Polymer chain Dynamics. Simulation Framework- user manual

Ofir Shukron

May 19, 2015

Contents

0.1 Simulation Framework 2

1 Classes **3**

1.1 RouseSimulatorFramework 3

1.1.1 Properties 3

1.1.2 Methods 3

1.2 SimulationDataRecorder 4

1.2.1 Properties 4

1.2.2 Methods 5

1.3 Rouse 5

1.3.1 Properties 5

1.3.2 Methods 6

1.3.3 Noise 6

1.4 The Recipe files 6

1.5 Chains' Association and Dissociation 6

1.6 Initialization 7

1.6.1 Domain initialization 7

1.6.2 Chain initialization 7

1.6.3 Initialization of beads on the domain's boundary 7

1.7 Domain Reflection 8

1.7.1 Sphere 8

1.7.2 Polygon 9

1.8 Object Manager 9

1.8.1 objects and members 9

1.8.2 Step process 10

1.9 Object Mapper 10

1.9.1 FindConnectedBeads 10

1.9.2 ConnectParticles 10

1.9.3 DisconnectParticles 10
1.9.4 SplitMember 10

0.1 Simulation Framework

The beads- In our system of a chain of beads connected by harmonic springs, the beads represent monomers of a polymer chain. Nucleosome- In Eukaryote cell, the nucleusome packs around 2 meters of DNA material into an accessible package called the nucleusome. It size is roughly $10\mu m$.

Chapter 1

Classes

In this chapter we review the different classes of the simulation framework and give details about the input/output of each method along with the properties of the class and concise explanation regarding the role of each of its methods.

1.1 RouseSimulatorFramework

This class is the backbone of the simulation framework. It coordinates the action between all classes participating in the simulations. The class receives the parameters of each of its participating classes, distributes them and initializes each class.

1.1.1 Properties

- **handles** -holds the handles for classes and graphical object. All classes' handles are held under the fields *handles.classes*, all graphical handles, e.g figure, axes, buttons, et. are held under *handles.graphical*.
- **params** - holds the parameters for each of the participating classes. The parameters are parsed at the classes initialization from an .xml file (see section) and arranged as a structure placed in this property.

1.1.2 Methods

- **PreRunActions**- Actions performed before a simulation round begins.

- **PostRunActions** activates a sequence of predefined commands after each simulation round. In the present release the post run action is predefined to record simulation end time using *SimulatorDataRecorder.SetsimulationEndTime* method, save results using *SimulatorDataRecorder.SaveResults* method and notify about the simulation successful ending by email using *SendMail* function, located in the 3rdParty folder.

1.2 SimulationDataRecorder

Handles recording of the simulation data.

1.2.1 Properties

- **simulationData** Structure containing the fields:
  1. **chainObj** the Rouse chain object initialized using the Rouse class(see section 1.3)
  2. **numChains** number of chains.
  3. **step** current simulation step.
  4. **time** current simulation time, calculated as $step \times \Delta t$
  5. **positions**- the current position of the beads in the chain
  6. **beadDist** pair-wise bead distance matrix. This in a multidimensional matrix of size $[numBead \times numBead \times step]$
- **simulationRound** the current round of simulation
- **params** contains the parameters for the class.

paramList

- saveType- [all/external/none] (see SaveResults method)
- resultsFolder the path to the results folder. Can be relative or absolute

1.2.2 Methods

- **SaveResults** Implements 4 types of saving: [all/external/internal/none]
    1. *all*- save all data. Data is stored on the class and exported
    2. *external*- save all data to .mat files. No data is saved on the class
    3. *internal*- data is saved only on the class
    4. *none* - don't save any data. No data is saved on the class and none is exported to .mat files
- **ClearCurrentSimulationData** Clears the data from the class properties

1.3 Rouse

1.3.1 Properties

- **time-** time of the simulation, defined as $step \times \Delta t$
- **step-** simulation step
-
- **positions**
    1. **beads-** the coordinates of the beads relative to $(0, 0, 0)$.
        (a) **cur-** bead position at the present simulation step
        (b) **prev-** bead position at the previous simulation step
    2. **springs-** The vectors defining the springs.
        (a) **angleBetweenSprings-** this is a sparse representation of a 3D matrix defining the angle between bead i, j, and k. The convention is the position (i, j, k) represent the angle between bead i, j, and k, where i is the row, j is the column, and k is the depth (height) of the matrix. Strings are defined by a vector composed of subtracting the bead position i from $i+1$.
        (b) **length**- the length of the springs is the norm of each of the vectors defining the springs

1.3.2 Methods

- **setBeadsMobilityMatrix**
- **GetNewBeadsPosition**

1.3.3 Noise

The noise terms can be determined to be any type of noise distribution, with mean and variance as parameters. The default values are drawn from a Gaussian distribution with mean zero and variance 1. To save computation time for each step, the noise terms are determined every N simulation steps.

1.4 The Recipe files

The recipe files are used to allow the user to insert simulation specific commands without changing the content of the code. Recipe files allow the user to insert 4 functions, which are executed in: PreSimulationBatchActions, PreRunActions, PostRunActions, PostSimulationBatchActions. of the simulation framework.

1.5 Chains' Association and Dissociation

To explain the data structure's dynamics for chain association and dissociation, we first make several definitions.

A system of N chains will be written as: $[C^{(1)}, C^{(2)}, ..., C^{(N)}]$, the coordinates of the beads in $C^{(i)}$ will be written as X^i. For each chain $C^{(i)}$ there exist a subset of 'sticky' beads $S^{(i)}$ that are allowed to interact with a subset $S^{(j)}, \quad j \in [1, N]$.For the present notation we do not allow self interactions. A subscript will indicate a member of the group, i.e $X_k^{(i)}$ will indicate the k^{th} coordinate in the coordinates of chain i, $S_k^{(i)}$ will indicate the k^{th} coordinate in the subset of interacting beads of chain i, etc...

An *association* between a chain i and j is the interaction between a member of $S^{(i)}$ and $S^{(j)}$ and will be denoted by $C^{(i)} \oplus C^{(j)}$, that is $\exists \alpha \in S^{(i)}, \beta \in S^{(j)}; |X_\alpha^{(i)} - X_\beta^{(j)}| < \epsilon$, with ϵ the interaction distance.

A *dissociation* of the structure

1.6 Initialization

All objects are initialized before simulation starts. Some objects might be initialized dynamically upon need. All objects are contained within some domain. Even, an unconstrained chain is contained within an open domain, for which no reflection is implemented. Chains are always associated with some domain, whether they are contained in it and constrained to stay inside according to the domains reflection rules or are interacting with it's surface from outside.

1.6.1 Domain initialization

A variable number of domains can be initialized. All domain are handled by the DomainHandler class. The list of functionality of domains is given in the DomainHandler class section. In short, domain exert on chains and object external forces such as diffusion, Lennard-Jones and Morse forces, and are responsible for relocating the particles after interaction between the particles and the domain (e.g in case of reflection)

To initialize a domain we have to register it in the DomainHandler class. The initialization of a domain is done by providing the DomainHandler a structure containing DomainHandlerParameters classes, one for each domain. For example, assume we would like to initialize two domains, one is a spherical domain with radius 5 and the second a cylindrical domain of radius 1 and hight 10. In addition, assume that we want chains to be reflected only from the spherical domain and the the cylindrical and we also want the cylindrical domain to not affect the particles by diffusion.

```
domainParams(1) =DomainHandlerParams('domainShape','sphere','domainWidth',10,'diffusionForce',true);
domainParams(2) = DomainHandlerParams('domainShape','cylinder','reflectionType','off',...
'diffusionForce',false,'domainWidth',1,'domainHeight',50);
```

These parameters will later be fed to the SimulatorFrameworkParams for initialization along with the parameters of the chains (to be explained in later subsection)

1.6.2 Chain initialization

1.6.3 Initialization of beads on the domain's boundary

For a set of beads $i, j, k, ...$ constrained to the surface ∂S of a sphere S, we have to find initial points on ∂S such that a chain can pass through them. This is accomplished by picking positions for the constrained bead by

diffusing on ∂S and then connect these positions with a Brownian bridges, completely contained within S. This procedure can be summarized as: For a surface of spherical domain ∂S, initialization of a chain with N_b beads $b_1, b_2, ...b_{N_b}$ is done as follows

1. list all constrain beads in ascending order $C = \{a_1, a_2, , ..., a_{N_c}\}, \quad a_i > a_{i-1}$
2. Choose a random position on ∂S for b_{a_1}.
3. For i from 2 to N_c, choose a position for b_{a_i} by diffusion on the boundary $a_i - a_{i-1}$ steps (see 1.6.3).
4. For i from 2 to N_c, if $a_i - a_{i-1} > 1$, construct a Brownian bridge $B(b_{a_{i-1}}, b_{a_i}) \in S$
5. if $a_1 > 1$, sequentially build a path from b_{a_1} to a random point in S
6. if $a_{N_c} < N_b$, sequentially build a path from $b_{a_{N_c}}$ to a random point in S

Diffusion on the surface of a sphere

We apply simple rules of diffusion for walking on the surface of a sphere ∂S. The coordinates of each particle, (x, y, z) are transformed to spherical coordinates (ϕ, θ, ρ), where we keep ρ constant throughout. The random walk is then a 'random walk' of the angles $\theta(t)$ and $\phi(t)$, with ϕ the angle between the path's ray and the positive z axis, and $theta$ is the angle between the projection of the path's ray onto the x-y plane and the positive x axis.

1.7 Domain Reflection

1.7.1 Sphere

First we find the intersection point between the path at two consecutive time steps and the spherical domain. Let A be the particle location at time t and B the tentative location of the particle at time $t + \Delta t$, at any step we check if the path $C = B - A$ have crossed the spherical boundary by finding the roots

of the quadratic equation for the intersection between a line and a sphere. setting $\alpha = \langle A, B\rangle - \langle A, A\rangle, \beta = \langle B - A, B - A\rangle$

$$t_{1,2} = \frac{-\alpha \pm \sqrt{\alpha^2 - \beta(\langle A, A\rangle - R^2)}}{\beta}$$

then we take the t such that $0 < t < 1$, to make sure the intersection is in the right direction and is between points A and B.

1.7.2 Polygon

The collision detection and reflection of two or more particles moving on a polygon is done according to the following steps:

1. *collision Detection*: we reduce the dimensionality of the problem to 1D collision detection. The polygon is parametrize in terms of arc-length (cumulative edge length). Two particles with positions $x_1(t), x_2(t)$ and $x_1(t+\Delta t), x_2(t+\Delta t)$ at time steps t and $t+\Delta t$ are considered collided if the value

   $$0 \leq t = \frac{\Delta t(x_2(t) - x_1(t)}{(x_1(t+\Delta t) - x_2(t+\Delta t)) - (x_1(t) - x_2(t))} \leq \Delta t$$

   which is equivalent to $0 \leq t/\Delta t \leq 1$

2. the collision position is determined in 1D simply as $x = v_t t$, for velocities of all particles for which the criteria in the previous point is met.

1.8 Object Manager

1.8.1 objects and members

objects in the framework can be one or several chains connected in an arbitrary manner. Objects are treated as a single unit when forces are applied. there are generally two types of objects in the simulation framework, wither chain(s) or domains.

Provisional: domains and chains should share a similar core such that chains could act as domains

1.8.2 Step process

first all objects (chains) move one step according to the internal forces (spring, bending) Then the external forces (diffusion, Lennard-Johns) are applied to all objects. The total displacement of each particle is considered by the additive contribution of the external and internal forces.

Provisional: The step process should be such that all external forces are applied iteratively on all the object they contain (including chains and domains), all the internal domains should apply forces on their internal objects ans so forth.

1.9 Object Mapper

this class serves as the mapping between object indexes, their member chains and the members' particles. several mapping exist: object-¿ members, object-¿particle, particle-¿ object, particle-¿member, member-¿ particle, particle-¿member

1.9.1 FindConnectedBeads

The function takes as argument the connectivity map of an object or a group of objects, and looks for connected particles outside the super and sub diagonal of the map.

1.9.2 ConnectParticles

1.9.3 DisconnectParticles

Two connected particle can be disconnected. Before disconnecting the particles we examine if the particles are members of the same chain, if not then we try to split the chain given that there are no other connections between the chain. since two connected members are part of the same object, we try to split the object assuming there are no other connections between the chains.

1.9.4 SplitMember

The split function splits a member from a composite object containing 2 or more members. The member that has been split from the object is assigned

a new object at the end of the object list.