes	36
4.8.3.2	strands	36
4.8.3.3	subchains	36
4.8.3.4	pslip_springs	36
4.8.3.5	node_types	36
4.8.3.6	bond_types	37
4.8.3.7	sorted_chains	37
4.9	NetworkNS::RanMars Class Reference	37
4.9.1	Detailed Description	38
4.9.2	Member Function Documentation	38
4.9.2.1	uniform()	38
4.9.2.2	gaussian()	38
4.9.2.3	modified_gaussian()	38
4.10	sBead_type Struct Reference	38
4.10.1	Detailed Description	39
4.11	sBond_type Struct Reference	39
4.11.1	Detailed Description	39

4.11.2	Member Data Documentation	39
4.11.2.1	spring_coeff	39
4.12	NetworkNS::sgrid_cell Struct Reference	39
4.12.1	Detailed Description	40
4.13	sNode Struct Reference	40
4.13.1	Detailed Description	40
4.13.2	Member Data Documentation	41
4.13.2.1	Type	41
4.13.2.2	node_cell	41
4.13.2.3	r_node	41
4.13.2.4	r_star	41
4.14	sStrand Struct Reference	41
4.14.1	Detailed Description	42
4.14.2	Member Data Documentation	42
4.14.2.1	Type	42
4.14.2.2	spring_coeff	42
4.14.2.3	sq_end_to_end	43
5	File Documentation	45
5.1	Auxiliary.cpp File Reference	45
5.1.1	Detailed Description	45
5.1.2	LICENSE	45
5.1.3	Function Documentation	45
5.1.3.1	tokenize()	46
5.2	Auxiliary.cpp	46
5.3	Auxiliary.h File Reference	46
5.3.1	Detailed Description	47
5.3.2	LICENSE	47
5.3.3	Function Documentation	47
5.3.3.1	tokenize()	47
5.4	Auxiliary.h	47
5.5	b3D_integrator.cpp File Reference	49
5.5.1	Detailed Description	49
5.5.2	LICENSE	50
5.6	b3D_integrator.cpp	50
5.7	b3D_integrator.h File Reference	57
5.7.1	Detailed Description	57
5.7.2	LICENSE	57
5.8	b3D_integrator.h	57
5.9	constants.h File Reference	59

5.9.1	Detailed Description	59
5.9.2	LICENSE	59
5.9.3	Variable Documentation	59
5.9.3.1	avogadro_constant	60
5.9.3.2	kg_to_amus	60
5.9.3.3	amu_to_kg	60
5.9.3.4	pi_tube_diameter	60
5.9.3.5	c1	60
5.9.3.6	c2	60
5.10	constants.h	60
5.11	distributions.cpp File Reference	61
5.11.1	Detailed Description	61
5.11.2	LICENSE	61
5.11.3	Function Documentation	61
5.11.3.1	f_gaussian()	62
5.11.3.2	e_gaussian()	62
5.12	distributions.cpp	62
5.13	distributions.h File Reference	63
5.13.1	Detailed Description	63
5.13.2	LICENSE	63
5.13.3	Function Documentation	63
5.13.3.1	f_gaussian()	64
5.13.3.2	e_gaussian()	64
5.14	distributions.h	64
5.15	domain.cpp File Reference	65
5.15.1	Detailed Description	65
5.15.2	LICENSE	65
5.16	domain.cpp	65
5.17	domain.h File Reference	67
5.17.1	Detailed Description	67
5.17.2	LICENSE	67
5.18	domain.h	67
5.19	dump.cpp File Reference	68
5.19.1	Detailed Description	68
5.19.2	LICENSE	68
5.20	dump.cpp	68
5.21	dump.h File Reference	69
5.21.1	Detailed Description	69
5.21.2	LICENSE	70
5.22	dump.h	70

5.23	grid.cpp File Reference	70
5.23.1	Detailed Description	70
5.23.2	LICENSE	71
5.24	grid.cpp	71
5.25	grid.h File Reference	73
5.25.1	Detailed Description	73
5.25.2	LICENSE	73
5.25.3	Typedef Documentation	74
5.25.3.1	grid_cell	74
5.26	grid.h	74
5.27	hopping.cpp File Reference	74
5.27.1	Detailed Description	75
5.28	hopping.cpp	75
5.29	hopping.h File Reference	82
5.29.1	Detailed Description	82
5.30	hopping.h	83
5.31	main.cpp File Reference	83
5.31.1	Detailed Description	84
5.31.2	LICENSE	84
5.31.3	Function Documentation	84
5.31.3.1	main()	84
5.32	main.cpp	84
5.33	net_types.h File Reference	85
5.33.1	Detailed Description	86
5.33.2	LICENSE	86
5.34	net_types.h	86
5.35	netmin.cpp File Reference	87
5.35.1	Detailed Description	87
5.35.2	LICENSE	88
5.36	netmin.cpp	88
5.37	netmin.h File Reference	89
5.37.1	Detailed Description	90
5.37.2	LICENSE	90
5.38	netmin.h	90
5.39	network.cpp File Reference	90
5.39.1	Detailed Description	91
5.39.2	LICENSE	91
5.39.3	Function Documentation	91
5.39.3.1	pred_strand_is_end()	91
5.40	network.cpp	91

5.41	network.h File Reference	95
5.41.1	Detailed Description	95
5.41.2	LICENSE	96
5.41.3	Function Documentation	96
5.41.3.1	pred_strand_is_end()	96
5.42	network.h	96
5.43	non_bonded_scheme_routines.cpp File Reference	96
5.43.1	Detailed Description	97
5.43.2	LICENSE	97
5.44	non_bonded_scheme_routines.cpp	97
5.45	non_bonded_scheme_routines.h File Reference	98
5.45.1	Detailed Description	98
5.46	non_bonded_scheme_routines.h	98
5.47	rng.cpp File Reference	99
5.47.1	Detailed Description	99
5.47.2	LICENSE	99
5.48	rng.cpp	99
5.49	rng.h File Reference	101
5.49.1	Detailed Description	101
5.49.2	LICENSE	102
5.50	rng.h	102
	Bibliography	103
	Index	105

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 $n_{\text{Kuhns/beat}}$ 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

$$A = A_b + A_{ss} + A_{nb} + A_{ps} + A_{pp}$$

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

$$A_b(r_{ij}^2) = \epsilon_b T \frac{r_{ij}^2}{\sigma_b^2} = \frac{3}{2} k_B T \frac{\mathbf{R}_{ij} \cdot \mathbf{R}_{ij}}{n_{\text{Kuhns/beat}} b^2}$$

with the parameters ϵ_b and σ_b read from the data file. In our approach $\epsilon_b = 0.012471$ kJ/mol/K and $\sigma_b^2 = 810 \text{ \AA}^2$ 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

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

with the parameters ϵ_{ss} measured in kJ/mol/K/Å² and σ_{ss} in Å². Chappa et al. [2], give the following values to the above coefficients. The distance term can be connected to our harmonic springs as

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

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

$$\begin{aligned} \epsilon_{ss} T \sigma_{ss}^2 &= \frac{k_{ss} r_{ss}^2}{2} = 17.28 k_B T \\ \epsilon_{ss} &= \frac{17.28 k_B}{\sigma_{ss}^2} = 3.0794 \times 10^{-5} \frac{\text{kJ}}{\text{mol K Å}^2} \end{aligned}$$

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

$$\epsilon_{ss} T \sigma_{ss}^2 = \frac{3}{2} k_B T$$

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:

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

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

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

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

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

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 $\mathbf{r}_j = (x_j, y_j, z_j)$. 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 $x_{\text{cell}}, y_{\text{cell}}$, and z_{cell} will be integer multiples of L_x, L_y and L_z , respectively.

In the following we will assume that

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

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:

$$\begin{aligned} x_{\text{cell}} - L_x &< x_j + \frac{R_j}{2} < x_{\text{cell}} + R_j \\ y_{\text{cell}} - L_y &< y_j + \frac{R_j}{2} < y_{\text{cell}} + R_j \\ z_{\text{cell}} - L_z &< z_j + \frac{R_j}{2} < z_{\text{cell}} + R_j \end{aligned}$$

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 occurring in parallel: (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] :

$$R_j(t_n + \Delta t) = R_j(t_n) + \frac{\Delta t}{k_B T} D_t F_j(t_n) + \frac{1}{2} \frac{\Delta t^2}{k_B T} D_t \dot{F}_j(t_n) + X_j^t(\Delta t)$$

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:

$$\langle (X_j^t)^2 \rangle = 2D_t \Delta t = 2 \frac{k_B T}{m_{\text{bead}} \gamma} \Delta t$$

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 $\zeta_0(500 \text{ K}) = m_{\text{monomer}} \gamma = 2.63 \times 10^{-12} \text{ kg/s}$, $\gamma(500 \text{ K}) = 2.325 \times 10^{13} \text{ s}^{-1}$ and thus $\gamma \Delta t = 23.25$ for $\Delta t = 10^{-12} \text{ s}$.

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:

$$\phi_j(t_n + \Delta t) = \phi_j(t_n) + \frac{\Delta t}{k_B T} D_r T_j(t_n) + \frac{1}{2} \frac{\Delta t^2}{k_B T} D_r \dot{T}_j + X_j^r(t_n)$$

where now D_r stands for the rotational diffusion coefficient of the particles, measured in $\text{rad}^2 \text{ s}^{-1}$, 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:

$$\langle (X_j^r)^2 \rangle = 2D_r \Delta t$$

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

$$D_r = \frac{k_B T}{f_r}$$

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

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

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:

$$D_{\text{Rouse}} = \frac{k_B T}{N \zeta}$$

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 $(n_{\text{Kuhns/bead}} b^2)^{1/2}$ in each direction by (see below for the definition of v_{diff})

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

Hence, one must have:

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

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:

$$k_{\text{hopping}} = v_0 \exp\left(-\frac{A_{1 \rightarrow 2}^\ddagger - A_{a_0-b_0}}{k_B T}\right) = v_{\text{diff}} \exp\left(-\frac{A_{a_0-b_0}}{k_B T}\right)$$

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, $A_{a_0-b_0}$, while the dependence on the height of the free energy barrier (i.e. $A_{1 \rightarrow 2}^\ddagger$) has been absorbed into the pre-exponential factor, v_{diff} .

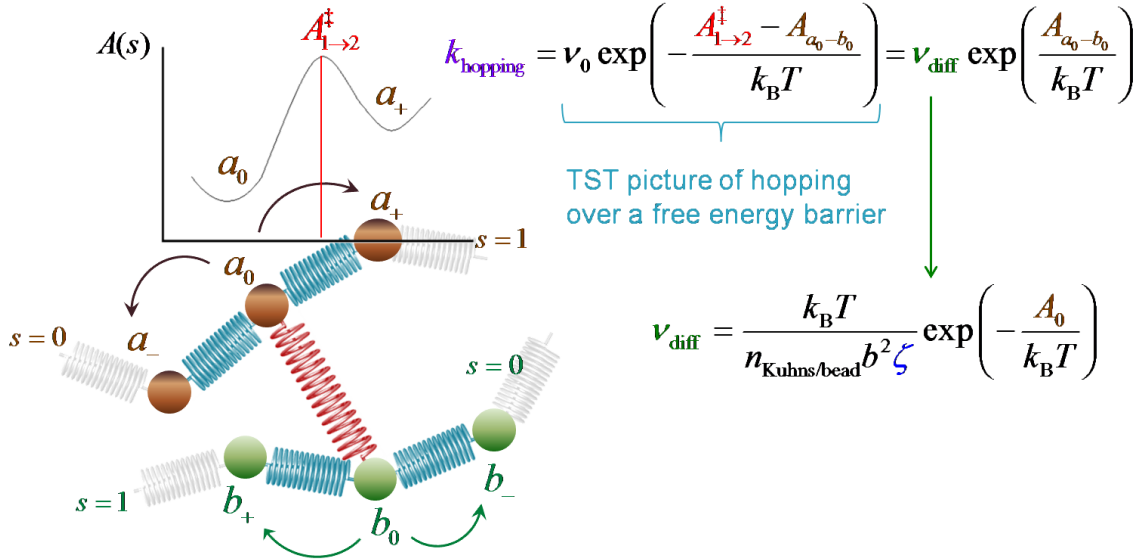


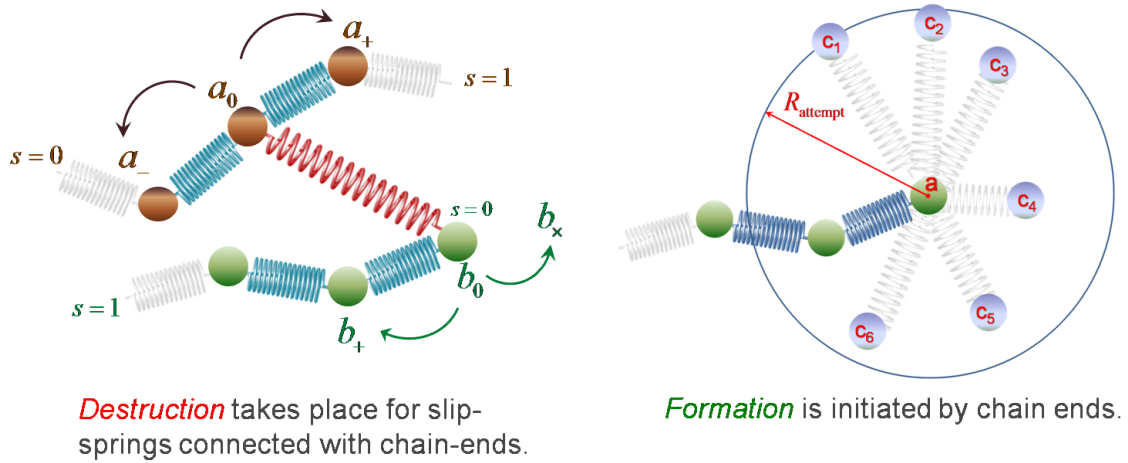
Figure 1.1: Slip-spring hopping schematic

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 $(n_{\text{Kuhns/bead}} b^2)^{1/2}$, corresponding to the last bead of the chain) with rate v_{diff} and there faces a transition which can take place with overall rate:

$$k_{\text{destruction}} = v_{\text{diff}} \exp\left(-\frac{A_{a_0-b_0}}{k_B T}\right)$$

where $A_{a_0-b_0}$ 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.



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

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] :

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

where F denotes the deformation gradient tensor and may be considered as a mapping of an infinitesimal vector $d\mathbf{x}$ of the initial configuration onto the infinitesimal vector $d\mathbf{x}$ 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 calculated 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:

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

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 interactions 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:

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

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

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

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:

$$\frac{\partial \rho_{cell}^{(1)}}{\partial \mathbf{F}} = \rho'_{cell} = \rho'_{cell} \det(\mathbf{F}) (\mathbf{F}^{-1})^T$$

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

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

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

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

and

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

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 contributions to the stress tensor

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

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

where $N_{particles}$ 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:

$$\eta_0 = \frac{V}{k_B T} \int_0^{+\infty} \langle \sigma_{\alpha\beta}(t) \sigma_{\alpha\beta}(0) \rangle dt$$

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

$$G^*(\omega) = G'(\omega) + iG''(\omega) = i\omega \frac{V}{k_B T} \int_0^{+\infty} e^{-i\omega t} \langle \sigma_{\alpha\beta}(t) \sigma_{\alpha\beta}(0) \rangle dt$$

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

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

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 $m_{\text{bead}} = 1460 \text{ g/mol}$. The simulation box size was fixed at a volume of $(R_{e,0})^3$ with $R_{e,0}$ 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, $b = 0.9 \text{ nm}$, 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:

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

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

$$D_{\text{Rouse}} = \frac{k_B T}{N \zeta_{\text{bead}}} = 1.56 \times 10^{-12} \frac{\text{m}^2}{\text{s}}$$

where our bead's friction coefficient, ζ_{bead} is $5.52 \times 10^{-11} \text{ kg/s}$. The introduction of nonbonded interactions does not seem to affect the scaling laws of the unentangled melt.

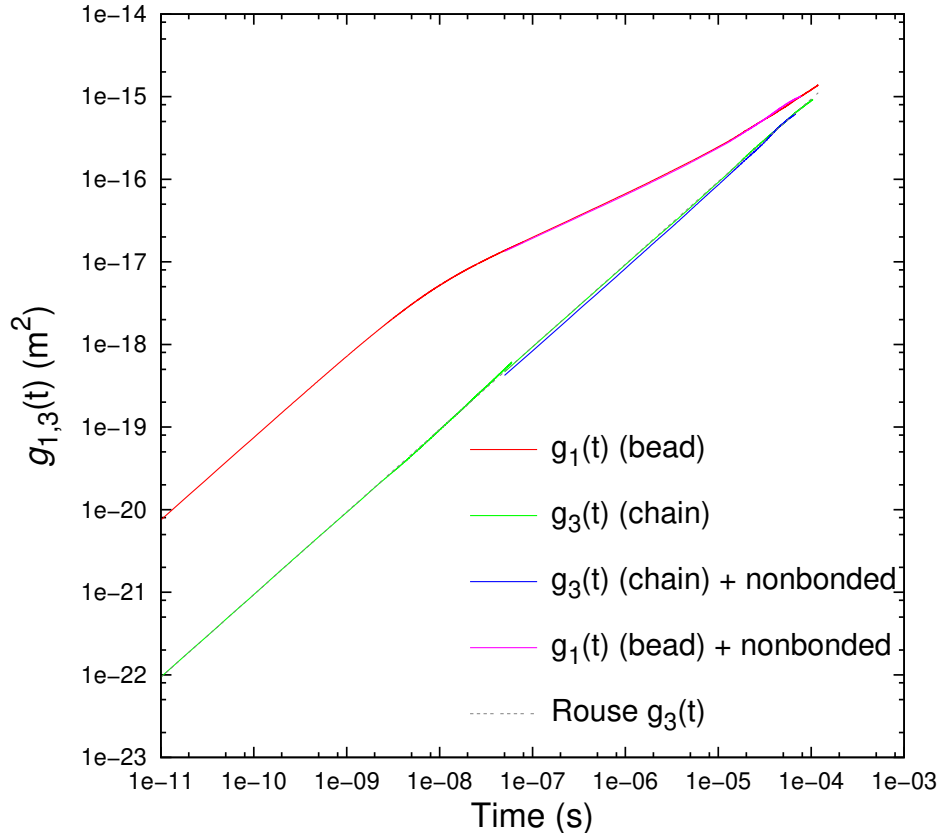


Figure 1.3: Time evolution of the mean-squared displacement of beads and center of mass of the chains for an unentangled PI melt.

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 calculated for the Rouse model in free space. Hence, $g_1(t)$ can be approximated as:

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

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} :

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

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:

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

where the characteristic times are:

$$\tau_R = \frac{\zeta_{\text{bead}} N^2 (n_{\text{Kuhns/bead}} b^2)}{3\pi^2 k_B T}$$

and:

$$\tau_d = \frac{\zeta_{\text{bead}} N^3 (n_{\text{Kuhns/bead}} b^2)^2}{\pi^2 k_B T \alpha_{\text{pp}}^2}$$

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

$$g_1(t) \simeq \frac{k_B T \alpha_{\text{pp}}^2}{N^2 \zeta_{\text{bead}} (n_{\text{Kuhns/bead}} b^2)} t$$

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.

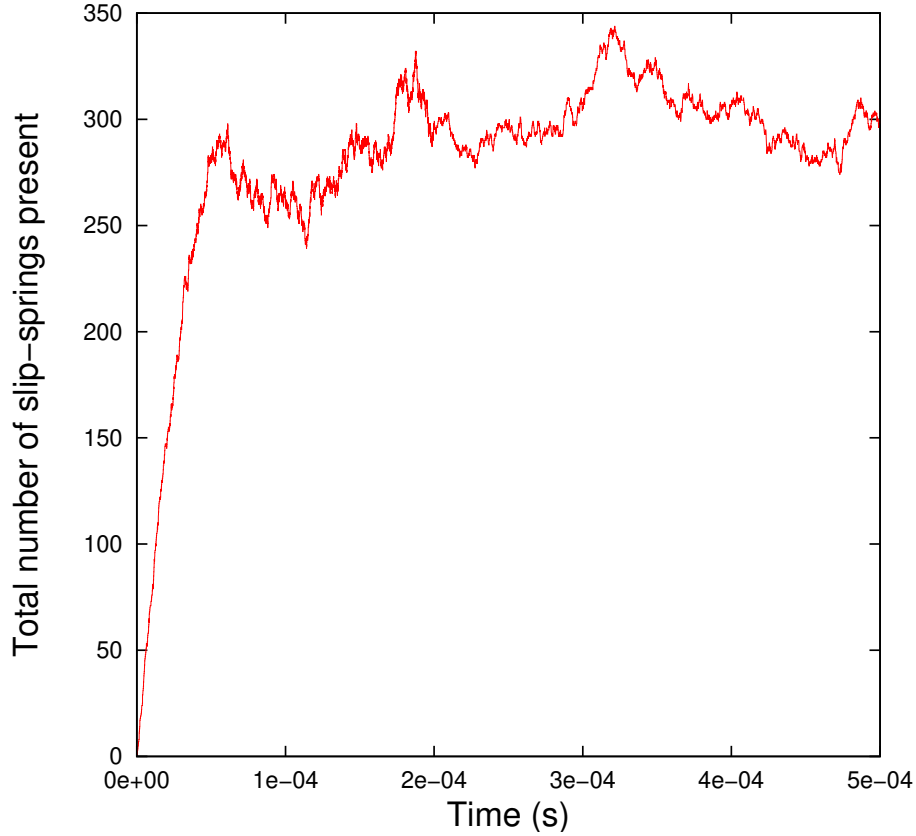


Figure 1.4: Number of slip-springs present in the system as a function of time.

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-springs 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.

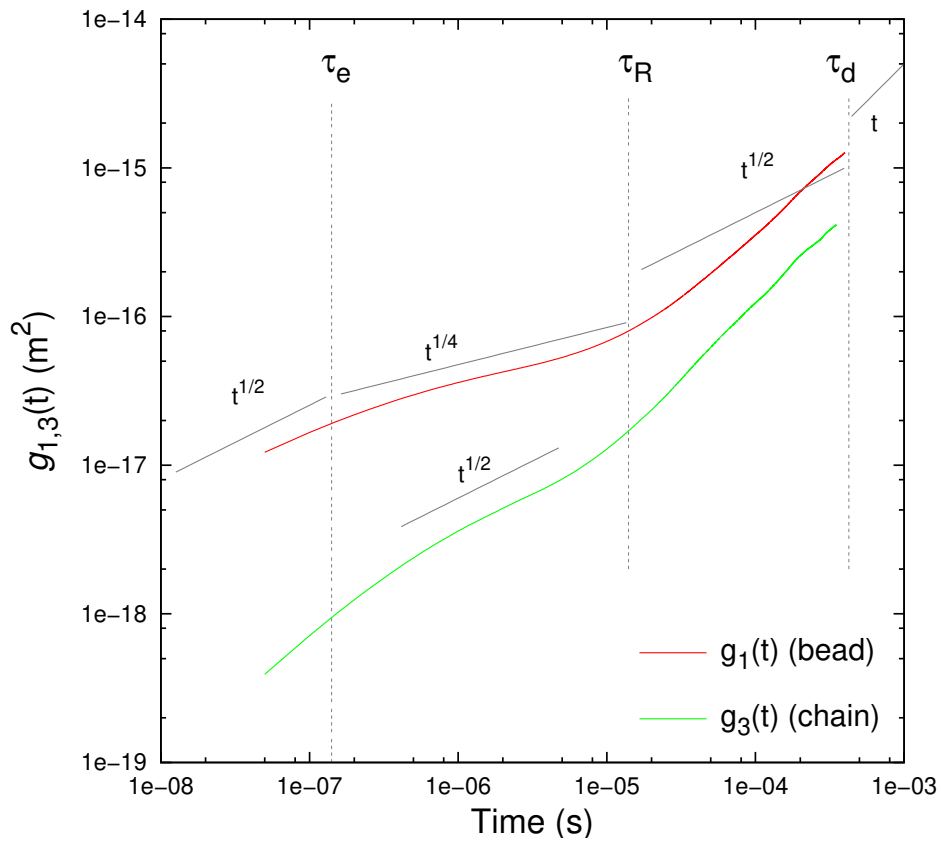


Figure 1.5: Time evolution of the mean-squared displacement of beads and center of mass of the chains for an entangled PI melt.

Chapter 2

Class Index

2.1 Class List

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

NetworkNS::cb3D_integrator	
The class of the Brownian Dynamics integrator	17
D3DVector< T >	
A class describing a vector with three components of type T	25
NetworkNS::Domain	
The class of the simulation domain	27
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	29
NetworkNS::Grid	
The nonbonded free energy estimation grid class	30
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	31
NetworkNS::NetwMin	
The class of the host application itself. It contains pointers to all constituents, e.g. the simulation domain, random number generator, etc	33
NetworkNS::Network	
The class which stores all information concerning the polymeric network	35
NetworkNS::RanMars	
The class of the pseudorandom number generator. It is based on Marsaglia's KISS design	37
sBead_type	
An elementary data type for reading in the information concerning a bead	38
sBond_type	
An elementary data type for reading in the information concerning a strand	39
NetworkNS::sgrid_cell	
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	40
sStrand	
The sStrand is the basic abstract data type keeping all relevant information concerning a strand of a chain	41

Chapter 3

File Index

3.1 File List

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

Auxiliary.cpp	The C++ source code file containing some auxiliary functions	45
Auxiliary.h	A C++ header file containing auxiliary type definitions and functions	46
b3D_integrator.cpp	C++ source file implementing the Brownian Dynamics simulation of PI	49
b3D_integrator.h	Header file accompanying the "b3D_integrator.cpp" C++ source file	57
constants.h	Header file containing the definitions of physical constants	59
distributions.cpp	C++ source file containing the implementation of bonded interactions	61
distributions.h	Header file accompanying the "distributions.cpp" C++ source file	63
domain.cpp	C++ source file containing the necessary functions for the manipulation of the simulation domain	65
domain.h	Header file containing the definitions of Domain class	67
dump.cpp	This file contains all routines necessary to write a trajectory file of the simulation. It uses a pretty standard LAMMPS trajectory format	68
dump.h	The header file for trajectory file keeping	69
grid.cpp	The source file containing the routines of the grid class used for the estimation of non-bonded interactions	70
grid.h	The class of the non-bonded energy estimation grid	73
hopping.cpp	This file contains the necessary routine for carrying out the slip-spring kinetic MC simulation	74
hopping.h	The header file of the slip-spring hopping kinetic Monte Carlo scheme	82
main.cpp	The main C++ source file driving the execution of the code	83
Makefile	??
net_types.h	The header file containing elementary data types of bead and strand	85

netmin.cpp	The routines of the "NetwMin" application class are defined here	87
netmin.h	The class of the network application itself	89
network.cpp	The C++ source file containing all functions relevant to the class Network	90
network.h	The header file of the Network class itself, which describes the topology of the system under consideration	95
non_bonded_scheme_routines.cpp	The C++ source file containing the functions of the smoothed non-bonded free energy estimation scheme	96
non_bonded_scheme_routines.h	The header file containing the definitions of the functions of the smoothed non-bonded free energy estimation scheme	98
rng.cpp	Random number generator based on Marsaglia's KISS (Keep it Simple and Stupid) algorithm .	99
rng.h	Random number generator based on Marsaglia's KISS (Keep it Simple and Stupid) algorithm .	101

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 initial 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	<i>*init_net</i>	A pointer to the application itself, From this pointer data concerning the network and the domain can be retrieved.
in	<i>temperature</i>	The temperature at which 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_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:

$$\log \zeta_0(T) = \log \zeta_\infty + \frac{C_1^g C_2^g}{T - T_g + C_2^g}$$

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

$$\zeta_0(T) = \gamma(T) \cdot m_{\text{monomer}}$$

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	<i>nsteps</i>	The number of integration steps to be carried out.
in	<i>dt</i>	Integration timestep in ps.
in	<i>nstout</i>	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 $\zeta_0(500 \text{ K}) = 2.63 \times 10^{-12}$ 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:

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

The coefficient $\Delta t / (\gamma(T)m_{\text{bead}})$ is measured in s²/(g/mol). The forces, are measured in kJ/(molÅ). 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:

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

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

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

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	<i>istep</i>	The current step of the Brownian Dynamics integration.
in	<i>b_energy</i>	The bonded energy of the system at the current timestep.
in	<i>nb_energy</i>	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	<i>stress_calc</i>	Boolean variable controlling whether stress calculation will take place.
----	--------------------	--

The stress tensor of the atom i , $\sigma_{i,a,b}$ 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:

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

where $N_b(i)$ 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:

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

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

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

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

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

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 $\mathbf{r}_j = (x_j, y_j, z_j)$. 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

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

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

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

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

$$\begin{aligned} x_{\text{cell}} - L_x &< x_j + \frac{R_j}{2} < x_{\text{cell}} + R_j \\ y_{\text{cell}} - L_y &< y_j + \frac{R_j}{2} < y_{\text{cell}} + R_j \\ z_{\text{cell}} - L_z &< z_j + \frac{R_j}{2} < z_{\text{cell}} + R_j \end{aligned}$$

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

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

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:

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

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, $xshift \in [0, L_x)$ 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, $yshift \in [0, L_y)$ 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, $zshift \in [0, L_z)$ 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, $\nabla_{\mathbf{r}_j} \rho_{\text{cell}}$.

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, $\nabla_{\mathbf{r}_j} \rho_{\text{cell}}$.

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, $\nabla_{\mathbf{r}_j} \rho_{\text{cell}}$.

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 subtraction 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 calculating 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	<i>netw_app</i>	A pointer to a network application, which contains all relevant information concerning the topology of the system under investigation.
in	<i>b3D</i>	A pointer to a Brownian Dynamics integrator containing the current positions of the bead in the course of the simulation.
in	<i>timestep</i>	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 nonbonded 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 [idx](#)
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	<i>netapp</i>	A pointer to the original application.
in	<i>b3D</i>	A pointer to a Brownian Dynamics simulation scheme, in order to extract the current positions of the beads.
in	<i>pos_array</i>	The array containing the positions of the beads.
in	<i>temperature</i>	The temperature at which the rates will be calculated.
in	<i>elapsed_time</i>	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:

$$D_{\text{Rouse}} = \frac{k_B T}{N \zeta}$$

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 $(n_{\text{Kuhns/bead}} b^2)^{1/2}$ in each direction by (see below for the definition of v_{diff})

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

Hence, one must have:

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

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	<i>filename</i>	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	<i>netw_min</i>	A pointer to the application which initializes the constructor. The application is a class NetwMin .
----	-----------------	--

Parameters

in	<i>filename</i>	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 description 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 uniformly 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 describing 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. $\langle R_e^2 \rangle$.

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

Copyright (c) 2014 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 [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     string buf;
00025     stringstream ss(input_string);
00026     std::vector<string> tokens;
00027
00028     while (ss >> buf)
00029         tokens.push_back(buf);
00030
00031     return tokens;
00032 }
00033
00034 void create_dir(string name_of_dir){
00035     struct stat st;
00036     if (stat(name_of_dir.c_str(), &st) != 0)
00037         mkdir(name_of_dir.c_str(), 0750);
00038 }
00039
00040
00041 double get_wall_time(){
00042     struct timeval time;
00043     if (gettimeofday(&time, NULL)){
00044         // Handle error
00045         return 0;
00046     }
00047     return (double)time.tv_sec + (double)time.tv_usec * .000001;
00048 }
00049
00050 double get_cpu_time(){
00051     return (double)clock() / CLOCKS_PER_SEC;
00052 }
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

Copyright (c) 2014 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 [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

```
00001
00015 #ifndef AUXILIARY_H
00016 #define AUXILIARY_H
00017
00018 #include<string>
00019 #include<sys/stat.h>
00020 #include<sstream>
00021 #include<string>
00022 #include<vector>
00023
00024
00025 using namespace std;
00026
00027 std::vector<string> tokenize(std::string input_string);
00028 double get_wall_time();
```

```

00030 double get_cpu_time();
00031
00032 template <typename T>
00033 T StringToNumber ( const string &Text )//Text not by const reference so that the function can be used with
00034 a                                     //character array as argument
00035 {
00036     stringstream ss(Text);
00037     T result;
00038     return ss >> result ? result : 0;
00039 }
00040 template <typename T>
00041 string NumberToString ( T Number )
00042 {
00043     stringstream ss;
00044     ss << Number;
00045     return ss.str();
00046 }
00047
00048 void create_dir(string name_of_dir);
00049
00050
00051
00052
00053 static inline double powint(const double &x, const int n) {
00054     double yy, ww;
00055
00056     if (x == 0.0) return 0.0;
00057     int nn = (n > 0) ? n : -n;
00058     ww = x;
00059
00060     for (yy = 1.0; nn != 0; nn >= 1, ww *= ww)
00061         if (nn & 1) yy *= ww;
00062
00063     return (n > 0) ? yy : 1.0 / yy;
00064 }
00065
00066
00067
00068
00069
00070
00071 template< class T >
00072 class D3DVector {
00073
00074 public :
00075     T x;
00076     T y;
00077     T z ;
00078
00079     D3DVector( T a , T b , T c ) {
00080         x = a ;
00081         y = b ;
00082         z = c ;
00083     }
00084
00085     D3DVector(){
00086         x = 0.0;
00087         y = 0.0;
00088         z = 0.0;
00089     }
00090
00091     void zero () {
00092         x = (T)0.0;
00093         y = (T)0.0;
00094         z = (T)0.0;
00095     }
00096
00097     D3DVector operator=(const D3DVector & rhs){
00098         return (D3DVector(rhs.x, rhs.y, rhs.z));
00099     }
00100
00101     D3DVector operator-(const D3DVector & rhs){
00102         return (D3DVector(x-rhs.x, y-rhs.y, z-rhs.z));
00103     }
00104
00105     D3DVector operator+(const D3DVector & rhs){
00106         return (D3DVector(x+rhs.x, y+rhs.y, z+rhs.z));
00107     }
00108
00109     D3DVector operator!(void){
00110         T inorm = 1.0 / sqrt(x*x + y*y + z*z);
00111         T nx = inorm * x;
00112         T ny = inorm * y;
00113         T nz = inorm * z;
00114         return (D3DVector(nx,ny,nz));
00115     }
00116
00117     T norm( ){
00118         return (sqrt(x*x + y*y + z*z));
00119     }

```

```

00124     }
00125
00126     T dotproduct (const D3DVector & rhs ) {
00127         T scalar = x * rhs.x + y * rhs.y + z * rhs.z ;
00128         return scalar ;
00129     }
00130
00131
00132     /* gvog: DOT PRODUCT operator is defined: */
00133     T operator*(const D3DVector & rhs){
00134         T scalar = x * rhs.x + y * rhs.y + z * rhs.z ;
00135         return scalar ;
00136     }
00137
00138     /* gvog: SCALAR PRODUCT operator is defined: */
00139     D3DVector operator*(T mult){
00140         return (D3DVector(x*mult, y*mult, z*mult));
00141     }
00142
00143     /* gvog: SCALAR DIVISION operator is defined: */
00144     D3DVector operator/(T mult){
00145         return (D3DVector(x/mult, y/mult, z/mult));
00146     }
00147
00148
00149     D3DVector crossproduct ( const D3DVector & rhs ) {
00150         T a = y * rhs.z - z * rhs.y ;
00151         T b = z * rhs.x - x * rhs.z ;
00152         T c = x * rhs.y - y * rhs.x ;
00153         D3DVector product ( a , b , c ) ;
00154         return product ;
00155     }
00156
00157     D3DVector operator^ (const D3DVector & rhs){
00158         return crossproduct(rhs);
00159     }
00160
00161     D3DVector triplevec( D3DVector & a , D3DVector & b ) {
00162         return crossproduct ( a.crossproduct( b ) ) ;
00163     }
00164
00165     T triplescal( D3DVector & a, D3DVector & b ) {
00166         return dotproduct ( a.crossproduct( b ) ) ;
00167     }
00168
00169 } ;
00170
00171 #endif /* AUXILIARY_H */
00172

```

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

Copyright (c) 2013 Grigorios Megariotis and 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 [b3D_integrator.cpp](#).

5.6 b3D_integrator.cpp

```

00001
00017 #include <time.h>
00018
00019 #include "b3D_integrator.h"
00020 #include "constants.h"
00021 #include "distributions.h"
00022 #include "hopping.h"
00023 #include "netmin.h"
00024
00025
00026
00027 using namespace std;
00028
00029 namespace NetworkNS{
00030
00031     cb3D_integrator::~cb3D_integrator() {
00032         free(bd_x);
00033         free(bd_x_ps);
00034         free(bd_mass);
00035         free(bd_f);
00036         free(bd_nb_f);
00037         free(bd_gamma);
00038
00039         return;
00040     }
00041
00047     cb3D_integrator::cb3D_integrator(class NetwMin *init_net, double temperature, double
slipspring_rate) {
00048
00049
00050         cur_bd_net = init_net;
00051         bd_cur_step = 0;
00052         dofs = cur_bd_net->network->nodes.size();
00053         dofs_3N = 3 * dofs;
00054
00055         xshift = (double*)malloc(dofs*sizeof(double));
00056         yshift = (double*)malloc(dofs*sizeof(double));
00057         zshift = (double*)malloc(dofs*sizeof(double));
00058         grid_cell = (int*)malloc(dofs*sizeof(int));
00059
00060         bd_x = (double*) malloc(dofs_3N * sizeof (double));
00061         bd_x_ps = (double*) malloc(dofs_3N * sizeof (double));
00062         bd_mass = (double*) malloc(dofs_3N * sizeof (double));
00063         bd_f = (double*) malloc(dofs_3N * sizeof (double));
00064         bd_f_ps = (double*) malloc(dofs_3N * sizeof (double));
00065         bd_nb_f = (double*) malloc(dofs_3N * sizeof (double));
00066         bd_gamma = (double*) malloc(dofs_3N * sizeof (double));
00067
00068
00069         bd_stress = (double**)malloc(dofs * sizeof(double*));
00070         for(unsigned int i = 0; i < dofs; i++)
00071             bd_stress[i] = (double*)malloc(6*sizeof(double));
00072
00073
00074         // Set the temperature:
00075         bd_temp = temperature;
00076
00077         double gamma;
00078         unsigned int inode = 0;
00079
00080
00081         // Monomeric friction coefficient in kg/s
00082         double monomeric_friction = 1.e-3 * pow(10.0, ((13.5 * 45.0)/(bd_temp-211.15+45)-10.4));
00083
00084         // convert monomeric friction coefficient (kg/s) to g/mol/s and then divide it with the mass
00085         // of a PI monomer
00086         gamma = monomeric_friction / (pi_monomer_mass * amu_to_kg); // s^{-1}
00087
00088         for (std::list<tNode>::iterator it = cur_bd_net->network->nodes.begin();
00089              it != cur_bd_net->network->nodes.end(); ++it) {

```

```

00124
00125     // positions in A
00126     bd_x[3 * inode + 0] = (*it).Pos[0];
00127     bd_x[3 * inode + 1] = (*it).Pos[1];
00128     bd_x[3 * inode + 2] = (*it).Pos[2];
00129
00130     // positions in A
00131     bd_x_ps[3 * inode + 0] = (*it).Pos[0];
00132     bd_x_ps[3 * inode + 1] = (*it).Pos[1];
00133     bd_x_ps[3 * inode + 2] = (*it).Pos[2];
00134
00135     // mass of nodal points in g/mol
00136     bd_mass[3 * inode + 0] = (*it).mass;
00137     bd_mass[3 * inode + 1] = (*it).mass;
00138     bd_mass[3 * inode + 2] = (*it).mass;
00139
00140     // All gamma are measured in s^{-1}
00141     bd_gamma[3 * inode + 0] = gamma;
00142     bd_gamma[3 * inode + 1] = gamma;
00143     bd_gamma[3 * inode + 2] = gamma;
00144
00145     inode++;
00146 }
00147
00148
00149 /* Check whether all beads have the same mass and the same friction coefficient. */
00150 double prev_val = bd_mass[0] * bd_gamma[0];
00151 gamma_mass_opt = true;
00152 for (inode = 1; inode < dofs_3N; inode++){
00153     if ( ((bd_mass[inode]*bd_gamma[inode])-prev_val)
00154         *((bd_mass[inode]*bd_gamma[inode])-prev_val) >= tol){
00155         gamma_mass_opt = false;
00156         break;
00157     }
00158     else
00159         prev_val = bd_mass[inode]*bd_gamma[inode];
00160 }
00161
00162 if (gamma_mass_opt)
00163     cout << "\n#: Brownian Dynamics integrator will run in the optimized way.\n"
00164         << "\n#: --- All beads have the same mass and friction coefficient.\n"
00165         << "\n#: --- The monomeric friction coefficient is " << monomeric_friction << " kg/s.\n"
00166         << "\n#: --- The friction coefficient is " << bd_gamma[0] << " s^{-1}.\n"
00167         << "\n#: --- The bead friction coefficient is " << bd_mass[0]*bd_gamma[0]*
amu_to_kg
00168         << " kg/s.\n"
00169         << "\n#: --- The expected Rouse diffusivity multiplied by N would be: "
00170         << boltz_const_Joule_K * bd_temp / bd_mass[0] / bd_gamma[0] /
amu_to_kg
00171         << " m^2/s.\n#:"<< endl;
00172
00173     // Initialize the coupled hopping scheme:
00174     // Here we can input the zeta parameter...
00175     if (cur_bd_net->network->pslip_springs.size() > 0)
00176         my_hopping_scheme = new Hopping(slip_spring_rate);
00177
00178     return;
00179 }
00180
00181
00182 void cb3D_integrator::integrate(unsigned int nsteps, double dt, unsigned int nstout) {
00183
00184     double current_energy = 0.0, current_nb_energy = 0.0;
00185
00186     density_cells = (double*) malloc(cur_bd_net->grid->ncells*sizeof(double));
00187     den_dx = (double*) malloc(27 * cur_bd_net->network->nodes.size() * sizeof(double));
00188     den_dy = (double*) malloc(27 * cur_bd_net->network->nodes.size() * sizeof(double));
00189     den_dz = (double*) malloc(27 * cur_bd_net->network->nodes.size() * sizeof(double));
00190
00191     // Keep the time:
00192     tbegin = clock();
00193     double dt_times_inv_mass_gamma, std;
00194     for (unsigned int idof = 0; idof < dofs_3N; idof++)
00195         bd_f_ps[idof] = 0.0;
00196
00197     // Ensure that there are slip-springs present in the network
00198     bool slip_spring_hopping = false;
00199     if (cur_bd_net->network->pslip_springs.size() > 0) {
00200         slip_spring_hopping = true;
00201         // and inform the user about the slip-spring hopping
00202         cout << "\n#: Slip-spring hopping has been enabled.\n";
00203     }
00204     else
00205         cout << "\n#: Slip-spring hopping has been disabled.\n";
00206
00207     bool out_step;

```

```

00230     for (unsigned int istep = 0; istep < nsteps; istep++) {
00231         /* Update the step counter: */
00232         bd_cur_step ++;
00233
00234         /* Check whether it is time to report the statistics.*/
00235         out_step = (istep % nstout == 0);
00236
00237         /* Calculate bonded and non-bonded interactions.*/
00238         current_energy = bonded_force_calculation(out_step);
00239         /* Calculate non-bonded interactions every 5 timesteps.*/
00240         if (istep % 5 == 0)
00241             current_nb_energy = simpler_scheme_non_bonded_force_calculation();
00242
00243         /* If it's time to report, do it: */
00244         if (out_step)
00245             report(istep, current_energy, current_nb_energy);
00246
00247         /* Keep a restart file: */
00248         if (istep % 20000 == 0){
00249             from_bd_x_to_polymer_network();
00250             cur_bd_net->write_network_to_lammps_data_file();
00251             //if (cur_bd_net->network->pslip_springs.size() > 1400)
00252             //    break;
00253         }
00254
00255         // Hopping starts here.
00256         if ((istep % 1000 == 0) && slippspring_hopping)
00257             my_hopping_scheme->hopping_step(cur_bd_net, this, bd_x, bd_temp, 1.e3*dt);
00258
00259
00260         /* Optimized integration scheme, in case every bead of the system has the same mass and
00261         * friction coefficient. */
00262         if (gamma_mass_opt){
00263             dt_times_inv_mass_gamma = (1.e-12*dt)/(bd_mass[0]*amu_to_kg*bd_gamma[0]); // s^2/kg
00264             std = sqrt(2.e20 * bd_temp * boltz_const_Joule_K * dt_times_inv_mass_gamma);
00265
00266             // A
00267             dt_times_inv_mass_gamma *= 1.e23 / avogadro_constant; // s^2 mol / kg
00268
00269             for (unsigned int i = 0; i < dofs_3N; i++) {
00270                 bd_x[i] = bd_x_ps[i]
00271                     + (bd_f[i] + bd_nb_f[i]) * dt_times_inv_mass_gamma
00272                     + 0.5 * (bd_f[i] + bd_nb_f[i] - bd_f_ps[i]) * dt_times_inv_mass_gamma * (1.e-12*dt)
00273                     + cur_bd_net->my_rnd_gen->gaussian()*std;
00274
00275                 bd_x_ps[i] = bd_x[i];
00276                 bd_f_ps[i] = bd_f[i] + bd_nb_f[i];
00277             }
00278
00279             else {
00280                 for (unsigned int i = 0; i < dofs_3N; i++) {
00281                     dt_times_inv_mass_gamma = (1.e-12 * dt) / (bd_mass[i]*amu_to_kg*bd_gamma[i]); //
00282                     s^2/kg
00283                     std = sqrt(2.e20*bd_temp*boltz_const_Joule_molK*
00284                     dt_times_inv_mass_gamma); // A
00285                     dt_times_inv_mass_gamma *= 1.e23 / avogadro_constant; // s^2 mol / kg
00286
00287                     bd_x[i] = bd_x_ps[i]
00288                         + (bd_f[i] + bd_nb_f[i]) * dt_times_inv_mass_gamma
00289                         + 0.5 * (bd_f[i] + bd_nb_f[i] - bd_f_ps[i]) * dt_times_inv_mass_gamma
00290                         + cur_bd_net->my_rnd_gen->gaussian()*std;
00291
00292                     bd_x_ps[i] = bd_x[i];
00293                     bd_f_ps[i] = bd_f[i] + bd_nb_f[i];
00294                 }
00295             }
00296         }
00297     }
00298 }
00299
00300 }
00301
00302
00303
00310     // Output the final statistics.
00311     current_energy = bonded_force_calculation(true);
00312     current_nb_energy = simpler_scheme_non_bonded_force_calculation();
00313
00314     report(nsteps, current_energy, current_nb_energy);
00315
00316     // Deallocate the arrays:
00317     free(den_dx);
00318     free(den_dy);
00319     free(den_dz);
00320     free(density_cells);
00321
00322     // Update the network with the new positions of the beads.
00323     from_bd_x_to_polymer_network();
00324     cur_bd_net->write_network_to_lammps_data_file();

```

```

00325
00326     return;
00327 }
00328
00329
00334 void cb3D_integrator::report(unsigned int istep, double b_energy, double nb_energy){
00335     double press_tens[6];
00336
00337     // gvog: Ask for the current time:
00338     clock_t tend = clock();
00339     calculate_pressure(press_tens);
00340
00341     double inv_vol = 1.0 / ( cur_bd_net->domain->XBoxLen
00342                             * cur_bd_net->domain->YBoxLen
00343                             * cur_bd_net->domain->ZBoxLen);
00344
00345     cout << istep << "\t" << b_energy << "\t" << nb_energy << "\t"
00346           << cur_bd_net->network->pslip_springs.size() << "\t"
00347           <<< " pressure " * inv_vol << "\t"
00348           << press_tens[0] * inv_vol << "\t"
00349           << press_tens[1] * inv_vol << "\t"
00350           << press_tens[2] * inv_vol << "\t"
00351           << press_tens[3] * inv_vol << "\t"
00352           << press_tens[4] * inv_vol << "\t"
00353           << press_tens[5] * inv_vol << "\t"
00354           << " # (" << (double) (tend - tbegin) / CLOCKS_PER_SEC << " s )" << endl;
00355
00356     //cur_bd_net->my_traj_file->add_snapshot_to_dump(cur_bd_net, this, istep);
00357
00358     return;
00359 }
00360
00361
00363 void cb3D_integrator::calculate_pressure(double *press_tens){
00364     // Initialize the pressure tensor to zero.
00365     for (unsigned int j = 0; j < 6; j++)
00366         press_tens[j] = 0.0;
00367
00368     // Accumulated the per-atom pressures to the global tensor:
00369     for (unsigned int i = 0; i < dofs; i++)
00370         for (unsigned int j = 0; j < 6; j++)
00371             press_tens[j] += bd_stress[i][j];
00372
00373     // Convert per-atom pressure*vol in atm*Angstrom
00374     // An interesting thread concerning loop unrolling in C++:
00375     // http://stackoverflow.com/questions/15275023/clang-force-loop-unroll-for-specific-loop
00376     for (unsigned int j = 0; j < 6; j++)
00377         press_tens[j] *= 1.e33 / avogadro_constant / 101.325e3;
00378
00379     return;
00380 }
00381
00382
00383
00384
00385
00386
00387
00388
00389
00390
00391
00392
00393
00394
00395
00396
00397
00404 double cb3D_integrator::bonded_force_calculation(bool stress_calc) {
00405     double fenergy = 0.0;
00406
00407     // Initialize the forces to zero.
00408     for (unsigned int i = 0; i < dofs_3N; i++)
00409         bd_f[i] = 0.0;
00410
00411     // Initialize the stresses to zero, if we have been asked for stress calculation:
00412     if (stress_calc)
00413         for (unsigned int i = 0; i < dofs; i++)
00414             for (unsigned int j = 0; j < 6; j++)
00415                 bd_stress[i][j] = 0.0;
00416
00417     int taga, taga3, tagb, tagb3;
00418     //double *sep_vec = (double*)malloc(3*sizeof(double));
00419     //double *grada = (double*)malloc(3*sizeof(double));
00420     //double *gradb = (double*)malloc(3*sizeof(double));
00421
00422     double sep_vec[3], grada[3], gradb[3];
00423
00424     for (std::list<tStrand>::iterator it = cur_bd_net->network->strands.begin();
00425          it != cur_bd_net->network->strands.end(); ++it) {
00426
00427         /* Ask for the tags of the nodes connected to the current strand. */
00428         taga = (*it).pEnds[0]->Id - 1;
00429         taga3 = 3*taga;
00430         tagb = (*it).pEnds[1]->Id - 1;
00431         tagb3 = 3*tagb;
00432
00433         /* Form the "strand" vector, based on the x vector coming from
00434          * the minimizer. */

```

```

00437     sep_vec[0] = bd_x[tagb3 + 0] - bd_x[taga3 + 0];
00438     sep_vec[1] = bd_x[tagb3 + 1] - bd_x[taga3 + 1];
00439     sep_vec[2] = bd_x[tagb3 + 2] - bd_x[taga3 + 2];
00440
00441     /* Apply minimum image convention. */
00442     cur_bd_net->domain->minimum_image(sep_vec[0], sep_vec[1], sep_vec[2]);
00443
00444 #ifdef FENE_SLS
00445     if ((*it).slip_spring)
00446         fenergy += f_fene( sep_vec, (*it).spring_coeff,
00447                           (*it).sq_end_to_end, bd_temp, grada, gradb);
00448     else
00449 #endif
00450     /* Calculate the spring's contribution to the free energy of the system.*/
00451     fenergy += f_gaussian( sep_vec, (*it).spring_coeff,
00452                           (*it).sq_end_to_end, bd_temp, grada, gradb);
00453
00454     // Accumulate the forces of the first atom:
00455     bd_f[taga3 + 0] += grada[0];
00456     bd_f[taga3 + 1] += grada[1];
00457     bd_f[taga3 + 2] += grada[2];
00458     // Accumulate the forces of the second atom:
00459     bd_f[tagb3 + 0] += gradb[0];
00460     bd_f[tagb3 + 1] += gradb[1];
00461     bd_f[tagb3 + 2] += gradb[2];
00462
00470     if (stress_calc) {
00471         bd_stress[taga][0] += 0.5 * sep_vec[0] * grada[0]; // xx
00472         bd_stress[taga][1] += 0.5 * sep_vec[1] * grada[1]; // yy
00473         bd_stress[taga][2] += 0.5 * sep_vec[2] * grada[2]; // zz
00474         bd_stress[taga][3] += 0.5 * sep_vec[0] * grada[1]; // xy
00475         bd_stress[taga][4] += 0.5 * sep_vec[0] * grada[2]; // xz
00476         bd_stress[taga][5] += 0.5 * sep_vec[1] * grada[2]; // yz
00477
00478         bd_stress[tagb][0] += 0.5 * sep_vec[0] * grada[0];
00479         bd_stress[tagb][1] += 0.5 * sep_vec[1] * grada[1];
00480         bd_stress[tagb][2] += 0.5 * sep_vec[2] * grada[2];
00481         bd_stress[tagb][3] += 0.5 * sep_vec[0] * grada[1];
00482         bd_stress[tagb][4] += 0.5 * sep_vec[0] * grada[2];
00483         bd_stress[tagb][5] += 0.5 * sep_vec[1] * grada[2];
00484     }
00485 }
00486
00487
00488
00489
00490     return (fenergy);
00491 }
00492
00493
00495 void cb3D_integrator::from_bd_x_to_polymer_network(void) {
00496
00497     unsigned int inode = 0;
00498
00499     for (std::list<Node>::iterator it = cur_bd_net->network->nodes.begin();
00500          it != cur_bd_net->network->nodes.end(); ++it) {
00501         (*it).Pos[0] = bd_x[3 * inode + 0];
00502         (*it).Pos[1] = bd_x[3 * inode + 1];
00503         (*it).Pos[2] = bd_x[3 * inode + 2];
00504         inode ++;
00505     }
00506
00507     return;
00508 }
00509
00510
00511
00513 double cb3D_integrator::simpler_scheme_non_bonded_force_calculation(void) {
00514
00515     // Variables holding the volume of a cell.
00516     double vcube_cell, vx, vy, vz;
00517
00518     // The nonbonded contribution to the free energy of the network.
00519     double f_nb_energy = 0.0;
00520
00521     //at this point we may need to call a subroutine that will convert bd_x elements to positions
00522
00523     int l; // indices used to find the parent cell and its first neighbours
00524
00525     for (int i = 0; i < cur_bd_net->grid->ncells; i++)
00526         density_cells[i] = 0.0;
00527
00528     int cur_node = 0, i, j, cur_elem, max_node = cur_bd_net->network->nodes.size();
00529     double dx, dy, dz, xl, yl, zl, half_rnode, mass_over_rnode3;
00530
00531     half_rnode = 0.5 * cur_bd_net->network->nodes.front().r_node;
00532     mass_over_rnode3 = cur_bd_net->network->nodes.front().n_mass

```



```

00593         / (cur_bd_net->network->nodes.front().r_node
00594         * cur_bd_net->network->nodes.front().r_node
00595         * cur_bd_net->network->nodes.front().r_node);
00596
00597
00598     for (cur_node = 0; cur_node < max_node; cur_node++) {
00599         /*loop over the (*it).node_cell itself and its first neighbors (it is always equal
00600         * to 27, or 26 starting the numbering from zero*/
00601         /*expressed in Angstrom^3. node coordinates have to be shifted
00602         * boxl/2.0 so as to be embedded into a grid extended from
00603         * zero to cur_bd_net->domain->XBoxLen */
00604
00605         //half_rnode = 0.5 * (*it).r_node;
00606         //mass_over_rnode3 = (*it).n_mass / (*it).r_node / (*it).r_node / (*it).r_node;
00607
00608         xshift[cur_node] = bd_x[3 * cur_node];
00609         yshift[cur_node] = bd_x[3 * cur_node + 1];
00610         zshift[cur_node] = bd_x[3 * cur_node + 2];
00611
00612         //return the nodes back into the primary box
00613         cur_bd_net->domain->minimum_image(xshift[cur_node], yshift[cur_node], zshift[cur_node]);
00614
00615         //shift the node position to a shifted simulation box that contains the grid
00616         xshift[cur_node] += 0.5 * cur_bd_net->domain->XBoxLen;
00617         yshift[cur_node] += 0.5 * cur_bd_net->domain->YBoxLen;
00618         zshift[cur_node] += 0.5 * cur_bd_net->domain->ZBoxLen;
00619
00620         grid_cell[cur_node] = cur_bd_net->grid->find_grid_cell(xshift[cur_node], yshift[cur_node]
00621
00622                                     zshift[cur_node]);
00623
00624     for (j = 0; j < 27; j++) {
00625         // find the neighbours of (*it).node_cell, zero corresponds to the cell itself
00626
00627         l = cur_bd_net->grid->cells[grid_cell[cur_node]].neigh[j];
00628
00629         /*find the intersection of cube formed by node (*it).Id, how to define
00630         * cell_vec[l][0:2]: vector of cell l, is used for the calculation of
00631         * vcube_cell,*/
00632
00633         // if statements for vx
00634         // for the computation of minimum images of xshift, yshift, zshift with respect
00635         // to xcell, ycell and zcell respectively
00636         dx = xshift[cur_node] - cur_bd_net->grid->cells[l].Vec[0];
00637         dy = yshift[cur_node] - cur_bd_net->grid->cells[l].Vec[1];
00638         dz = zshift[cur_node] - cur_bd_net->grid->cells[l].Vec[2];
00639
00640         // minimum images in a box from zero to box_l
00641         //where xl, yl and zl updated values, due to minimum image, of xshift, yshift and zshift
00642         cur_bd_net->domain->zero_to_length_minimum_image(dx, dy, dz);
00643
00644         xl = cur_bd_net->grid->cells[l].Vec[0] + dx;
00645         yl = cur_bd_net->grid->cells[l].Vec[1] + dy;
00646         zl = cur_bd_net->grid->cells[l].Vec[2] + dz;
00647
00648
00649         vx = max(min(xl + half_rnode, cur_bd_net->grid->cells[l].Vec[0])
00650                 - max(xl-half_rnode, cur_bd_net->grid->cells[l].Vec[0]-cur_bd_net->grid->dlx), 0.0);
00651
00652         vy = max(min(yl + half_rnode, cur_bd_net->grid->cells[l].Vec[1])
00653                 - max(yl-half_rnode, cur_bd_net->grid->cells[l].Vec[1]-cur_bd_net->grid->dly), 0.0);
00654
00655         vz = max(min(zl + half_rnode, cur_bd_net->grid->cells[l].Vec[2])
00656                 - max(zl-half_rnode, cur_bd_net->grid->cells[l].Vec[2]-cur_bd_net->grid->dlz), 0.0);
00657
00658         vcube_cell = vx * vy * vz;
00659
00660         cur_elem = 27 * cur_node + j;
00661
00662         if ((xl > cur_bd_net->grid->cells[l].Vec[0] - cur_bd_net->grid->dlx - half_rnode) &&
00663             (xl < cur_bd_net->grid->cells[l].Vec[0] - cur_bd_net->grid->dlx + half_rnode))
00664             den_dx[cur_elem] = mass_over_rnode3 * vx * vz / cur_bd_net->grid->vcell;
00665         else if ((xl > cur_bd_net->grid->cells[l].Vec[0] - half_rnode) &&
00666                 (xl < cur_bd_net->grid->cells[l].Vec[0] + half_rnode))
00667             den_dx[cur_elem] = -mass_over_rnode3 * vx * vz / cur_bd_net->grid->vcell;
00668         else
00669             den_dx[cur_elem] = 0.0;
00670
00671         if ((yl > cur_bd_net->grid->cells[l].Vec[1] - cur_bd_net->grid->dly - half_rnode) &&
00672             (yl < cur_bd_net->grid->cells[l].Vec[1] - cur_bd_net->grid->dly + half_rnode))
00673             den_dy[cur_elem] = mass_over_rnode3 * vx * vz / cur_bd_net->grid->vcell;
00674         else if ((yl > cur_bd_net->grid->cells[l].Vec[1] - half_rnode) &&
00675                 (yl < cur_bd_net->grid->cells[l].Vec[1] + half_rnode))
00676             den_dy[cur_elem] = -mass_over_rnode3 * vx * vz / cur_bd_net->grid->vcell;
00677         else
00678             den_dy[cur_elem] = 0.0;
00679
00680         if ((zl > cur_bd_net->grid->cells[l].Vec[2] - cur_bd_net->grid->dlz - half_rnode) &&
00681             (zl < cur_bd_net->grid->cells[l].Vec[2] - cur_bd_net->grid->dlz + half_rnode))
00682             den_dz[cur_elem] = mass_over_rnode3 * vx * vy / cur_bd_net->grid->vcell;
00683         else if ((zl > cur_bd_net->grid->cells[l].Vec[2] - half_rnode) &&
00684                 (zl < cur_bd_net->grid->cells[l].Vec[2] + half_rnode))
00685             den_dz[cur_elem] = -mass_over_rnode3 * vx * vy / cur_bd_net->grid->vcell;
00686         else
00687             den_dz[cur_elem] = 0.0;
00688
00689         // calculate the density of the cell
00690         den[cur_elem] = den_dx[cur_elem] + den_dy[cur_elem] + den_dz[cur_elem];
00691
00692         // calculate the mass of the cell
00693         mass[cur_elem] = (*it).n_mass;
00694
00695         // calculate the volume of the cell
00696         vol[cur_elem] = vcube_cell;
00697
00698         // calculate the density of the cell
00699         den[cur_elem] = den_dx[cur_elem] + den_dy[cur_elem] + den_dz[cur_elem];
00700
00701         // calculate the mass of the cell
00702         mass[cur_elem] = (*it).n_mass;
00703
00704         // calculate the volume of the cell
00705         vol[cur_elem] = vcube_cell;

```

```

00706
00707     if ((z1 > cur_bd_net->grid->cells[1].Vec[2] - cur_bd_net->grid->dlz - half_rnode) &&
00708         (z1 < cur_bd_net->grid->cells[1].Vec[2] - cur_bd_net->grid->dlz + half_rnode))
00709         den_dz[cur_elem] = mass_over_rnode3 * vx * vy / cur_bd_net->grid->vcell;
00710     else if ((z1 > cur_bd_net->grid->cells[1].Vec[2] - half_rnode) &&
00711         (z1 < cur_bd_net->grid->cells[1].Vec[2] + half_rnode))
00712         den_dz[cur_elem] = -mass_over_rnode3 * vx * vy / cur_bd_net->grid->vcell;
00713     else
00714         den_dz[cur_elem] = 0.0;
00715
00724     density_cells[1] += mass_over_rnode3*vcube_cell;
00725 }
00726 }
00727
00736 for (i = 0; i < cur_bd_net->grid->ncells; i++) {
00737     density_cells[i] *= cur_bd_net->grid->ivcell; //expressed in kuhn segments per Angstrom^3
00738     f_nb_energy += cur_bd_net->grid->vcell
00739         * (c1 * density_cells[i] + c2 * density_cells[i] * density_cells[i]);
00740 }
00741
00742
00751 // new nested for-loops for updating the 3N vector of derivatives
00752 double fx, fy, fz; // force components on a nod due to non-bonded interactions
00753 // used for updating the array of derivatives[3N]
00754 for (cur_node = 0; cur_node < max_node; cur_node++){
00755     fx = 0.0;
00756     fy = 0.0;
00757     fz = 0.0;
00758
00768     for (j = 0; j < 27; j++) {
00769
00770         cur_elem = 27 * cur_node + j;
00771
00772         l = cur_bd_net->grid->cells[grid_cell[cur_node]].neigh[j];
00773
00774         fx -= cur_bd_net->grid->vcell * (c1 + 2.0 * c2 * density_cells[l]) * den_dx[cur_elem];
00775         fy -= cur_bd_net->grid->vcell * (c1 + 2.0 * c2 * density_cells[l]) * den_dy[cur_elem];
00776         fz -= cur_bd_net->grid->vcell * (c1 + 2.0 * c2 * density_cells[l]) * den_dz[cur_elem];
00777     }
00778
00779     bd_nb_f[3 * cur_node] = fx;
00780     bd_nb_f[3 * cur_node + 1] = fy;
00781     bd_nb_f[3 * cur_node + 2] = fz;
00782 }
00783
00784
00785     return (f_nb_energy);
00786 }
00787
00788
00789
00790 void cb3D_integrator::cell_density_nodal_points() {
00791
00792     double xnew, ynew, znew;
00793     int Id;
00794     int *hist_grid = (int*) malloc(cur_bd_net->grid->ncells * sizeof (int));
00795
00796     for (int i = 0; i < cur_bd_net->grid->ncells; i++)
00797         hist_grid[i] = 0;
00798
00799     for (unsigned int inode = 0; inode < dofs; inode++) {
00800
00801         xnew = bd_x[3 * inode + 0];
00802         ynew = bd_x[3 * inode + 1];
00803         znew = bd_x[3 * inode + 2];
00804         cur_bd_net->domain->minimum_image(xnew, ynew, znew);
00805         xnew = xnew + cur_bd_net->domain->XBoxLen / 2.0;
00806         ynew = ynew + cur_bd_net->domain->YBoxLen / 2.0;
00807         znew = znew + cur_bd_net->domain->ZBoxLen / 2.0;
00808         Id = cur_bd_net->grid->find_grid_cell(xnew, ynew, znew);
00809         hist_grid[Id] = hist_grid[Id] + 1;
00810     }
00811
00812
00813     for (int i = 0; i < cur_bd_net->grid->ncells; i++)
00814         fprintf(p_cell_density, "%d " " %d\n", i, hist_grid[i]);
00815
00816     return;
00817 }
00818
00819 }

```

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

```
00001
00016 #ifndef _B3D_INTEGRATOR_H
00017 #define _B3D_INTEGRATOR_H
00018
00019 #include <cmath>
00020 #include <cstdio>
00021 #include <list>
00022 #include <stdlib.h>
00023 #include <vector>
00024
00025 #include "hopping.h"
00026 #include "net_types.h"
00027 #include "network.h"
00028 #include "domain.h"
00029 #include "rng.h"
00030
00031 namespace NetworkNS {
00032
```

```

00037     class cb3D_integrator{
00038     public:
00039
00040         cb3D_integrator(class NetwMin *, double, double);
00041         ~cb3D_integrator();
00042
00043         void integrate(unsigned int nsteps, double dt, unsigned int nstout);
00044
00045         void extract_positions(double *x);
00046
00047         void compute_stresses (void);
00048
00049         void report(unsigned int, double, double);
00050
00051         double bonded_force_calculation(bool);
00052
00053         double simpler_scheme_non_bonded_force_calculation(void);
00054
00055
00056
00057         void calculate_pressure(double *);
00058
00059         double *bd_gamma;
00060
00061         double *bd_mass;
00062
00063         double *bd_x;
00064         double **bd_stress;
00065
00066         unsigned int bd_cur_step;
00067
00068     private:
00069
00070         class NetwMin *cur_bd_net;
00071
00072         unsigned int dofs;
00073         unsigned int dofs_3N;
00074
00075
00076         double *bd_f;
00077
00078         double *bd_nb_f;
00079
00080         double *bd_x_ps;
00081
00082         double *bd_f_ps;
00083
00084         class Hopping *my_hopping_scheme;
00085
00086         clock_t tbegin;
00087
00088         double *xshift;
00089
00090         double *yshift;
00091
00092         double *zshift;
00093
00094         int *grid_cell;
00095
00096         double bd_temp;
00097
00098         bool gamma_mass_opt;
00099
00100         double *density_cells;
00101
00102         double *den_dx;
00103
00104         double *den_dy;
00105
00106         double *den_dz;
00107
00108         FILE * p_cell_density;
00109
00110         void from_bd_x_to_polymer_network(void); // Convert elements of array
00111         bd_x to positions of the polymer network
00112
00113         void cell_density_nodal_points(void); // compute the density in each cell of the orthogonal
00114         grid
00115
00116     };
00117
00118 }
00119
00120 #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](#) = 3.1415926535897932384626433
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

Copyright (c) 2013 Grigorios Megariotis and 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 [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, $1\text{kg} =$

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. $1\text{g/mol} = 1.660538921 \times 10^{-27}\text{kg}$

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: $\mathcal{V}_{\text{nb}}(\rho) = c_1\rho + c_2\rho^2$.

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: $\mathcal{V}_{\text{nb}}(\rho) = c_1\rho + c_2\rho^2$.

Definition at line 46 of file [constants.h](#).

5.10 constants.h

```
00001
00018 #ifndef CONSTANTS_H
```

```

00019 #define  CONSTANTS_H
00020
00021 double const avogadro_constant = 6.02214129e23;
00022
00024 double const kg_to_amus = 6.022141129e26;
00025
00027 double const amu_to_kg = 1.660538921e-27;
00028
00031 double const pi_monomer_mass = 68.12;
00032
00033 double const boltz_const_Joule_molK = 8.3144621;
00034 double const boltz_const_kJoule_molK = 8.3144621e-3;
00035 double const boltz_const_Joule_K = 1.3806488e-23;
00036 double const pi_tube_diameter = 80.39;
00037
00041 const double c1=-5.4e02;
00042
00046 const double c2=7.0e04;
00047
00052 const double tol = 1.e-4;
00053
00054 const double PI = 3.1415926535897932384626433;
00055
00056 double const hopping_attempt_radius = 60.0;
00057
00058 #endif  /* CONSTANTS_H */
00059

```

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

Copyright (c) 2013 Grigorios Megariotis and 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 [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	<i>rij</i>	the separation vector between beads i and j, i.e. \mathbf{R}_{ij} .
in	<i>coeff</i>	the strength of the entropic springs $\varepsilon_b = 3/2 k_B$, measured in kJ/mol/K.
in	<i>sq_ete</i>	the equilibrium mean-squared end-to-end length of the strand, i.e. $\sigma_b = n_{\text{Kuhns/beat}} b^2$, measured in \AA^2 .
in	<i>temp</i>	the temperature of the simulation, T , in K.
out	<i>gradi</i>	the force acted on the bead i, due to its bond with bead j
out	<i>gradj</i>	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:

$$\mathcal{V}_b(r_{ij}^2) = \varepsilon_b T \frac{r_{ij}^2}{\sigma_b^2} = \frac{3}{2} k_B T \frac{\mathbf{R}_{ij} \cdot \mathbf{R}_{ij}}{n_{\text{Kuhns/beat}} b^2}$$

with the parameters ε_b and σ_b read from the data file. In our approach $\varepsilon_b = 0.012471$ kJ/mol/K and $\sigma_b = 810 \text{\AA}^2$ 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	<i>rij</i>	the separation vector between beads i and j, i.e. \mathbf{R}_{ij} .
in	<i>coeff</i>	the strength of the entropic springs $\varepsilon_b = 3/2 k_B$, measured in kJ/mol/K.
in	<i>sq_ete</i>	the equilibrium mean-squared end-to-end length of the strand, i.e. $\sigma_b = n_{\text{Kuhns/beat}} b^2$, measured in \AA^2 .
in	<i>temp</i>	the temperature of the simulation, T , in K.

Definition at line 71 of file [distributions.cpp](#).

5.12 distributions.cpp

00001


```

00014 #include <cmath>
00015 #include <iostream>
00016
00017 #include "distributions.h"
00018
00019 using namespace std;
00020
00031 double f_gaussian(const double *rij, const double &coeff, const double &sq_ete,
00032                  const double &temp, double *gradi, double *gradj) {
00033
00044     // Compute the distance between positional vectors ri and rj
00045     double rsq = rij[0]*rij[0] + rij[1]*rij[1] + rij[2]*rij[2];
00046
00047     // Compute the free energy of the Gaussian spring.
00048     double p = coeff * temp / sq_ete;
00049
00050     // Compute the forces as gradients of the Gaussian distribution along ri and rj direction.
00051     gradi[0] = 2.0 * p * rij[0];
00052     gradi[1] = 2.0 * p * rij[1];
00053     gradi[2] = 2.0 * p * rij[2];
00054
00055     gradj[0] = -gradi[0];
00056     gradj[1] = -gradi[1];
00057     gradj[2] = -gradi[2];
00058
00059     return (p*rsq);
00060 }
00061
00062
00071 double e_gaussian(const double *rij, const double &coeff, const double &sq_ete,
00072                  const double &temp) {
00073     //compute the distance between positional vectors ri and rj
00074     double rsq = rij[0]*rij[0] + rij[1]*rij[1] + rij[2]*rij[2];
00075
00076     // Compute only the free energy of the Gaussian spring.
00077     return (coeff * temp * rsq / sq_ete);
00078 }
00079

```

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

Copyright (c) 2014 Grigorios Megariotis and 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 [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	<i>rij</i>	the separation vector between beads i and j, i.e. \mathbf{R}_{ij} .
in	<i>coeff</i>	the strength of the entropic springs $\varepsilon_b = 3/2 k_B$, measured in kJ/mol/K.
in	<i>sq_ete</i>	the equilibrium mean-squared end-to-end length of the strand, i.e. $\sigma_b = n_{\text{Kuhns/beat}} b^2$, measured in \AA^2 .
in	<i>temp</i>	the temperature of the simulation, T , in K.
out	<i>gradi</i>	the force acted on the bead i, due to its bond with bead j
out	<i>gradj</i>	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:

$$\mathcal{V}_b(r_{ij}^2) = \varepsilon_b T \frac{r_{ij}^2}{\sigma_b^2} = \frac{3}{2} k_B T \frac{\mathbf{R}_{ij} \cdot \mathbf{R}_{ij}}{n_{\text{Kuhns/beat}} b^2}$$

with the parameters ε_b and σ_b read from the data file. In our approach $\varepsilon_b = 0.012471$ kJ/mol/K and $\sigma_b = 810 \text{\AA}^2$ 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	<i>rij</i>	the separation vector between beads i and j, i.e. \mathbf{R}_{ij} .
in	<i>coeff</i>	the strength of the entropic springs $\varepsilon_b = 3/2 k_B$, measured in kJ/mol/K.
in	<i>sq_ete</i>	the equilibrium mean-squared end-to-end length of the strand, i.e. $\sigma_b = n_{\text{Kuhns/beat}} b^2$, measured in \AA^2 .
in	<i>temp</i>	the temperature of the simulation, T , in K.

Definition at line 71 of file [distributions.cpp](#).

5.14 distributions.h

00001

```

00015 #ifndef _DISTRIBUTIONS_H_
00016 #define _DISTRIBUTIONS_H_
00017
00018 double f_gaussian(const double *, const double &, const double &, const double &, double *,
    double *);
00019 double e_gaussian(const double *, const double &, const double &, const double &);
00020
00021 #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

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 [domain.cpp](#).

5.16 domain.cpp

```

00001
00015 #include <cmath>
00016 #include <fstream>
00017 #include <iostream>
00018 #include <sstream>
00019 #include <stdlib.h>
00020 #include <stdio.h>
00021 #include <vector>
00022
00023 #include "domain.h"
00024
00025 using namespace std;
00026
00027 namespace NetworkNS {
00028
00029     Domain::Domain(std::string filename) {
00030
00031         ifstream data_file(filename.c_str(), ifstream::in);
00032         /* Define an array of strings to hold the contents of the file. */
00033         std::vector<string> lines_of_file;
00034
00035         for (int i = 0; i < 11; i++) {
00036             /* A temporary string for the current line of the file. */
00037             std::string current_line;
00038             getline(data_file, current_line);

```

```

00039         /* Add current line to file's array of lines. */
00040         lines_of_file.push_back(current_line);
00041     }
00042
00043     /* x box dimension is stored in the 9th line: */
00044     string buf;
00045     stringstream ss;
00046     vector<string> tokens;
00047
00048     // http://www.cplusplus.com/faq/sequences/strings/split/#boost-split
00049
00050     for (int i = 0; i < 3; i++) {
00051         ss.flush();
00052         tokens.clear();
00053         ss << lines_of_file[8 + i];
00054         while (ss >> buf)
00055             tokens.push_back(buf);
00056         BoxLow[i] = atof(tokens[0].c_str());
00057         BoxHigh[i] = atof(tokens[1].c_str());
00058     }
00059
00060
00061     XBoxLen = BoxHigh[0] - BoxLow[0];
00062     iXBoxLen = 1.0 / XBoxLen;
00063
00064     YBoxLen = BoxHigh[1] - BoxLow[1];
00065     iYBoxLen = 1.0 / YBoxLen;
00066
00067     ZBoxLen = BoxHigh[2] - BoxLow[2];
00068     iZBoxLen = 1.0 / ZBoxLen;
00069
00070     BoxExists = 1;
00071     NonPeriodic = 0;
00072     Xperiodic = Yperiodic = Zperiodic = 1;
00073     Periodicity[0] = Xperiodic;
00074     Periodicity[1] = Yperiodic;
00075     Periodicity[2] = Zperiodic;
00076
00077     Boundary[0][0] = Boundary[0][1] = 0;
00078     Boundary[1][0] = Boundary[1][1] = 0;
00079     Boundary[2][0] = Boundary[2][1] = 0;
00080
00081     return;
00082 }
00083
00084 Domain::~Domain() {
00085     return;
00086 }
00087
00088 void Domain::minimum_image(double &x, double &y, double &z) {
00089     x = x - XBoxLen * round(x * iXBoxLen);
00090     y = y - YBoxLen * round(y * iYBoxLen);
00091     z = z - ZBoxLen * round(z * iZBoxLen);
00092     return;
00093 }
00094
00095 void Domain::zero_to_length_minimum_image(double &x, double&y, double &z)
00096 {
00097     x = x - XBoxLen * round(x * iXBoxLen);
00098     y = y - YBoxLen * round(y * iYBoxLen);
00099     z = z - ZBoxLen * round(z * iZBoxLen);
00100 }
00101
00102
00103 void Domain::put_in_primary_box(double *coords, int *pcoeffs){
00104
00105     // Calculate the distance from the center of the simulation box.
00106     double distx = coords[0] - (BoxLow[0] + 0.5*XBoxLen);
00107     double disty = coords[1] - (BoxLow[1] + 0.5*YBoxLen);
00108     double distz = coords[2] - (BoxLow[2] + 0.5*ZBoxLen);
00109
00110     // Calculate the periodic continuation coefficients:
00111     pcoeffs[0] = round(distx * iXBoxLen);
00112     pcoeffs[1] = round(disty * iYBoxLen);
00113     pcoeffs[2] = round(distz * iZBoxLen);
00114
00115     coords[0] += (double)pcoeffs[0] * XBoxLen + BoxLow[0];
00116     coords[1] += (double)pcoeffs[1] * YBoxLen + BoxLow[1];
00117     coords[2] += (double)pcoeffs[2] * ZBoxLen + BoxLow[2];
00118
00119     return;
00120 }
00121
00122 }

```

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

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 [domain.h](#).

5.18 domain.h

```
00001
00015 #ifndef DOMAIN_H
00016 #define DOMAIN_H
00017
00018 #include <string>
00019
00020 namespace NetworkNS {
00021
00026     class Domain{
00027     public:
00028         int BoxExists;
00029         int NonPeriodic;
00031
00035         int Xperiodic;
00036         int Yperiodic;
00037         int Zperiodic;
00038
00039         int Periodicity[3];
00040
00041         int Boundary[3][2];
00042
00045         double BoxLow[3];
00046         double BoxHigh[3];
00047
00048
00049         double XBoxLen;
00050         double YBoxLen;
00051         double ZBoxLen;
00052
00053         double iXBoxLen;
00054         double iYBoxLen;
00055         double iZBoxLen;
00056
```

```

00057
00058     Domain(std::string);
00059
00061     virtual ~Domain();
00062
00063     void minimum_image(double &x, double &y, double &z);
00065
00066     void zero_to_length_minimum_image(double &, double&, double &);
00068
00069     void put_in_primary_box(double *, int *);
00071
00072 };
00073 }
00074
00075 #endif /* DOMAIN_H */
00076

```

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

Copyright (c) 2013 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 [dump.cpp](#).

5.20 dump.cpp

```

00001
00016 #include "dump.h"
00017 #include "network.h"
00018 #include "netmin.h"
00019 #include "b3D_integrator.h"
00020
00021 using namespace std;
00022
00023 namespace NetworkNS {
00024
00025     dump::dump(std::string filename_to_open) {
00026
00027         my_file.open(filename_to_open.c_str(), ofstream::out);
00028         if (not my_file.good())
00029             cout << "Specified dump file does not exist!" << endl;
00030
00031     }

```

```

00032
00040 void dump::add_snapshot_to_dump(const class NetwMin *netw_app, const class
      cb3D_integrator *b3D,
00041                                unsigned int timestep){
00042
00043     my_file << "ITEM: TIMESTEP\n";
00044     my_file << timestep << "\n";
00045     my_file << "ITEM: NUMBER OF ATOMS\n";
00046     my_file << netw_app->network->nodes.size() << "\n";
00047     my_file << "ITEM: BOX BOUNDS pp pp pp\n";
00048     my_file << netw_app->domain->BoxLow[0] << " " << netw_app->
domain->BoxHigh[0] << "\n";
00049     my_file << netw_app->domain->BoxLow[1] << " " << netw_app->
domain->BoxHigh[1] << "\n";
00050     my_file << netw_app->domain->BoxLow[2] << " " << netw_app->
domain->BoxHigh[2] << "\n";
00051     my_file << "ITEM: ATOMS id x y z ix iy iz Vor VorNeigh xx yy zz xy xz yz\n";
00052
00053     // TODO: Here we can ask for a Voronoi tessellation of the simulation box in order to
00054     // calculate atomic volumes.
00055
00056     my_file.precision(6);
00057     for (std::list<tNode>::iterator it = netw_app->network->nodes.begin();
00058          it != netw_app->network->nodes.end(); ++it){
00059
00060         unsigned int ibead = (*it).Id - 1;
00061
00062         my_file << (*it).Id << " "
00063             << b3D->bd_x[3*ibead + 0] << " "
00064             << b3D->bd_x[3*ibead + 1] << " "
00065             << b3D->bd_x[3*ibead + 2] << " "
00066             << " 0 0 0 1.0 0 "
00067             << b3D->bd_stress[ibead][0] << " "
00068             << b3D->bd_stress[ibead][1] << " "
00069             << b3D->bd_stress[ibead][2] << " "
00070             << b3D->bd_stress[ibead][3] << " "
00071             << b3D->bd_stress[ibead][4] << " "
00072             << b3D->bd_stress[ibead][5] << endl;
00073     }
00074 }
00075
00076 dump::dump(const dump& orig) {
00077
00078 }
00079
00080 dump::~dump() {
00081     my_file.close();
00082 }
00083
00084 }

```

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

Copyright (c) 2013 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 [dump.h](#).

5.22 dump.h

```
00001
00014 #ifndef DUMP_H
00015 #define DUMP_H
00016
00017 #include <fstream>
00018 #include <iostream>
00019 #include <string>
00020
00021 #include "b3D_integrator.h"
00022
00023
00024 namespace NetworkNS {
00025
00030     class dump {
00031     public:
00032
00033         dump(std::string);
00034         dump(const dump& orig);
00035         void add_snapshot_to_dump(const class NetwMin *, const class
00036         cb3D_integrator *, unsigned int);
00037         virtual ~dump();
00038
00039
00040     private:
00041         std::ofstream my_file;
00042
00043
00044     };
00045 }
00046 #endif /* DUMP_H */
00048
```

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

Copyright (c) 2014 Grigorios Megariotis and 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 [grid.cpp](#).

5.24 grid.cpp

```

00001
00016 #include <cstdlib>
00017 #include "grid.h"
00018
00019 namespace NetworkNS {
00020
00021 Grid::Grid(double Lx, double Ly, double Lz, int nx, int ny, int nz) {
00022
00023     /* Initialize the arrays with the cells: */
00024     ncellx = nx;
00025     ncelly = ny;
00026     ncellz = nz;
00027     ncells = ncellx * ncelly * ncellz;
00028     cells = (grid_cell*)malloc(ncells * sizeof(grid_cell));
00029
00030     /*define dlx, dly, dlz and vecell: all of these are common parameters for each subcell
00031      * and thus they are defined only once*/
00032
00033     dlx = Lx / double(ncellx);
00034     idlx = 1.0 / dlx;
00035     dly = Ly / double(ncelly);
00036     idly = 1.0 / dly;
00037     dlz = Lz / double(ncellz);
00038     idlz = 1.0 / dlz;
00039     vcell = dlx * dly * dlz;
00040     ivcell = 1.0 / vcell;
00041
00042     int counter = 0;
00043     //define Id, and the vector of each cell
00044
00045     for (int k = 0; k < ncellz; k++)
00046
00047         for (int j = 0; j < ncelly; j++)
00048
00049             for (int i = 0; i < ncellx; i++) {
00050
00051
00052                 cells[counter].Id = k * ncelly * ncellx + j * ncellx + i;
00053                 cells[counter].Vec[0] = double(i + 1) * dlx;
00054                 cells[counter].Vec[1] = double(j + 1) * dly;
00055                 cells[counter].Vec[2] = double(k + 1) * dlz;
00056
00057
00058
00059                 counter = counter + 1;
00060             }
00061
00062
00063
00064
00065
00066     //define the first neighbours of all nodes
00067
00068
00069     int icell, jcell, kcell, klower, kupper, jlower, jupper, ilower, iupper, c_id;
00070
00071     for (kcell = 0; kcell < ncellz; kcell++) {
00072
00073         kupper = kcell + 1;
00074         if ((kcell + 1) > (ncellz - 1)) kupper = 0;
00075         klower = kcell - 1;
00076         if ((kcell - 1) < 0) klower = ncellz - 1;
00077
00078         for (jcell = 0; jcell < ncelly; jcell++) {
00079
00080             jupper = jcell + 1;
00081             if ((jcell + 1) > (ncelly - 1)) jupper = 0;
00082             jlower = jcell - 1;
00083             if ((jcell - 1) < 0) jlower = ncelly - 1;
00084

```

```

00085
00086     for (icell = 0; icell < ncellx; icell++) {
00087
00088         iupper = icell + 1;
00089         if ((icell + 1) > ncellx - 1) iupper = 0;
00090         ilower = icell - 1;
00091         if ((icell - 1) < 0) ilower = ncellx - 1;
00092
00093         c_id = (kcell) * ncelly * ncellx + (jcell) * ncellx + icell;
00094
00095         // 1: z, y, x:
00096         //neigh[cell_id][0] = (kcell)*ncelly*ncellx + (jcell)*ncellx + icell;
00097         cells[c_id].neigh[0] = (kcell) * ncelly * ncellx + (jcell) * ncellx + icell;
00098         //2: same z, same y, x+
00099         //neigh[cell_id][1] = (kcell)*ncelly*ncellx + (jcell)*ncellx + iupper;
00100         cells[c_id].neigh[1] = (kcell) * ncelly * ncellx + (jcell) * ncellx + iupper;
00101         // 3: same z, same y, x-
00102         //neigh[cell_id][2] = (kcell)*ncelly*ncellx + (jcell)*ncellx + ilower;
00103         cells[c_id].neigh[2] = (kcell) * ncelly * ncellx + (jcell) * ncellx + ilower;
00104         //4: same z, y+, same x
00105         //neigh[cell_id][3] = (kcell)*ncelly*ncellx + (jupper)*ncellx + icell;
00106         cells[c_id].neigh[3] = (kcell) * ncelly * ncellx + (jupper) * ncellx + icell;
00107         //5: same z, y-, same x
00108         //neigh[cell_id][4] = (kcell)*ncelly*ncellx + (jlower)*ncellx + icell;
00109         cells[c_id].neigh[4] = (kcell) * ncelly * ncellx + (jlower) * ncellx + icell;
00110         //6: same z, y+, x+
00111         //neigh[cell_id][5] = (kcell)*ncelly*ncellx + (jupper)*ncellx + iupper;
00112         cells[c_id].neigh[5] = (kcell) * ncelly * ncellx + (jupper) * ncellx + iupper;
00113         //7: same z, y+, x-
00114         //neigh[cell_id][6] = (kcell)*ncelly*ncellx + (jupper)*ncellx + ilower;
00115         cells[c_id].neigh[6] = (kcell) * ncelly * ncellx + (jupper) * ncellx + ilower;
00116         //8: same z, y-, x+
00117         //neigh[cell_id][7] = (kcell)*ncelly*ncellx + (jlower)*ncellx + iupper;
00118         cells[c_id].neigh[7] = (kcell) * ncelly * ncellx + (jlower) * ncellx + iupper;
00119         //9: same z, y-, x-
00120         //neigh[cell_id][8] = (kcell)*ncelly*ncellx + (jlower)*ncellx + ilower;
00121         cells[c_id].neigh[8] = (kcell) * ncelly * ncellx + (jlower) * ncellx + ilower;
00122
00123         //10: z+1, y, x :
00124         //neigh[cell_id][9] = (kupper)*ncelly*ncellx + (jcell)*ncellx + icell;
00125         cells[c_id].neigh[9] = (kupper) * ncelly * ncellx + (jcell) * ncellx + icell;
00126         //11: z+1, same y, x+
00127         //neigh[cell_id][10] = (kupper)*ncelly*ncellx + (jcell)*ncellx + iupper;
00128         cells[c_id].neigh[10] = (kupper) * ncelly * ncellx + (jcell) * ncellx + iupper;
00129         //12: z+1, same y, x-
00130         //neigh[cell_id][11] = (kupper)*ncelly*ncellx + (jcell)*ncellx + ilower;
00131         cells[c_id].neigh[11] = (kupper) * ncelly * ncellx + (jcell) * ncellx + ilower;
00132         //13: z+1, y+, same x
00133         //neigh[cell_id][12] = (kupper)*ncelly*ncellx + (jupper)*ncellx + icell;
00134         cells[c_id].neigh[12] = (kupper) * ncelly * ncellx + (jupper) * ncellx + icell;
00135         //14: z+1, y-, same x
00136         //neigh[cell_id][13] = (kupper)*ncelly*ncellx + (jlower)*ncellx + icell;
00137         cells[c_id].neigh[13] = (kupper) * ncelly * ncellx + (jlower) * ncellx + icell;
00138         //15: z+1, y+, x+
00139         //neigh[cell_id][14] = (kupper)*ncelly*ncellx + (jupper)*ncellx + iupper;
00140         cells[c_id].neigh[14] = (kupper) * ncelly * ncellx + (jupper) * ncellx + iupper;
00141         //16: z+1, y+, x-
00142         //neigh[cell_id][15] = (kupper)*ncelly*ncellx + (jupper)*ncellx + ilower;
00143         cells[c_id].neigh[15] = (kupper) * ncelly * ncellx + (jupper) * ncellx + ilower;
00144         //17: z+1, y-, x+
00145         //neigh[cell_id][16] = (kupper)*ncelly*ncellx + (jlower)*ncellx + iupper;
00146         cells[c_id].neigh[16] = (kupper) * ncelly * ncellx + (jlower) * ncellx + iupper;
00147         //18: z+1, y-, x-
00148         //neigh[cell_id][17] = (kupper)*ncelly*ncellx + (jlower)*ncellx + ilower;
00149         cells[c_id].neigh[17] = (kupper) * ncelly * ncellx + (jlower) * ncellx + ilower;
00150
00151         //19: z-1, y, x :
00152         //neigh[cell_id][18] = (klower)*ncelly*ncellx + (jcell)*ncellx + icell;
00153         cells[c_id].neigh[18] = (klower) * ncelly * ncellx + (jcell) * ncellx + icell;
00154         //20: z-1, same y, x+
00155         //neigh[cell_id][19] = (klower)*ncelly*ncellx + (jcell)*ncellx + iupper;
00156         cells[c_id].neigh[19] = (klower) * ncelly * ncellx + (jcell) * ncellx + iupper;
00157         //21: z-1, same y, x-
00158         //neigh[cell_id][20] = (klower)*ncelly*ncellx + (jcell)*ncellx + ilower;
00159         cells[c_id].neigh[20] = (klower) * ncelly * ncellx + (jcell) * ncellx + ilower;
00160         //22: z-1, y+, same x
00161         //neigh[cell_id][21] = (klower)*ncelly*ncellx + (jupper)*ncellx + icell;
00162         cells[c_id].neigh[21] = (klower) * ncelly * ncellx + (jupper) * ncellx + icell;
00163         //23: z-1, y-, same x
00164         //neigh[cell_id][22] = (klower)*ncelly*ncellx + (jlower)*ncellx + icell;
00165         cells[c_id].neigh[22] = (klower) * ncelly * ncellx + (jlower) * ncellx + icell;
00166         //24: z-1, y+, x+
00167         //neigh[cell_id][23] = (klower)*ncelly*ncellx + (jupper)*ncellx + iupper;
00168         cells[c_id].neigh[23] = (klower) * ncelly * ncellx + (jupper) * ncellx + iupper;
00169         //25: z-1, y+, x-
00170         //neigh[cell_id][24] = (klower)*ncelly*ncellx + (jupper)*ncellx + ilower;
00171         cells[c_id].neigh[24] = (klower) * ncelly * ncellx + (jupper) * ncellx + ilower;

```

```

00172             //26: z-1, y-, x+
00173             //neigh[cell_id][25] = (klower)*ncelly*ncellx + (jlower)*ncellx + iupper;
00174             cells[c_id].neigh[25] = (klower) * ncellx * ncellx + (jlower) * ncellx + iupper;
00175             //27: z-1, y-, x-
00176             //neigh[cell_id][26] = (klower)*ncelly*ncellx + (jlower)*ncellx + ilower;
00177             cells[c_id].neigh[26] = (klower) * ncellx * ncellx + (jlower) * ncellx + ilower;
00178         }
00179     }
00180 }
00181 }
00182 }
00183 }
00184 }
00185 }
00186 }
00187 }
00188 }
00189 int Grid::find_grid_cell(const double &xnode, const double &ynode,
00190                         const double &znode) {
00191     double xid = xnode * idlx;
00192     double yid = ynode * idly;
00193     double zid = znode * idlz;
00194     return(int(zid)*(ncelly)*(ncellx) + int(yid) * ncellx + int(xid));
00195 }
00196 }
00197 }
00198 }
00199 }
00200 Grid::~Grid(){
00201     free(cells);
00202 }
00203     return;
00204 }
00205 }

```

5.25 grid.h File Reference

The class of the non-bonded energy estimation grid.

Classes

- struct [NetworkNS::sgrid_cell](#)
 - class [NetworkNS::Grid](#)
- The nonbonded 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

```
00001
00014 #ifndef GRID_H
00015 #define GRID_H
00016
00017
00018 namespace NetworkNS {
00019
00020
00023     typedef struct sgrid_cell {
00024         int Id;
00025         double Vec[3];
00026         int neigh[27];
00027     } grid_cell;
00028
00033     class Grid {
00034     public:
00035
00036         Grid(double, double, double, int, int, int);
00037         virtual ~Grid();
00038
00039         double dlx;
00040         double dly;
00041         double dlz;
00042
00043         double idlx;
00044
00045         double idly;
00046
00047         double idlz;
00048
00049         double vcell;
00050         double ivcell;
00051
00052         int find_grid_cell(const double &xnode, const double &ynode, const double &znode);
00053
00054         int ncellx;
00055         int ncelly;
00056         int ncellz;
00057         int ncells;
00058
00059         grid_cell *cells;
00060     };
00061 }
00062
00063
00064
00065
00066
00067
00068
00069
00070
00071 #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

```

00001
00011 #include <iostream>
00012 #include <list>
00013 #include <vector>
00014 #include <cmath>
00015 #include "constants.h"
00016 #include "distributions.h"
00017 #include "b3D_integrator.h"
00018 #include "network.h"
00019 #include "netmin.h"
00020 #include "hopping.h"
00021
00022 #include "stdlib.h"
00023
00024 using namespace std;
00025 using namespace NetworkNS;
00026
00027
00028 namespace NetworkNS {
00029
00031     Hopping::Hopping(double hopping_rate_constant) {
00032
00033         /* Open a file to write the life-time of slip-springs. */
00034         lifetimes_file.open("ss_lifetimes.txt", ofstream::out);
00035         if (not lifetimes_file.good())
00036             cout << "Specified slip-springs lifetime file does not exist!" << endl;
00037
00038         events_file.open("events.txt", ofstream::out);
00039         if (not events_file.good())
00040             cout << "Specified events file does not exist!" << endl;
00041
00042         nu_hopping_times_exp_of_barrier = hopping_rate_constant; // s-1
00043
00044         return;
00045     }
00046
00048     Hopping::~Hopping() {
00049         /* Close the lifetimes file. */

```

```

00050     lifetimes_file.close();
00051     // gvog: Close the events file:
00052     events_file.close();
00053
00054     return;
00055 }
00056
00057
00065 void Hopping::hopping_step(NetwMin *netapp, const cb3D_integrator *b3D,
00066     double *pos_array, double temperature, double elapsed_time) {
00067
00092     double dr[3], rsq;
00093     unsigned int transitions_performed = 0; // transitions counter
00094     double time_in_seconds = elapsed_time * 1.e-12; // simulation time in s
00095     double feng_old = 0.0;
00096
00097     #ifndef CONST_SL_SCHEME
00098         unsigned int new_slipsprings = 0;
00099     #endif
00100
00108     // gvog: A list of strands to be deleted at the current time step.
00109     #ifdef CONST_SL_SCHEME
00110         std::list<std::pair<tStrand *, unsigned int > > to_be_deleted;
00111     #else
00112         std::list<tStrand *> to_be_deleted;
00113     #endif
00114
00115
00116     /* gvog: Loop over all slip-springs. */
00117     for (std::list<tStrand *>::iterator it = netapp->network->
pslip_springs.begin();
00118         it != netapp->network->pslip_springs.end(); ++it) {
00119
00120         //compute the initial free-energy of the current slip-spring
00121         dr[0] = pos_array[3 * ((*it)->pEnds[0]->Id - 1) + 0]
00122             - pos_array[3 * ((*it)->pEnds[1]->Id - 1) + 0];
00123
00124         dr[1] = pos_array[3 * ((*it)->pEnds[0]->Id - 1) + 1]
00125             - pos_array[3 * ((*it)->pEnds[1]->Id - 1) + 1];
00126
00127         dr[2] = pos_array[3 * ((*it)->pEnds[0]->Id - 1) + 2]
00128             - pos_array[3 * ((*it)->pEnds[1]->Id - 1) + 2];
00129
00130         // gvog: Ask for the shortest distance between the two beads:
00131         netapp->domain->minimum_image(dr[0], dr[1], dr[2]);
00132
00133         // Calculate the free energy the spring contributes to the total free energy of the system.
00134         #ifdef FENE_SLS
00135             feng_old = e_fene (dr, (*it)->spring_coeff, (*it)->sq_end_to_end, temperature);
00136         #else
00137             feng_old = e_gaussian(dr, (*it)->spring_coeff, (*it)->sq_end_to_end, temperature);
00138         #endif
00139
00140         double cur_slipspring_sq_distance = dr[0]*dr[0] + dr[1]*dr[1] + dr[2]*dr[2];
00141
00142         // gvog: Calculate the exponential of the slip-spring free energy.
00143         double spring_boltz_factor = exp(feng_old/boltz_const_kJoule_molK/
temperature);
00144         // gvog: Calculate the probability in a timestep of Delta t
00145         double hopping_prob = nu_hopping_times_exp_of_barrier * spring_boltz_factor * time_in_seconds;
00146
00147         /* REMINDER:
00148          * Type = 1 for chain ends.
00149          * Type = 2 for internal beads.
00150          * Type = 3 for crosslinks.
00151          */
00152
00153         // gvog: Allocate a list of possible candidates to jump on for every end of the slip-spring:
00154         vector<list<pair<tNode*, double> > > sls_attchmnts(2);
00155
00156         /* Loop over both ends of the slip-spring. */
00157         for (unsigned int iend = 0; iend < 2; iend++) {
00158
00159             tNode* cur_sls_end = (*it)->pEnds[iend];
00160
00161             /* Check whether the slip-spring has any of its ends connected to a chain end. */
00162             if (cur_sls_end->Type == 1)
00163                 /* The one possible point of attachment is the vacuum, so set it accordingly. */
00164                 sls_attchmnts[iend].push_back(pair<tNode*,double>(static_cast<tNode *>(0), hopping_prob));
00165
00166             if (cur_sls_end->Type != 3) {
00167                 // Loop over all strands connected to the other end of the slip-spring
00168                 for (vector<tStrand *>::iterator end_inc_strand = cur_sls_end->
pStrands.begin();
00169                     end_inc_strand != cur_sls_end->pStrands.end(); ++end_inc_strand){
00170
00171                     // check which end of the incident strand we should consider:

```

```

00172         if ((*end_inc_strand)->pEnds[0] == cur_sls_end) {
00173             if ((*end_inc_strand)->pEnds[1]->Type != 3)
00174                 sls_attachmnts[iend].push_back(pair<tNode*, double> ((*end_inc_strand)->pEnds[1],
hopping_prob));
00175         }
00176         else {
00177             if ((*end_inc_strand)->pEnds[0]->Type != 3)
00178                 sls_attachmnts[iend].push_back(pair<tNode*, double> ((*end_inc_strand)->pEnds[0],
hopping_prob));
00179         }
00180     }
00181 }
00182 else /* (*it)->pEnds[0]->Type == 3 */
00183     cout << "?: hopping.cpp: Slip-spring attached to crosslink detected!\n";
00184 }
00185
00186 // #define DEBUG_HOPPING
00187 #ifdef DEBUG_HOPPING
00188     cout << " ---- ---- ---- ----" << endl;
00189     cout << "Slip - spring: " << (*it)->Id << endl;
00190     cout << "possible transitions for left end: " ;
00191     for (list<pair<tNode*, double> >::iterator isls = sls_attachmnts[0].begin();
00192          isls != sls_attachmnts[0].end(); ++isls){
00193         if ((*isls).first != 0)
00194             cout << "( " << (*isls).first->Id << ", " << (*isls).second << "), ";
00195         else
00196             cout << "( vacuum, " << (*isls).second << "), ";
00197     }
00198     cout << endl;
00199
00200     cout << "possible transitions for right end: ";
00201     for (list<pair<tNode*, double> >::iterator isls = sls_attachmnts[1].begin();
00202          isls != sls_attachmnts[1].end(); ++isls){
00203         if ((*isls).first != 0)
00204             cout << "( " << (*isls).first->Id << ", " << (*isls).second << "), ";
00205         else
00206             cout << "( vacuum, " << (*isls).second << "), ";
00207     }
00208     cout << endl << " ---- ---- ---- ----" << endl << endl;
00209 #endif
00210
00211     // gvog: Check whether the sum of the transition probabilities is greater than 1.0.
00212     double summer = 0.0;
00213     for (vector<list<pair<tNode*, double > > >::iterator iend = sls_attachmnts.begin(); iend !=
sls_attachmnts.end(); ++iend)
00214         for (list<pair<tNode*, double > >::iterator end_ineigh = (*iend).begin(); end_ineigh != (*iend)
.end(); ++end_ineigh)
00215             summer += (*end_ineigh).second;
00216
00217     // gvog: If the sum of the probabilities exceeded 1.0, normalize it to unity.
00218     if (summer > 1.0) {
00219         // TODO: Replace the following quick fix with something smarter:
00220
00221         cout << "?: warning: Please change the transition probabilities. Sum = " << summer << " > 1.0\n";
00222     }
00223     for (vector<list<pair<tNode*, double > > >::iterator iend = sls_attachmnts.begin(); iend !=
sls_attachmnts.end(); ++iend)
00224         for (list<pair<tNode*, double > >::iterator end_ineigh = (*iend).begin(); end_ineigh !=
(*iend).end(); ++end_ineigh)
00225             (*end_ineigh).second /= summer;
00226 }
00227
00228
00229 bool perform_transition = false;
00230 double cum_prob = 0.0;
00231 double ran_num = netapp->my_rnd_gen->uniform();
00232
00233 unsigned int end_to_jump = 0;
00234 tNode* target_node = 0;
00235
00236 for (unsigned int iend = 0; iend < 2; iend++)
00237     for (list<pair<tNode*, double > >::iterator end_ineigh = sls_attachmnts[iend].begin();
end_ineigh != sls_attachmnts[iend].end(); ++end_ineigh)
00238
00239         if (!perform_transition) {
00240             // gvog: Add to the cumulative probability:
00241             cum_prob += (*end_ineigh).second;
00242
00243             if (cum_prob > ran_num) {
00244                 // Here I have to store the pair (itr,jtr) and break.
00245                 end_to_jump = iend;
00246                 target_node = (*end_ineigh).first;
00247
00248                 perform_transition = true;
00249             }
00250         }
00251 }

```

```

00252
00253 // gvog: In case the probability is too small, try another loop iteration.
00254 // gvog: pre 2016/02/02 - bug pointed by Aris Sgouros:
00255 // if ((cum_prob < 1.e-12) || (end_to_jump == 0 && target_node == 0))
00256 //     continue;
00257 // gvog: post 2016/02/02: no boolean condition
00258
00259 bool destroy_slpsprng = false;
00260
00261 if (perform_transition) {
00262
00263     // gvog: Increase the counter of transitions performed.
00264     transitions_performed++;
00265
00266     // gvog: Write the ID of the slip-spring to the file:
00267     events_file << (*it)->Id << " " << ((double)b3D->bd_cur_step) << endl;
00268
00269 #ifndef DEBUG_HOPPING
00270     if (target_node)
00271         cout << "end " << end_to_jump << " has hopped to " << target_node->
00272             Id << endl;
00273     else
00274         cout << "end " << end_to_jump << " has gone to vacuum" << endl;
00275     cout << "random number = " << ran_num << endl;
00276     exit(0);
00277 #endif
00278
00279     /* In case the attachment point exists: */
00280     if (target_node)
00281         (*it)->pEnds[end_to_jump] = target_node;
00282     else
00283         destroy_slpsprng = true;
00284
00285     if (destroy_slpsprng)
00286         // gvog: The slip-spring can be deleted only if its distance is smaller than the attempt
00287         radius:
00288             if (cur_slipspring_sq_distance < (hopping_attempt_radius * hopping_attempt_radius))
00289 #ifndef CONST_SL_SCHEME
00290             to_be_deleted.push_back(std::pair<tStrand *, unsigned int>((*it), end_to_jump));
00291 #else
00292             to_be_deleted.push_back((*it));
00293 #endif
00294         }
00295     }
00296
00297 #ifndef CONST_SL_SCHEME
00298     for (std::list<tStrand *>::iterator it = to_be_deleted.begin();
00299          it != to_be_deleted.end(); ++it) {
00300
00301         // Find the element to be deleted:
00302         for (std::list<tStrand *>::iterator jt = netapp->network->strands.begin();
00303              jt != netapp->network->strands.end(); ++jt)
00304             if (&(*jt) == (*it)) {
00305                 netapp->network->strands.erase(jt);
00306                 break;
00307             }
00308
00309         /* Write the lifetime of the slip-spring to the file: */
00310         lifetimes_file << (int) (b3D->bd_cur_step - (*it)->tcreation) << endl;
00311
00312         // gvog: Finally, we can delete the slip-spring from the pointer array:
00313         netapp->network->pslip_springs.remove((*it));
00314     }
00315 #else
00316     for (std::list<std::pair<tStrand *, unsigned int>>::iterator it = to_be_deleted.begin();
00317          it != to_be_deleted.end(); ++it) {
00318
00319         /*
00320          * gvog: In the constant number of slip-springs scheme, we have to check whether the
00321          * end-to-end distance of the spring to be destroyed is smaller than the capture
00322          * radius, for the detailed balance to hold.
00323          */
00324         dr[0] = pos_array[3 * ((*it).first->pEnds[1]->Id - 1) + 0]
00325             - pos_array[3 * ((*it).first->pEnds[0]->Id - 1) + 0];
00326
00327         dr[1] = pos_array[3 * ((*it).first->pEnds[1]->Id - 1) + 1]
00328             - pos_array[3 * ((*it).first->pEnds[0]->Id - 1) + 1];
00329
00330         dr[2] = pos_array[3 * ((*it).first->pEnds[1]->Id - 1) + 2]
00331             - pos_array[3 * ((*it).first->pEnds[0]->Id - 1) + 2];
00332
00333         /* Apply minimum image convention to the separation vector. */
00334         netapp->domain->minimum_image(dr[0], dr[1], dr[2]);
00335         rsq = dr[0] * dr[0] + dr[1] * dr[1] + dr[2] * dr[2];
00336

```



```

00337      /*
00338      * gvog: Only if the distance of the strand to be deleted is smaller than the tube diameter the
00339      *       whole deletion/creation procedure can proceed.
00340      */
00341      if (rsq <= hopping_attempt_radius * hopping_attempt_radius) {
00342
00343          // gvog: Initialize Rosenbluth weights to zero.
00344          double rosen_old = 0.0, rosen_new = 0.0;
00345
00346          // gvog: Calculate the Rosenbluth weight in the old configuration:
00347          for (std::list<tNode>::iterator jat = netapp->network->nodes.begin();
00348               jat != netapp->network->nodes.end(); ++jat) {
00349
00350              /*
00351              * gvog: We allow internal and end-end connections. Only crosslinks (Type = 3) are excluded
00352              *       from possible candidates. Moreover, connections along the same chain are allowed,
00353              *       even with the same bead.
00354              */
00355              #ifdef ALLOW_INTRAMOLECULAR
00356                  // gvog: Here we exclude sites belonging to the same chain:
00357                  if ((*jat).Type == 3)
00358              #else
00359                  if (((*jat).Type == 3) || ((*jat).OrChains[0] == (*it).first->pEnds[(*it).second]->OrChains[
00360                      0]))
00361              #endif
00362                      continue;
00363
00364                  /* check the distance from the end to the bead: */
00365                  dr[0] = pos_array[3 * ((*jat).Id - 1) + 0]
00366                      - pos_array[3 * ((*it).first->pEnds[(*it).second]->Id - 1) + 0];
00367
00368                  dr[1] = pos_array[3 * ((*jat).Id - 1) + 1]
00369                      - pos_array[3 * ((*it).first->pEnds[(*it).second]->Id - 1) + 1];
00370
00371                  dr[2] = pos_array[3 * ((*jat).Id - 1) + 2]
00372                      - pos_array[3 * ((*it).first->pEnds[(*it).second]->Id - 1) + 2];
00373                  /* Apply minimum image convention to the separation vector. */
00374                  netapp->domain->minimum_image(dr[0], dr[1], dr[2]);
00375                  rsq = dr[0] * dr[0] + dr[1] * dr[1] + dr[2] * dr[2];
00376
00377                  // gvog: Add the contribution of the strand to the Rosenbluth weight
00378                  rosen_old += exp(-e_gaussian(dr, (*it).first->spring_coeff, (*it).first->
00379                      sq_end_to_end, temperature)
00380                      / boltz_const_kJoule_molK / temperature);
00381              }
00382
00383              // gvog: NEW CONFIGURATION
00384              // gvog: Select randomly one of the chain ends
00385              unsigned int nends = netapp->network->sorted_chains.size() * 2;
00386              unsigned int iend = (unsigned int) (netapp->my_rnd_gen->
00387                  uniform() * double(nends));
00388              unsigned int ichain = (unsigned int) (iend / 2);
00389
00390              tNode* new_end = 0;
00391
00392              // gvog: Bug fix proposed by Aris Sgouros on January, 29
00393              if (iend % 2 == 0)
00394                  // gvog: Select the end of the chain:
00395                  new_end = netapp->network->sorted_chains[ichain].back()->pEnds[1];
00396              else
00397                  // gvog: Select the start of the chain:
00398                  new_end = netapp->network->sorted_chains[ichain].front()->pEnds[0];
00399
00400              /* Create a vector to store the candidates for slip-spring bridging. */
00401              std::vector<std::pair<tNode *, double> > candidates;
00402              std::pair<tNode *, double> cur_cand;
00403
00404              /* Loop over all beads: */
00405              for (std::list<tNode>::iterator jat = netapp->network->nodes.begin();
00406                   jat != netapp->network->nodes.end(); ++jat) {
00407
00408                  // gvog: check whether the candidate is a crosslink, or belongs to the same chain.
00409
00410              #ifdef ALLOW_INTRAMOLECULAR
00411                  if ((*jat).Type == 3)
00412              #else
00413                  if (((*jat).Type == 3) || ((*jat).OrChains[0] == (new_end->
00414                      OrChains[0])))
00415              #endif
00416                      continue;
00417
00418                  /* check the distance from the end to the bead: */
00419                  dr[0] = pos_array[3 * ((*jat).Id - 1) + 0]
00420                      - pos_array[3 * (new_end->Id - 1) + 0];
00421
00422                  dr[1] = pos_array[3 * ((*jat).Id - 1) + 1]
00423                      - pos_array[3 * (new_end->Id - 1) + 1];

```

```

00419         dr[2] = pos_array[3 * ((*jat).Id - 1) + 2]
00420             - pos_array[3 * (new_end->Id - 1) + 2];
00421
00422
00423         /* Apply minimum image convention to the separation vector. */
00424         netapp->minimum_image(dr[0], dr[1], dr[2]);
00425         rsq = dr[0] * dr[0] + dr[1] * dr[1] + dr[2] * dr[2];
00426
00427         /* If the distance is smaller than the tube diameter of the polyisoprene,
00428          * append this neighbor to the candidates list. */
00429         if (rsq <= hopping_attempt_radius * hopping_attempt_radius) {
00430             cur_cand.first = &(*jat);
00431             cur_cand.second = e_gaussian(dr, (*it).first->spring_coeff, (*it).first->
sq_end_to_end, temperature);
00432
00433             rosen_new += exp(-cur_cand.second / boltz_const_kJoule_molK /
temperature);
00434
00435             candidates.push_back(cur_cand);
00436         }
00437     }
00438
00439     /* Calculate the cumulative probability of each candidate: */
00440     double cum_prob = 0.0, ran_num = netapp->my_rnd_gen->
uniform();
00441     tNode *sel_candidate = 0;
00442     for (std::vector<std::pair<tNode *, double> >::iterator icand = candidates.begin();
00443          icand != candidates.end(); ++icand) {
00444         cum_prob += exp(-(*icand).second / boltz_const_kJoule_molK /
temperature) / rosen_new;
00445         if (cum_prob > ran_num) {
00446             sel_candidate = (*icand).first;
00447             feng_new = (*icand).second;
00448             break;
00449         }
00450     }
00451
00452     // gvog: rosen_old -> old rosenbluth weight
00453     // gvog: rosen_new -> new rosenbluth weight
00454     // gvog: feng_old -> old energy
00455     // gvog: feng_new -> new energy
00456     double criterion = exp((feng_new-feng_old)/boltz_const_kJoule_molK/
temperature)*rosen_new/rosen_old;
00457
00458     // gvog: Ask for a random number:
00459     ran_num = netapp->my_rnd_gen->uniform();
00460     if (criterion >= ran_num) {
00461         // gvog: The move has been accepted, update the connectivity of the slip-spring.
00462         (*it).first->pEnds[0] = new_end;
00463         (*it).first->pEnds[1] = sel_candidate;
00464     }
00465
00466 }
00467 else
00468     cout << "## warning: increase capture radius for the hopping scheme." << endl;
00469 }
00470 #endif
00471
00472 // cout << "##: hopping.cpp: slip-springs deleted = " << to_be_deleted.size() << endl;
00473
00474
00475 #ifndef CONST_SL_SCHEME
00494     /* gvog:
00495      * Slip-spring creation event:
00496      * 1. loop over all chain ends of the system.
00497      * 2. for every chain end search in a sphere of prescribed radius for other beads
00498      * 3. calculate the probability of creating a new slip-spring and select one of the candidates at
random
00500      */
00501     std::vector<std::list<tStrand * > >::iterator ich;
00502     std::list<tNode>::iterator jat;
00503     std::vector<tNode * > ich_ends(2);
00504
00505
00506     for (ich = netapp->network->sorted_chains.begin();
00507          ich != netapp->network->sorted_chains.end(); ++ich) {
00508
00509         /* Extract the first and the last bead of chain ich: */
00510         ich_ends[0] = (*ich).front()->pEnds[0];
00511         ich_ends[1] = (*ich).back()->pEnds[1];
00512
00513
00514         /* Loop over both ends of the chain: */
00515         for (std::vector<tNode * >::iterator iend = ich_ends.begin();
00516              iend != ich_ends.end(); ++iend) {
00517

```

```

00518         /* Create a vector to store the candidates for slip-spring bridging. */
00519         std::list<tNode *> candidates;
00520
00521         /* Loop over all beads of the system: */
00522         for (jat = netapp->network->nodes.begin();
00523              jat != netapp->network->nodes.end(); ++jat) {
00524
00525             // gvog: Do not construct a slip-spring with a crosslink:
00526             #ifdef ALLOW_INTRAMOLECULAR
00527                 if ((*jat).Type == 3)
00528             #else
00529                 if ((*jat).Type == 3 || ((*jat).OrChains[0] == (*iend)->OrChains[0]))
00530             #endif
00531                 continue;
00532
00533             /* check the distance from the end to the bead: */
00534             dr[0] = pos_array[3 * ((*jat).Id - 1) + 0]
00535                 - pos_array[3 * ((*iend)->Id - 1) + 0];
00536
00537             dr[1] = pos_array[3 * ((*jat).Id - 1) + 1]
00538                 - pos_array[3 * ((*iend)->Id - 1) + 1];
00539
00540             dr[2] = pos_array[3 * ((*jat).Id - 1) + 2]
00541                 - pos_array[3 * ((*iend)->Id - 1) + 2];
00542
00543
00544             /* Apply minimum image convention to the separation vector. */
00545             netapp->domain->minimum_image(dr[0], dr[1], dr[2]);
00546             rsq = dr[0]*dr[0] + dr[1]*dr[1] + dr[2]*dr[2];
00547
00548             /* If the distance is smaller than the tube diameter of the polyisoprene,
00549              * append this neighbor to the candidates list. */
00550             if (rsq <= hopping_attempt_radius * hopping_attempt_radius)
00551                 candidates.push_back(&(*jat));
00552         }
00553
00554         // gvog: Calculate the creation probability for the current chain end:
00555         double creation_prob = nu_hopping_times_exp_of_barrier * (double)candidates.size() *
time_in_seconds;
00556
00557         // gvog: and check that it is lower than 1.0
00558         if (creation_prob > 1.0) {
00559             cout << " #: hopping.cpp: Please change the rate prefactor. Creation probability = " <<
creation_prob << " > 1.0\n";
00560             //exit(EXIT_FAILURE);
00561         }
00562
00563         // gvog: ask for a random number to decide whether the creation should be attempted:
00564         double ran_num = netapp->my_rnd_gen->uniform();
00565         tNode *sel_candidate = 0;
00566         unsigned int isel_cand = 0;
00567
00568         if (creation_prob > ran_num){
00569             // gvog: we have to select one of the candidates to bridge our end with:
00570             ran_num = netapp->my_rnd_gen->uniform();
00571             isel_cand = (unsigned int)(ran_num*(double)candidates.size());
00572
00573             unsigned int icand = 0;
00574             for (list<tNode *>::iterator it = candidates.begin(); it!=candidates.end(); ++it){
00575                 sel_candidate = (*it);
00576                 icand++;
00577                 if (icand > isel_cand)
00578                     break;
00579             }
00580         }
00581
00582
00583         /* Create a new strand and append it the appropriate lists. */
00584         if (sel_candidate) {
00585             /* Here I have to create a new strand, and update the corresponding arrays.*/
00586             tStrand new_slip_spring;
00587             new_slip_spring.Id = netapp->network->strands.back().Id + new_slipsprings +
1;
00588             new_slip_spring.pEnds.resize(2);
00589             new_slip_spring.pEnds[0] = *iend;
00590             new_slip_spring.pEnds[1] = sel_candidate;
00591             new_slip_spring.slip_spring = true;
00592             new_slip_spring.OrChain = 0;
00593             new_slip_spring.tcreation = b3D->bd_cur_step;
00594
00595             if (netapp->network->pslip_springs.size() > 0){
00596                 new_slip_spring.spring_coeff = netapp->network->
pslip_springs.back()->spring_coeff;
00597                 new_slip_spring.sq_end_to_end = netapp->network->
pslip_springs.back()->sq_end_to_end;
00598                 new_slip_spring.kuhn_length = netapp->network->
pslip_springs.back()->kuhn_length;

```

```

00599         }
00600     else {
00601         cout << "#: (warning): The first slip-spring of the system has been initialized with "
00602             << "hard-coded coefficients.\n";
00603         new_slip_spring.spring_coeff = netapp->network->
strands.front().spring_coeff;
00604         new_slip_spring.kuhn_length = 9.58;
00605         new_slip_spring.sq_end_to_end = pi_tube_diameter *
pi_tube_diameter;
00606
00607         /* Also, create a new bond type: */
00608         tBond_type new_bond_type;
00609         new_bond_type.spring_coeff = new_slip_spring.
spring_coeff;
00610         new_bond_type.kuhn1 = new_slip_spring.kuhn_length;
00611         new_bond_type.sq_ete = new_slip_spring.sq_end_to_end;
00612
00613         netapp->network->bond_types.push_back(new_bond_type);
00614     }
00615
00616
00617     // gvog: and add it to the list of slip-springs
00618     netapp->network->strands.push_back(new_slip_spring);
00619     netapp->network->pslip_springs.push_back(&(netapp->
network->strands.back()));
00620     new_slipsprings++;
00621 }
00622 }
00623 }
00624
00625 //cout << "# hopping.cpp: new slip-springs created: " << new_slipsprings << endl;
00626
00627
00628 /* Re-count all springs in order to achieve continuous enumeration.*/
00629 unsigned int start_tag = netapp->network->strands.size() - netapp->
network->pslip_springs.size();
00630
00631 for (std::list<tStrand *>::iterator it = netapp->network->
pslip_springs.begin();
00632      it != netapp->network->pslip_springs.end(); ++it)
00633     (*it)->Id = start_tag++;
00634 #endif
00635
00636 //cout << "#: hopping.cpp: transitions performed = " << transitions_performed - to_be_deleted.size()
<< endl;
00637
00638     return;
00639 }
00640 }
00641 }
00642 }
00643

```

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

```

00001
00002 #ifndef HOPPING_H
00003 #define HOPPING_H
00004
00005 #include <fstream>
00006 #include <iostream>
00007 #include <string>
00008
00009 #include "b3D_integrator.h"
00010 #include "net_types.h"
00011 #include "network.h"
00012
00013 using namespace std;
00014 using namespace NetworkNS;
00015
00016 namespace NetworkNS{
00017
00018     class Hopping {
00019     public:
00020         Hopping(double);
00021         ~Hopping();
00022
00023         void hopping_step(class NetwMin *netapp, const class
00024         cb3D_integrator *b3D, double *pos_array,
00025                             double temperature, double elapsed_time);
00026
00027     private:
00028         std::ofstream lifetimes_file;
00029         std::ofstream events_file;
00030         double nu_hopping_times_exp_of_barrier;
00031     };
00032 }
00033
00034 #endif /* HOPPING_H */
00035

```

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

Copyright (c) 2012 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 [main.cpp](#).

5.31.3 Function Documentation

5.31.3.1 main()

```
int main (
    int argc,
    char ** argv )
```

Parameters

in	<i>argc</i>	The count of arguments provided through the command line.
in	<i>argv</i>	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]);
```

```

00043     else{
00044         cout << " #: Data file has not been specified from command line.\n";
00045         filename = "emsipon.data";
00046         cout << " #: --- ' " << filename << "' will be used as data file.\n";
00047     }
00048
00049     double creation_rate;
00050     if (argc > 2)
00051         creation_rate = StringToNumber<double>(argv[2]);
00052     else
00053         return(1);
00054
00055     if (argc > 3)
00056         bd3D_nsteps = atol(argv[3]);
00057     else
00058         bd3D_nsteps = 2000000000;
00059
00060     if (argc > 4)
00061         bd3D_write_every = atol(argv[4]);
00062     else
00063         bd3D_write_every = 1000;
00064
00065
00066 #ifdef FENE_SLS
00067     cout << " #: (warning): FENE potential will be used for slip-springs.\n";
00068 #endif
00069
00070     // Define a new pointer to the class:
00071     NetwMin netw_min(filename);
00072     cb3D_integrator b3D_integrator(&netw_min, 500.0, creation_rate);
00073
00074     cout << " #: Integrator has been initialized. Simulation will start for "<<bd3D_nsteps<< " steps.\n";
00075     cout << " #: The rate for the slip-spring creation is: " << creation_rate << endl;
00076
00077     // Simulation starts here:
00078     double timestep = 1.0; // integration timestep in ps.
00079     cout << " #: The timestep of the integrator will be: " << timestep << " ps.\n";
00080     cout << " #: Report will be written every: " << bd3D_write_every << " steps.\n";
00081
00082
00083     b3D_integrator.integrate(bd3D_nsteps, timestep, bd3D_write_every);
00084
00085     // Close trajectory file:
00086     netw_min.my_traj_file->~dump();
00087
00088
00089     return (EXIT_SUCCESS);
00090 }
00091

```

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

```

00001
00014 #ifndef _NET_TYPES_H
00015 #define _NET_TYPES_H
00016
00017 #include <list>
00018 #include <vector>
00019
00020 // Forward declaration, in order to create a pointer...
00021 using namespace std;
00022
00023
00024 /* Forward declaration of structure (object) strand in order to
00025  * create a pointer to this kind inside structure (object) node. */
00026 struct sStrand;
00027 struct sChain; //new addition, November 2012
00028
00029
00033 typedef struct sNode{
00034     int Id;
00035     int Type;
00036
00040     double Pos[3];
00041
00042     std::vector<int> OrChains;
00043
00044     std::vector<sStrand*> pStrands;
00045
00046     std::vector<int> SubCh;
00047
00048     std::vector<sChain*> pChains;

```



```

00049
00050     int   node_cell;
00051
00053     double n_mass;
00054
00055     double mass;
00056
00057     double r_node;
00058
00060     double r_star;
00061 }tNode;
00064
00065
00067 typedef struct sStrand{
00068
00069     int Id;
00070
00071     int Type;
00072
00075     int OrChain;
00076
00077     bool slip_spring;
00078
00079     unsigned int tcreation;
00080
00081     double spring_coeff;
00082
00085     double sq_end_to_end;
00086
00088     double kuhn_length;
00089
00090     std::vector <tNode *> pEnds;
00091
00092     double * pChain;
00093
00094 } tStrand;
00095
00097 typedef std::list <tStrand> tSubCh;
00098
00100 typedef struct sBead_type{
00101
00102     double mass;
00103     double n_mass;
00104     double r_node;
00105 } tBead_type;
00106
00108 typedef struct sBond_type{
00109
00110     double spring_coeff;
00111     double sq_ete;
00114     double kuhnl;
00115 } tBond_type;
00116
00117 #endif /* _NET_TYPES_H */
00118

```

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

Copyright (c) 2013 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 [netmin.cpp](#).

5.36 netmin.cpp

```

00001
00014 #include <string>
00015 #include "netmin.h"
00016
00017 using namespace std;
00018
00019 namespace NetworkNS{
00020
00023     NetwMin::NetwMin(std::string filename){
00024
00025         domain = new Domain(filename);
00026         cout << " #: Domain characteristics have been read.\n";
00027         network = new Network(this,filename);
00028         cout << " #: Network has been read.\n";
00029         my_rnd_gen = new RanMars(11111);
00030
00031         cout << " #: Particles have been read.\n";
00032
00033         my_traj_file = new dump("dump_b3D.lammpstrj");
00034         cout << " #: 'dump_b3D.lammpstrj' will be used as trajectory file.\n";
00035
00036         grid = new Grid(domain->XBoxLen, domain->YBoxLen, domain->ZBoxLen, 5, 5, 5);
00037         write_network_to_lammps_data_file();
00038
00039         return;
00040     }
00041
00042
00043     void NetwMin::write_network_to_lammps_data_file() {
00044
00045         double wrapped_coords[3];
00046         int pcoeffs[3];
00047
00048         pcoeffs[0] = 0;
00049         pcoeffs[1] = 0;
00050         pcoeffs[2] = 0;
00051
00052         FILE *lammps_file = fopen("restart.data", "wt");
00053
00054
00055         /* Write header to LAMMPS data file: */
00056
00057         fprintf(lammps_file, "LAMMPS data file of crosslinked coarse-grained network.\n\n");
00058         fprintf(lammps_file, "%ld atoms\n", network->nodes.size());
00059         fprintf(lammps_file, "%ld bonds\n", network->strands.size());
00060         fprintf(lammps_file, "\n");
00061
00062         fprintf(lammps_file, "%lu atom types\n", network->node_types.size());
00063
00064         if (network->pslip_springs.size() > 0)
00065             fprintf(lammps_file, "2 bond types\n");
00066         else
00067             fprintf(lammps_file, "1 bond types\n");
00068         fprintf(lammps_file, "\n");
00069
00070         fprintf(lammps_file, "%lf %lf xlo xhi\n",
00071             domain->BoxLow[0], domain->BoxHigh[0]);
00072         fprintf(lammps_file, "%lf %lf ylo yhi\n",
00073             domain->BoxLow[1], domain->BoxHigh[1]);
00074         fprintf(lammps_file, "%lf %lf zlo zhi\n",
00075             domain->BoxLow[2], domain->BoxHigh[2]);
00076
00077
00078         fprintf(lammps_file, "\nMasses\n\n");
00079         for (unsigned int i = 0; i < network->node_types.size(); i++)
00080             fprintf(lammps_file, "%d \t %lf \t %-1.4e \t # n_Kuhns/bead - g/mol\n",
00081                 i+1, network->node_types[i].n_mass, network->node_types[i].mass);
00082
00083
00084         fprintf(lammps_file, "\nBond Coeffs\n\n");

```

```

00085     for (unsigned int i = 0; i < network->bond_types.size(); i++)
00086         fprintf(lammps_file,
00087             "%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",
00088             i+1, network->bond_types[i].spring_coeff,
00089             network->bond_types[i].sq_ete, network->bond_types[i].kuhn1);
00090
00091
00092     fprintf(lammps_file, "\nPair Coeffs\n\n");
00093     for (unsigned int i = 0; i < network->node_types.size(); i++)
00094         fprintf(lammps_file, "%d \t %lf \t # r_node = b*n_j^(1/2)\n", i+1,
00095             network->node_types[i].r_node);
00096
00097
00098     fprintf(lammps_file, "\nAtoms\n\n");
00099     int inode = 0;
00100     for (std::list<tNode> ::iterator it = network->nodes.begin();
00101          it != network->nodes.end(); ++it) {
00102         inode++;
00103         (*it).Id = inode;
00104
00105         wrapped_coords[0] = (*it).Pos[0];
00106         wrapped_coords[1] = (*it).Pos[1];
00107         wrapped_coords[2] = (*it).Pos[2];
00108         //domain->put_in_primary_box(wrapped_coords, pcoeffs);
00109
00110         fprintf(lammps_file, "%d %d %d 0.000 %.10f %.10f %.10f %d %d %d\n",
00111             (*it).Id, (*it).OrChains[0], (*it).Type,
00112             wrapped_coords[0], wrapped_coords[1], wrapped_coords[2],
00113             pcoeffs[0], pcoeffs[1], pcoeffs[2]);
00114     }
00115
00116     fprintf(lammps_file, "\nBonds\n\n");
00117
00118     int istrand = 0;
00119     for (std::list<tStrand> ::iterator it = network->strands.begin();
00120          it != network->strands.end(); ++it) {
00121         istrand++;
00122
00123         if ((*it).slip_spring)
00124             fprintf(lammps_file, "%d 2 %d %d # %d\n",
00125                 istrand, (*it).pEnds[0]->Id, (*it).pEnds[1]->Id, (*it).OrChain);
00126         else
00127             fprintf(lammps_file, "%d 1 %d %d # %d\n",
00128                 istrand, (*it).pEnds[0]->Id, (*it).pEnds[1]->Id, (*it).OrChain);
00129     }
00130     fclose(lammps_file);
00131
00132     return;
00133 }
00134
00135 }

```

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

Copyright (c) 2013 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 [netmin.h](#).

5.38 netmin.h

```

00001
00015 #ifndef NETWMIN_H
00016 #define NETWMIN_H
00017
00018 #include <vector>
00019
00020 #include "domain.h"
00021 #include "dump.h"
00022 #include "grid.h"
00023 #include "hopping.h"
00024 #include "rng.h"
00025 #include "network.h"
00026
00027
00028 namespace NetworkNS {
00029
00030
00036     class NetwMin {
00037     public:
00038         class Network *network;
00039
00041         class Domain *domain;
00042
00044         class Grid *grid;
00045
00047         class RanMars *my_rnd_gen;
00049         class dump *my_traj_file;
00050
00052         NetwMin(std::string);
00053
00055         void write_network_to_lammps_data_file();
00056
00060     };
00061 }
00062
00063 #endif /* NETWMIN_H */
00064

```

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

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 `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	<i>current</i>	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

```
00001
00015 #include <algorithm>
00016 #include <cstdio>
00017 #include <fstream>
00018 #include <iostream>
00019 #include <iterator>
00020 #include <list>
00021 #include <sstream>
00022 #include <vector>
00023 #include <cmath>
00024
```

```

00025 #include "constants.h"
00026 #include "netmin.h"
00027 #include "Auxiliary.h"
00028
00029
00030 using namespace std;
00031
00032
00033 namespace NetworkNS {
00034
00035     Network::~Network() {
00036         nodes.clear();
00037         strands.clear();
00038     }
00039
00040
00041     Network::Network(class NetwMin *netw_min, std::string filename) {
00042
00043         int nnodes = 0, nstrands = 0, atoms_line_start = 0, bonds_line_start = 0,
00044             bond_coeffs_line_start = 0, masses_line_start = 0, pair_coeffs_line_start = 0,
00045             atom_types_line = 0, bond_types_line = 0, ihelp;
00046
00047         nstrands = 0;
00048         cout << " #: Initializing network.\n";
00049         /* Define and open the desired file. */
00050         ifstream data_file(filename.c_str(), ifstream::in);
00051         /* Define an array of strings to hold the contents of the file. */
00052         std::vector<string> lines_of_file;
00053         std::vector<string> tokens;
00054
00055         /* Read and search file. */
00056         int iline = 0;
00057
00058         while (data_file.good()) {
00059             /* A temporary string for the current line of the file. */
00060             std::string current_line;
00061             getline(data_file, current_line);
00062             /* Add current line to file's array of lines. */
00063             lines_of_file.push_back(current_line);
00064
00065             if (current_line.find("atoms") != string::npos)
00066                 nnodes = atoi(current_line.c_str());
00067             if (current_line.find("bonds") != string::npos)
00068                 nstrands = atoi(current_line.c_str());
00069             if (current_line.find("Atoms") != string::npos)
00070                 atoms_line_start = iline;
00071             if (current_line.find("Bonds") != string::npos)
00072                 bonds_line_start = iline;
00073             if (current_line.find("Masses") != string::npos)
00074                 masses_line_start = iline;
00075             if ((current_line.find("Bond") != string::npos)
00076                 && (current_line.find("Coeffs") != string::npos))
00077                 bond_coeffs_line_start = iline;
00078             if ((current_line.find("Pair") != string::npos)
00079                 && (current_line.find("Coeffs") != string::npos))
00080                 pair_coeffs_line_start = iline;
00081             if ((current_line.find("atom") != string::npos)
00082                 && (current_line.find("types") != string::npos))
00083                 atom_types_line = iline;
00084             if ((current_line.find("bond") != string::npos)
00085                 && (current_line.find("types") != string::npos))
00086                 bond_types_line = iline;
00087
00088             iline++;
00089         }
00090         data_file.close();
00091
00092         /* Read the masses of the nodes from the corresponding session of the data file. */
00093         tokens = tokenize(lines_of_file[atom_types_line]);
00094         ihelp = atoi(tokens[0].c_str());
00095         cout << " #: --- Number of bead types: " << ihelp << " .\n";
00096         node_types.resize(ihelp);
00097         for (int i = 0; i < ihelp; i++) {
00098             tokens = tokenize(lines_of_file[masses_line_start + i + 2]);
00099             node_types[i].n_mass = atof(tokens[1].c_str());
00100             node_types[i].mass = atof(tokens[2].c_str());
00101
00102             tokens = tokenize(lines_of_file[pair_coeffs_line_start+i+2]);
00103             node_types[i].r_node = atof(tokens[1].c_str());
00104         }
00105
00106         /* Read the bond coefficients from the corresponding section of the data file. */
00107         tokens = tokenize(lines_of_file[bond_types_line]);
00108         ihelp = atoi(tokens[0].c_str());
00109         cout << " #: --- Number of bond types: " << ihelp << " .\n";
00110         bond_types.resize(ihelp);
00111         for (int i = 0; i < ihelp; i++){

```

```

00117         tokens = tokenize(lines_of_file[bond_coeffs_line_start + i + 2]);
00118         bond_types[i].spring_coeff = atof(tokens[1].c_str());
00119         bond_types[i].sq_ete = atof(tokens[2].c_str());
00120         bond_types[i].kuhn1 = atof(tokens[3].c_str());
00121     }
00122
00123     /* Read atom sections which corresponds to nodal points: */
00124     for (int i = 0; i < nnodes; i++) {
00125         /* Read and split the line from the input file: */
00126         tokens = tokenize(lines_of_file[atoms_line_start + 2 + i]);
00127
00128
00129         //int pbcx = atoi(tokens[7].c_str());
00130         //int pbcy = atoi(tokens[8].c_str());
00131         //int pbcz = atoi(tokens[9].c_str());
00132
00133         /* Set the attributes of the current node object. */
00134         tNode current_node;
00135         current_node.Id = atoi(tokens[0].c_str());
00136         current_node.Type = atoi(tokens[2].c_str());
00137         current_node.Pos[0] = atof(tokens[4].c_str()); //+ (double)pbcx*netw_min->domain->XBoxLen;
00138         current_node.Pos[1] = atof(tokens[5].c_str()); //+ (double)pbcy*netw_min->domain->YBoxLen;
00139         current_node.Pos[2] = atof(tokens[6].c_str()); //+ (double)pbcz*netw_min->domain->ZBoxLen;
00140         current_node.node_cell = 0;
00141
00142         // Read the mass of the nodal point from the "Masses" section of the file.
00143         current_node.n_mass = node_types[current_node.Type-1].n_mass;
00144         current_node.mass = node_types[current_node.Type-1].mass;
00145
00146         /* Pair coeffs come from the data file in Angstrom. */
00147         current_node.r_node = node_types[current_node.Type-1].r_node;
00148         current_node.r_star = current_node.r_node;
00149
00150         nodes.push_back(current_node);
00151     }
00152     cout << "#: --- " << nnodes << " nodes have been added to the network structure.\n";
00153
00154     int nchains = 0;
00155
00156     for (int i = 0; i < nstrands; i++) {
00157         /* Read and split the line from the input file: */
00158         tokens = tokenize(lines_of_file[bonds_line_start + 2 + i]);
00159
00160         /* Create a temporary object: */
00161         tStrand current_strand;
00162         current_strand.Id = atoi(tokens[0].c_str());
00163         current_strand.Type = atoi(tokens[1].c_str());
00164         current_strand.OrChain = atoi(tokens[5].c_str());
00165         if (current_strand.OrChain > nchains)
00166             nchains = current_strand.OrChain;
00167
00168         if (current_strand.OrChain == 0)
00169             current_strand.slip_spring = true;
00170         else
00171             current_strand.slip_spring = false;
00172
00173         /* Read the tags of the start and the end of the strand. */
00174         int snode = atoi(tokens[2].c_str());
00175         int enode = atoi(tokens[3].c_str());
00176
00177         if (snode > (nnodes) || enode > (nnodes)){
00178             cout << ": --- There is a problem at line: " << (bonds_line_start + 2 + i + 1)
00179                 << " of the data file.\n" << lines_of_file[bonds_line_start + 2 + i] << "\n";
00180             exit(0);
00181         }
00182     }
00183
00184     /* We have to find the nodal point whose tag is equal to snode: */
00185     int found = 0;
00186     for (std::list<tNode>::iterator it = nodes.begin(); it != nodes.end(); ++it) {
00187         if (found > 1)
00188             break;
00189         if ((*it).Id == snode || (*it).Id == enode) {
00190             found++;
00191             (*it).OrChains.push_back(current_strand.OrChain);
00192             current_strand.pEnds.push_back(&(*it));
00193         }
00194     }
00195
00196     /* Spring coeffs comes from the data file in kJ/mol/K/A^2 */
00197     current_strand.spring_coeff = bond_types[current_strand.
Type-1].spring_coeff;
00198     /* Length of the strand (n*b) comes from the data file in Angstroms. */
00199     current_strand.sq_end_to_end = bond_types[current_strand.
Type-1].sq_ete;
00200     current_strand.kuhn_length = bond_types[current_strand.Type-1].kuhn1;
00201

```

```

00202      /* Set the creation time of the slip-spring: */
00203      if (current_strand.slip_spring)
00204          current_strand.tcreation = 0;
00205
00206      /* and push it back to the list: */
00207      strands.push_back(current_strand);
00208
00209      /* 2013/04/09: Also, push back a pointer to the last element of "strands" vector, if
00210       * the inserted strand is a slip spring. */
00211      if (current_strand.slip_spring)
00212          pslip_springs.push_back(&(strands.back()));
00213  }
00214
00215
00216
00217  for (std::list<tStrand> ::iterator it = strands.begin(); it != strands.end(); ++it) {
00218      /* 2013/04/09: We exclude slip strings from the local registry of strands connected to
00219       * a specific node. Array "pStrands" contains only chain strands,
00220       * not slip-springs!! */
00221      if (!(*it).slip_spring) {
00222          (*it).pEnds[0]->pStrands.push_back(&(*it));
00223          (*it).pEnds[1]->pStrands.push_back(&(*it));
00224      }
00225  }
00226
00227  for (std::list<tStrand> ::iterator it = strands.begin(); it != strands.end(); ++it) {
00228      if ((*it).slip_spring)
00229          continue;
00230
00231      double dxx = (*it).pEnds[1]->Pos[0] - (*it).pEnds[0]->Pos[0];
00232      double dyy = (*it).pEnds[1]->Pos[1] - (*it).pEnds[0]->Pos[1];
00233      double dzz = (*it).pEnds[1]->Pos[2] - (*it).pEnds[0]->Pos[2];
00234      netw_min->domain->minimum_image(dxx, dyy, dzz);
00235
00236      double dist = sqrt(dxx * dxx + dyy * dyy + dzz * dzz);
00237      if (dist - (*it).sq_end_to_end / (*it).kuhn_length > tol)
00238          cout << "Strand " << (*it).Id << " has length " << (*it).sq_end_to_end
00239              / (*it).kuhn_length << " but real dist " << dist << endl;
00240  }
00241
00242
00243      /* Categorize strands into chains: */
00244      std::vector<std::list<tStrand * > > chains(nchains);
00245      for (std::list<tStrand> ::iterator it = strands.begin(); it != strands.end(); ++it)
00246          if (!(*it).slip_spring)
00247              chains[(*it).OrChain - 1].push_back(&(*it));
00248
00249
00250      /* Loop over all chains: */
00251      sorted_chains.resize(nchains);
00252
00253      for (unsigned int ich = 0; ich < chains.size(); ich++) {
00254          /* Search to find an end of the chain: */
00255          std::list<tStrand * >::iterator jt;
00256          jt = find_if(chains[ich].begin(), chains[ich].end(), pred_strand_is_end);
00257          if (jt == chains[ich].end()) {
00258              cout << "Chain with no ends was found: " << ich << " .\n";
00259              cout << "Number of strands: " << chains[ich].size() << " .\n";
00260              return;
00261          }
00262
00263          if ((*jt)->pEnds[0]->Type != 1) {
00264              tNode *temp;
00265              temp = (*jt)->pEnds[0];
00266              (*jt)->pEnds[0] = (*jt)->pEnds[1];
00267              (*jt)->pEnds[1] = temp;
00268          }
00269          sorted_chains[ich].push_back((*jt));
00270          if (chains[ich].size() > 1)
00271              chains[ich].erase(jt);
00272          else
00273              continue;
00274
00275          int nelems = chains[ich].size();
00276          for (int i = 0; i < nelems; i++) {
00277              /* Set the iterator to the end of the sorted list: */
00278              it = sorted_chains[ich].end();
00279              --it;
00280
00281              /* Find the next element: */
00282              for (jt = chains[ich].begin(); jt != chains[ich].end(); ++jt) {
00283                  /* check whether is connected to the preceeding strand: */
00284                  if ((*jt)->pEnds[0] == (*it)->pEnds[1]) {
00285                      /* We can append the found strand as is: */
00286                      sorted_chains[ich].push_back((*jt));
00287                      /* an delete it from the previous list */
00288                      jt = chains[ich].erase(jt);

```



```

00289             break;
00290         }
00291
00292         if ((*jt)->pEnds[1] == (*it)->pEnds[1]) {
00293             /* We have to change the order of the pointers. */
00294             tNode *temp;
00295             temp = (*jt)->pEnds[0];
00296             (*jt)->pEnds[0] = (*jt)->pEnds[1];
00297             (*jt)->pEnds[1] = temp;
00298
00299             /* And add it to the list: */
00300             sorted_chains[ich].push_back((*jt));
00301             jt = chains[ich].erase(jt);
00302             break;
00303         }
00304     }
00305 }
00306 }
00307
00308     return;
00309 }
00310 }
00311
00312
00315 bool pred_strand_is_end(tStrand * current) {
00316     bool outcome;
00317     if (current->pEnds[0]->Type == 1 || current->pEnds[1]->Type == 1)
00318         outcome = true;
00319     else
00320         outcome = false;
00321
00322     return outcome;
00323 }
00324
00325 }

```

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

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 [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	<i>current</i>	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

```
00001
00015 #ifndef _NETWORK_H
00016 #define _NETWORK_H
00017
00018 #include "net_types.h"
00019
00020 namespace NetworkNS {
00021
00025     class Network {
00026     public:
00027
00028         Network(class NetwMin *, std::string);
00029         virtual ~Network();
00030
00031         std::list<tNode> nodes;
00032         std::list<tStrand> strands;
00033         std::list<tSubCh> subchains;
00034         std::list<tStrand *> pslip_springs;
00035         std::vector<tBead_type> node_types;
00036         std::vector<tBond_type> bond_types;
00037
00045         std::vector<std::list<tStrand *> > sorted_chains;
00046
00050     };
00051
00052     bool pred_strand_is_end(tStrand *);
00053 }
00054
00055 #endif /* NETWORK_H */
00056
```

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

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 [non_bonded_scheme_routines.cpp](#).

5.44 non_bonded_scheme_routines.cpp

```

00001
00015 #include <iostream>
00016 #include <cmath>
00017
00018 void phi_function(double x, double xj, double Rj, double delta, double &phi_value) {
00019
00020     double inv_delta = 1.0 / delta;
00021
00022     if (x <= xj - 0.5 * Rj - 0.5 * delta)
00023         phi_value = 0.0;
00024
00025     else if (x > xj - 0.5 * Rj - 0.5 * delta && x <= xj - 0.5 * Rj + 0.5 * delta) {
00026         double temp = 2.0 * (x - xj) * inv_delta + Rj*inv_delta;
00027         double tempsq = temp*temp;
00028
00029         phi_value = (8.0 + 15.0 * temp
00030                     - 10.0 * tempsq * temp + 3.0 * tempsq * tempsq * temp) / 16.0 / Rj;
00031     } else if (x > xj - 0.5 * Rj + 0.5 * delta && x <= xj + 0.5 * Rj - 0.5 * delta)
00032         phi_value = 1.0 / Rj;
00033
00034     else if (x > xj + 0.5 * Rj - 0.5 * delta && x <= xj + 0.5 * Rj + 0.5 * delta) {
00035         double temp = 2.0 * (xj - x) * inv_delta + Rj*inv_delta;
00036         double tempsq = temp*temp;
00037
00038         phi_value = (8.0 + 15.0 * temp
00039                     - 10.0 * tempsq * temp + 3.0 * tempsq * tempsq * temp) / 16.0 / Rj;
00040     }
00041     else if (x > xj + 0.5 * Rj + 0.5 * delta)
00042         phi_value = 0.0;
00043 }
00044
00045 void integral_minus_infinity_to_x(double xasterisk, double xj, double Rj, double delta,
00046     double &integral_value) {
00047
00048     if (xasterisk <= xj - Rj / 2.0 - delta / 2.0)
00049         integral_value = 0.0;
00050
00051     else if (xasterisk > xj - Rj / 2.0 - delta / 2.0 && xasterisk <= xj - Rj / 2.0 + delta / 2.0)
00052         integral_value = delta * (5.0 / 4.0 + 4.0 * (2.0 * (xasterisk - xj) / delta + Rj / delta)
00053             + 15.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 2)\
00054             - 5.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 4)
00055             + 1.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 6)) / 16.0 / Rj;
00056
00057     else if (xasterisk > xj - Rj / 2.0 + delta / 2.0 && xasterisk <= xj + Rj / 2.0 - delta / 2.0)
00058         integral_value = (xasterisk - xj) / Rj + 1.0 / 2.0;
00059
00060     else if (xasterisk > xj + Rj / 2.0 - delta / 2.0 && xasterisk <= xj + Rj / 2.0 + delta / 2.0)

```

```

00061     integral_value = 1.0 - delta * (5.0 / 4.0 + 4.0 * (2.0 * (xj - xasterisk) / delta + Rj / delta)
00062     + 15.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 2)
00063     - 5.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 4)
00064     + 1.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 6)) / 16.0 / Rj;
00065
00066     else if (xasterisk > xj + Rj / 2.0 + delta / 2.0)
00067         integral_value = 1.0;
00068
00069 }
00070
00071 void integral_x_to_plus_infinity(double xasterisk, double xj, double Rj,
00072     double delta, double &integral_value) {
00073
00074     if (xasterisk <= xj - Rj / 2.0 - delta / 2.0)
00075         integral_value = 1.0;
00076
00077     else if (xasterisk > xj - Rj / 2.0 - delta / 2.0 && xasterisk <= xj - Rj / 2.0 + delta / 2.0)
00078         integral_value = 1.0 - delta * (5.0 / 4.0 + 4.0 * (2.0 * (xasterisk - xj) / delta + Rj / delta)
00079         + 15.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 2)
00080         - 5.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 4)
00081         + 1.0 / 4.0 * pow(2.0 * (xasterisk - xj) / delta + Rj / delta, 6)) / 16.0 / Rj;
00082
00083     else if (xasterisk > xj - Rj / 2.0 + delta / 2.0 && xasterisk <= xj + Rj / 2.0 - delta / 2.0)
00084         integral_value = (xj - xasterisk) / Rj + 1.0 / 2.0;
00085
00086     else if (xasterisk > xj + Rj / 2.0 - delta / 2.0 && xasterisk <= xj + Rj / 2.0 + delta / 2.0)
00087         integral_value = delta * (5.0 / 4.0 + 4.0 * (2.0 * (xj - xasterisk) / delta + Rj / delta)
00088         + 15.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 2)
00089         - 5.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 4)
00090         + 1.0 / 4.0 * pow(2.0 * (xj - xasterisk) / delta + Rj / delta, 6)) / 16.0 / Rj;
00091
00092     else if (xasterisk > xj + Rj / 2.0 + delta / 2.0)
00093         integral_value = 0.0;
00094
00095 }

```

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

```

00001
00007 #ifndef NEW_SCHEME_ROUTINES_H
00008 #define NEW_SCHEME_ROUTINES_H
00009
00010 void phi_function(double x, double xj, double Rj, double delta, double &phi_value);
00011 void integral_minus_infinity_to_x(double xasterisk, double xj, double Rj, double delta, double &
    integral_value);
00012 void integral_x_to_plus_infinity(double xasterisk, double xj, double Rj, double delta, double &
    integral_value);

```

```

00013
00014 #endif    /* NEW_SCHEME_ROUTINES_H */
00015

```

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

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 [rng.cpp](#).

5.48 rng.cpp

```

00001
00015 #include <iostream>
00016 #include <cmath>
00017 #include <fcntl.h>
00018 #include <stdio.h>
00019 #include <stdlib.h>
00020 #include <stdint.h>
00021 #include <unistd.h>
00022 #include <time.h>
00023 #include <math.h>
00024 #include <cmath>
00025 #include "rng.h"
00026 #define PHI 0x9e3779b9
00027
00028
00029 using namespace std;
00030
00031 namespace NetworkNS {
00032
00033     unsigned int RanMars::devrand()
00034     {

```

```

00035
00036     int fn;
00037     unsigned int r;
00038     fn = open("/dev/urandom", O_RDONLY);
00039     if (fn == -1)
00040         exit(-1); /* Failed! */
00041     if (read(fn, &r, 4) != 4)
00042         exit(-1); /* Failed! */
00043     close(fn);
00044     return r;
00045 }
00046
00047 unsigned int RanMars::uint_rand()
00048 {
00049     unsigned long long t;
00050     x = 314527869 * x + 1234567;
00051     y ^= y << 5;
00052     y ^= y >> 7;
00053     y ^= y << 22;
00054     t = 4294584393ULL * z + c;
00055     c = t >> 32;
00056     z = t;
00057     return x + y + z;
00058 }
00059
00060
00061
00062 RanMars::RanMars(int seed) {
00063     /* Seed variables */
00064     x = 123456789;
00065     y = 987654321;
00066     z = 43219876;
00067     c = 6543217;
00068
00069     x = devrand();
00070     while (!(y = devrand())); /* y must not be zero! */
00071     z = devrand();
00072
00073     /* We don't really need to set c as well but let's anyway... */
00074     /* NOTE: offset c by 1 to avoid z=c=0 */
00075     c = devrand() % 698769068 + 1; /* Should be less than 698769069 */
00076
00077     // gvog: And warm up the generator:
00078     /* Also, discard the first values... */
00079     int i, nelements = 1000000;
00080     //unsigned int temp;
00081     for (i = 0; i < nelements; i++)
00082         uint_rand();
00083
00084     return;
00085 }
00086
00087 /* ----- */
00088
00089 RanMars::~RanMars() {
00090     delete [] u;
00091 }
00092
00093 /* -----
00094     uniform RN
00095 ----- */
00097 double RanMars::uniform() {
00098
00099     double x;
00100     unsigned long long a;
00101     a = ((unsigned long long) uint_rand() << 32) + uint_rand();
00102     a = (a >> 12) | 0x3FF0000000000000ULL; /* Take upper 52 bits */
00103     *((unsigned long long *) &x) = a; /* Make a double from bits */
00104
00105     return x - 1.0;
00106 }
00107
00108 /* -----
00109     gaussian RN
00110 ----- */
00111
00112 double RanMars::gaussian() {
00113     double first, v1, v2, rsq, fac;
00114
00115     if (!save) {
00116         int again = 1;
00117         while (again) {
00118             v1 = 2.0 * uniform() - 1.0;
00119             v2 = 2.0 * uniform() - 1.0;
00120             rsq = v1 * v1 + v2*v2;
00121             if (rsq < 1.0 && rsq != 0.0) again = 0;
00122         }

```

```

00123         fac = sqrt(-2.0 * log(rsq) / rsq);
00124         second = v1*fac;
00125         first = v2*fac;
00126         save = 1;
00127     } else {
00128         first = second;
00129         save = 0;
00130     }
00131     return first;
00132 }
00133
00134 double RanMars::modified_gaussian(double mean, double stdev) {
00135     double first, v1, v2, rsq, fac;
00136
00137     if (!save) {
00138         int again = 1;
00139         while (again) {
00140             v1 = 2.0 * uniform() - 1.0;
00141             v2 = 2.0 * uniform() - 1.0;
00142             rsq = v1 * v1 + v2*v2;
00143             if (rsq < 1.0 && rsq != 0.0) again = 0;
00144         }
00145         fac = sqrt(-2.0 * log(rsq) / rsq);
00146         second = mean + stdev * v1*fac;
00147         first = mean + stdev * v2*fac;
00148         //cout << "in_marsaglia=" << mean << endl;
00149         save = 1;
00150     } else {
00151         first = second;
00152         save = 0;
00153     }
00154     return first;
00155 }
00156
00157 double RanMars::rand_gauss(void) {
00158     double v1, v2, s;
00159
00160     do {
00161         v1 = 2.0 * uniform() - 1;
00162         v2 = 2.0 * uniform() - 1;
00163
00164         s = v1 * v1 + v2*v2;
00165     } while (s >= 1.0);
00166
00167     if (s == 0.0)
00168         return 0.0;
00169     else
00170         return (v1 * sqrt(-2.0 * log(s) / s));
00171 }
00172
00173
00174 }

```

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

Copyright (c) 2014 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 [rng.h](#).

5.50 rng.h

```
00001
00015 #ifndef MARSAGLIA_H
00016 #define MARSAGLIA_H
00017
00018 namespace NetworkNS {
00019
00023     class RanMars {
00024     public:
00025
00026         RanMars(int);
00027
00028         ~RanMars();
00029
00030         double uniform();
00032         double gaussian();
00036         double modified_gaussian(double mean, double stdev);
00040
00041         double rand_gauss(void);
00042
00043         unsigned int devrand();
00044
00045         unsigned int uint_rand();
00046
00047     private:
00048         int seed;
00049         int save;
00050         double second;
00051         double *u;
00052         int i97, j97;
00053
00054         unsigned int x;
00055         unsigned int y;
00056         unsigned int z;
00057         unsigned int c;
00058
00059     };
00060
00061 }
00062
00063
00064 #endif /* MARSAGLIA_H */
00065
```


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

- ~Hopping
 - NetworkNS::Hopping, 32
- add_snapshot_to_dump
 - NetworkNS::dump, 29
- amu_to_kg
 - constants.h, 60
- Auxiliary.cpp, 45, 46
 - tokenize, 45
- Auxiliary.h, 46, 47
 - tokenize, 47
- avogadro_constant
 - constants.h, 59
- b3D_integrator.cpp, 49, 50
- b3D_integrator.h, 57
- bd_f
 - NetworkNS::cb3D_integrator, 23
- bd_f_ps
 - NetworkNS::cb3D_integrator, 24
- bd_gamma
 - NetworkNS::cb3D_integrator, 23
- bd_mass
 - NetworkNS::cb3D_integrator, 23
- bd_nb_f
 - NetworkNS::cb3D_integrator, 23
- bd_x
 - NetworkNS::cb3D_integrator, 23
- bd_x_ps
 - NetworkNS::cb3D_integrator, 24
- bond_types
 - NetworkNS::Network, 36
- bonded_force_calculation
 - NetworkNS::cb3D_integrator, 20
- Boundary
 - NetworkNS::Domain, 28
- BoxExists
 - NetworkNS::Domain, 28
- c1
 - constants.h, 60
- c2
 - constants.h, 60
- calculate_pressure
 - NetworkNS::cb3D_integrator, 22
- cb3D_integrator
 - NetworkNS::cb3D_integrator, 18
- constants.h, 59, 60
 - amu_to_kg, 60
 - avogadro_constant, 59
 - c1, 60
 - c2, 60
 - kg_to_amus, 60
 - pi_tube_diameter, 60
- D3DVector
 - operator*, 26
- D3DVector< T >, 25
- den_dx
 - NetworkNS::cb3D_integrator, 25
- den_dy
 - NetworkNS::cb3D_integrator, 25
- den_dz
 - NetworkNS::cb3D_integrator, 25
- distributions.cpp, 61, 62
 - e_gaussian, 62
 - f_gaussian, 61
- distributions.h, 63, 64
 - e_gaussian, 64
 - f_gaussian, 63
- Domain
 - NetworkNS::Domain, 28
- domain
 - NetworkNS::NetwMin, 34
- domain.cpp, 65
- domain.h, 67
- dump.cpp, 68
- dump.h, 69, 70
- e_gaussian
 - distributions.cpp, 62
 - distributions.h, 64
- f_gaussian
 - distributions.cpp, 61
 - distributions.h, 63
- from_bd_x_to_polymer_network
 - NetworkNS::cb3D_integrator, 23
- gamma_mass_opt
 - NetworkNS::cb3D_integrator, 24
- gaussian
 - NetworkNS::RanMars, 38
- grid
 - NetworkNS::NetwMin, 34
- grid.cpp, 70, 71
- grid.h, 73, 74
 - grid_cell, 74
- grid_cell
 - grid.h, 74

- Hopping
 - NetworkNS::Hopping, 31
- hopping.cpp, 74, 75
- hopping.h, 82, 83
- hopping_step
 - NetworkNS::Hopping, 32
- integrate
 - NetworkNS::cb3D_integrator, 19
- kg_to_amus
 - constants.h, 60
- main
 - main.cpp, 84
- main.cpp, 83, 84
 - main, 84
- modified_gaussian
 - NetworkNS::RanMars, 38
- my_traj_file
 - NetworkNS::NetwMin, 34
- net_types.h, 85, 86
- netmin.cpp, 87, 88
- netmin.h, 89, 90
- NetwMin
 - NetworkNS::NetwMin, 33
- Network
 - NetworkNS::Network, 35
- network
 - NetworkNS::NetwMin, 34
- network.cpp, 90, 91
 - pred_strand_is_end, 91
- network.h, 95, 96
 - pred_strand_is_end, 96
- NetworkNS::Domain, 27
 - Boundary, 28
 - BoxExists, 28
 - Domain, 28
 - NonPeriodic, 28
- NetworkNS::Grid, 30
 - vcell, 30
- NetworkNS::Hopping, 31
 - ~Hopping, 32
 - Hopping, 31
 - hopping_step, 32
- NetworkNS::NetwMin, 33
 - domain, 34
 - grid, 34
 - my_traj_file, 34
 - NetwMin, 33
 - network, 34
 - write_network_to_lammps_data_file, 34
- NetworkNS::Network, 35
 - bond_types, 36
 - Network, 35
 - node_types, 36
 - nodes, 36
 - pslip_springs, 36
- sorted_chains, 37
- strands, 36
- subchains, 36
- NetworkNS::RanMars, 37
 - gaussian, 38
 - modified_gaussian, 38
 - uniform, 38
- NetworkNS::cb3D_integrator, 17
 - bd_f, 23
 - bd_f_ps, 24
 - bd_gamma, 23
 - bd_mass, 23
 - bd_nb_f, 23
 - bd_x, 23
 - bd_x_ps, 24
 - bonded_force_calculation, 20
 - calculate_pressure, 22
 - cb3D_integrator, 18
 - den_dx, 25
 - den_dy, 25
 - den_dz, 25
 - from_bd_x_to_polymer_network, 23
 - gamma_mass_opt, 24
 - integrate, 19
 - report, 20
 - simpler_scheme_non_bonded_force_calculation, 21
 - xshift, 24
 - yshift, 24
 - zshift, 24
- NetworkNS::dump, 29
 - add_snapshot_to_dump, 29
- NetworkNS::sgrid_cell, 39
- node_cell
 - sNode, 41
- node_types
 - NetworkNS::Network, 36
- nodes
 - NetworkNS::Network, 36
- non_bonded_scheme_routines.cpp, 96, 97
- non_bonded_scheme_routines.h, 98
- NonPeriodic
 - NetworkNS::Domain, 28
- operator*
 - D3DVector, 26
- pi_tube_diameter
 - constants.h, 60
- pred_strand_is_end
 - network.cpp, 91
 - network.h, 96
- pslip_springs
 - NetworkNS::Network, 36
- r_node
 - sNode, 41
- r_star
 - sNode, 41

report
 NetworkNS::cb3D_integrator, 20
rng.cpp, 99
rng.h, 101, 102

sBead_type, 38
sBond_type, 39
 spring_coeff, 39
sNode, 40
 node_cell, 41
 r_node, 41
 r_star, 41
 Type, 41
sStrand, 41
 spring_coeff, 42
 sq_end_to_end, 42
 Type, 42
simpler_scheme_non_bonded_force_calculation
 NetworkNS::cb3D_integrator, 21
sorted_chains
 NetworkNS::Network, 37
spring_coeff
 sBond_type, 39
 sStrand, 42
sq_end_to_end
 sStrand, 42
strands
 NetworkNS::Network, 36
subchains
 NetworkNS::Network, 36

tokenize
 Auxiliary.cpp, 45
 Auxiliary.h, 47
Type
 sNode, 41
 sStrand, 42

uniform
 NetworkNS::RanMars, 38

vcell
 NetworkNS::Grid, 30

write_network_to_lammps_data_file
 NetworkNS::NetwMin, 34

xshift
 NetworkNS::cb3D_integrator, 24

yshift
 NetworkNS::cb3D_integrator, 24

zshift
 NetworkNS::cb3D_integrator, 24