Reconstructing 3D Chromatin Structure from Chromosome Conformation Capture Data

Flagship Project InterOmics – WP1 CNR-ISTI

Claudia Caudai [1], Emanuele Salerno [1], Monica Zoppè [2], and Anna Tonazzini [1]

National Research Council of Italy
[1]Institute of Information Science and Technologies, and [2]Institute of Clinical Physiology, Pisa, Italy
claudia.caudai@isti.cnr.it

1 Introduction

DNA is the central repository of information to keep cells and organisms alive. In human cells, the 46 chromosomes amount to a length of about 2 m, with a diameter of 2 nm, and are packed in a way that allows for access by transcription, replication and repair machinery, fitting within a globular nucleus with a radius of 5 to 10 μm. Efficiency of packing is obtained by several levels of packing mechanisms (Figure 1), both general (due to general principles, irrespective of DNA sequence) and specific, i.e. mediated by proteins that recognize specific motives (sequences) and bring in close proximity parts of DNA that may be very distant in the genomic sequence. In both cases, general packing and specific aggregation, the underlying mechanisms are not entirely described or understood. The first level, mediated by histon octamers, produces a fiber of 11 nm, which in turn is organized into a 30 nm-wide structure. Further packing is at work in cells, and the research community engaged in the study of chromatin conformation is producing increasing knowledge that will finally allow for a clear vision of the nuclear machinery that regulates DNA metabolism.

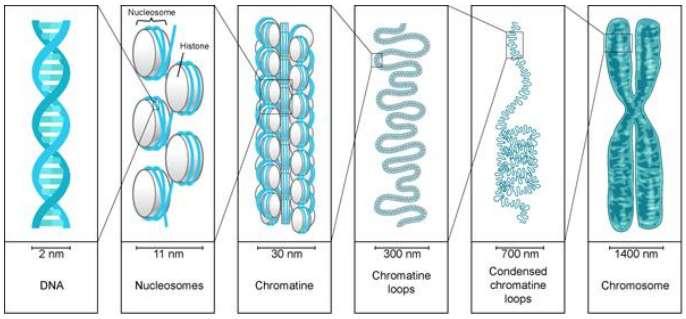

Figure1. DNA at different scales.

The three-dimensional conformation of DNA can be studied in different cells, and most current information relates to a static view, usually referred to a large number of cells which may differ to some degree from one another. DNA metabolism, however, is to be seen as a dynamic equilibrium that changes from moment to moment in the same cell, both to respond to external stimuli (allowing for either transcription regulation or DNA repairs, if necessary), and to allow for regular compaction, as clearly recognizable at large scale during mitosis. It is well established that, in metabolically active cells, most chromosomal DNA is organized in *chromosome territories* [23], and it is increasingly apparent that chromosomal organization is one of the factors involved in regulation of gene function. A step ahead towards an understanding of this spatial organization has been enabled by fluorescence in-situ hybridization techniques (FISH) [1,24], which can be used to locate specific DNA sequences in the genome and measure the distances between pairs of fragments. More recently, Chromosome Conformation Capture (3C, [7]) and a number of related techniques (4C, 5C, HI-C [4,9,45], Figure 2) fostered a major boost in chromatin studies, as they provide high-throughput, high resolution contact data for a full genome at a relatively very low cost.

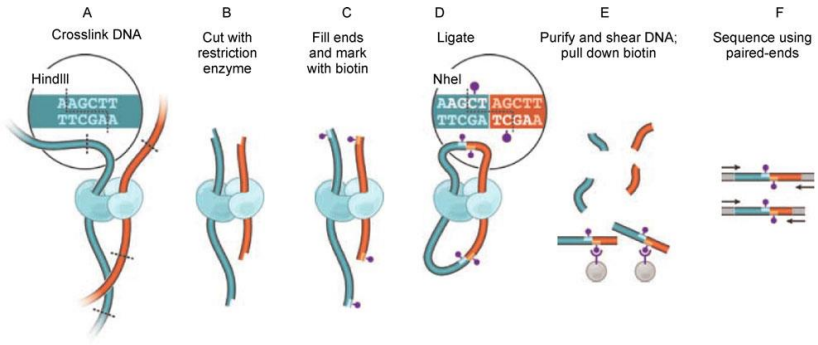

Figure2. Hi-C experiment procedure (direct copy from [26]).

The output of each such experiment is a matrix of contact frequencies between pairs of DNA fragments in a uniform population of cells. The average size of the individual fragments produced by the restriction enzymes in our data is about 4 kbp, therefore the raw contact matrices produced by these techniques have a very high genomic resolution. These contact data, however, come from millions of individual cells. To get stable results, the matrices are thus binned to lower resolutions (typically, 100 kbp), Figure 3. This kind of data carry information about the spatial configuration of the chromatin chain, and many research groups in the last decade have been trying to develop specific reconstruction algorithms.

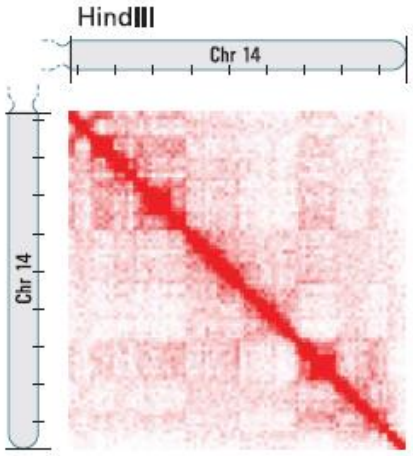

Figure3. Intra-chromosomal interactions in chromosome 14. Each pixel represents all interactions between a 1 Mb locus and another 1 Mb locus (direct copy from [26]).

Recent studies identify large, megabase-sized, local chromatin interaction domains (Dixon *et al.*, [8]). These domains are stable across different cell types and highly conserved across species, indicating that topological domains are an inherent property of mammalian genomes (Figure 4).

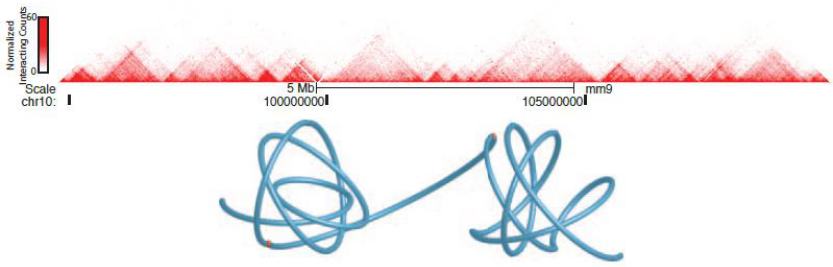

Figure4. Topological domains in Chromosome 10 (direct copy from [8]).

2 Historical Overview

In the last decade, the problem of 3D reconstruction of chromatin structure from contact data has been addressed through three main approaches:

1. Constrained optimization
2. Bayesian inference
3. Polymer models

In our work, we have tried to take a cue from all three approaches and summarize their advantages. Most of the attempts made so far solve the structure reconstruction problem as a distance-to-geometry problem, by translating the contact frequency data into Euclidean distances between pairs of loci. Intuitively, this choice is motivated by the fact that two loci that show very frequent contacts must be close to each other, whereas pairs of loci that are seldom in contact should not be close. As we will show, this assumption has its disadvantages, and also raises the problem of finding a correct strategy to translate contacts into distances.

2.1 Constrained Optimization

The earliest attempts to construct a 3D model of the chromatin used constrained optimization techniques. Dekker *et al.* [7] presented the first such approach, inferring a coarse 3D structure from a few 3C data (Figure 5). Spatial distances were obtained by converting contact frequencies through a theoretical expression for worm-like chains [34], and then used to solve a molecular distance to geometry problem.

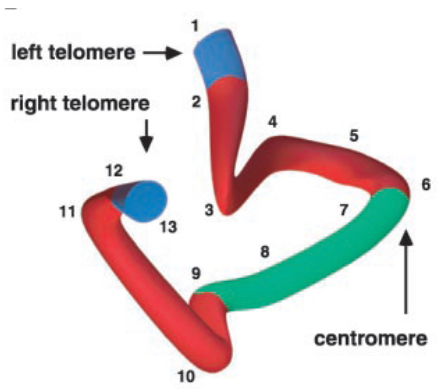

Figure5. 3D model of chromosome III (direct copy from [7]).

Later, Fraser *et al.* [12] assumed an inverse proportionality relation between spatial distances and contact frequencies obtained by 5C experiments, and optimized via gradient descent a piecewise linear curve representing the gene cluster under study.

Duan *et al.* [11] made the first attempt to build a genome wide three dimensional model in yeast from 4C data. They still used the inverse proportionality relation between interaction frequencies and Euclidean distances, and modeled chromatin as a chain of beads, enforcing various constraints of known geometric and topological features of the yeast genome organization (Figure 6).

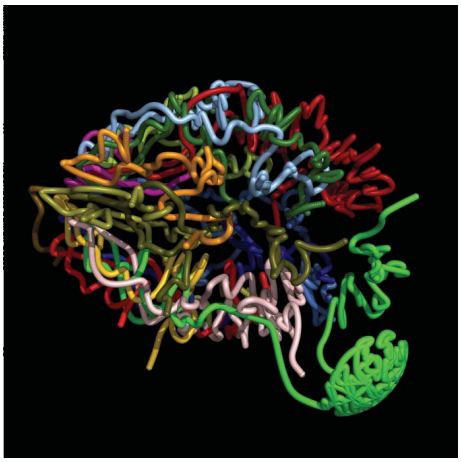

Figure6. Three dimensional model of the yeast genome (direct copy from [11]).

A bead-chain model of chromatin was also employed by Baù *et al.* [2], for 5C data. In this model, the beads interact through harmonic restraints with strengths and equilibrium lengths depending on the inverse of the Z-score of the contact frequencies, and subject to certain constraints based on the structure of the chromatin fiber (Figure 7). Using Hi-C data Tanizawa *et al.* [36] determine the 3D structure of the full genome of fission yeast with a bead-chain model and a method similar to that of Duan. Spatial distances were calculated from contact frequencies through a calibration curve obtained by fitting a double exponential decay function to distance measurements obtained by FISH. Kalhor *et al.* [20], propose to correlate contact frequencies with the presence or absence of chromatin contacts rather than with average spatial distances, and fitted the Hi-C data by learning a set of 3D structures instead of one single structure. Within a regularization approach, Zhang *et al.* [44] apply semi-definite programming techniques to find the best structure fitting the observed data in the noise free case. They use golden section search to find the correct parameter for converting the contact frequencies to spatial distances, and also prove that this parameter changes under different resolutions.
Although constrained optimization approaches have the merit of converting a set of noisy contact frequencies measurements into more interpretable data, they

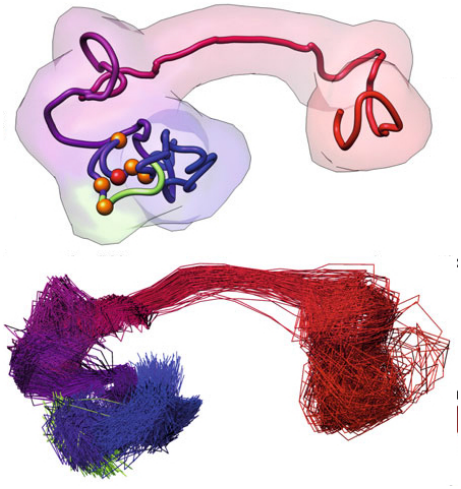

Figure7. Cluster of solutions of -globin 3D-models and their centroid representation (direct copy from [2]).

have several limitations: they tend to get trapped in local modes due to the very high dimensionality, the objective function (usually root mean square deviation) assumes that each contact measurement is equally reliable, and no confidence intervals can be computed to measure the uncertainty of the structure obtained.

2.2 Bayesian Inference

Besides the above mentioned drawbacks, the constrained optimization approaches have the further limitation that they do not take systematic biases into consideration. Without a probabilistic model accounting for noise in the contact frequencies, the set of sampled structures might not be representative of statistically significant conformational features. The first probabilistic model was proposed by Rousseau *et al.* [35] for both 5C and Hi-C data. They applied a Markov Chain Monte Carlo sampling method called MCMC5C to generate ensembles of structures consistent with a Gaussian distribution of spatial distances, still obtained from experimental contact frequencies by assuming an inverse power law relation. Nevertheless, the Hi-C interaction matrix does not contain enough information to accurately estimate the variance of each read count, and, like the constrained optimization methods, MCMC5C only focuses on reconstructing consensus 3D chromosomal structures, without evaluating structural variations of chromatin at different resolution scales.
In a more recent work based on Bayesian inference, Hu *et al.* [16] develop an algorithm called Bayesian 3D Constructor for Hi-C data (BACH) to build consensus 3D structures at the level of the individual topological domains, which

they believe to have conservative functional forms (Figure 8). They also propose the BACH-MIX algorithm to model interactions between adjacent topological domains. The method adopts a Bayesian approach with non-informative prior and a likelihood in the form of a Poisson distribution, whose rate accounts for an inverse relationship between contact measurements and Euclidean distances, but also for some partially known biases, e.g. fragment end length effect, GC content effect and mappability effect. Thus, BACH represents the first approach where the biases are included in the data model. However, the large number of parameters to be estimated and the multiple heuristic sampling processes involved make it a rather complex method. In addition, it does not exploit any constraint on the predicted structures.

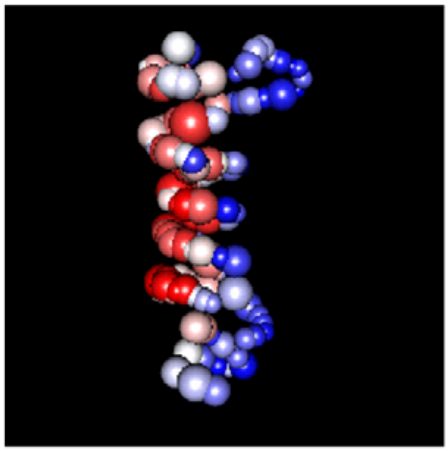

Figure8. Spatial organization of genome-nuclear lamina interaction (direct copy from [16]).

2.3 Polymer Models

Integrating polymer physics into the 3D chromatin structure model could be a key point. However, although the approaches of Duan *et al.* [11] and Tanizawa *et al.* [36] use simple sets of constraints derived from polymer physics, and models of polymers for the chromatin fiber have recently been proposed [19,25,28,37,38], it is not yet fully understood how well they are able to reproduce the biological properties of chromatin structures [3], and to what extent these models are informative at the large scales usually considered (1 Mb in the case of the Hi-C data). Another aspect to be investigated is that almost all the computational approaches reviewed above rely on converting the measured contact frequencies

into spatial distances, assuming a functional relation that describes the behavior of free linear chains. These kinds of relations may not be valid for polymers subjected to looping and other external constraints, or may not properly consider the mechanical properties of the chromatin fiber. In [30], Meluzzi and Arya propose using polymer models without requiring conversion of contact frequencies to mean spatial distances. They estimate contact frequencies by simulating a coarse-grained bead-chain polymer model that approximates the physical behavior of a 30 nm chromatin fiber, and then use an approach from adaptive filter theory to iteratively adjust the parameters of this polymer model until the estimated contact frequencies match the given ones. However, validation has only been performed against reference data sets obtained from simulations of systems with up to 45 beads and 4 loops (Figure 9). Another effort to obtain chromatin conformation ensembles with models and without using a distance-contact relationship was recently reported by Gehlen *et al.* [13] for real data from the entire *S. cerevisiae* genome. Finally, Nagano *et al.* [32] adopted a new experimental Hi-C protocol applied to individual cells, pointing out that the intra-chromosomal structures are substantially stable across different cells, whereas a marked variability of inter-chromosomal interactions has been revealed. The 3D reconstruction of the chromosomal structure is obtained by restrained molecular dynamics simulations, at fine or coarse resolutions, where the restraints are flexible target distances derived from the Hi-C data.

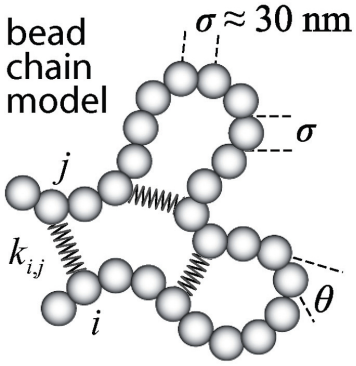

Figure9. Bead-chain polymer model used for Brownian Dynamics simulations of a 30 nm chromatin fiber subjected to looping constraints (direct copy from [30]).

2.4 Two Main Problems

The reconstruction of 3D conformations of the chromatin from contact matrices poses two major difficulties: i) accounting for the presence of biases in the data; ii) converting contact frequencies into coherent geometrical information.

The Bias Problem In Hi-C experiments, five major sources of bias were identified [6]: i) the length of restriction fragments, ii) the length of fragment ends, iii) GC content of the paired-end reads, iv) the length of Dna segments at the circularization steps, and v) the mappability of the sequence reads.

i) In theory, as the number of positions accessible to fixating along a restriction fragment increases with its size, the interaction probability increases linearly with restriction fragment size. In fact, this is true for restriction fragments under 800 bp; for longer restriction fragments, a plateau is reached, suggesting that the maximum probability for at least one cross-linking event to occur along that length is reached.

ii) The sizes of cross-linked fragment ends affect their ligation efficiency. Ligation efficiency is low when fragment ends are too long or too short. Optimal ligation takes place when both cross-linked fragment ends have intermediate length.

iii) Another potential bias comes from the GC content of the restriction fragments. Fragments with extreme GC content are underrepresented in the final interaction reads[1]. Deep sequencing appears to favor reads with a GC content of about 45%.

iv) The fourth bias is dependent on the length of Dna segments at the circularization steps (most human Hi-C protocols do not require a circularization step). The mechanical properties of the Dna polymer dictate that too small fragments will be poorly ligated due to high bending resistance, whereas too long fragments will also disfavor ligation due to entropic contribution to the free energy. Optimal circularization is achieved at around 500 bp. In addition, this bias is highly non-monotone in cycles of 10.5 bp. For instance, it favors the circularization length of 261 bp, but disfavors circularization length of 266 bp and again favors circularization length of 271 bp.

v) The last bias is associated with highly repetitive regions. Regions with low level of unique sequences are usually unmappable and hence underrepresented in the final interaction reads.

To eliminate these biases, Yaffe and Tanay [42] developed a probabilistic model to generate a more accurate interaction matrix in silico. This method was designed for human Hi-C data analysis and corrects for fragment ends length, GC content and mappability. It is computationally expensive and only accounts for known sources of bias. An iterative correction method (described in detail in 4.1) has been proposed later [18]. This method is less computationally demanding and does not rely on prior knowledge of the sources of bias. Both methods are generally accepted and applied in recent studies.

The Geometrical Coherence Problem Data from Hi-C experiments derive from millions of cells, and a deterministic translation from contact frequencies to

[1] However it is well known that CG content is also involved in chromatin behavior. It is therefore difficutl to separate an effect due to chromatin packaging from a supposed effect due to experimental procedure (i.e. the difficulty in sequenciung CGrich tracts).

Euclidean distances leads unavoidably to geometrical inconsistencies. The main problem is a violation of metric conditions such as the triangular inequality. This problem has been described in detail in [5]. Duggal *et al.* [10] developed an algorithm to select subsets of interactions that obey metric constraints of various strictness, embedding the set of distances in a subgraph with a lower number of metric violations. This kind of solution, although very interesting, is very computationally expensive.

3 Our Approach

3.1 The Model

We adopt a multi-scale modified bead-chain chromatin model, exploiting the fact that, in some genomic regions, the DNA sequences show many internal contacts between pairs of loci, and very weak interactions with the rest of the genome [8]. This entails a contact frequency matrix with a number of diagonal blocks with relatively large entries, and rows and columns whose entries are much smaller, almost anywhere else. Each diagonal block represents the mutual contacts within one of the above-mentioned regions (a topological domain), whose 3D configuration does not depend on the remainder of the chain. Thus, the structure of each topological domain can only be reconstructed from the data in the related diagonal block. Once the structures of all the topological domains have been reconstructed, their mutual relationships depend on the data outside the diagonal blocks. To account for such a lower resolution structure, we consider each topological domain as a single locus and bin the contact matrix so that each locus corresponds to a single entry. Then, a new block structure can be identified and estimated. This procedure can be repeated recursively until the lowest significant resolution is reached (corresponding to a full matrix, where no isolated diagonal blocks can be identified). The result is a chromatin model whose structure can be represented at multiple resolutions. As already done in many studies (see, e.g. [33]), we consider each locus, at any resolution, as a bead in a chain, but not as a simple sphere. Given a chromatin fiber composed of sub-chains of known structure, we try to find the mutual positions of the sub-chains without changing their internal configurations, so we treat them as non-deformable structures. To do this, we consider the geometric centroid, the start and the end-points of each subchain; these three points and their mutual positions constitute one bead of our model. The lengths of the segments joining the endpoints with the centroid, and the related angle, cannot be changed during the evolution of the model. Conversely, the planar and dihedral angles defining the position of each bead with respect to the adjacent ones (see Figure 11) can be varied, subject to possible constraints establishing flexibility and mutual distance ranges. The beads are linked respecting their biological order, with the end point of each bead coinciding with the start point of the next. Figure 10 illustrates how four consecutive sub-chains are schematized as modified beads and then connected to form a chain at a lower resolution. Of course, at the maximum allowed resolution, the structure of the topological domains is not known, so the centroid

and the endpoints of each subchain collapse into a single point or, equivalently, the beads in our chain are simply spheres. The advantages offered by this model consist in a better accuracy in the reconstruction of the chain at successive resolutions. The size of each bead is derived from its genomic span and from the related entry in the contact matrix. Indeed, if the contact matrix bin related to a particular bead shows many internal contacts, then it must have a small volume; otherwise, a few internal contacts mean that the related structure is less compact. Each bead is bound to its immediate neighbors in the genomic sequence; the length of each bond is such that the bead cannot penetrate its neighbors and cannot be too far apart from them. The angles between adjacent bonds are constrained so that the chain curvature cannot be higher than that permitted by biological and physical constraints. Finally, the overall size of the chain in its 3D configuration cannot exceed a certain value, for example, the size of the nucleus. Our approach is different from the one adopted by Rousseau *et al.* [35] and Hu *et al.* [16]: our constraints limit the feasible positions of any subset of loci, although they are not explicitly included in the energy function, which is constituted of a data-fit function only.

3.2 Data-Fit Function

For building our data-fit function, we tried to bypass the presence of bias and geometrical inconsistencies (see Section 2.4). To do this, we take into consideration only contact frequencies higher than a certain threshold, so we can assume that the error derived from neglecting the biases is not so big, and we do not convert frequencies into distances, but include the contact frequencies $n_{i,j}$ directly into the criterion. We assume that the bead pairs characterized by contact numbers above a certain threshold are likely to be in contact, whereas we do not say anything on the pairs whose contacts are below that threshold. The rationale for this choice is twofold: first, whatever their entity, the biases introduce the largest errors in the smallest contact frequencies; second, we do not try to enforce any target distance between the beads of any particular pair. We just say that pairs with higher contact frequencies must be in close contact, so we try to minimize this distance, weighted by the related contact frequency, and subject to all the constraints imposed on the whole chain. In formulas, let $\mathcal{C}$ be the 3D configuration of the chromatin segment under study (a matrix containing the coordinates of all the bead centers), $d_{i,j}$ be the Euclidean distance between the i-th and the j-th beads, and $\mathcal{L}$ be the set of pairs whose contact frequencies are above the threshold. Our data fit term is

$$\Phi(\mathcal{C}) = \sum_{i,j \in \mathcal{L}} n_{i,j} \cdot d_{i,j} \tag{1}$$

If $\mathbf{x}_i$ and $\mathbf{x}_j$ identify two bead centers, it is $d_{i,j} = ||\mathbf{x}_i - \mathbf{x}_j||$. Eq. (1) does not imply any restriction on the value of the threshold. Accepting a contact frequency to vanish simply means that the corresponding pair does not affect the data fit. Of course, all the configurations with $d_{i,j}$ vanishing for each (i, j)

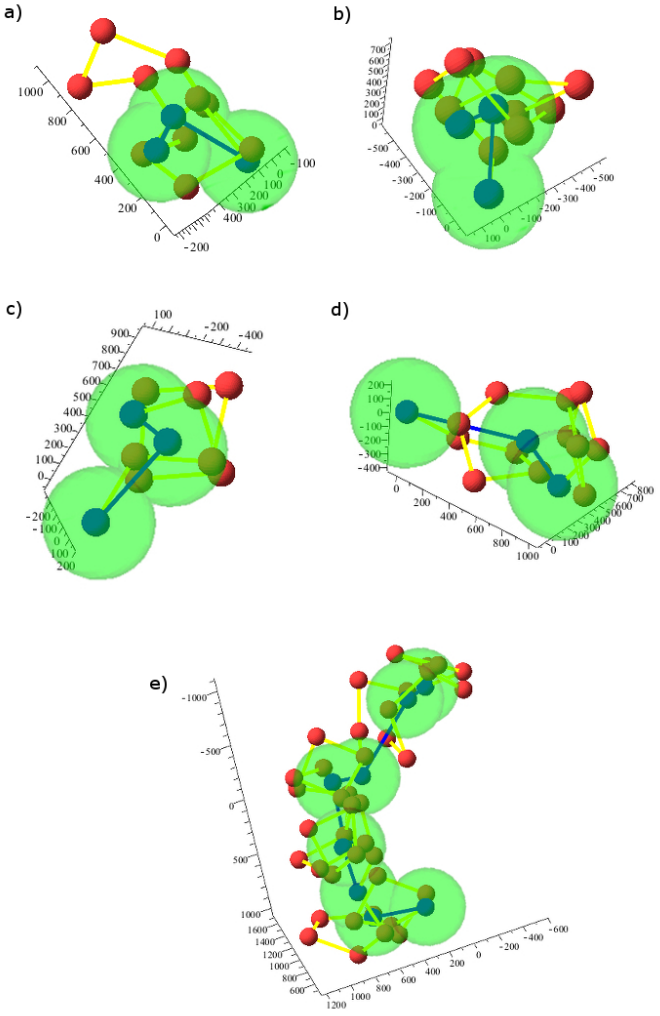

Figure10. a-d) Consecutive fragments of the chromatin fiber, represented as bead sequences (red balls linked by yellow segments), and as centroid end points triples (blue balls linked by blue segments). The green spheres represent the assumed sizes for the beads at the lower resolution. e) Lower-resolution chain composed by the fragments in a-d).

in $\mathcal{L}$ are unconstrained minimizers of (1). Each such configuration has all the pairs of loci in $\mathcal{L}$ in contact, and all the others in arbitrary positions. Constraining the set of feasible solutions through physical and biological knowledge, that is, describing any configuration as an instance of a geometric chromatin model, avoids these minima and produces solutions that are consistent with both the data and the prior knowledge. We do not look for a unique configuration because the experimental Hi-C data are not derived from a single cell, but from millions of cells, so we need to widely explore the solution space, in order to obtain different conformations corresponding to different minima of the data fit function.

3.3 Sampling the Solution Space

Let $\mathcal{C}$ be the configuration of a bead chain at any resolution. In our present implementation, we estimate the configuration $\mathcal{C}$ through the following probability density function

$$p(\mathcal{C}) \propto \ \mathrm{e}^{-\Phi(\mathcal{C})} \tag{2}$$

A number of high-likelihood configurations are obtained through a classical annealing scheme, where the distribution at each temperature is sampled by a Metropolis algorithm [22,31]. In synthesis, given the current chain configuration, a randomly perturbed configuration is proposed (see 3.4) and included in the sample upon a probabilistic test. During the iteration, the data fit term $\Phi(\mathcal{C})$ is modified by dividing it by a decreasing temperature parameter, in order to make the distribution more peaked around its maxima. When the temperature has reached its minimum value, the samples generated should be clustered around the set of absolute maxima of the distribution. The last configuration extracted is representative of one of these clusters and can be assumed as a highly plausible configuration for the subchain under study.

3.4 Evolution with Quaternions

To evolve our model, we use quaternions rather than Euler angles. Quaternions are an extension of the complex algebra, and represent a simple framework to understand and visualize rotations. They offer a number of advantages, being an intuitive way to represent rotations, through an angle and an axis of rotation, and allow a continuous evolution of the structure that is intrinsically compatible with the topological constraints. Furthermore, for their uniform coverage of the orientation space, quaternions avoid several problems involving rotations, such as singularities and numerical instabilities related to orthonormal matrices (e.g., gimbal lock [14]). Finally, using quaternions is less expensive than using Euler angles, as they only need to store 4 real numbers, as opposed to the 9 required by orthogonal matrices, and composing two rotations needs 16 multiplications and 12 additions, as opposed to the 27 multiplications and 18 additions needed

in matrix representation.[2] Quaternions are employed in many fields, including molecular dynamics and bioinformatics [15, 21, 27]. In appendix A we report an essential account on how they are used in our case. Further details on quaternion algebra can be found in [41]. To see how these properties are applied to perturb our model, let us consider the quadruple of consecutive beads in Fig. 11. The two triples $B_1 - B_2 - B_3$ and $B_2 - B_3 - B_4$ determine, respectively, the planes P_1 and P_2 and the associated planar angles ψ_1 and ψ_2. Planes P_1 and P_2, in turn, determine the dihedral angle φ_1. Our model is a series of concatenated quadruples of this type. Once all the distances between the centers of consecutive beads and all the planar and dihedral angles are fixed, the position of each bead with respect to all the others is defined, and can easily be perturbed to obtain different chain conformations complying with the constraints. The planar angles are perturbed by rotations around the normals to the corresponding planes, and the dihedral angles are perturbed by rotations around the intersection of the relevant planes. As an example, referring again to Fig. 11, a perturbation of angle ψ_1 is obtained by rotating vector $\overrightarrow{B_3B_2}$ around the direction of the cross product between $\overrightarrow{B_1B_2}$ and $\overrightarrow{B_3B_2}$; a perturbation of angle φ_1 is obtained by rotating vector $\overrightarrow{B_4B_3}$ around vector $\overrightarrow{B_3B_2}$. These operations maintain the chain topology, so the only constraint to be checked, if relevant, is the one that excludes spatial interference between beads.

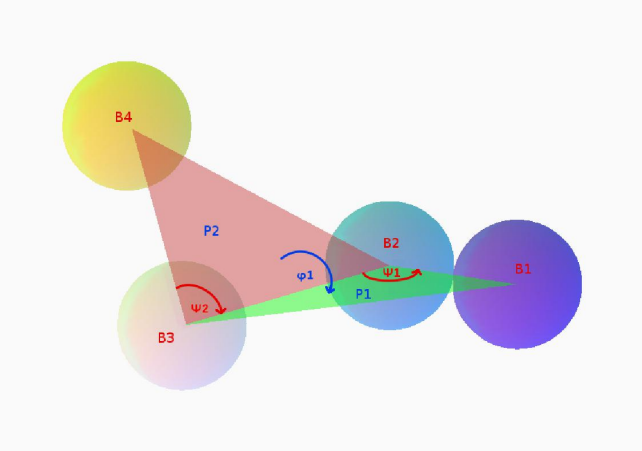

Figure11. Quadruple of consecutive beads in a chain model, identifying the planes P_1 and P_2, the planar angles ψ_1 and ψ_2 and the dihedral angle φ_1.

[2] http://www.geometrictools.com/Documentation/RotationIssues.pdf

4 Method

We adopted a recursive procedure, described in the following pseudo-code

`structure` = *procedure*(`cont.matr`, `constraints`)
1) *extract the diagonal blocks from* `cont.matr`
2) For all the extracted blocks

a) *Populate set* $\mathcal{L}$;
b) *Set the initial bead chain configuration* $\mathcal{C}_0$;
c) *Compute* $\Phi(\mathcal{C}_0)$ *as in* Eq. (1);
d) Iterate in i (assuming a cooling schedule $T_0 \rightarrow \ldots T_n \rightarrow \ldots$)
- *Check stop criterion: if* `stop criterion` *is satisfied*
  save $\mathcal{C}_i$
  leave
- *Generate* $\mathcal{C}^*$ *by perturbing randomly the bond lengths, the planar and the dihedral angles of the current configuration* $\mathcal{C}_i$
- *In the perturbed configuration, evaluate the distances between the beads belonging to the pairs in* $\mathcal{L}$;
- *Compute* $\Phi(\mathcal{C}^*)$
- *if* $\{\Phi(\mathcal{C}^*) < \Phi(\mathcal{C}_i)$ *or* $\text{random}[0,1] < \text{e}^{[\frac{\Phi(\mathcal{C}_i)-\Phi(\mathcal{C}^*)}{T_i}]}\}$ *and* `constraints` *are satisfied*
  $\mathcal{C}_{i+1} = \mathcal{C}^*$
  else
  $\mathcal{C}_{i+1} = \mathcal{C}_i$

3) *if # of diagonal blocks* $= 1$
`structure` $= \mathcal{C}$ (hierarchical composition of all the saved configurations)
output `structure`
leave
4) `constraints` = *geometrical features of all the sub-chains + parameters and constraints at the new resolution* (Fig. 11 a-d)
5) `cont.matr` = *bin*(`cont.matr`) (binning in accordance to the current blocks)
6) `structure` = *procedure*(`cont.matr`, `constraints`)

We wrote the procedure in Maple$^{\text{TM}}$ (release 17) and in Python (release 2.7.2), implementing this recursion for two hierarchical levels. The integral codes are reported in appendix B. At start, we can use external information on possible topological domains to extract the diagonal blocks at the maximum resolution; further binnings should be based on the values assumed by the matrix elements, possibly using some appropriately chosen threshold. Note that step 3 can be performed in parallel for all the extracted blocks. This means that possible parallel computing capabilities can be fully exploited. Note also that this procedure produces one overall structure, at maximum resolution, per run. As per the remarks in the previous section, different runs normally produce different structures. Another way to proceed, for each data and parameter set, is to save all the stable

configurations of all the subchains at any resolution, and then sample each such set to produce the structures at the subsequent resolution. This strategy allows us to produce a potentially very large set of solutions, while saving much computation time.

4.1 Future Improvements of the Method

Among the many possible improvements of the method, the main and more pressing are the following:

The Problem of The Biases In the present version of the algorithm, we do not take into consideration the issue of the biases. However the adoption of some biases normalization method is necessary, because more precise data could lead to more biologically realistic outputs. To solve the biases problem, many approaches have been adopted. Yaffe and Tanay, and Dixon developed two different algorithms to preliminarily clear data from biases. Yaffe and Tanay [42] introduced a probabilistic model and algorithm to estimate, via maximum-likelihood, the parameters needed to normalize empirical raw contact matrices. Dixon *et al.* [8] analyzed mammalian 3D genome organization and observed that, in the data normalized through their algorithm, TD (Topological Domains) are more evident with respect to the raw data. Another approach was adopted by Hu *et al.* in [16]. They developed a Bayesian method with a non-informative prior and a likelihood in the form of a Poisson distribution, to find consensus 3D conformations of Chromatin (BACH, BAyesian 3D Constructor for Hi-C data). They start from raw contact data, then biases are treated as model parameters to be estimated. Another approach was adopted by Imakaev *et al.* [18]. They developed a data-driven method for iterative correction of biases (ICE, Iterative Correction and Eigenvector decomposition). They do not assume specific sources of biases, and correct collectively for all the factors affecting experimental data. They assume and demonstrate that biases of contacts between two regions can be represented as the product of the individual biases of these regions: the expected contact frequency E_{ij}, for every pair of regions (i, j), can be written as $E_{ij} = B_i B_j T_{ij}$, where B_i and B_j are the biases in the two regions, and $\mathbf{T} = (T_{ij})$ is the sought matrix of relative contact probabilities. The maximum-likelihood estimate of B_i for every region is obtained by iteratively solving a system of equations. Imakaev *et al.* also assume that, being the known biases factorisable, also uncharacterized biases are likely to be factorisable, and can be removed by the ICE algorithm. The algorithm also decomposes the correct contact probability matrix $\mathbf{T}$ into independent interaction genomic tracks, to reveal the main features of chromosomal organization. The weights of the contributions of these tracks are represented by the eigenvalues of matrix $\mathbf{T}$. Sorting eigenvalues in descendant magnitude order, the first 13 are statistically significant ($p < 0.001$) and explain 72% of the total variance. Focusing on the first three, the first eigenvector provides a genomic track of inter-chromosomal interactions along the genome, and is correlated with many genomic features like replication

timing and many histone marks (epigenetic features). The second eigenvector is relative to the error propagation of the algorithm, and the third captures both centromere-centromere and telomere-telomere enrichment (especially for mouse data sets). We consider Imakaev's method the most efficient among the ones cited above, and intend to use it in our method.

Automatic Detection of Topological Domains In the present implementation, the topological domains are computed off-line and provided as an input to the algorithm. Adopting a bias cleaning method, the addition of a code that recognizes Topological Domains (starting from the block structure of the contact matrix) and automatically performs the division in sub-chains, would be simplify the algorithm.

5 Experiments

For our first experiments, we selected Hi-C data from the long arm of human chromosome 1, made available by Lieberman-Aiden *et al.* [26]. The original resolution of these data was 100 kbp. At each experiment, we partitioned these data with the help of the topological domains identified by Dixon *et al.* [8], thus obtaining 25 sub-chains of sizes ranging from 700 kbp to 1.8 Mbp, reconstructed at the original resolution. The data matrix was then binned so as to make a single entry from each of the blocks in the first partition, and the algorithm was run again to estimate the structure of the entire chain at the new resolution. The structures of the original and the binned matrices are shown in Figure 12.

We run a series of experiments to settle the most appropriate set of constraints, and parameters. The result is reported in Table 1.

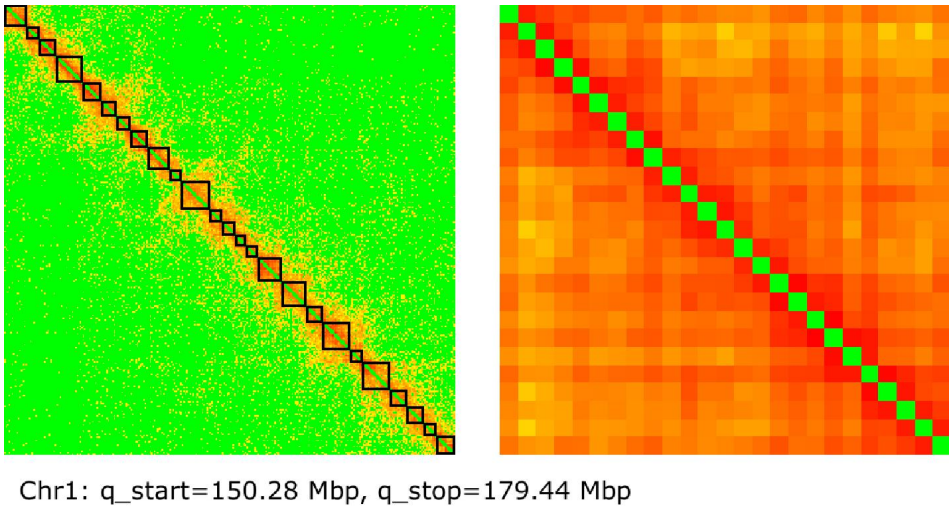

Figure12. Heat-maps, in logarithmic scale, representing the contact matrix for the long arm of chromosome 1, from a Hi-C experiment on human lymphoblastoid cells (GM06990, Lieberman-Aiden *et al.* [26]). Main diagonal removed for visualization convenience. Left: Original 292×292 matrix at a resolution of 100 kbp. The 25 highlighted diagonal blocks represent the contacts in the topological domains used for binning. Right: 25×25 matrix obtained by binning the original.

Table1: Values of parameters used in the experiment	
Initemp = initial temperature for simulated annealing	10000
Numiterationskb = max number of iterations for sub-chains	100000
NumiterationsMb = max number of iterations for low resolution chain	100000
Energythresholdkb = high resolution steps threshold without energy decrease	50
EnergythresholdMb = low resolution steps threshold without energy decrease	50
NumiterationsTemp = number of iterations for increasing temperature check	200
Epsilonchi = acceptance delay parameter for temperature increasing stop	0.10
Incremtemp = temperature increasing rate	1.50
Dimrate = temperature decreasing rate	0.99
Mincontactskb = min number of contacts in topological domains	15
Numcontkb = max number of contacts in topological domains	20
SparseMb = sparsity factor in low resolution binned contact matrix	8
Diagskb = number of neglected diagonals in sub-chains	2
DiagsMb = number of neglected diagonals in low resolution chain	2
Mindistkb = compenetration distance in sub-chains (nm)	120
MindistMb = compenetration distance in low resolution chain (nm)	120
Maxdistkb = max distances between beads (nucleus diameter, nm)	10000
Maxangle = max curvature angle (in degrees)	100

Using the parameters in Table 1, we produced 40 different estimated configurations. To check the plausibility of these results we used experimental data from cellular lines compatible with the ones used for our experiments. The HTMseq repository [3] stores the distribution of genes and their expression levels for a large number of human cells. Cells from normal bone marrow were selected for validation, as they belong to the lineage leading to immature B cells, the same type of our diploid human B-lymphoblastoid cell line GM06990 (Figure 13). Two stretches of about 3.5 Mbp were then selected as representative of, respectively, highly expressed and poorly expressed genomic domains, that is:
Dna from $q = 151.5$ Mbp to $q = 155.1$ Mbp, and Dna from $q = 162$ Mbp to $q = 165.5$ Mbp.

[3] http://bioinfo.amc.uva.nl/HTMseq/controller

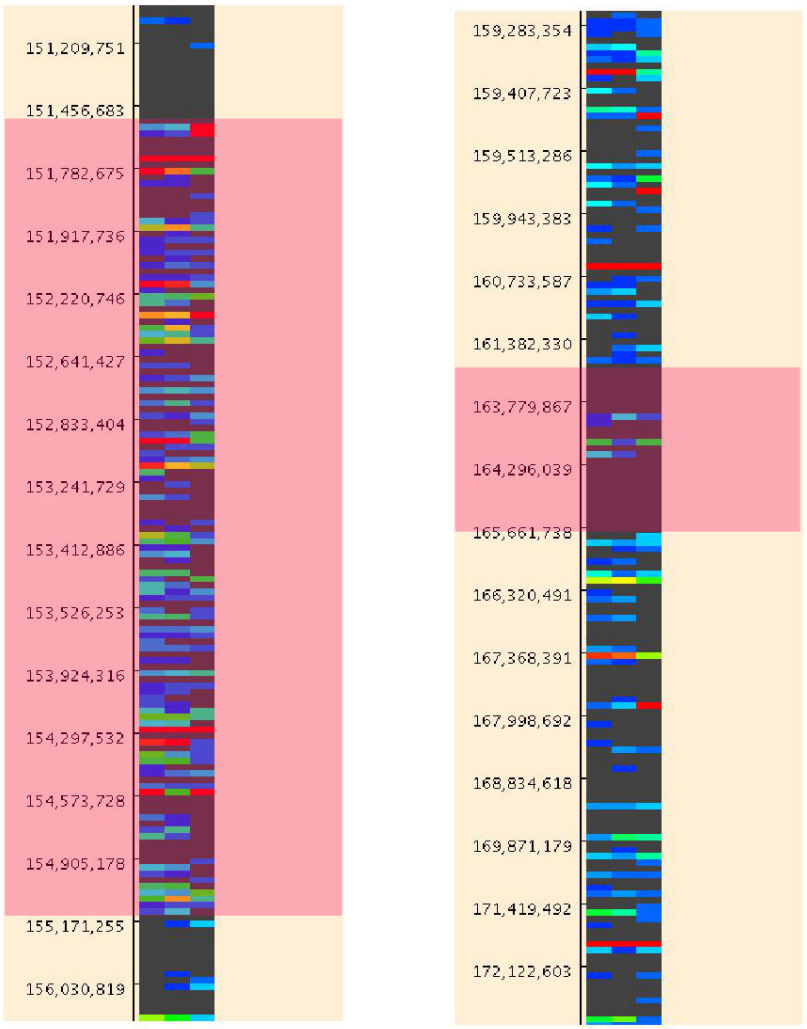

Figure13. Genomic expression levels representation for three bone marrow cellular lines. Left: stretch of highly expressed region from 151.5 Mbp to 155.1 Mbp. Right: stretch of poorly expressed region from 162 Mbp to 165.6 Mbp. Both regions have a span of 3.6 Mbp.

The highly expressed domains are known to be much less packed than the domains poor in genes or with low transcriptional activity [40]. Figure 13 represents a heat-map (from blue to red, low to high) of the genic expression levels in two regions of chromosome 1, one highly expressed (left) and one poorly ex-

pressed (right). To verify the existence of this property in our reconstructions, we follow Mateos-Langerak *et al.* [29] by comparing the genomic distance between pairs of loci with their Euclidean distance. Our reconstructed configurations are basically of two types, exemplified in Figure 14, top panels, with the highly and poorly transcribed regions marked in different colors. The plots in the bottom panels represent, for the two stretches, the mean-square Euclidean distances between pairs of loci as functions of their genomic distance. The difference between the two types of configurations is apparent from these plots: in the configuration on the left, the highly expressed domain is actually spread on a larger distance than the poorly expressed domain; in the configuration on the right, conversely, the highly expressed domain occupies a volume much smaller than expected. This could depend on insufficient constraints imposed in the estimation or could capture real configurations assumed in some phase of the cellular cycle. Figure 15 shows the two main clusters of output configurations, on the left the pear-like and on the right the globular ones. Clusters have been done by calculating the ratio between the integrals (on the whole range from 0 to 3.6 Mbp) of mean-square Euclidean distances between pairs of beads and genomic distances (see bottom of Figure 14). Figure 16 shows the box-plots for the two stretches, summarizing all the results from our 40 configurations. Apparently, the two stretches show a substantially different statistical behavior, and the poorly expressed region normally occupies much less space than the highly expressed region, although their genomic spans are nearly the same. The configurations obtained are compatible with biological information that has not been introduced into the problem. Indeed, the geometrical constraints we introduce are uniform along the chain, so the structural differences only depend on data.

For what concerns the computational cost, at present our Maple™ procedure with the matrix in Figure 12, a 292×292 contact matrix (representing the long arm of chromosome 1 binned at 100 Kb) subdivided in 25 domains of dimensions varying from 700 kbp to 1.8 Mbp, has the following elapsed times:

- $\sim$ 20 hours for the evolution of sub-chains ($\sim$ 4000 evolution steps)
- $\sim$ 30-40 hours for the evolution of the lower resolution chain ($\sim$ 5000 evolution steps)

Sub-chains simulations are performed in parallel on a 32Ghz core PC. The whole task in Python requires $\sim$ 1.5 hours on a normal 3Ghz core laptop, with serial data processing. Figure 17 shows the behavior of the energy (values of the data-fit function) during the evolution of the lower resolution chains. Looking at the contact matrix used for our experiments, we would expect the poorly expressed regions to have more contacts, since in our outputs they always result packed. This does not happen, as the number of contacts is about the same in the two regions; this means that the relationship between the number of contacts and the 3D structure is not direct. This relationship necessarily exists, and must be investigated; probably it depend on intrinsic and hidden structures of the contact matrix.

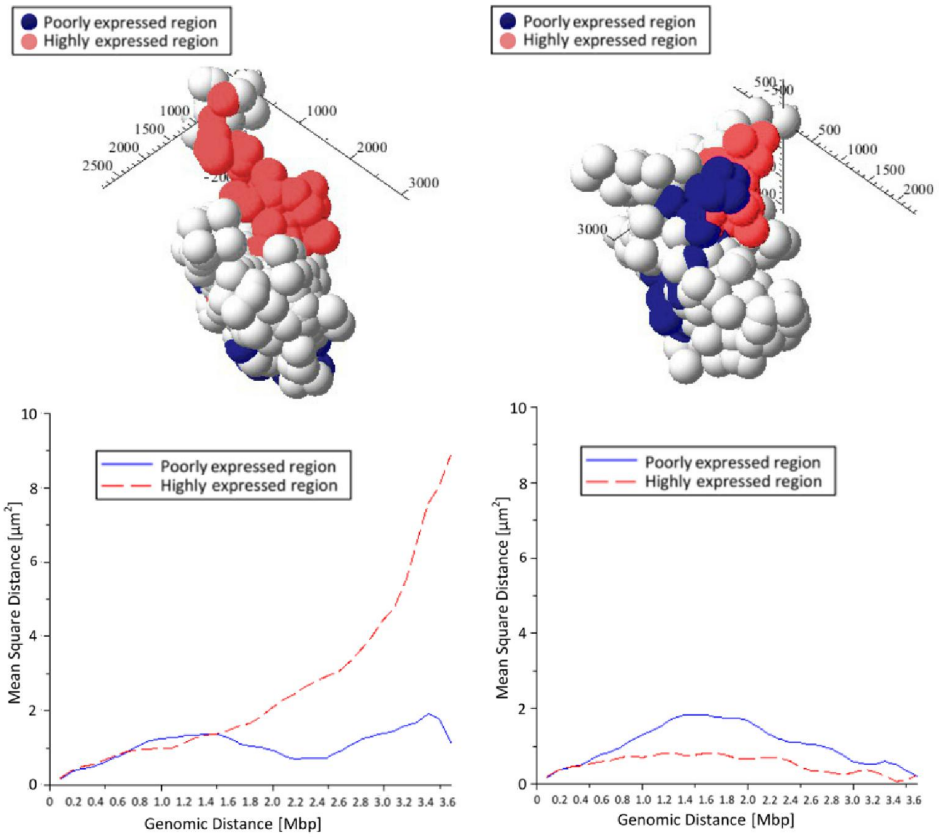

Figure14. Top: the two typical configurations resulting from the data in Figure 13 (measurements in nm). The model parameters used are: a) cardinality of sets $\mathcal{L}$ for the high-resolution sub-chains: 20; b) cardinality of set $\mathcal{L}$ for the low-resolution chain: 40; c) minimum distance between beads, at maximum resolution, to avoid interference: 120 nm; d) maximum distance between any two beads: 10 μm; e) maximum angle between two consecutive bead pairs (curvature): 100°. The red and blue beads belong, respectively, to a highly expressed and a poorly expressed regions. Bottom: mean-square Euclidean distances between pairs of beads, as functions of their genomic distances, for the red and the blue regions.

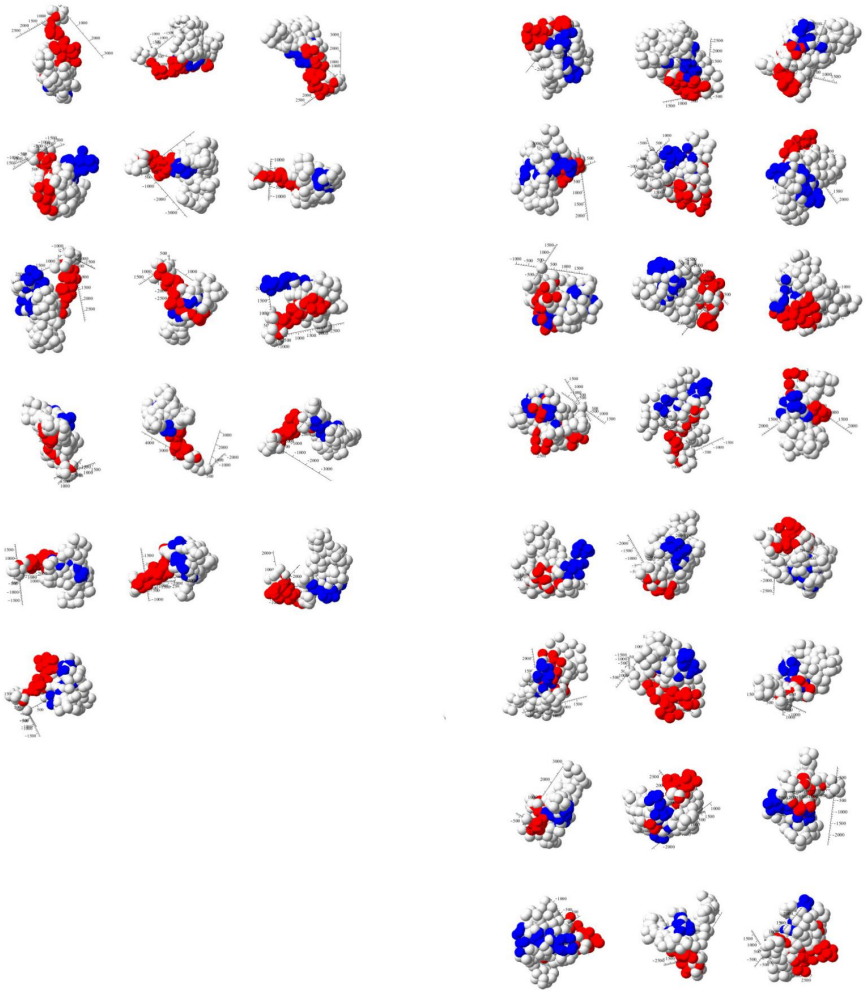

Figure15. Left: pear like configurations. Right: globular configurations.

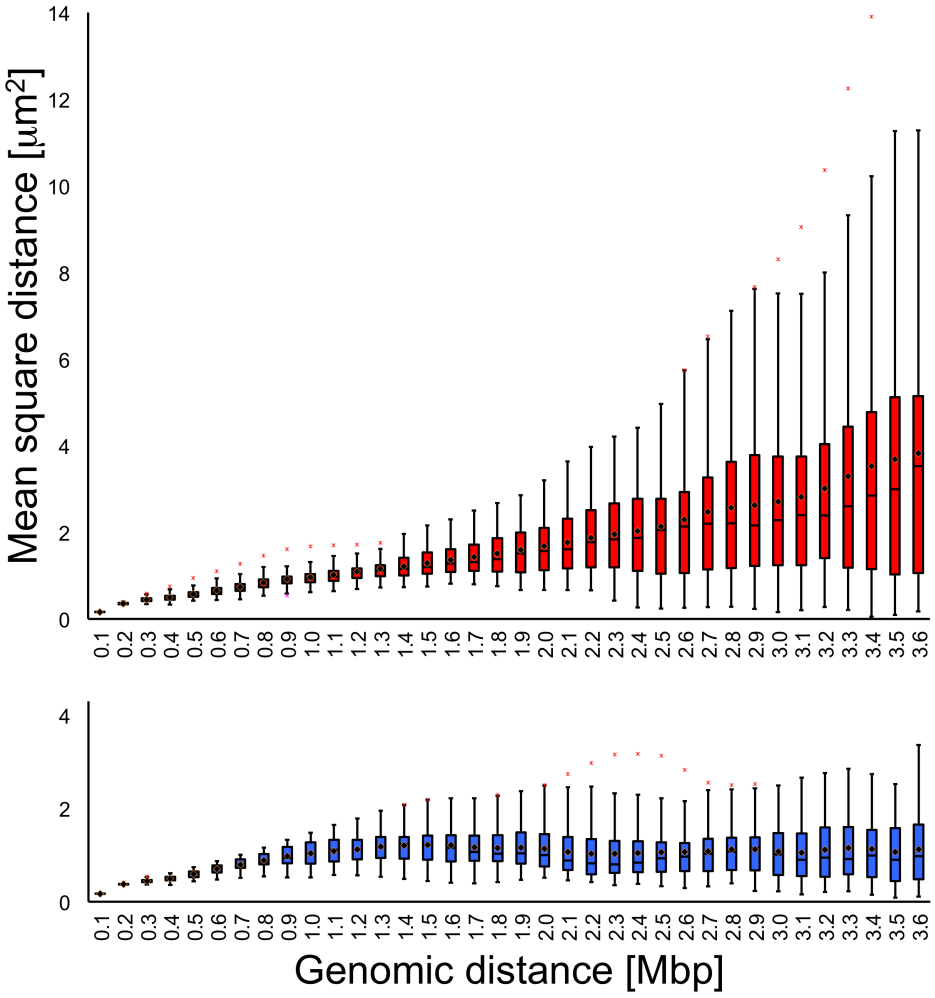

Figure16. Box-plots from 40 results obtained through the same parameter set. Top: highly expressed domain. Bottom: poorly expressed domain. The boxes include the second and third quartiles; the whiskers extend for (at most) 1.5 times the interquartile range above the 3rd and below the 2nd quartiles. Cross marks: extreme outliers; Diamond marks: mean values.

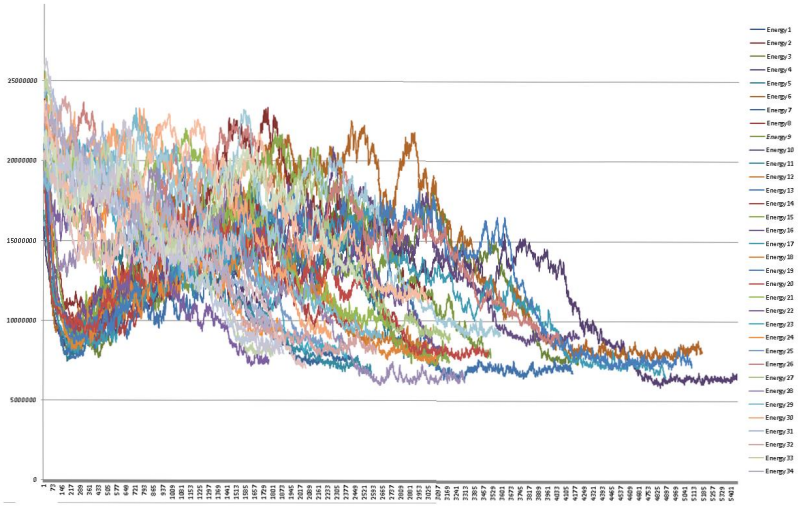

Figure17. Plot of energy for the lower resolution chains for the 40 configurations with the same parameter set. On the horizontal axes the steps, on the vertical axes the energy values.

6 Conclusions

We propose a new approach to estimate chromatin configurations from contact frequency data. The novelties introduced are:

- The modified bead-chain model evolved by quaternion operators.
- The data-fit function that does not require to translate frequencies into distances.
- The recursive structure of the algorithm that can be applied at different resolutions.

In order to keep the model compliant with known physical and biological features, any prior information available must be translated into geometrical constraints. Our first results from real Hi-C data are compatible with biological information. We demonstrated that structurally different regions in our reconstructions highly correlate with functionally different regions, as known from literature and genomic repositories. Besides extending the experimentation to more data and target features, our future activity will deal with the optimization of our code, as explained in section 4.1, in order to facilitate the choice of the most appropriate parameters, include all the available biological knowledge in the problem, and allow the structure to be estimated for larger and larger genomic regions.

Funding & Acknowledgments

This work has been funded by the Italian Ministry of Education, University and Research, and by the National Research Council of Italy, Flagship Project InterOmics, PB.P05 [4].
The authors are indebted to Luigi Bedini and Aurora Savino for helpful discussions.

[4] http://www.interomics.eu

References

1. Amann,R., Fuchs,B.M. (2008): Single-cell identification in microbial communities by improved fluorescence in situ hybridization techniques, Nature Reviews Microbiology 6: 339-348.
2. Baù,D., Marti-Renom,M.A. (2011): Structure determination of genomic domains by satisfaction of spatial restraints, Chromosome Research 19: 25-35.
3. Barbieri, M. et al. (2012): Complexity of chromatin folding is captured by the strings and binders switch model. Proc. Natl. Acad. Sci. U.S.A. 109: 16173-16178.
4. van Berkum,N.L. et al. (2010): Hi-C: a method to study the three-dimensional architecture of genomes, J. Vis. Exp. 39: 1869-1875.
5. Caudai,C. (2014): Ricostruzione tridimensionale della struttura della cromatina da dati tipo Chromosome Conformation Capture, InterOmics Flagship Project, Report cnr.isti/2014-PR-003, National Research Council of Italy - ISTI, Pisa.
6. Cournac, A., Marie-Nelly, H., Marbouty, M., Koszul, R., Mozziconacci, J. (2012): Normalization of a chromosomal contact map. BMC Genomics 13, 436.
7. Dekker,J. et al. (2002): Capturing chromosome conformation. Science 295: 1306-1311.
8. Dixon,J.R. et al. (2012): Topological domains in mammalian genomes identified by analysis ofchromatin interactions, Nature 485: 376-380.
9. Dostie,J., Dekker,J. (2007): Mapping networks of physical interactions between genomic elements using 5C technology, Nat. Protoc. 2: 988-1002.
10. Duggal,G. et al. (2013): Resolving spatial inconsistencies in chromosome conformation measurements, Algorithms for Molecular Biology 8, 8.
11. Duan,Z. et al. (2010): A three-dimensional model of the yeast genome, Nature 465: 363367.
12. Fraser,J. et al. (2009): Chromatin conformation signatures of cellular differentiation, Genome Biology 10, R37.
13. Gehlen,L.R. et al. (2012): Chromosome positioning and the clustering of functionally related lociin yeast is driven by chromosomal interactions, Nucleus 3: 370-383.
14. Grassia,F.S. (1998): Practical parameterization of rotations using the exponential map, J. Graph.Tools 3: 29-48.
15. Hanson,A.J., Thakur,S. (2012): Quaternion Maps of Global Protein Structure, J. Mol. Graph. Model. 38: 256-278.
16. Hu,M. et al. (2013): Bayesian inference of Spatial organizations of chromosomes, PLOS Comp. Biol. 9, 1002-893.
17. Hu,M. et al. (2013): Understanding spatial organizations of chromosomes via statistical analysis ofHi-C data, Quantitative Biology 1: 156-174.
18. Imakaev, M. et al. (2012): Iterative correction of Hi-C data reveals hallmarks of chromosome organization. Nat. Methods 9: 999-1003.
19. Iyer,B., et al. (2011): Hierarchies in eukaryotic genome organization: insights from polymer theory and simulations. BMC Biophys. 4, 8.
20. Kalhor,R. et al. (2012): Genome architectures revealed by tethered chromosome conformation capture and population-based modeling, Nature Biotechnol. 30: 90-100.
21. Karney,C.F. (2007): Quaternions in molecular modeling, J. Mol. Graph. Model. 25: 595-604.
22. Kirkpatrick,S. et al. (1983): Optimization by Simulated Annealing, Science 229: 671-680.

23. Lamond,A.I., Earnshaw,W.C. (1998): Structure and Function in the Nucleus, Science 280: 547-553.
24. Langer-Safer,P.R. et al. (1982): Immunological method for mapping genes on Drosophila polytenechromosomes, Proc. Natl. Acad. Sci. U.S.A. 79: 4381-4385.
25. Langowski,J. and Heermann,D.W. (2007): Computational modeling of the chromatin fiber. Semin. Cell. Dev. Biol. 18: 659-667.
26. Lieberman-Aiden,E. et al. (2009): Comprehensive Mapping of Long-Range Interactions Reveals Folding Principles of the Human Genome, Science 326: 289-293.
27. Magarshak,Y. (1993): Quaternion representation of RNA sequences and tertiary structures, Biosystems 30: 21-29.
28. Marti-Renom,M.A., Mirny,L.A. (2011): Bridging the resolution gap in structural modeling of 3D genome organization, PLoS Comput. Biol. 7.
29. Mateos-Langerak,J. et al. (2009): Spatially confined folding of chromatin in the interphase nucleus, PNAS 106: 3812-3817.
30. Meluzzi,D., Arya,G. (2013): Recovering ensembles of chromatin conformations from contact probabilities, Nucleic Acid Res. 41: 63-75.
31. Metropolis,N. et al. (1953): Equations of State Calculations by Fast Computing Machines, J. Chem. Phys. 21: 1087-1091.
32. Nagano, T., Lubling, Y., Stevens, T.J., Schoenfelder, S., Yaffe, E., Dean, W., Laue, E.D., Tanay, A., Fraser, P.(2013): Single-cell hi-c reveals cell-to-cell variability in chromosome structure. Nature 502: 59-64
33. Olins,A.L., Olins,D.E. (1974): Spheroid chromatin units (v bodies), Science 183: 330-332.
34. Rippe,K. (2001): Making contacts on a nucleic acid polymer. Trends Biochem. Sci. 26: 733-740.
35. Rousseau,M. et al. (2011): Three-dimensional modeling of chromatin structure from interaction frequency data using Markov chain Monte Carlo sampling, BMC Bioinformatics 12: 414-429.
36. Tanizawa,H. et al. (2010): Mapping of long-range associations throughout the fission yeast genome reveals global genome organization linked to transcriptional regulation. Nucleic Acids Res. 38: 8164-8177.
37. Tark-Dame,M. et al. (2011): Chromatin foldingfrom biology to polymer models and back. J. Cell. Sci. 124: 839-845.
38. Tokuda,N. et al. (2012): Dynamical modeling of three-dimensional genome organization in interphase budding yeast, Biophys. J. 102: 296-304.
39. Varoquaux,N. et al. (2014): A statistical approach for inferring the 3D structure of the genome, Bioinformatics 30: i26-i33.
40. Versteeg,R. et al. (2003): The Human Transcriptome Map Reveals Extremes in Gene Density, Intron Length, GC Content, and Repeat Pattern for Domains of Highly and Weakly Expressed Genes, Genome Res. 13: 1998-2004.
41. Vince,J.A. (2008): Geometric Algebra for Computer Graphics, Springer.
42. Yaffe,E., Tanay,A. (2011): Probabilistic modeling of Hi-C contact maps eliminates systematic biases to characterize global chromosomal architecture, Nature Genetics 43: 1059-1067.
43. Zhang, Y. et al. (2012): Spatial organization of the mouse genome and its role in recurrent chromosomal translocations. Cell 148: 908-921.
44. Zhang,Z.Z. et al. (2013): Inference of Spatial Organizations of Chromosomes Using Semi-definite Embedding Approach and Hi-C Data. In Deng, M. et al. (eds.), Research in Computational Molecular Biology, Springer-Verlag, Berlin Heidelberg 7821: 317-332.

45. 44. Zhao,Z. et al. (2006): Circular chromosome conformation capture (4C) uncovers extensive networks of epigenetically regulated intra- and inter-chromosomal interactions, Nat. Genet. 38: 1341-1347.

A Quaternion Algebra

Quaternions were introduced by sir William Rowan Hamilton in 1843, as an extension of the complex algebra.

A complex number has the form

$$\mathbf{z} = a + ib \tag{3}$$

where a and b are real numbers.
Hamilton extended the definition in 3 to a 4-uple

$$\mathbf{z} = a + ib + jc + kd \tag{4}$$

where a, b, c and d are real numbers.
Imaginary units obey to the following rules

$$\begin{gathered} ij = k \quad jk = i \quad ki = j \quad ji = -k \quad kj = -i \\ ik = -ji2 = j2 = k2 = ijk = -1 \end{gathered} \tag{5}$$

Rules 5 make Quaternions an anticommuting Algebra.

A.1 Adding Quaternions

Two quaternions $\mathbf{q}_1$ and $\mathbf{q}_2$

$$\mathbf{q}_1 = s_1 + ix_1 + jy_1 + kz_1 \tag{6}$$

$$\mathbf{q}_2 = s_2 + ix_2 + jy_2 + kz_2 \tag{7}$$

are equal if and only if their corresponding terms are equal.
Furthermore, like vectors, they can be added or subtracted as follows:

$$\mathbf{q}_1 \pm \mathbf{q}_2 = [(s_1 \pm s_2) + i(x_1 \pm x_2) + j(y_1 \pm y_2) + k(z_1 \pm z_2)] \tag{8}$$

A.2 The Quaternion Product

Given two quaternions $\mathbf{q}_1$ and $\mathbf{q}_2$ as in 6 and in 7, their product is given by

$$\begin{gathered} \mathbf{q}_1\mathbf{q}_2 = (s_1 + ix_1 + jy_1 + kz_1)(s_2 + ix_2 + jy_2 + kz_2) = \\ = s_1s_2 + is_1x_2 + js_1y_2 + ks_1z_2 + \\ + ix_1s_2 + i_1^{2x}x_2 + ijx_1y_2 + ikx_1z_2 + \\ + jy_1s_2 + jiy_1x_2 + j_1^{2y}y_2 + jky_1z_2 + \\ + kz_1s_2 + kiz_1x_2 + kjz_1y_2 + k_1^{2z}z_2 + \end{gathered} \tag{9}$$

substituting 5 and collecting

$$\begin{aligned}\mathbf{q}_1\mathbf{q}_2 = s_1s_2 - (x_1x_2 + y_1y_2 + z_1z_2) + \\ +s_1(ix_2 + jy_2 + kz_2) + s_2(ix_1 + jy_1 + kz_1) + \\ +j(y_1z_2 + z_1y_2) + j(z_1x_2 + x_1z_2) + k(x_1y_2 + y_1x_2)\end{aligned} \tag{10}$$

The product $\mathbf{q}_1\mathbf{q}_2$ is still a quaternion. The product $\mathbf{q}_1\mathbf{q}_2$ is not equal to the product $\mathbf{q}_2\mathbf{q}_1$ because the scalar product commutes, but the vector product anticommutes.

Pure quaternions are quaternions with the scalar term equal to zero. The product of two pure quaternions is no longer a pure quaternion. Algebra of pure quaternions is not closed.

A.3 The Magnitude of a Quaternion

Given the quaternion

$$\mathbf{q} = s + ix + jy + kz \tag{11}$$

Its magnitude is defined as

$$||\mathbf{q}|| = \sqrt{s^2 + x^2 + y^2 + z^2} \tag{12}$$

Unit quaternions are quaternions with magnitude equal to 1.

A.4 The Conjugate of a Quaternion

Given the quaternion $\mathbf{q}$ defined in 11, its conjugate is defined as

$$\bar{\mathbf{q}} = s - ix - jy - kz \tag{13}$$

A.5 The Inverse of a Quaternion

Given the quaternion $\mathbf{q}$ defined in 11, its inverse is defined as

$$\mathbf{q}^{-1} = \frac{s - ix - jy - kz}{||d||^2} = \frac{\bar{\mathbf{q}}}{||d||^2} \tag{14}$$

A.6 Quaternion Algebra

Closure
For all $\mathbf{q}_1$ and $\mathbf{q}_2$ in the quaternion algebra Q

$$\begin{aligned} &\textit{addition} && \mathbf{q}_1 + \mathbf{q}_2 \in Q \\ &\textit{multiplication} && \mathbf{q}_1\mathbf{q}_2 \in Q \end{aligned} \tag{15}$$

Identity

For each $\mathbf{q} \in Q$ there is an identity element for the addition and for the multiplication

$$\begin{array}{ll} \textit{addition} & \mathbf{q} + \mathbf{0} = \mathbf{0} + \mathbf{q} = \mathbf{q} \qquad \mathbf{0} = 0 + 0i + 0j + 0k \\ \textit{multiplication} & \mathbf{q1} = \mathbf{1q} = \mathbf{q} \qquad \mathbf{1} = 1 + 0i + 0j + 0k \end{array} \tag{16}$$

Inverse

For each $\mathbf{q} \in Q$ there is an inverse element for the addition and for the multiplication

$$\begin{array}{ll} \textit{addition} & \mathbf{q} + (-\mathbf{q}) = -\mathbf{q} + \mathbf{q} = \mathbf{0} \\ \textit{multiplication} & \mathbf{q}(\mathbf{q}^{-1}) = \mathbf{q}^{-1}\mathbf{q} = \mathbf{1} \quad (\mathbf{q} \neq \mathbf{0}) \end{array} \tag{17}$$

Associativity

For all $\mathbf{q}_1$, $\mathbf{q}_2$ and $\mathbf{q}_3$

$$\begin{array}{ll} \textit{addition} & \mathbf{q}_1 + (\mathbf{q}_2 + \mathbf{q}_3) = (\mathbf{q}_1 + \mathbf{q}_2) + \mathbf{q}_3 \\ \textit{multiplication} & \mathbf{q}_1(\mathbf{q}_2\mathbf{q}_3) = (\mathbf{q}_1\mathbf{q}_2)\mathbf{q}_3 \end{array} \tag{18}$$

Commutativity

For all $\mathbf{q}_1$ and $\mathbf{q}_2$

$$\begin{array}{ll} \textit{addition} & \mathbf{q}_1 + \mathbf{q}_2 = \mathbf{q}_2 + \mathbf{q}_1 \\ \textit{multiplication} & \mathbf{q}_1\mathbf{q}_2 \neq \mathbf{q}_2\mathbf{q}_1 \end{array} \tag{19}$$

Distributivity

For all $\mathbf{q}_1$, $\mathbf{q}_2$ and $\mathbf{q}_3$

$$\begin{array}{ll} \textit{addition} & \mathbf{q}_1(\mathbf{q}_2 + \mathbf{q}_3) = \mathbf{q}_1\mathbf{q}_2 + \mathbf{q}_1\mathbf{q}_3 \\ \textit{multiplication} & (\mathbf{q}_1 + \mathbf{q}_2)\mathbf{q}_3 = \mathbf{q}_1 q_3 + \mathbf{q}_2\mathbf{q}_3 \end{array} \tag{20}$$

A.7 Rotating Vectors using Quaternions

The rotation of a position vector $\mathbf{v}$, represented as the pure quaternion $\mathbf{v} = v_x i + v_y j + v_z k$, of an angle θ around a direction identified by the pure quaternion $\mathbf{u} = u_x i + u_y j + u_z k$, can be expressed through the map $\mathbf{v} \longrightarrow \mathbf{R}_q(\mathbf{v}) = \mathbf{q}\mathbf{v}\bar{\mathbf{q}}$, where $\mathbf{q}$ is the unit quaternion

$$\mathbf{q} = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}u_x i + \sin\frac{\theta}{2}u_y j + \sin\frac{\theta}{2}u_z k \tag{21}$$

and $\bar{\mathbf{q}}$ is the unit quaternion

$$\bar{\mathbf{q}} = \cos\frac{\theta}{2} - \sin\frac{\theta}{2}u_x i - \sin\frac{\theta}{2}u_y j - \sin\frac{\theta}{2}u_z k \tag{22}$$

The composition of two rotations is simply obtained through the product of the corresponding quaternions: $\mathbf{R_p}(\mathbf{R_q}(\mathbf{v})) = \mathbf{R_{pq}}(\mathbf{v})$. Note that, as the quaternion product is non-commutative, it is $\mathbf{R_{pq}} \neq \mathbf{R_{qp}}$. Also, since quaternions $\mathbf{q}$ and $-\mathbf{q}$ give the same rotation (changing the sign of $\mathbf{q}$ is increasing θ of 2π), it is $\mathbf{R_q} = \mathbf{R_{-q}}$.

B Integral Codes

B.1 Code in Maple™ (release 17)

```
> numcontkb := 20;
> sparseMb := 8;
> #sparsekb:=2:
> diagskb := 2;
> diagsMb := 2;
> mindistkb := 120;
> mindistMb := 110;
> maxdistkb := 5000;
> maxangle := 100;
> alpha0 := readdata("C:/Users/claudia/Desktop/Maple-Chromatin/Chain-Maple/
ini_values/alpha.txt");
> alpha0 := convert(alpha0, Vector);
> beta0 := readdata("C:/Users/claudia/Desktop/Maple-Chromatin/Chain-Maple/
ini_values/beta.txt");
> beta0 := convert(beta0, Vector);
> domin := readdata("C:/Users/claudia/Desktop/Maple-Chromatin/Chr1_11/
dom_chr1.txt", 2);
> m := trunc(domin[nops(domin), 2]);
> HM := readdata("C:/Users/claudia/Desktop/Maple-Chromatin/Chr1_11/
100kb_chr1_short_txt.txt", m);
> HM := convert(HM, Matrix);
> nbinMb := nops(domin);
> numcontMb := trunc(nbinMb^2/(2*sparseMb));
> dom := Matrix(nbinMb, 2);
> for l to nbinMb do
dom(l, 1) := trunc(domin[l, 1]);
dom(l, 2) := trunc(domin[l, 2]) end do;
> arad := evalf(convert(maxangle*degrees, radians));
> rot := proc (v, q) options operator, arrow; ((2*q[1]^2-1+2*q[2]^2)*v[1]+
(2*q[2]*q[3]-2*q[1]*q[4])*v[2]+
(2*q[2]*q[4]+2*q[1]*q[3])*v[3], (2*q[2]*q[3]+2*q[1]*q[4])*v[1]+
(2*q[1]^2-1+2*q[3]^2)*v[2]+
(2*q[3]*q[4]-2*q[1]*q[2])*v[3], (2*q[2]*q[4]-2*q[1]*q[3])*v[1]+
(2*q[3]*q[4]+2*q[1]*q[2])*v[2]+
(2*q[1]^2-1+2*q[4]^2)*v[3]) end proc;
> fincoords := []; findists := []; collisionskb := []; stepsenkb := [];
> a := rand(-10 .. 10);
> segm:=proc(numchrom::integer,frag::integer,n1::integer,n2::integer,
mincontacts::integer,numcontkb::integer,diags::integer,mindist::integer,
maxdist::integer,numiterations::integer)
```

```
local MD,diam,vect,pHM, pHMM,inienergy;
local l,t,s, no,vectinizkb, HM2, n,minkb,contenergy;
local M,c,cc,energy,vt,v1,al,be,Mcoord,temp;
local finalcoords, b, g, en, den, pen, contiter, contacc, chitemp, gg;
global fincoord, collisionkb,stepenkb;
n:=n2-n1+1:
MD:=Matrix():
diam:=Vector(n):
vect:=Vector(n-1):
pHM:=[]:
pHMM:=[]:
collisionkb:=[]:
stepenkb:=[]:
fincoord:=Matrix(n,3):
lprint('il frammento ',frag,' ha ',n,' bin'):
for l from 1 to n do
diam(l):=500-(HM(n1+l-1,n1+l-1)*300)/(1000);
end do:
for l from 1 to (n-1) do
vectinizkb(l):=(diam(l))/(2)+(diam(l+1))/(2);
vect(l):=(diam(l))/(2)+(diam(l+1))/(2);
end do:
for l from 1 to n-2 do
alpha(l):=alpha0(l):
beta(l):=beta0(l):
end do:
for l from 1 to n-1 do
for t from 1 to n-1 do
if (t-l)>=diags then
pHM:=[HM(n1+l,n1+t),op(pHM)];
end if:  end do:  end do:
pHM:=Sort(pHM,order=descending):
if pHM[1]<=mincontacts then
minkb:=pHM[1]:
else   if nops(pHM)>=numcontkb then
minkb:=pHM[numcontkb]:
else   minkb:=pHM[nops(pHM)]:
end if:  end if:
for l from 1 to n-1 do
for t from 1 to n-1 do
if (HM(n1+l,n1+t)>=minkb and (t-l)>=diags) then
lprint('nel frammento',frag,'del cromosoma',numchrom,'ci sono',
HM(n1+l,n1+t),'contatti fra il bin', n1+l,'e il bin',n1+t):
pHMM:=[<HM(n1+l,n1+t),l+1,t+1>,op(pHMM)];
end if:  end do:  end do:
```

```
no:=nops(pHMM);
M:=Matrix(n,n):
c:=Vector(numiterations):
cc:=Vector(numiterations):
energy:=Vector(numiterations):
vt:=Vector(n-1):v1:=Vector(n-1):
al:=Vector(n-2):be:=Vector(n-2):
Mcoord:=Matrix(n,3):
temp:=initemp:
contiter:=0:
contacc:=0:
gg:=0:
contenergy:=0:
s:=0:
while (contenergy<energytresholdkb and s<=numiterations) do
s:=s+1:
contenergy:=contenergy+1:
contiter:=contiter+1:
chitemp:=evalf((contacc+1)/(contiter)):
if contiter=numiterationsTemp and gg=0 then
if chitemp<(1-epsilonchi) then
lprint('dopo',contiter,'iterazioni il rapporto tra transizioni
accettate e proposte risulta',chitemp):
contiter:=0:
contacc:=0:
contenergy:=0:
temp:=temp*incremtemp:
lprint('la temperatura sale a',temp):
else
lprint('al passo',s,'il rapporto risulta',chitemp,'e la temperatura
 inizia a diminuire con un valore di',temp):
contiter:=0:
contacc:=0:
contenergy:=0:
gg:=1:
end if:     end if:
if gg=1 then
temp:=temp*dimrate:
end if:
for t from 1 to n-1 do
b:=a():
if vectinizkb(t)-20<=vect(t)+evalf(b)<=vectinizkb(t)+20 then
vt(t):=vect(t)+evalf(b) else
vt(t):=vect(t)-evalf(b)
end if:     end do:
```

```
for t from 1 to n-2 do
b:=(a())/(100):
if -arad<= evalf(alpha(t))+evalf(b)<=arad then
al(t):=alpha(t)+evalf(b) else
al(t):=alpha(t)-evalf(b)
end if:     end do:
for t from 1 to n-2 do
b:=(a())/(100):
be(t):=beta(t)+evalf(b):
end do:
#‘PLANAR ANGLES‘
p[al](1):=<0,0,0>:
p[al](2):=<vt(1),0,0>:
alp(1):=al(1):
for t from 3 to n do
alp(t-2):=evalf(add(al(x),x=1..t-2)):
p[al](t):=<p[al](t-1)[1]+evalf((cos(alp(t-2)))*vt(t-1)),
p[al](t-1)[2]+evalf((sin(alp(t-2)))*vt(t-1)),0>:
d(t):=<p[al](t)(1)-p[al](t-1)(1),p[al](t)(2)-p[al](t-1)(2),
p[al](t)(3)-p[al](t-1)(3)>:
normpa(t,s):=norm(d(t),2);    od:
for t from 1 to n do
lprint(alp(t),"alp",p[al](t),"pal",d(t),"dist",t):
#‘DIHEDRAL ANGLES‘
pr(1):=p[al](1):
pr(2):=p[al](2):
qua(1):=1:qua(2):=1:
for t from 2 to n do
v1(t-1):=<p[al](t)[1]-p[al](t-1)[1],p[al](t)[2]-p[al](t-1)[2],
p[al](t)[3]-p[al](t-1)[3]>:
v(t-1):=<(p[al](t)[1]-p[al](t-1)[1])/(vt(t-1)),
(p[al](t)[2]-p[al](t-1)[2])/
(vt(t-1)),(p[al](t)[3]-p[al](t-1)[3])/(vt(t-1))>:    od:
for t from 3 to n do
 q(t):=Qunit(cos(be(t-2)/2)+((v(t-2)[1])*i+(v(t-2)[2])*j+
(v(t-2)[3])*k)*sin(be(t-2)/2)):
qua(t):=evalf(qua(t-1)*q(t)):
p(t):=rot(v1(t-1),
<Qscalar(qua(t)),Qicoeff(qua(t)),Qjcoeff(qua(t)),Qkcoeff(qua(t))>):
 pr(t):=<p(t)[1]+pr(t-1)[1],p(t)[2]+pr(t-1)[2],p(t)[3]+pr(t-1)[3]>:
od:
for t from 2 to n do
 d(t):=<pr(t)[1]-pr(t-1)[1],pr(t)[2]-pr(t-1)[2],pr(t)[3]-pr(t-1)[3]>:
normpr(t,s):=norm(d(t),2):     od:
#‘DISTANCE MATRIX ‘
```

```
for g from 1 to n do
for t from 1 to n do
 d(g,t):=<pr(g)[1]-pr(t)[1],pr(g)[2]-pr(t)[2],pr(g)[3]-pr(t)[3]>:
M(g,t):=norm(d(g,t),2):
od:   od:
lprint(M):
for l from 1 to n do
Mcoord(l,1):=pr(l)(1):
Mcoord(l,2):=pr(l)(2):
Mcoord(l,3):=pr(l)(3):   od:
c(s):=0:cc(s):=0:
for g from 1 to n do
for t from g+1 to n do
if M(g,t)<=mindist then
c(s):=c(s)+1:
lprint(M(g,t),'minimdist',s):
collisionkb:=[op(collisionkb),<frag,g,t,s>]:
elif M(g,t)> maxdist then
cc(s):=cc(s)+1:
lprint(M(g,t),'maximdist',s):
end if:     od:   od:
en:=0:
for l from 1 to no do
en:=en+M(pHMM[l][2],pHMM[l][3])*HM(pHMM[l][2],pHMM[l][3]):
end do:
energy(s):=en:
if s=1 then
lprint('energia iniziale', energy(s));
inienergy:=energy(s):   end if:
if s>1 and (c(s)=0 and cc(s)=0)  then
#'if s>1 then'
den:=energy(s)-inienergy:
#lprint('crom',numchrom,'framm',frag,'ener al passo', s, energy(s),
'ener iniziale', inienergy,'differ',den):
if den<=0 then
inienergy:=energy(s):
for t from 1 to n-1 do
vect(t):=evalf(vt(t)):
end do:
for t from 1 to n-2 do
alpha(t):=evalf(al(t)):
beta(t):=evalf(be(t)):
end do:
contacc:=contacc+1:
#lprint('calo',contacc,contiter,den):
```

```
contenergy:=0:
#ExportMatrix(coords[s],Mcoord,target=delimited):
stepenkb:=[op(stepenkb),<frag,energy(s),den,s>]:
for l from 1 to n do
fincoord(l,1):=Mcoord(l,1):
fincoord(l,2):=Mcoord(l,2):
fincoord(l,3):=Mcoord(l,3):
od:
else
pen:=exp(-den/(temp)):
if s=1 or pen>=|(a())/(10)| then
inienergy:=energy(s):
for t from 1 to n-1 do
vect(t):=evalf(vt(t)):
end do:
for t from 1 to n-2 do
alpha(t):=evalf(al(t)):
beta(t):=evalf(be(t)):
end do:
contacc:=contacc+1:
#lprint('probab',contacc,contiter,den,temp,pen):
contenergy:=0:
#ExportMatrix(coords[s],Mcoord,target=delimited):
stepenkb:=[op(stepenkb),<frag,energy(s),den,s>]:
for l from 1 to n do
fincoord(l,1):=Mcoord(l,1):
fincoord(l,2):=Mcoord(l,2):
fincoord(l,3):=Mcoord(l,3):
od:
end if:    end if:  end if:
#lprint('contatore energia',contenergy): end do:
lprint('la simulazione si `` interrotta allo step',contiter,'dopo',
contenergy,'steps di energia stabile'):
lprint('temperatura finale',temp,'energia finale', energy(s)):
finalcoords:=convert(cat("Chrom",numchrom,"Binkb",frag),name):
ExportMatrix(finalcoords,Mcoord,target=delimited):
end proc:
> for l to nbinMb do
segm(1, l, dom(l, 1), dom(l, 2), mincontactskb, numcontkb,
 diagskb, mindistkb, maxdistkb, numiterationskb);
fincoords := [op(fincoords), fincoord];
findists := [op(findists), findist];
collisionskb := [op(collisionskb), collisionkb];
stepsenkb := [op(stepsenkb), stepenkb] end do;
> stepsenkb := convert(stepsenkb, Matrix);
```

```
ExportMatrix("stepsenkb.txt", stepsenkb, target = delimited);
> collisionskb := convert(collisionskb, Matrix);
ExportMatrix("collisionskb.txt", collisionskb, target = delimited);
> plan := Vector(nbinMb);
CHI := Vector(nbinMb);
> B := []; vect1 := []; vect2 := [];
> for l from 1 to nbinMb do
#‘ BARYCENTERS ‘
 ps:=[]:
nl:=nops(convert(Column(fincoords[l],1),list));
for t from 1 to nl do
ps:=[op(ps),point(convert(cat("point",t),name),
fincoords[l](t,1),fincoords[l](t,2),fincoords[l](t,3))]:   end do:
B:=[op(B),coordinates(centroid(C,ps))]:
#VECTORS
v1:=<B[l,1],B[l,2],B[l,3]>:
v2:=<fincoords[l](nl,1)-B[l,1],fincoords[l](nl,2)-B[l,2],
fincoords[l](nl,3)-B[l,3]>:
vect1:=[op(vect1),norm(v1,2)]:
#lprint(vect1[l],vector1,l):
vect2:=[op(vect2),norm(v2,2)]:
#lprint(vect2[l],vector2,l):
#‘PLANAR ANGLES‘
vl1:=convert(v1,list):
vl2:=convert(v2,list):
plan(l):=sign(a())*Re(evalf(arccos(DotProduct(vl1,vl2)/
(vect1[l]*vect2[l])))):
print(plan(l),planar angle,l):
#‘DHIEDRAL ANGLES‘
if l=1 then
CHI(l):=0:print(CHI(l),dhiedral angle,l):
else
vs1:=<B[l-1][1]-fincoords[l-1](1,1),B[l-1][2]-fincoords[l-1](1,2),
B[l-1][3]-fincoords[l-1](1,3)>:
vs2:=<fincoords[l](nl,1)-B[l-1][1],fincoords[l](nl,2)-B[l-1][2],
fincoords[l](nl,3)-B[l-1][3]>:
vvett1:=<CrossProduct(vs1,vs2)[1],CrossProduct(vs1,vs2)[2],
CrossProduct(vs1,vs2)[3]>:
 lvett1:=convert(vvett1,list):
vvett2:=<CrossProduct(v1,v2)[1],CrossProduct(v1,v2)[2],
CrossProduct(v1,v2)[3]>:
lvett2:=convert(vvett2,list):
vsintor:=<CrossProduct(vvett2,vs1)[1],
CrossProduct(vvett2,vs1)[2],
CrossProduct(vvett2,vs1)[3]>:
```

```
lsintor:=convert(vsintor,list):
sinsin:=DotProduct(lsintor,lvett1):
if sinsin >= 0   then
CHI(l):=Re(evalf(arccos(DotProduct(lvett1,lvett2)/
(norm(vvett1,2)*(norm(vvett2,2))))))
else
CHI(l):=Re(-evalf(arccos(DotProduct(lvett1,lvett2)/
(norm(vvett1,2)*(norm(vvett2,2))))))
end if:
print(CHI(l),dhiedral angle,l):
end if: end do:
> HMM := Matrix(nbinMb);
> for l to nbinMb do for s to nbinMb do
HMM[l, s] := add(add(HM[r, g], r = dom(l, 1) .. dom(l, 2)),
g = dom(s, 1) .. dom(s, 2))
end do end do;
> mainchain := proc (numchrom::integer, frag::integer, n1::integer,
 n2::integer, numcontMb::integer, diags::integer, mindist::integer,
maxdist::integer, numiterations::integer)
local MD, vect, pHM, pHMM, inienergy, contenergy, l, t, s, no,
vectinizMb, n, minMb, M, c, cc, d, energy, vt, v1, al, be,
Mcoord, temp, b, g, en, den, pen, contiter, contacc, chitemp, gg;
global finchi, coordMb, stepenMb, collisionMb;
n := 2*n2-2*n1+3;
MD := Matrix(n);
vect := Vector(n);
pHM := [];
pHMM := [];
stepenMb := [];
collisionMb := [];
finchi := Vector(n-1);
coordMb := Matrix(n, 3);
for l to n-1 do if `mod`(l, 2) = 1 then
vectinizMb(l) := vect1[(1/2)*l+1/2];
vect(l) := vect1[(1/2)*l+1/2] else
vectinizMb(l) := vect2[(1/2)*l];
vect(l) := vect2[(1/2)*l]
end if end do;
for l to (1/2)*n-1/2 do
for t from l to (1/2)*n-1/2 do
if t = l then MD(t, l) := 0
else MD(t, l) := add(vect(g), g = l+1 .. t);
MD(l, t) := MD(t, l)
end if end do end do;
for l to (1/2)*n-3/2 do
```

```
for t to (1/2)*n-3/2 do
if diags <= t-l then
pHM := [HMM(n1+l, n1+t), op(pHM)]
end if end do end do;
pHM := Sort(pHM, order = descending);
if numcontMb <= nops(pHM) then
minMb := pHM[numcontMb]
else minMb := pHM[nops(pHM)] end if;
for l to (1/2)*n-3/2 do
for t to (1/2)*n-3/2 do
if minMb <= HMM(n1+l, n1+t) and diags <= t-l then
lprint('nel*frammento', frag, 'del*cromosoma', numchrom,
'il*numero*di*contatti*risulta',
HMM(n1+l, n1+t), 'fra*il*bin', n1+l, 'e*il*bin', n1+t);
pHMM := [(HMM(n1+l, n1+t), l+1, t+1), op(pHMM)]
end if end do end do;
no := nops(pHMM);
M := Matrix(n, n);
v1 := Vector(n);
c := Vector(numiterations);
cc := Vector(numiterations);
energy := Vector(numiterations);
vt := Vector(n-1);
al := Vector(n-2);
be := Vector(n-2);
Mcoord := Matrix(n, 3);
d := Matrix(n, n);
temp := initemp;
contiter := 0;
contacc := 0;
gg := 0;
contenergy := 0;
for t to n-1 do
vt(t) := vect(t) end do;
for t to n-2 do
if `mod`(t, 2) = 1 then
alpha(t) := plan((1/2)*t+1/2)
else alpha(t) := 0 end if end do;
for t to n-2 do
if `mod`(t, 2) = 1 then
beta(t) := CHI((1/2)*t+1/2) else
beta(t) := 0
end if end do;
s := 0;
while contenergy < energytresholdMb and s <= numiterations do
```

```
s := s+1;
contenergy := contenergy+1;
contiter := contiter+1;
chitemp := evalf((contacc+1)/contiter);
if contiter = numiterationsTemp and gg = 0 then
if chitemp < 1-epsilonchi then
lprint('dopo', contiter, 'iterazioni*il*rapporto*tra*transizioni*
accettate*e*proposte*risulta', chitemp); contiter := 0;
contacc := 0;
contenergy := 0;
temp := temp*incremtemp;
lprint('la*temperatura*sale*a', temp) else
lprint('al*passo', s, 'il*rapporto*risulta', chitemp, 'e*la*
temperatura*inizia*a*diminuire*con*un*valore*di', temp);
contiter := 0;
contacc := 0;
contenergy := 0;
gg := 1
end if end if;
if gg = 1 then
temp := temp*dimrate end if;
al(1) := alpha(1);
be(1) := beta(1);
for t from 2 to n-2 do
if `mod`(t, 2) = 1 then
al(t) := alpha(t);
b := (1/100)*a();
be(t) := beta(t)+evalf(b) else
be(t) := beta(t);
b := (1/100)*a();
if -arad <= evalf(alpha(t))+evalf(b) and
evalf(alpha(t))+evalf(b) <= arad then
al(t) := alpha(t)+evalf(b) else
al(t) := alpha(t)-evalf(b)
end if end if end do;
p[al](1) := (0, 0, 0);
p[al](2) := (vt(1), 0, 0);
for t from 3 to n do
alp(t-2) := evalf(add(al(x), x = 1 .. t-2));
p[al](t) := (p[al](t-1)[1]+evalf(cos(alp(t-2))*vt(t-1)),
p[al](t-1)[2]+evalf(sin(alp(t-2))*vt(t-1)), 0);
d(t) := (p[al](t)[1]-p[al](t-1)[1], p[al](t)[2]-
p[al](t-1)[2], p[al](t)[3]-p[al](t-1)[3]);
normpa(t, s) := norm(d(t), 2) end do;
pr(1) := p[al](1);
```

```
pr(2) := p[al](2);
qua(1) := 1;
qua(2) := 1;
for t from 2 to n do
v1(t-1) := (p[al](t)[1]-p[al](t-1)[1], p[al](t)[2]-p[al](t-1)[2],
 p[al](t)[3]-p[al](t-1)[3]);
v(t-1) := ((p[al](t)[1]-p[al](t-1)[1])/vt(t-1),
(p[al](t)[2]-p[al](t-1)[2])/vt(t-1),
(p[al](t)[3]-p[al](t-1)[3])/vt(t-1)) end do;
for t from 3 to n do
q(t) :=Qunit(cos((1/2)*be(t-2))+(v(t-2)[1]*i+v(t-2)[2]*j+
v(t-2)[3]*k)*sin((1/2)*be(t-2)));
qua(t) := evalf(qua(t-1)*q(t));
p(t) := rot(v1(t-1), (Qscalar(qua(t)), Qicoeff(qua(t)),
Qjcoeff(qua(t)), Qkcoeff(qua(t))));
pr(t) := (p(t)[1]+pr(t-1)[1], p(t)[2]+pr(t-1)[2],
p(t)[3]+pr(t-1)[3]) end do;
for t from 2 to n do
d(t) := (pr(t)[1]-pr(t-1)[1], pr(t)[2]-pr(t-1)[2], pr(t)[3]-pr(t-1)[3]);
normpr(t, s) := norm(d(t), 2) end do;
for g to n do
for t to n do
d(g, t) := (pr(g)[1]-pr(t)[1], pr(g)[2]-pr(t)[2], pr(g)[3]-pr(t)[3]);
M(g, t) := norm(d(g, t), 2)
end do end do; for l to n do
Mcoord(l, 1) := (pr(l))(1);
Mcoord(l, 2) := (pr(l))(2);
Mcoord(l, 3) := (pr(l))(3) end do;
c(s) := 0;
cc(s) := 0;
for g to n do
for t from g+1 to n do
if M(g, t) <= mindist then
c(s) := c(s)+1;
lprint(M(g, t), 'minimdist', s);
collisionMb := [op(collisionMb), (frag, g, t, s)] elif
maxdist < M(g, t) then
cc(s) := cc(s)+1;
lprint(M(g, t), 'maximdist', s)
end if end do end do;
en := 0;
for l to no do
en := en+M(2*pHMM[l][2], 2*pHMM[l][3])*HMM(pHMM[l][2],
pHMM[l][3]) end do;
energy(s) := en;
```

```
if s = 1 then
lprint('energia*iniziale', energy(s));
inienergy := energy(s) end if;
if 1 < s and c(s) = 0 and cc(s) = 0 then
den := energy(s)-inienergy;
if den <= 0 then inienergy := energy(s);
for t to n-1 do
vect(t) := evalf(vt(t)) end do;
for t to n-2 do
alpha(t) := evalf(al(t)) end do;
for t to n-2 do
beta(t) := evalf(be(t)) end do;
contacc := contacc+1;
contenergy := 0;
stepenMb := [op(stepenMb), (energy(s), den, s)];
for l to n do
coordMb(l, 1) := Mcoord(l, 1);
coordMb(l, 2) := Mcoord(l, 2);
coordMb(l, 3) := Mcoord(l, 3) end do
else pen := exp(-den/temp);
if s = 1 or abs((1/10)*a()) <= pen then
inienergy := energy(s);
for t to n-1 do
vect(t) := evalf(vt(t)) end do;
for t to n-2 do
alpha(t) := evalf(al(t))
end do;
for t to n-2 do
beta(t) := evalf(be(t)) end do;
contacc := contacc+1;
contenergy := 0;
stepenMb := [op(stepenMb), (energy(s), den, s)];
for l to n do
coordMb(l, 1) := Mcoord(l, 1);
coordMb(l, 2) := Mcoord(l, 2);
coordMb(l, 3) := Mcoord(l, 3) end do
end if end if end if end do;
for t to n-2 do
finchi(t) := beta(t) end do;
if energytresholdMb < contenergy then
lprint('la*simulazione*si**interrotta*allo*step', contiter, 'dopo',
 contenergy, 'steps*di*energia*stabile') end if;
lprint('temperatura*finale', temp, 'energia*finale', energy(s));
ExportMatrix("Mbcoords.txt", Mcoord, target = delimited)
end proc
```

```
> mainchain(1, 1, 1, nbinMb, numcontMb, diagsMb, mindistMb,
maxdistkb, numiterationsMb);
> stepenMb := convert(stepenMb, Matrix);
ExportMatrix("stepenMb.txt", stepenMb, target = delimited);
collisionMb := convert(collisionMb, Matrix):
ExportMatrix("collisionMb.txt", collisionsMb, target = delimited):
> ve(nbinMb+2) := (coordMb(2*nbinMb+1, 1),
 coordMb(2*nbinMb+1, 2), coordMb(2*nbinMb+1, 3));
> ve(1) := (coordMb(1, 1), coordMb(1, 2), coordMb(1, 3));
> for s to nbinMb do
ve(s+1) := (coordMb(2*s, 1), coordMb(2*s, 2), coordMb(2*s, 3))
end do;
> pointplot3d([ve(1), ve(2), ve(3), ve(4), ve(5), ve(6), ve(7), ve(8),
ve(9), ve(10), ve(11), ve(12), ve(13), ve(14), ve(15), ve(16), ve(17),
ve(18), ve(19), ve(20), ve(21), ve(22), ve(23), ve(24), ve(25), ve(26),
ve(27)], color = [blue], scaling = constrained, axes = normal);
> pfincoords := [];
> numbin := Vector(nbinMb);
> for s from 1 to nbinMb do
numbin(s):=dom(s,2)-dom(s,1)+1:
end do:
> for s to nbinMb do
nb := numbin(s);
pfin := Matrix(nb, 3);
if s = 1 then
v1 := (B[1, 1], B[1, 2], B[1, 3]);
v2 := (fincoords[1](nb, 1)-B[1, 1], fincoords[1](nb, 2)-B[1, 2],
fincoords[1](nb, 3)-B[1, 3]);
v3 := CrossProduct(v1, v2);
w1 := (coordMb[2*s, 1], coordMb[2*s, 2], coordMb[2*s, 3]);
w2 := (coordMb[2*s+1, 1]-coordMb[2*s, 1], coordMb[2*s+1, 2]-coordMb[2*s, 2],
coordMb[2*s+1, 3]-coordMb[2*s, 3]);
w3 := CrossProduct(w1, w2);
M1 := Matrix([v1, v2, v3]);
M2 := Matrix([w1, w2, w3]);
MI1 := MatrixInverse(M1);
L := MatrixMatrixMultiply(M2, MI1);
for t to nb do
pfin(t, 1) := MatrixVectorMultiply(L,
 (fincoords[1](t, 1), fincoords[1](t, 2), fincoords[1](t, 3)))[1];
pfin(t, 2) := MatrixVectorMultiply(L,
 (fincoords[1](t, 1), fincoords[1](t, 2), fincoords[1](t, 3)))[2];
pfin(t, 3) := MatrixVectorMultiply(L,
 (fincoords[1](t, 1), fincoords[1](t, 2), fincoords[1](t, 3)))[3] end do
else v1 := (B[s, 1], B[s, 2], B[s, 3]);
```

```
v2 := (fincoords[s](nb, 1)-B[s, 1],
 fincoords[s](nb, 2)-B[s, 2], fincoords[s](nb, 3)-B[s, 3]);
v3 := CrossProduct(v1, v2);
w1 := (coordMb[2*s, 1]-coordMb[2*s-1, 1], coordMb[2*s, 2]-coordMb[2*s-1, 2],
coordMb[2*s, 3]-coordMb[2*s-1, 3]);
w2 := (coordMb[2*s+1, 1]-coordMb[2*s, 1], coordMb[2*s+1, 2]-coordMb[2*s, 2],
coordMb[2*s+1, 3]-coordMb[2*s, 3]);
w3 := CrossProduct(w1, w2);
M1 := Matrix([v1, v2, v3]);
M2 := Matrix([w1, w2, w3]);
MI1 := MatrixInverse(M1);
L := MatrixMatrixMultiply(M2, MI1);
for t to nb do
pfin(t, 1) := MatrixVectorMultiply(L,
 (fincoords[s](t, 1), fincoords[s](t, 2), fincoords[s](t, 3)))[1]+
pfincoords[s-1](numbin(s-1), 1);
pfin(t, 2) := MatrixVectorMultiply(L,
(fincoords[s](t, 1), fincoords[s](t, 2), fincoords[s](t, 3)))[2]+
pfincoords[s-1](numbin(s-1), 2);
pfin(t, 3) := MatrixVectorMultiply(L,
(fincoords[s](t, 1), fincoords[s](t, 2), fincoords[s](t, 3)))[3]+
pfincoords[s-1](numbin(s-1), 3)
end do end if; pfincoords := [op(pfincoords), pfin] end do;
> for t to nbinMb-1 do print(pfincoords[t][numbin(t), 1]);
print(pfincoords[t+1][1, 1]) end do;
> finalpt := Matrix(m, 3);
finalv := Vector(m);
> for t to nbinMb do for l to numbin(t)-1 do
ipt := add(numbin(p)-1, p = 1 .. t-1);
finalpt(l+ipt, 1) := pfincoords[t][l, 1];
finalpt(l+ipt, 2) := pfincoords[t][l, 2];
finalpt(l+ipt, 3) := pfincoords[t][l, 3];
finalv(l+ipt) := (finalpt(l+ipt, 1), finalpt(l+ipt, 2), finalpt(l+ipt, 3))
end do end do;
> finalpt(m,1):=pfincoords[nbinMb][numbin(nbinMb),1]:
finalpt(m,2):=pfincoords[nbinMb][numbin(nbinMb),2]:
finalpt(m,3):=pfincoords[nbinMb][numbin(nbinMb),3]:
finalv(m):=<finalpt(m,1),finalpt(m,2),finalpt(m,3)>: ;
> finalcoordinates := Matrix(m, 3);
> for l to m do
finalcoordinates(l, 1) := finalv(l)[1];
finalcoordinates(l, 2) := finalv(l)[2];
finalcoordinates(l, 3) := finalv(l)[3] end do;
> ExportMatrix("finalmatrix.txt", finalcoordinates, target = delimited);
> for l from 2 to m do
```

```
normv(l):=norm(<finalv(l)[1]-finalv(l-1)[1],finalv(l)[2]-finalv(l-1)[2],
finalv(l)[3]-finalv(l-1)[3]>,2):
print(normv(l),l-1): end do:
> pointplot3d([finalv(1), finalv(2), finalv(3), finalv(4), finalv(5),
finalv(6), finalv(7), finalv(8), finalv(9), finalv(10), finalv(11),
finalv(12), finalv(13), finalv(14), finalv(15), finalv(16), finalv(17),
finalv(18), finalv(19), finalv(20), finalv(21), finalv(22), finalv(23),
finalv(24), finalv(25), finalv(26), finalv(27), finalv(28), finalv(29),
finalv(30), finalv(31), finalv(32), finalv(33), finalv(34), finalv(35),
finalv(36), finalv(37), finalv(38), finalv(39), finalv(40), finalv(41),
finalv(42), finalv(43), finalv(44), finalv(45), finalv(46), finalv(47),
finalv(48), finalv(49), finalv(50), finalv(51), finalv(52), finalv(53),
finalv(54), finalv(55), finalv(56), finalv(57), finalv(58), finalv(59),
finalv(60), finalv(61), finalv(62), finalv(63), finalv(64), finalv(65),
finalv(66), finalv(67), finalv(68), finalv(69), finalv(70), finalv(71),
finalv(72), finalv(73), finalv(74), finalv(75), finalv(76), finalv(77),
finalv(78), finalv(79), finalv(80), finalv(81), finalv(82), finalv(83),
finalv(84), finalv(85), finalv(86), finalv(87), finalv(88), finalv(89),
finalv(90), finalv(91), finalv(92), finalv(93), finalv(94), finalv(95),
finalv(96), finalv(97), finalv(98), finalv(99), finalv(100), finalv(101),
finalv(102), finalv(103), finalv(104), finalv(105), finalv(106),
finalv(107), finalv(108), finalv(109), finalv(110), finalv(111),
finalv(112), finalv(113), finalv(114), finalv(115), finalv(116),
finalv(117), finalv(118), finalv(119), finalv(120), finalv(121),
finalv(122), finalv(123), finalv(124), finalv(125), finalv(126),
finalv(127), finalv(128), finalv(129), finalv(130), finalv(131),
finalv(132), finalv(133), finalv(134), finalv(135), finalv(136),
finalv(137), finalv(138), finalv(139), finalv(140), finalv(141),
finalv(142), finalv(143), finalv(144), finalv(145), finalv(146),
finalv(147), finalv(148), finalv(149), finalv(150), finalv(151),
finalv(152), finalv(153), finalv(154), finalv(155), finalv(156),
finalv(157), finalv(158), finalv(159), finalv(160), finalv(161),
finalv(162), finalv(163), finalv(164), finalv(165), finalv(166),
finalv(167), finalv(168), finalv(169), finalv(170), finalv(171),
finalv(172), finalv(173),finalv(174), finalv(175), finalv(176),
finalv(177), finalv(178), finalv(179), finalv(180), finalv(181),
finalv(182), finalv(183), finalv(184), finalv(185), finalv(186),
finalv(187), finalv(188), finalv(189), finalv(190), finalv(191),
finalv(192), finalv(193), finalv(194), finalv(195), finalv(196),
finalv(197), finalv(198), finalv(199), finalv(200), finalv(201),
finalv(202), finalv(203), finalv(204), finalv(205), finalv(206),
finalv(207), finalv(208), finalv(209), finalv(210), finalv(211),
finalv(212), finalv(213), finalv(214), finalv(215), finalv(216),
finalv(217), finalv(218), finalv(219), finalv(220), finalv(221),
finalv(222), finalv(223), finalv(224), finalv(225), finalv(226),
```

```
finalv(227), finalv(228), finalv(229),finalv(230), finalv(231),
finalv(232), finalv(233), finalv(234), finalv(235), finalv(236),
finalv(237), finalv(238), finalv(239), finalv(240), finalv(241),
finalv(242), finalv(243), finalv(244), finalv(245), finalv(246),
finalv(247), finalv(248), finalv(249), finalv(250), finalv(251),
finalv(252), finalv(253), finalv(254), finalv(255), finalv(256),
finalv(257), finalv(258), finalv(259), finalv(260), finalv(261),
finalv(262), finalv(263), finalv(264), finalv(265), finalv(266),
finalv(267), finalv(268), finalv(269), finalv(270), finalv(271),
finalv(272), finalv(273), finalv(274), finalv(275), finalv(276),
finalv(277), finalv(278), finalv(279), finalv(280), finalv(281),
finalv(282), finalv(283), finalv(284), finalv(285), finalv(286),
finalv(287), finalv(288), finalv(289), finalv(290), finalv(291),
finalv(292)], color = [blue], scaling = constrained, axes = normal);
```

B.2 Code in Python (release 2.7.2)

```
# Claudia Caudai (ISTI-CNR Pisa) claudia.caudai@isti.cnr.it

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.

# This program takes as input the contact Matrix of a chromosome and the file
# containing the division in Topological Domains
# and gives as output a possible 3D conformation of the Chromatin fiber.
# The fiber is modeled as a bead chain, divided in subchains representing TDs.
# The evolution of the system is performed by a Simulated Annealing algorithm.


#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------

# This first part of the code is devoted to the definition of 2D and 3D vector,
# matrix, quaternion and geometry module (euclid3-0.01)
# downloaded, copied and pasted from https://pypi.python.org/pypi/euclid3/0.01


__docformat__ = 'restructuredtext'
__version__ = '$Id: euclid.py 37 2011-08-21 22:24:05Z elfnor@gmail.com $'
__revision__ = '$Revision: 37 $'


import math
import operator
import types

# Some magic here.  If _use_slots is True, the classes will derive from
# object and will define a __slots__ class variable.  If _use_slots is
# False, classes will be old-style and will not define __slots__.
#
# _use_slots = True:   Memory efficient, probably faster in future versions
#                      of Python, "better".
# _use_slots = False:  Ordinary classes, much faster than slots in current
#                      versions of Python (2.4 and 2.5).
```

```
_use_slots = True

# If True, allows components of Vector2 and Vector3 to be set via swizzling;
# e.g.  v.xyz = (1, 2, 3).  This is much, much slower than the more verbose
# v.x = 1; v.y = 2; v.z = 3,  and slows down ordinary element setting as
# well.  Recommended setting is False.
_enable_swizzle_set = False

# Requires class to derive from object.
if _enable_swizzle_set:
    _use_slots = True

# Implement _use_slots magic.
class _EuclidMetaclass(type):
    def __new__(cls, name, bases, dct):
        if '__slots__' in dct:
            dct['__getstate__'] = cls._create_getstate(dct['__slots__'])
            dct['__setstate__'] = cls._create_setstate(dct['__slots__'])
        if _use_slots:
            return type.__new__(cls, name, bases + (object,), dct)
        else:
            if '__slots__' in dct:
                del dct['__slots__']
            return types.ClassType.__new__(types.ClassType, name, bases, dct)

    @classmethod
    def _create_getstate(cls, slots):
        def __getstate__(self):
            d = {}
            for slot in slots:
                d[slot] = getattr(self, slot)
            return d
        return __getstate__

    @classmethod
    def _create_setstate(cls, slots):
        def __setstate__(self, state):
            for name, value in state.items():
                setattr(self, name, value)
        return __setstate__

__metaclass__ = _EuclidMetaclass

class Vector2:
    __slots__ = ['x', 'y']
```

```
__hash__ = None

def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

def __copy__(self):
    return self.__class__(self.x, self.y)

copy = __copy__

def __repr__(self):
    return 'Vector2(%.2f, %.2f)' % (self.x, self.y)

def __eq__(self, other):
    if isinstance(other, Vector2):
        return self.x == other.x and \
               self.y == other.y
    else:
        assert hasattr(other, '__len__') and len(other) == 2
        return self.x == other[0] and \
               self.y == other[1]

def __ne__(self, other):
    return not self.__eq__(other)

def __nonzero__(self):
    return self.x != 0 or self.y != 0

def __len__(self):
    return 2

def __getitem__(self, key):
    return (self.x, self.y)[key]

def __setitem__(self, key, value):
    l = [self.x, self.y]
    l[key] = value
    self.x, self.y = l

def __iter__(self):
    return iter((self.x, self.y))

def __getattr__(self, name):
    try:
```

```
        return tuple([(self.x, self.y)['xy'.index(c)] \
                      for c in name])
    except ValueError:
        raise AttributeError(name)

if _enable_swizzle_set:
    # This has detrimental performance on ordinary setattr as well
    # if enabled
    def __setattr__(self, name, value):
        if len(name) == 1:
            object.__setattr__(self, name, value)
        else:
            try:
                l = [self.x, self.y]
                for c, v in map(None, name, value):
                    l['xy'.index(c)] = v
                self.x, self.y = l
            except ValueError:
                raise AttributeError(name)

def __add__(self, other):
    if isinstance(other, Vector2):
        # Vector + Vector -> Vector
        # Vector + Point -> Point
        # Point + Point -> Vector
        if self.__class__ is other.__class__:
            _class = Vector2
        else:
            _class = Point2
        return _class(self.x + other.x,
                      self.y + other.y)
    else:
        assert hasattr(other, '__len__') and len(other) == 2
        return Vector2(self.x + other[0],
                       self.y + other[1])
__radd__ = __add__

def __iadd__(self, other):
    if isinstance(other, Vector2):
        self.x += other.x
        self.y += other.y
    else:
        self.x += other[0]
        self.y += other[1]
    return self
```

```
def __sub__(self, other):
    if isinstance(other, Vector2):
        # Vector - Vector -> Vector
        # Vector - Point -> Point
        # Point - Point -> Vector
        if self.__class__ is other.__class__:
            _class = Vector2
        else:
            _class = Point2
        return _class(self.x - other.x,
                      self.y - other.y)
    else:
        assert hasattr(other, '__len__') and len(other) == 2
        return Vector2(self.x - other[0],
                       self.y - other[1])


def __rsub__(self, other):
    if isinstance(other, Vector2):
        return Vector2(other.x - self.x,
                       other.y - self.y)
    else:
        assert hasattr(other, '__len__') and len(other) == 2
        return Vector2(other.x - self[0],
                       other.y - self[1])

def __mul__(self, other):
    assert type(other) in (int, int, float)
    return Vector2(self.x * other,
                   self.y * other)

__rmul__ = __mul__

def __imul__(self, other):
    assert type(other) in (int, int, float)
    self.x *= other
    self.y *= other
    return self

def __div__(self, other):
    assert type(other) in (int, int, float)
    return Vector2(operator.div(self.x, other),
                   operator.div(self.y, other))
```

```
def __rdiv__(self, other):
    assert type(other) in (int, int, float)
    return Vector2(operator.div(other, self.x),
                   operator.div(other, self.y))

def __floordiv__(self, other):
    assert type(other) in (int, int, float)
    return Vector2(operator.floordiv(self.x, other),
                   operator.floordiv(self.y, other))


def __rfloordiv__(self, other):
    assert type(other) in (int, int, float)
    return Vector2(operator.floordiv(other, self.x),
                   operator.floordiv(other, self.y))

def __truediv__(self, other):
    assert type(other) in (int, int, float)
    return Vector2(operator.truediv(self.x, other),
                   operator.truediv(self.y, other))


def __rtruediv__(self, other):
    assert type(other) in (int, int, float)
    return Vector2(operator.truediv(other, self.x),
                   operator.truediv(other, self.y))

def __neg__(self):
    return Vector2(-self.x,
                    -self.y)

__pos__ = __copy__

def __abs__(self):
    return math.sqrt(self.x ** 2 + \
                     self.y ** 2)

magnitude = __abs__

def magnitude_squared(self):
    return self.x ** 2 + \
           self.y ** 2

def normalise(self):
```

```
            d = self.magnitude()
            if d:
                self.x /= d
                self.y /= d
            return self

        def normalised(self):
            d = self.magnitude()
            if d:
                return Vector2(self.x / d,
                               self.y / d)
            return self.copy()

        def dot(self, other):
            assert isinstance(other, Vector2)
            return self.x * other.x + \
                   self.y * other.y

        def cross(self):
            return Vector2(self.y, -self.x)

        def reflect(self, normal):
            # assume normal is normalised
            assert isinstance(normal, Vector2)
            d = 2 * (self.x * normal.x + self.y * normal.y)
            return Vector2(self.x - d * normal.x,
                           self.y - d * normal.y)

        def angle(self, other):
            """Return the angle to the vector other"""
            return math.acos(self.dot(other) / (self.magnitude()*other.magnitude()))

        def project(self, other):
            """Return one vector projected on the vector other"""
            n = other.normalised()
            return self.dot(n)*n

    class Vector3:
        __slots__ = ['x', 'y', 'z']
        __hash__ = None

        def __init__(self, x=0, y=0, z=0):
            self.x = x
            self.y = y
            self.z = z
```

```
def __copy__(self):
    return self.__class__(self.x, self.y, self.z)

copy = __copy__

def __repr__(self):
    return 'Vector3(%.2f, %.2f, %.2f)' % (self.x,
                                          self.y,
                                          self.z)

def __eq__(self, other):
    if isinstance(other, Vector3):
        return self.x == other.x and \
               self.y == other.y and \
               self.z == other.z
    else:
        assert hasattr(other, '__len__') and len(other) == 3
        return self.x == other[0] and \
               self.y == other[1] and \
               self.z == other[2]

def __ne__(self, other):
    return not self.__eq__(other)

def __nonzero__(self):
    return self.x != 0 or self.y != 0 or self.z != 0

def __len__(self):
    return 3

def __getitem__(self, key):
    return (self.x, self.y, self.z)[key]

def __setitem__(self, key, value):
    l = [self.x, self.y, self.z]
    l[key] = value
    self.x, self.y, self.z = l

def __iter__(self):
    return iter((self.x, self.y, self.z))

def __getattr__(self, name):
    try:
        return tuple([(self.x, self.y, self.z)['xyz'.index(c)] \
```

```
                    for c in name])
    except ValueError:
        raise AttributeError(name)

if _enable_swizzle_set:
    # This has detrimental performance on ordinary setattr as well
    # if enabled
    def __setattr__(self, name, value):
        if len(name) == 1:
            object.__setattr__(self, name, value)
        else:
            try:
                l = [self.x, self.y, self.z]
                for c, v in map(None, name, value):
                    l['xyz'.index(c)] = v
                self.x, self.y, self.z = l
            except ValueError:
                raise AttributeError(name)


def __add__(self, other):
    if isinstance(other, Vector3):
        # Vector + Vector -> Vector
        # Vector + Point -> Point
        # Point + Point -> Vector
        if self.__class__ is other.__class__:
            _class = Vector3
        else:
            _class = Point3
        return _class(self.x + other.x,
                      self.y + other.y,
                      self.z + other.z)
    else:
        assert hasattr(other, '__len__') and len(other) == 3
        return Vector3(self.x + other[0],
                       self.y + other[1],
                       self.z + other[2])
__radd__ = __add__

def __iadd__(self, other):
    if isinstance(other, Vector3):
        self.x += other.x
        self.y += other.y
        self.z += other.z
    else:
```

```
            self.x += other[0]
            self.y += other[1]
            self.z += other[2]
        return self

    def __sub__(self, other):
        if isinstance(other, Vector3):
            # Vector - Vector -> Vector
            # Vector - Point -> Point
            # Point - Point -> Vector
            if self.__class__ is other.__class__:
                _class = Vector3
            else:
                _class = Point3
            return Vector3(self.x - other.x,
                           self.y - other.y,
                           self.z - other.z)
        else:
            assert hasattr(other, '__len__') and len(other) == 3
            return Vector3(self.x - other[0],
                           self.y - other[1],
                           self.z - other[2])


    def __rsub__(self, other):
        if isinstance(other, Vector3):
            return Vector3(other.x - self.x,
                           other.y - self.y,
                           other.z - self.z)
        else:
            assert hasattr(other, '__len__') and len(other) == 3
            return Vector3(other.x - self[0],
                           other.y - self[1],
                           other.z - self[2])

    def __mul__(self, other):
        if isinstance(other, Vector3):
            # TODO component-wise mul/div in-place and on Vector2; docs.
            if self.__class__ is Point3 or other.__class__ is Point3:
                _class = Point3
            else:
                _class = Vector3
            return _class(self.x * other.x,
                          self.y * other.y,
                          self.z * other.z)
```

```
    else:
        assert type(other) in (int, int, float)
        return Vector3(self.x * other,
                       self.y * other,
                       self.z * other)

__rmul__ = __mul__

def __imul__(self, other):
    assert type(other) in (int, int, float)
    self.x *= other
    self.y *= other
    self.z *= other
    return self

def __div__(self, other):
    assert type(other) in (int, int, float)
    return Vector3(operator.div(self.x, other),
                   operator.div(self.y, other),
                   operator.div(self.z, other))


def __rdiv__(self, other):
    assert type(other) in (int, int, float)
    return Vector3(operator.div(other, self.x),
                   operator.div(other, self.y),
                   operator.div(other, self.z))

def __floordiv__(self, other):
    assert type(other) in (int, int, float)
    return Vector3(operator.floordiv(self.x, other),
                   operator.floordiv(self.y, other),
                   operator.floordiv(self.z, other))


def __rfloordiv__(self, other):
    assert type(other) in (int, int, float)
    return Vector3(operator.floordiv(other, self.x),
                   operator.floordiv(other, self.y),
                   operator.floordiv(other, self.z))

def __truediv__(self, other):
    assert type(other) in (int, int, float)
    return Vector3(operator.truediv(self.x, other),
                   operator.truediv(self.y, other),
```

```
                    operator.truediv(self.z, other))


def __rtruediv__(self, other):
    assert type(other) in (int, int, float)
    return Vector3(operator.truediv(other, self.x),
                   operator.truediv(other, self.y),
                   operator.truediv(other, self.z))

def __neg__(self):
    return Vector3(-self.x,
                    -self.y,
                    -self.z)

__pos__ = __copy__

def __abs__(self):
    return math.sqrt(self.x ** 2 + \
                     self.y ** 2 + \
                     self.z ** 2)

magnitude = __abs__

def magnitude_squared(self):
    return self.x ** 2 + \
           self.y ** 2 + \
           self.z ** 2

def normalise(self):
    d = self.magnitude()
    if d:
        self.x /= d
        self.y /= d
        self.z /= d
    return self

def normalised(self):
    d = self.magnitude()
    if d:
        return Vector3(self.x / d,
                       self.y / d,
                       self.z / d)
    return self.copy()

def dot(self, other):
```

```
        assert isinstance(other, Vector3)
        return self.x * other.x + \
               self.y * other.y + \
               self.z * other.z

    def cross(self, other):
        assert isinstance(other, Vector3)
        return Vector3(self.y * other.z - self.z * other.y,
                       -self.x * other.z + self.z * other.x,
                       self.x * other.y - self.y * other.x)

    def reflect(self, normal):
        # assume normal is normalised
        assert isinstance(normal, Vector3)
        d = 2 * (self.x * normal.x + self.y * normal.y + self.z * normal.z)
        return Vector3(self.x - d * normal.x,
                       self.y - d * normal.y,
                       self.z - d * normal.z)

    def rotate_around(self, axis, theta):
        """Return the vector rotated around axis through angle theta.
Right hand rule applies"""

        # Adapted from equations published by Glenn Murray.
        # http://inside.mines.edu/~gmurray/ArbitraryAxisRotation/
ArbitraryAxisRotation.html
        x, y, z = self.x, self.y,self.z
        u, v, w = axis.x, axis.y, axis.z

        # Extracted common factors for simplicity and efficiency
        r2 = u**2 + v**2 + w**2
        r = math.sqrt(r2)
        ct = math.cos(theta)
        st = math.sin(theta) / r
        dt = (u*x + v*y + w*z) * (1 - ct) / r2
        return Vector3((u * dt + x * ct + (-w * y + v * z) * st),
                       (v * dt + y * ct + ( w * x - u * z) * st),
                       (w * dt + z * ct + (-v * x + u * y) * st))

    def angle(self, other):
        """Return the angle to the vector other"""
        return math.acos(self.dot(other) / (self.magnitude()*other.magnitude()))

    def project(self, other):
        """Return one vector projected on the vector other"""
```

```
        n = other.normalised()
        return self.dot(n)*n

# a b c
# e f g
# i j k

class Matrix3:
    __slots__ = list('abcefgijk')

    def __init__(self):
        self.identity()

    def __copy__(self):
        M = Matrix3()
        M.a = self.a
        M.b = self.b
        M.c = self.c
        M.e = self.e
        M.f = self.f
        M.g = self.g
        M.i = self.i
        M.j = self.j
        M.k = self.k
        return M

    copy = __copy__
    def __repr__(self):
        return ('Matrix3([% 8.2f % 8.2f % 8.2f\n'  \
                '         % 8.2f % 8.2f % 8.2f\n'  \
                '         % 8.2f % 8.2f % 8.2f])') \
                % (self.a, self.b, self.c,
                   self.e, self.f, self.g,
                   self.i, self.j, self.k)

    def __getitem__(self, key):
        return [self.a, self.e, self.i,
                self.b, self.f, self.j,
                self.c, self.g, self.k][key]

    def __setitem__(self, key, value):
        L = self[:]
        L[key] = value
        (self.a, self.e, self.i,
         self.b, self.f, self.j,
```

```
     self.c, self.g, self.k) = L

def __mul__(self, other):
    if isinstance(other, Matrix3):
        # Caching repeatedly accessed attributes in local variables
        # apparently increases performance by 20%.  Attrib: Will McGugan.
        Aa = self.a
        Ab = self.b
        Ac = self.c
        Ae = self.e
        Af = self.f
        Ag = self.g
        Ai = self.i
        Aj = self.j
        Ak = self.k
        Ba = other.a
        Bb = other.b
        Bc = other.c
        Be = other.e
        Bf = other.f
        Bg = other.g
        Bi = other.i
        Bj = other.j
        Bk = other.k
        C = Matrix3()
        C.a = Aa * Ba + Ab * Be + Ac * Bi
        C.b = Aa * Bb + Ab * Bf + Ac * Bj
        C.c = Aa * Bc + Ab * Bg + Ac * Bk
        C.e = Ae * Ba + Af * Be + Ag * Bi
        C.f = Ae * Bb + Af * Bf + Ag * Bj
        C.g = Ae * Bc + Af * Bg + Ag * Bk
        C.i = Ai * Ba + Aj * Be + Ak * Bi
        C.j = Ai * Bb + Aj * Bf + Ak * Bj
        C.k = Ai * Bc + Aj * Bg + Ak * Bk
        return C
    elif isinstance(other, Point2):
        A = self
        B = other
        P = Point2(0, 0)
        P.x = A.a * B.x + A.b * B.y + A.c
        P.y = A.e * B.x + A.f * B.y + A.g
        return P
    elif isinstance(other, Vector2):
        A = self
        B = other
```

```
            V = Vector2(0, 0)
            V.x = A.a * B.x + A.b * B.y
            V.y = A.e * B.x + A.f * B.y
            return V
        else:
            other = other.copy()
            other._apply_transform(self)
            return other

    def __imul__(self, other):
        assert isinstance(other, Matrix3)
        # Cache attributes in local vars (see Matrix3.__mul__).
        Aa = self.a
        Ab = self.b
        Ac = self.c
        Ae = self.e
        Af = self.f
        Ag = self.g
        Ai = self.i
        Aj = self.j
        Ak = self.k
        Ba = other.a
        Bb = other.b
        Bc = other.c
        Be = other.e
        Bf = other.f
        Bg = other.g
        Bi = other.i
        Bj = other.j
        Bk = other.k
        self.a = Aa * Ba + Ab * Be + Ac * Bi
        self.b = Aa * Bb + Ab * Bf + Ac * Bj
        self.c = Aa * Bc + Ab * Bg + Ac * Bk
        self.e = Ae * Ba + Af * Be + Ag * Bi
        self.f = Ae * Bb + Af * Bf + Ag * Bj
        self.g = Ae * Bc + Af * Bg + Ag * Bk
        self.i = Ai * Ba + Aj * Be + Ak * Bi
        self.j = Ai * Bb + Aj * Bf + Ak * Bj
        self.k = Ai * Bc + Aj * Bg + Ak * Bk
        return self

    def identity(self):
        self.a = self.f = self.k = 1.
        self.b = self.c = self.e = self.g = self.i = self.j = 0
        return self
```

```
def scale(self, x, y):
    self *= Matrix3.new_scale(x, y)
    return self

def translate(self, x, y):
    self *= Matrix3.new_translate(x, y)
    return self

def rotate(self, angle):
    self *= Matrix3.new_rotate(angle)
    return self

# Static constructors
def new_identity(cls):
    self = cls()
    return self
new_identity = classmethod(new_identity)

def new_scale(cls, x, y):
    self = cls()
    self.a = x
    self.f = y
    return self
new_scale = classmethod(new_scale)

def new_translate(cls, x, y):
    self = cls()
    self.c = x
    self.g = y
    return self
new_translate = classmethod(new_translate)

def new_rotate(cls, angle):
    self = cls()
    s = math.sin(angle)
    c = math.cos(angle)
    self.a = self.f = c
    self.b = -s
    self.e = s
    return self
new_rotate = classmethod(new_rotate)

def determinant(self):
    return (self.a*self.f*self.k
```

```
                + self.b*self.g*self.i
                + self.c*self.e*self.j
                - self.a*self.g*self.j
                - self.b*self.e*self.k
                - self.c*self.f*self.i)

    def inverse(self):
        tmp = Matrix3()
        d = self.determinant()

        if abs(d) < 0.001:
            # No inverse, return identity
            return tmp
        else:
            d = 1.0 / d

            tmp.a = d * (self.f*self.k - self.g*self.j)
            tmp.b = d * (self.c*self.j - self.b*self.k)
            tmp.c = d * (self.b*self.g - self.c*self.f)
            tmp.e = d * (self.g*self.i - self.e*self.k)
            tmp.f = d * (self.a*self.k - self.c*self.i)
            tmp.g = d * (self.c*self.e - self.a*self.g)
            tmp.i = d * (self.e*self.j - self.f*self.i)
            tmp.j = d * (self.b*self.i - self.a*self.j)
            tmp.k = d * (self.a*self.f - self.b*self.e)

            return tmp

# a b c d
# e f g h
# i j k l
# m n o p

class Matrix4:
    __slots__ = list('abcdefghijklmnop')

    def __init__(self):
        self.identity()

    def __copy__(self):
        M = Matrix4()
        M.a = self.a
        M.b = self.b
        M.c = self.c
        M.d = self.d
```

```
    M.e = self.e
    M.f = self.f
    M.g = self.g
    M.h = self.h
    M.i = self.i
    M.j = self.j
    M.k = self.k
    M.l = self.l
    M.m = self.m
    M.n = self.n
    M.o = self.o
    M.p = self.p
    return M

copy = __copy__


def __repr__(self):
    return ('Matrix4([% 8.2f % 8.2f % 8.2f % 8.2f\n'  \
            '         % 8.2f % 8.2f % 8.2f % 8.2f\n'  \
            '         % 8.2f % 8.2f % 8.2f % 8.2f\n'  \
            '         % 8.2f % 8.2f % 8.2f % 8.2f])') \
            % (self.a, self.b, self.c, self.d,
               self.e, self.f, self.g, self.h,
               self.i, self.j, self.k, self.l,
               self.m, self.n, self.o, self.p)

def __getitem__(self, key):
    return [self.a, self.e, self.i, self.m,
            self.b, self.f, self.j, self.n,
            self.c, self.g, self.k, self.o,
            self.d, self.h, self.l, self.p][key]

def __setitem__(self, key, value):
    L = self[:]
    L[key] = value
    (self.a, self.e, self.i, self.m,
     self.b, self.f, self.j, self.n,
     self.c, self.g, self.k, self.o,
     self.d, self.h, self.l, self.p) = L

def __mul__(self, other):
    if isinstance(other, Matrix4):
        # Cache attributes in local vars (see Matrix3.__mul__).
        Aa = self.a
```

```
Ab = self.b
Ac = self.c
Ad = self.d
Ae = self.e
Af = self.f
Ag = self.g
Ah = self.h
Ai = self.i
Aj = self.j
Ak = self.k
Al = self.l
Am = self.m
An = self.n
Ao = self.o
Ap = self.p
Ba = other.a
Bb = other.b
Bc = other.c
Bd = other.d
Be = other.e
Bf = other.f
Bg = other.g
Bh = other.h
Bi = other.i
Bj = other.j
Bk = other.k
Bl = other.l
Bm = other.m
Bn = other.n
Bo = other.o
Bp = other.p
C = Matrix4()
C.a = Aa * Ba + Ab * Be + Ac * Bi + Ad * Bm
C.b = Aa * Bb + Ab * Bf + Ac * Bj + Ad * Bn
C.c = Aa * Bc + Ab * Bg + Ac * Bk + Ad * Bo
C.d = Aa * Bd + Ab * Bh + Ac * Bl + Ad * Bp
C.e = Ae * Ba + Af * Be + Ag * Bi + Ah * Bm
C.f = Ae * Bb + Af * Bf + Ag * Bj + Ah * Bn
C.g = Ae * Bc + Af * Bg + Ag * Bk + Ah * Bo
C.h = Ae * Bd + Af * Bh + Ag * Bl + Ah * Bp
C.i = Ai * Ba + Aj * Be + Ak * Bi + Al * Bm
C.j = Ai * Bb + Aj * Bf + Ak * Bj + Al * Bn
C.k = Ai * Bc + Aj * Bg + Ak * Bk + Al * Bo
C.l = Ai * Bd + Aj * Bh + Ak * Bl + Al * Bp
C.m = Am * Ba + An * Be + Ao * Bi + Ap * Bm
```

```
            C.n = Am * Bb + An * Bf + Ao * Bj + Ap * Bn
            C.o = Am * Bc + An * Bg + Ao * Bk + Ap * Bo
            C.p = Am * Bd + An * Bh + Ao * Bl + Ap * Bp
            return C
        elif isinstance(other, Point3):
            A = self
            B = other
            P = Point3(0, 0, 0)
            P.x = A.a * B.x + A.b * B.y + A.c * B.z + A.d
            P.y = A.e * B.x + A.f * B.y + A.g * B.z + A.h
            P.z = A.i * B.x + A.j * B.y + A.k * B.z + A.l
            return P
        elif isinstance(other, Vector3):
            A = self
            B = other
            V = Vector3(0, 0, 0)
            V.x = A.a * B.x + A.b * B.y + A.c * B.z
            V.y = A.e * B.x + A.f * B.y + A.g * B.z
            V.z = A.i * B.x + A.j * B.y + A.k * B.z
            return V
        else:
            other = other.copy()
            other._apply_transform(self)
            return other

    def __imul__(self, other):
        assert isinstance(other, Matrix4)
        # Cache attributes in local vars (see Matrix3.__mul__).
        Aa = self.a
        Ab = self.b
        Ac = self.c
        Ad = self.d
        Ae = self.e
        Af = self.f
        Ag = self.g
        Ah = self.h
        Ai = self.i
        Aj = self.j
        Ak = self.k
        Al = self.l
        Am = self.m
        An = self.n
        Ao = self.o
        Ap = self.p
        Ba = other.a
```

```
    Bb = other.b
    Bc = other.c
    Bd = other.d
    Be = other.e
    Bf = other.f
    Bg = other.g
    Bh = other.h
    Bi = other.i
    Bj = other.j
    Bk = other.k
    Bl = other.l
    Bm = other.m
    Bn = other.n
    Bo = other.o
    Bp = other.p
    self.a = Aa * Ba + Ab * Be + Ac * Bi + Ad * Bm
    self.b = Aa * Bb + Ab * Bf + Ac * Bj + Ad * Bn
    self.c = Aa * Bc + Ab * Bg + Ac * Bk + Ad * Bo
    self.d = Aa * Bd + Ab * Bh + Ac * Bl + Ad * Bp
    self.e = Ae * Ba + Af * Be + Ag * Bi + Ah * Bm
    self.f = Ae * Bb + Af * Bf + Ag * Bj + Ah * Bn
    self.g = Ae * Bc + Af * Bg + Ag * Bk + Ah * Bo
    self.h = Ae * Bd + Af * Bh + Ag * Bl + Ah * Bp
    self.i = Ai * Ba + Aj * Be + Ak * Bi + Al * Bm
    self.j = Ai * Bb + Aj * Bf + Ak * Bj + Al * Bn
    self.k = Ai * Bc + Aj * Bg + Ak * Bk + Al * Bo
    self.l = Ai * Bd + Aj * Bh + Ak * Bl + Al * Bp
    self.m = Am * Ba + An * Be + Ao * Bi + Ap * Bm
    self.n = Am * Bb + An * Bf + Ao * Bj + Ap * Bn
    self.o = Am * Bc + An * Bg + Ao * Bk + Ap * Bo
    self.p = Am * Bd + An * Bh + Ao * Bl + Ap * Bp
    return self

def transform(self, other):
    A = self
    B = other
    P = Point3(0, 0, 0)
    P.x = A.a * B.x + A.b * B.y + A.c * B.z + A.d
    P.y = A.e * B.x + A.f * B.y + A.g * B.z + A.h
    P.z = A.i * B.x + A.j * B.y + A.k * B.z + A.l
    w =   A.m * B.x + A.n * B.y + A.o * B.z + A.p
    if w != 0:
        P.x /= w
        P.y /= w
        P.z /= w
```

```
        return P

    def identity(self):
        self.a = self.f = self.k = self.p = 1.
        self.b = self.c = self.d = self.e = self.g = self.h = \
        self.i = self.j = self.l = self.m = self.n = self.o = 0
        return self

    def scale(self, x, y, z):
        self *= Matrix4.new_scale(x, y, z)
        return self

    def translate(self, x, y, z):
        self *= Matrix4.new_translate(x, y, z)
        return self

    def rotatex(self, angle):
        self *= Matrix4.new_rotatex(angle)
        return self

    def rotatey(self, angle):
        self *= Matrix4.new_rotatey(angle)
        return self

    def rotatez(self, angle):
        self *= Matrix4.new_rotatez(angle)
        return self

    def rotate_axis(self, angle, axis):
        self *= Matrix4.new_rotate_axis(angle, axis)
        return self

    def rotate_euler(self, heading, attitude, bank):
        self *= Matrix4.new_rotate_euler(heading, attitude, bank)
        return self

    def rotate_triple_axis(self, x, y, z):
        self *= Matrix4.new_rotate_triple_axis(x, y, z)
        return self

    def transpose(self):
        (self.a, self.e, self.i, self.m,
         self.b, self.f, self.j, self.n,
         self.c, self.g, self.k, self.o,
         self.d, self.h, self.l, self.p) = \
```

```
    (self.a, self.b, self.c, self.d,
     self.e, self.f, self.g, self.h,
     self.i, self.j, self.k, self.l,
     self.m, self.n, self.o, self.p)

def transposed(self):
    M = self.copy()
    M.transpose()
    return M

# Static constructors
def new(cls, *values):
    M = cls()
    M[:] = values
    return M
new = classmethod(new)

def new_identity(cls):
    self = cls()
    return self
new_identity = classmethod(new_identity)

def new_scale(cls, x, y, z):
    self = cls()
    self.a = x
    self.f = y
    self.k = z
    return self
new_scale = classmethod(new_scale)

def new_translate(cls, x, y, z):
    self = cls()
    self.d = x
    self.h = y
    self.l = z
    return self
new_translate = classmethod(new_translate)

def new_rotatex(cls, angle):
    self = cls()
    s = math.sin(angle)
    c = math.cos(angle)
    self.f = self.k = c
    self.g = -s
    self.j = s
```

```
    return self
new_rotatex = classmethod(new_rotatex)

def new_rotatey(cls, angle):
    self = cls()
    s = math.sin(angle)
    c = math.cos(angle)
    self.a = self.k = c
    self.c = s
    self.i = -s
    return self
new_rotatey = classmethod(new_rotatey)

def new_rotatez(cls, angle):
    self = cls()
    s = math.sin(angle)
    c = math.cos(angle)
    self.a = self.f = c
    self.b = -s
    self.e = s
    return self
new_rotatez = classmethod(new_rotatez)

def new_rotate_axis(cls, angle, axis):
    assert(isinstance(axis, Vector3))
    vector = axis.normalised()
    x = vector.x
    y = vector.y
    z = vector.z

    self = cls()
    s = math.sin(angle)
    c = math.cos(angle)
    c1 = 1. - c

    # from the glRotate man page
    self.a = x * x * c1 + c
    self.b = x * y * c1 - z * s
    self.c = x * z * c1 + y * s
    self.e = y * x * c1 + z * s
    self.f = y * y * c1 + c
    self.g = y * z * c1 - x * s
    self.i = x * z * c1 - y * s
    self.j = y * z * c1 + x * s
    self.k = z * z * c1 + c
```

```
    return self
new_rotate_axis = classmethod(new_rotate_axis)

def new_rotate_euler(cls, heading, attitude, bank):
    # from http://www.euclideanspace.com/
    ch = math.cos(heading)
    sh = math.sin(heading)
    ca = math.cos(attitude)
    sa = math.sin(attitude)
    cb = math.cos(bank)
    sb = math.sin(bank)

    self = cls()
    self.a = ch * ca
    self.b = sh * sb - ch * sa * cb
    self.c = ch * sa * sb + sh * cb
    self.e = sa
    self.f = ca * cb
    self.g = -ca * sb
    self.i = -sh * ca
    self.j = sh * sa * cb + ch * sb
    self.k = -sh * sa * sb + ch * cb
    return self
new_rotate_euler = classmethod(new_rotate_euler)

def new_rotate_triple_axis(cls, x, y, z):
  m = cls()

  m.a, m.b, m.c = x.x, y.x, z.x
  m.e, m.f, m.g = x.y, y.y, z.y
  m.i, m.j, m.k = x.z, y.z, z.z

  return m
new_rotate_triple_axis = classmethod(new_rotate_triple_axis)

def new_look_at(cls, eye, at, up):
  z = (eye - at).normalised()
  x = up.cross(z).normalised()
  y = z.cross(x)

  m = cls.new_rotate_triple_axis(x, y, z)
  m.d, m.h, m.l = eye.x, eye.y, eye.z
  return m
new_look_at = classmethod(new_look_at)
```

```
    def new_perspective(cls, fov_y, aspect, near, far):
        # from the gluPerspective man page
        f = 1 / math.tan(fov_y / 2)
        self = cls()
        assert near != 0.0 and near != far
        self.a = f / aspect
        self.f = f
        self.k = (far + near) / (near - far)
        self.l = 2 * far * near / (near - far)
        self.o = -1
        self.p = 0
        return self
    new_perspective = classmethod(new_perspective)

    def determinant(self):
        return ((self.a * self.f - self.e * self.b)
              * (self.k * self.p - self.o * self.l)
              - (self.a * self.j - self.i * self.b)
              * (self.g * self.p - self.o * self.h)
              + (self.a * self.n - self.m * self.b)
              * (self.g * self.l - self.k * self.h)
              + (self.e * self.j - self.i * self.f)
              * (self.c * self.p - self.o * self.d)
              - (self.e * self.n - self.m * self.f)
              * (self.c * self.l - self.k * self.d)
              + (self.i * self.n - self.m * self.j)
              * (self.c * self.h - self.g * self.d))

    def inverse(self):
        tmp = Matrix4()
        d = self.determinant();

        if abs(d) < 0.001:
            # No inverse, return identity
            return tmp
        else:
            d = 1.0 / d;

            tmp.a = d * (self.f * (self.k * self.p - self.o * self.l) + self.j *
(self.o * self.h - self.g * self.p) + self.n *
         (self.g * self.l - self.k * self.h));
            tmp.e = d * (self.g * (self.i * self.p - self.m * self.l) + self.k *
(self.m * self.h - self.e * self.p) + self.o *
(self.e * self.l - self.i * self.h));
            tmp.i =  d * (self.h * (self.i * self.n - self.m * self.j) + self.l *
```

```
(self.m * self.f - self.e * self.n) + self.p *
(self.e * self.j - self.i * self.f));
            tmp.m = d * (self.e * (self.n * self.k - self.j * self.o) + self.i *
  (self.f * self.o - self.n * self.g) + self.m *
(self.j * self.g - self.f * self.k));
            tmp.b = d * (self.j * (self.c * self.p - self.o * self.d) + self.n *
(self.k * self.d - self.c * self.l) + self.b *
(self.o * self.l - self.k * self.p));
            tmp.f = d * (self.k * (self.a * self.p - self.m * self.d) + self.o *
(self.i * self.d - self.a * self.l) + self.c *
(self.m * self.l - self.i * self.p));
            tmp.j = d * (self.l * (self.a * self.n - self.m * self.b) + self.p *
(self.i * self.b - self.a * self.j) + self.d *
(self.m * self.j - self.i * self.n));
            tmp.n = d * (self.i * (self.n * self.c - self.b * self.o) + self.m *
(self.b * self.k - self.j * self.c) + self.a *
(self.j * self.o - self.n * self.k));
            tmp.c = d * (self.n * (self.c * self.h - self.g * self.d) + self.b *
(self.g * self.p - self.o * self.h) + self.f *
(self.o * self.d - self.c * self.p));
            tmp.g = d * (self.o * (self.a * self.h - self.e * self.d) + self.c *
(self.e * self.p - self.m * self.h) + self.g *
(self.m * self.d - self.a * self.p));
            tmp.k = d * (self.p * (self.a * self.f - self.e * self.b) + self.d *
(self.e * self.n - self.m * self.f) + self.h *
(self.m * self.b - self.a * self.n));
            tmp.o = d * (self.m * (self.f * self.c - self.b * self.g) + self.a *
(self.n * self.g - self.f * self.o) + self.e *
(self.b * self.o - self.n * self.c));
            tmp.d = d * (self.b * (self.k * self.h - self.g * self.l) + self.f *
(self.c * self.l - self.k * self.d) + self.j *
(self.g * self.d - self.c * self.h));
            tmp.h = d * (self.c * (self.i * self.h - self.e * self.l) + self.g *
(self.a * self.l - self.i * self.d) + self.k *
(self.e * self.d - self.a * self.h));
            tmp.l = d * (self.d * (self.i * self.f - self.e * self.j) + self.h *
(self.a * self.j - self.i * self.b) + self.l *
(self.e * self.b - self.a * self.f));
            tmp.p = d * (self.a * (self.f * self.k - self.j * self.g) + self.e *
(self.j * self.c - self.b * self.k) + self.i *
(self.b * self.g - self.f * self.c));
        return tmp;


class Quaternion:
```

```
# All methods and naming conventions based off
# http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions

# w is the real part, (x, y, z) are the imaginary parts
__slots__ = ['w', 'x', 'y', 'z']

def __init__(self, w=1, x=0, y=0, z=0):
    self.w = w
    self.x = x
    self.y = y
    self.z = z

def __copy__(self):
    Q = Quaternion()
    Q.w = self.w
    Q.x = self.x
    Q.y = self.y
    Q.z = self.z
    return Q

copy = __copy__

def __repr__(self):
    return 'Quaternion(real=%.2f, imag=<%.2f, %.2f, %.2f>)' % \
        (self.w, self.x, self.y, self.z)

def __mul__(self, other):
    if isinstance(other, Quaternion):
        Ax = self.x
        Ay = self.y
        Az = self.z
        Aw = self.w
        Bx = other.x
        By = other.y
        Bz = other.z
        Bw = other.w
        Q = Quaternion()
        Q.x =  Ax * Bw + Ay * Bz - Az * By + Aw * Bx
        Q.y = -Ax * Bz + Ay * Bw + Az * Bx + Aw * By
        Q.z =  Ax * By - Ay * Bx + Az * Bw + Aw * Bz
        Q.w = -Ax * Bx - Ay * By - Az * Bz + Aw * Bw
        return Q
    elif isinstance(other, Vector3):
        w = self.w
        x = self.x
```

```
            y = self.y
            z = self.z
            Vx = other.x
            Vy = other.y
            Vz = other.z
            ww = w * w
            w2 = w * 2
            wx2 = w2 * x
            wy2 = w2 * y
            wz2 = w2 * z
            xx = x * x
            x2 = x * 2
            xy2 = x2 * y
            xz2 = x2 * z
            yy = y * y
            yz2 = 2 * y * z
            zz = z * z
            return other.__class__(\
               ww * Vx + wy2 * Vz - wz2 * Vy + \
               xx * Vx + xy2 * Vy + xz2 * Vz - \
               zz * Vx - yy * Vx,
               xy2 * Vx + yy * Vy + yz2 * Vz + \
               wz2 * Vx - zz * Vy + ww * Vy - \
               wx2 * Vz - xx * Vy,
               xz2 * Vx + yz2 * Vy + \
               zz * Vz - wy2 * Vx - yy * Vz + \
               wx2 * Vy - xx * Vz + ww * Vz)
        else:
            other = other.copy()
            other._apply_transform(self)
            return other

    def __imul__(self, other):
        assert isinstance(other, Quaternion)
        Ax = self.x
        Ay = self.y
        Az = self.z
        Aw = self.w
        Bx = other.x
        By = other.y
        Bz = other.z
        Bw = other.w
        self.x =  Ax * Bw + Ay * Bz - Az * By + Aw * Bx
        self.y = -Ax * Bz + Ay * Bw + Az * Bx + Aw * By
        self.z =  Ax * By - Ay * Bx + Az * Bw + Aw * Bz
```

```
        self.w = -Ax * Bx - Ay * By - Az * Bz + Aw * Bw
        return self

    def __abs__(self):
        return math.sqrt(self.w ** 2 + \
                         self.x ** 2 + \
                         self.y ** 2 + \
                         self.z ** 2)

    magnitude = __abs__

    def magnitude_squared(self):
        return self.w ** 2 + \
               self.x ** 2 + \
               self.y ** 2 + \
               self.z ** 2

    def identity(self):
        self.w = 1
        self.x = 0
        self.y = 0
        self.z = 0
        return self

    def rotate_axis(self, angle, axis):
        self *= Quaternion.new_rotate_axis(angle, axis)
        return self

    def rotate_euler(self, heading, attitude, bank):
        self *= Quaternion.new_rotate_euler(heading, attitude, bank)
        return self

    def rotate_matrix(self, m):
        self *= Quaternion.new_rotate_matrix(m)
        return self

    def conjugated(self):
        Q = Quaternion()
        Q.w = self.w
        Q.x = -self.x
        Q.y = -self.y
        Q.z = -self.z
        return Q

    def normalise(self):
```

```
        d = self.magnitude()
        if d != 0:
            self.w /= d
            self.x /= d
            self.y /= d
            self.z /= d
        return self

    def normalised(self):
        d = self.magnitude()
        if d != 0:
            Q = Quaternion()
            Q.w = self.w / d
            Q.x = self.x / d
            Q.y = self.y / d
            Q.z = self.z / d
            return Q
        else:
            return self.copy()

    def get_angle_axis(self):
        if self.w > 1:
            self = self.normalised()
        angle = 2 * math.acos(self.w)
        s = math.sqrt(1 - self.w ** 2)
        if s < 0.001:
            return angle, Vector3(1, 0, 0)
        else:
            return angle, Vector3(self.x / s, self.y / s, self.z / s)

    def get_euler(self):
        t = self.x * self.y + self.z * self.w
        if t > 0.4999:
            heading = 2 * math.atan2(self.x, self.w)
            attitude = math.pi / 2
            bank = 0
        elif t < -0.4999:
            heading = -2 * math.atan2(self.x, self.w)
            attitude = -math.pi / 2
            bank = 0
        else:
            sqx = self.x ** 2
            sqy = self.y ** 2
            sqz = self.z ** 2
            heading = math.atan2(2 * self.y * self.w - 2 * self.x * self.z,
```

```
                            1 - 2 * sqy - 2 * sqz)
        attitude = math.asin(2 * t)
        bank = math.atan2(2 * self.x * self.w - 2 * self.y * self.z,
                          1 - 2 * sqx - 2 * sqz)
    return heading, attitude, bank

def get_matrix(self):
    xx = self.x ** 2
    xy = self.x * self.y
    xz = self.x * self.z
    xw = self.x * self.w
    yy = self.y ** 2
    yz = self.y * self.z
    yw = self.y * self.w
    zz = self.z ** 2
    zw = self.z * self.w
    M = Matrix4()
    M.a = 1 - 2 * (yy + zz)
    M.b = 2 * (xy - zw)
    M.c = 2 * (xz + yw)
    M.e = 2 * (xy + zw)
    M.f = 1 - 2 * (xx + zz)
    M.g = 2 * (yz - xw)
    M.i = 2 * (xz - yw)
    M.j = 2 * (yz + xw)
    M.k = 1 - 2 * (xx + yy)
    return M

# Static constructors
def new_identity(cls):
    return cls()
new_identity = classmethod(new_identity)

def new_rotate_axis(cls, angle, axis):
    assert(isinstance(axis, Vector3))
    axis = axis.normalised()
    s = math.sin(angle / 2)
    Q = cls()
    Q.w = math.cos(angle / 2)
    Q.x = axis.x * s
    Q.y = axis.y * s
    Q.z = axis.z * s
    return Q
new_rotate_axis = classmethod(new_rotate_axis)
```

```
def new_rotate_euler(cls, heading, attitude, bank):
    Q = cls()
    c1 = math.cos(heading / 2)
    s1 = math.sin(heading / 2)
    c2 = math.cos(attitude / 2)
    s2 = math.sin(attitude / 2)
    c3 = math.cos(bank / 2)
    s3 = math.sin(bank / 2)

    Q.w = c1 * c2 * c3 - s1 * s2 * s3
    Q.x = s1 * s2 * c3 + c1 * c2 * s3
    Q.y = s1 * c2 * c3 + c1 * s2 * s3
    Q.z = c1 * s2 * c3 - s1 * c2 * s3
    return Q
new_rotate_euler = classmethod(new_rotate_euler)

def new_rotate_matrix(cls, m):
  if m[0*4 + 0] + m[1*4 + 1] + m[2*4 + 2] > 0.00000001:
    t = m[0*4 + 0] + m[1*4 + 1] + m[2*4 + 2] + 1.0
    s = 0.5/math.sqrt(t)

    return cls(
      s*t,
      (m[1*4 + 2] - m[2*4 + 1])*s,
      (m[2*4 + 0] - m[0*4 + 2])*s,
      (m[0*4 + 1] - m[1*4 + 0])*s
      )

  elif m[0*4 + 0] > m[1*4 + 1] and m[0*4 + 0] > m[2*4 + 2]:
    t = m[0*4 + 0] - m[1*4 + 1] - m[2*4 + 2] + 1.0
    s = 0.5/math.sqrt(t)

    return cls(
      (m[1*4 + 2] - m[2*4 + 1])*s,
      s*t,
      (m[0*4 + 1] + m[1*4 + 0])*s,
      (m[2*4 + 0] + m[0*4 + 2])*s
      )

  elif m[1*4 + 1] > m[2*4 + 2]:
    t = -m[0*4 + 0] + m[1*4 + 1] - m[2*4 + 2] + 1.0
    s = 0.5/math.sqrt(t)

    return cls(
      (m[2*4 + 0] - m[0*4 + 2])*s,
```

```
      (m[0*4 + 1] + m[1*4 + 0])*s,
      s*t,
      (m[1*4 + 2] + m[2*4 + 1])*s
      )

  else:
    t = -m[0*4 + 0] - m[1*4 + 1] + m[2*4 + 2] + 1.0
    s = 0.5/math.sqrt(t)

    return cls(
      (m[0*4 + 1] - m[1*4 + 0])*s,
      (m[2*4 + 0] + m[0*4 + 2])*s,
      (m[1*4 + 2] + m[2*4 + 1])*s,
      s*t
      )
new_rotate_matrix = classmethod(new_rotate_matrix)

def new_interpolate(cls, q1, q2, t):
    assert isinstance(q1, Quaternion) and isinstance(q2, Quaternion)
    Q = cls()

    costheta = q1.w * q2.w + q1.x * q2.x + q1.y * q2.y + q1.z * q2.z
    if costheta < 0.:
        costheta = -costheta
        q1 = q1.conjugated()
    elif costheta > 1:
        costheta = 1

    theta = math.acos(costheta)
    if abs(theta) < 0.01:
        Q.w = q2.w
        Q.x = q2.x
        Q.y = q2.y
        Q.z = q2.z
        return Q

    sintheta = math.sqrt(1.0 - costheta * costheta)
    if abs(sintheta) < 0.01:
        Q.w = (q1.w + q2.w) * 0.5
        Q.x = (q1.x + q2.x) * 0.5
        Q.y = (q1.y + q2.y) * 0.5
        Q.z = (q1.z + q2.z) * 0.5
        return Q

    ratio1 = math.sin((1 - t) * theta) / sintheta
```

```
        ratio2 = math.sin(t * theta) / sintheta

        Q.w = q1.w * ratio1 + q2.w * ratio2
        Q.x = q1.x * ratio1 + q2.x * ratio2
        Q.y = q1.y * ratio1 + q2.y * ratio2
        Q.z = q1.z * ratio1 + q2.z * ratio2
        return Q
    new_interpolate = classmethod(new_interpolate)

# Geometry
# Much maths thanks to Paul Bourke, http://astronomy.swin.edu.au/~pbourke
# --------------------------------------------------------------------------

class Geometry:
    def _connect_unimplemented(self, other):
        raise AttributeError( 'Cannot connect %s to %s' % \
            (self.__class__, other.__class__))

    def _intersect_unimplemented(self, other):
        raise AttributeError( 'Cannot intersect %s and %s' % \
            (self.__class__, other.__class__))

    _intersect_point2 = _intersect_unimplemented
    _intersect_line2 = _intersect_unimplemented
    _intersect_circle = _intersect_unimplemented
    _connect_point2 = _connect_unimplemented
    _connect_line2 = _connect_unimplemented
    _connect_circle = _connect_unimplemented

    _intersect_point3 = _intersect_unimplemented
    _intersect_line3 = _intersect_unimplemented
    _intersect_sphere = _intersect_unimplemented
    _intersect_plane = _intersect_unimplemented
    _connect_point3 = _connect_unimplemented
    _connect_line3 = _connect_unimplemented
    _connect_sphere = _connect_unimplemented
    _connect_plane = _connect_unimplemented

    def intersect(self, other):
        raise NotImplementedError

    def connect(self, other):
        raise NotImplementedError

    def distance(self, other):
```

```
        c = self.connect(other)
        if c:
            return c.length
        return 0.0

def _intersect_point2_circle(P, C):
    return abs(P - C.c) <= C.r

def _intersect_line2_line2(A, B):
    d = B.v.y * A.v.x - B.v.x * A.v.y
    if d == 0:
        return None

    dy = A.p.y - B.p.y
    dx = A.p.x - B.p.x
    ua = (B.v.x * dy - B.v.y * dx) / d
    if not A._u_in(ua):
        return None
    ub = (A.v.x * dy - A.v.y * dx) / d
    if not B._u_in(ub):
        return None

    return Point2(A.p.x + ua * A.v.x,
                  A.p.y + ua * A.v.y)

def _intersect_line2_circle(L, C):
    a = L.v.magnitude_squared()
    b = 2 * (L.v.x * (L.p.x - C.c.x) + \
             L.v.y * (L.p.y - C.c.y))
    c = C.c.magnitude_squared() + \
        L.p.magnitude_squared() - \
        2 * C.c.dot(L.p) - \
        C.r ** 2
    det = b ** 2 - 4 * a * c
    if det < 0:
        return None
    sq = math.sqrt(det)
    u1 = (-b + sq) / (2 * a)
    u2 = (-b - sq) / (2 * a)
    if not L._u_in(u1):
        u1 = max(min(u1, 1.0), 0.0)
    if not L._u_in(u2):
        u2 = max(min(u2, 1.0), 0.0)

    # Tangent
```

```
    if u1 == u2:
        return Point2(L.p.x + u1 * L.v.x,
                      L.p.y + u1 * L.v.y)

    return LineSegment2(Point2(L.p.x + u1 * L.v.x,
                               L.p.y + u1 * L.v.y),
                        Point2(L.p.x + u2 * L.v.x,
                               L.p.y + u2 * L.v.y))

def _connect_point2_line2(P, L):
    d = L.v.magnitude_squared()
    assert d != 0
    u = ((P.x - L.p.x) * L.v.x + \
         (P.y - L.p.y) * L.v.y) / d
    if not L._u_in(u):
        u = max(min(u, 1.0), 0.0)
    return LineSegment2(P,
                        Point2(L.p.x + u * L.v.x,
                               L.p.y + u * L.v.y))

def _connect_point2_circle(P, C):
    v = P - C.c
    v.normalise()
    v *= C.r
    return LineSegment2(P, Point2(C.c.x + v.x, C.c.y + v.y))

def _connect_line2_line2(A, B):
    d = B.v.y * A.v.x - B.v.x * A.v.y
    if d == 0:
        # Parallel, connect an endpoint with a line
        if isinstance(B, Ray2) or isinstance(B, LineSegment2):
            p1, p2 = _connect_point2_line2(B.p, A)
            return p2, p1
        # No endpoint (or endpoint is on A), possibly choose arbitrary point
        # on line.
        return _connect_point2_line2(A.p, B)

    dy = A.p.y - B.p.y
    dx = A.p.x - B.p.x
    ua = (B.v.x * dy - B.v.y * dx) / d
    if not A._u_in(ua):
        ua = max(min(ua, 1.0), 0.0)
    ub = (A.v.x * dy - A.v.y * dx) / d
    if not B._u_in(ub):
        ub = max(min(ub, 1.0), 0.0)
```

```
    return LineSegment2(Point2(A.p.x + ua * A.v.x, A.p.y + ua * A.v.y),
                        Point2(B.p.x + ub * B.v.x, B.p.y + ub * B.v.y))

def _connect_circle_line2(C, L):
    d = L.v.magnitude_squared()
    assert d != 0
    u = ((C.c.x - L.p.x) * L.v.x + (C.c.y - L.p.y) * L.v.y) / d
    if not L._u_in(u):
        u = max(min(u, 1.0), 0.0)
    point = Point2(L.p.x + u * L.v.x, L.p.y + u * L.v.y)
    v = (point - C.c)
    v.normalise()
    v *= C.r
    return LineSegment2(Point2(C.c.x + v.x, C.c.y + v.y), point)

def _connect_circle_circle(A, B):
    v = B.c - A.c
    d = v.magnitude()
    if A.r >= B.r and d < A.r:
        #centre B inside A
        s1,s2 = +1, +1
    elif B.r > A.r and d < B.r:
        #centre A inside B
        s1,s2 = -1, -1
    elif d >= A.r and d >= B.r:
        s1,s2 = +1, -1
    v.normalise()
    return LineSegment2(Point2(A.c.x + s1 * v.x * A.r, A.c.y + s1 * v.y * A.r),
                        Point2(B.c.x + s2 * v.x * B.r, B.c.y + s2 * v.y * B.r))


class Point2(Vector2, Geometry):
    def __repr__(self):
        return 'Point2(%.2f, %.2f)' % (self.x, self.y)

    def intersect(self, other):
        return other._intersect_point2(self)

    def _intersect_circle(self, other):
        return _intersect_point2_circle(self, other)

    def connect(self, other):
        return other._connect_point2(self)
```

```
    def _connect_point2(self, other):
        return LineSegment2(other, self)

    def _connect_line2(self, other):
        c = _connect_point2_line2(self, other)
        if c:
            return c._swap()

    def _connect_circle(self, other):
        c = _connect_point2_circle(self, other)
        if c:
            return c._swap()

class Line2(Geometry):
    __slots__ = ['p', 'v']

    def __init__(self, *args):
        if len(args) == 3:
            assert isinstance(args[0], Point2) and \
                   isinstance(args[1], Vector2) and \
                   type(args[2]) == float
            self.p = args[0].copy()
            self.v = args[1] * args[2] / abs(args[1])
        elif len(args) == 2:
            if isinstance(args[0], Point2) and isinstance(args[1], Point2):
                self.p = args[0].copy()
                self.v = args[1] - args[0]
            elif isinstance(args[0], Point2) and isinstance(args[1], Vector2):
                self.p = args[0].copy()
                self.v = args[1].copy()
            else:
                raise AttributeError( '%r' % (args,))
        elif len(args) == 1:
            if isinstance(args[0], Line2):
                self.p = args[0].p.copy()
                self.v = args[0].v.copy()
            else:
                raise AttributeError( '%r' % (args,))
        else:
            raise AttributeError( '%r' % (args,))

        if not self.v:
            raise AttributeError( 'Line has zero-length vector')

    def __copy__(self):
```

```
        return self.__class__(self.p, self.v)

    copy = __copy__

    def __repr__(self):
        return 'Line2(<%.2f, %.2f> + u<%.2f, %.2f>)' % \
            (self.p.x, self.p.y, self.v.x, self.v.y)

    p1 = property(lambda self: self.p)
    p2 = property(lambda self: Point2(self.p.x + self.v.x,
                                      self.p.y + self.v.y))

    def _apply_transform(self, t):
        self.p = t * self.p
        self.v = t * self.v

    def _u_in(self, u):
        return True

    def intersect(self, other):
        return other._intersect_line2(self)

    def _intersect_line2(self, other):
        return _intersect_line2_line2(self, other)

    def _intersect_circle(self, other):
        return _intersect_line2_circle(self, other)

    def connect(self, other):
        return other._connect_line2(self)

    def _connect_point2(self, other):
        return _connect_point2_line2(other, self)

    def _connect_line2(self, other):
        return _connect_line2_line2(other, self)

    def _connect_circle(self, other):
        return _connect_circle_line2(other, self)

class Ray2(Line2):
    def __repr__(self):
        return 'Ray2(<%.2f, %.2f> + u<%.2f, %.2f>)' % \
            (self.p.x, self.p.y, self.v.x, self.v.y)
```

```
    def _u_in(self, u):
        return u >= 0.0

class LineSegment2(Line2):
    def __repr__(self):
        return 'LineSegment2(<%.2f, %.2f> to <%.2f, %.2f>)' % \
            (self.p.x, self.p.y, self.p.x + self.v.x, self.p.y + self.v.y)

    def _u_in(self, u):
        return u >= 0.0 and u <= 1.0

    def __abs__(self):
        return abs(self.v)

    def magnitude_squared(self):
        return self.v.magnitude_squared()

    def _swap(self):
        # used by connect methods to switch order of points
        self.p = self.p2
        self.v *= -1
        return self

    length = property(lambda self: abs(self.v))

class Circle(Geometry):
    __slots__ = ['c', 'r']

    def __init__(self, center, radius):
        assert isinstance(center, Vector2) and type(radius) == float
        self.c = center.copy()
        self.r = radius

    def __copy__(self):
        return self.__class__(self.c, self.r)

    copy = __copy__

    def __repr__(self):
        return 'Circle(<%.2f, %.2f>, radius=%.2f)' % \
            (self.c.x, self.c.y, self.r)

    def _apply_transform(self, t):
        self.c = t * self.c
```

```
    def intersect(self, other):
        return other._intersect_circle(self)

    def _intersect_point2(self, other):
        return _intersect_point2_circle(other, self)

    def _intersect_line2(self, other):
        return _intersect_line2_circle(other, self)

    def connect(self, other):
        return other._connect_circle(self)

    def _connect_point2(self, other):
        return _connect_point2_circle(other, self)

    def _connect_line2(self, other):
        c = _connect_circle_line2(self, other)
        if c:
            return c._swap()

    def _connect_circle(self, other):
        return _connect_circle_circle(other, self)

# 3D Geometry
# -----------------------------------------------------------------------------

def _connect_point3_line3(P, L):
    d = L.v.magnitude_squared()
    assert d != 0
    u = ((P.x - L.p.x) * L.v.x + \
         (P.y - L.p.y) * L.v.y + \
         (P.z - L.p.z) * L.v.z) / d
    if not L._u_in(u):
        u = max(min(u, 1.0), 0.0)
    return LineSegment3(P, Point3(L.p.x + u * L.v.x,
                                  L.p.y + u * L.v.y,
                                  L.p.z + u * L.v.z))

def _connect_point3_sphere(P, S):
    v = P - S.c
    v.normalise()
    v *= S.r
    return LineSegment3(P, Point3(S.c.x + v.x, S.c.y + v.y, S.c.z + v.z))

def _connect_point3_plane(p, plane):
```

```
    n = plane.n.normalised()
    d = p.dot(plane.n) - plane.k
    return LineSegment3(p, Point3(p.x - n.x * d, p.y - n.y * d, p.z - n.z * d))

def _connect_line3_line3(A, B):
    assert A.v and B.v
    p13 = A.p - B.p
    d1343 = p13.dot(B.v)
    d4321 = B.v.dot(A.v)
    d1321 = p13.dot(A.v)
    d4343 = B.v.magnitude_squared()
    denom = A.v.magnitude_squared() * d4343 - d4321 ** 2
    if denom == 0:
        # Parallel, connect an endpoint with a line
        if isinstance(B, Ray3) or isinstance(B, LineSegment3):
            return _connect_point3_line3(B.p, A)._swap()
        # No endpoint (or endpoint is on A), possibly choose arbitrary
        # point on line.
        return _connect_point3_line3(A.p, B)

    ua = (d1343 * d4321 - d1321 * d4343) / denom
    if not A._u_in(ua):
        ua = max(min(ua, 1.0), 0.0)
    ub = (d1343 + d4321 * ua) / d4343
    if not B._u_in(ub):
        ub = max(min(ub, 1.0), 0.0)
    return LineSegment3(Point3(A.p.x + ua * A.v.x,
                               A.p.y + ua * A.v.y,
                               A.p.z + ua * A.v.z),
                        Point3(B.p.x + ub * B.v.x,
                               B.p.y + ub * B.v.y,
                               B.p.z + ub * B.v.z))

def _connect_line3_plane(L, P):
    d = P.n.dot(L.v)
    if not d:
        # Parallel, choose an endpoint
        return _connect_point3_plane(L.p, P)
    u = (P.k - P.n.dot(L.p)) / d
    if not L._u_in(u):
        # intersects out of range, choose nearest endpoint
        u = max(min(u, 1.0), 0.0)
        return _connect_point3_plane(Point3(L.p.x + u * L.v.x,
                                            L.p.y + u * L.v.y,
                                            L.p.z + u * L.v.z), P)
```

```
    # Intersection
    return None

def _connect_sphere_line3(S, L):
    d = L.v.magnitude_squared()
    assert d != 0
    u = ((S.c.x - L.p.x) * L.v.x + \
         (S.c.y - L.p.y) * L.v.y + \
         (S.c.z - L.p.z) * L.v.z) / d
    if not L._u_in(u):
        u = max(min(u, 1.0), 0.0)
    point = Point3(L.p.x + u * L.v.x, L.p.y + u * L.v.y, L.p.z + u * L.v.z)
    v = (point - S.c)
    v.normalise()
    v *= S.r
    return LineSegment3(Point3(S.c.x + v.x, S.c.y + v.y, S.c.z + v.z),
                        point)

def _connect_sphere_sphere(A, B):
    v = B.c - A.c
    d = v.magnitude()
    if A.r >= B.r and d < A.r:
        #centre B inside A
        s1,s2 = +1, +1
    elif B.r > A.r and d < B.r:
        #centre A inside B
        s1,s2 = -1, -1
    elif d >= A.r and d >= B.r:
        s1,s2 = +1, -1

    v.normalise()
    return LineSegment3(Point3(A.c.x + s1* v.x * A.r,
                               A.c.y + s1* v.y * A.r,
                               A.c.z + s1* v.z * A.r),
                        Point3(B.c.x + s2* v.x * B.r,
                               B.c.y + s2* v.y * B.r,
                               B.c.z + s2* v.z * B.r))

def _connect_sphere_plane(S, P):
    c = _connect_point3_plane(S.c, P)
    if not c:
        return None
    p2 = c.p2
    v = p2 - S.c
    v.normalise()
```

```
    v *= S.r
    return LineSegment3(Point3(S.c.x + v.x, S.c.y + v.y, S.c.z + v.z),
                        p2)

def _connect_plane_plane(A, B):
    if A.n.cross(B.n):
        # Planes intersect
        return None
    else:
        # Planes are parallel, connect to arbitrary point
        return _connect_point3_plane(A._get_point(), B)

def _intersect_point3_sphere(P, S):
    return abs(P - S.c) <= S.r

def _intersect_line3_sphere(L, S):
    a = L.v.magnitude_squared()
    b = 2 * (L.v.x * (L.p.x - S.c.x) + \
             L.v.y * (L.p.y - S.c.y) + \
             L.v.z * (L.p.z - S.c.z))
    c = S.c.magnitude_squared() + \
        L.p.magnitude_squared() - \
        2 * S.c.dot(L.p) - \
        S.r ** 2
    det = b ** 2 - 4 * a * c
    if det < 0:
        return None
    sq = math.sqrt(det)
    u1 = (-b + sq) / (2 * a)
    u2 = (-b - sq) / (2 * a)
    if not L._u_in(u1):
        u1 = max(min(u1, 1.0), 0.0)
    if not L._u_in(u2):
        u2 = max(min(u2, 1.0), 0.0)
    return LineSegment3(Point3(L.p.x + u1 * L.v.x,
                               L.p.y + u1 * L.v.y,
                               L.p.z + u1 * L.v.z),
                        Point3(L.p.x + u2 * L.v.x,
                               L.p.y + u2 * L.v.y,
                               L.p.z + u2 * L.v.z))

def _intersect_line3_plane(L, P):
    d = P.n.dot(L.v)
    if not d:
        # Parallel
```

```
        return None
    u = (P.k - P.n.dot(L.p)) / d
    if not L._u_in(u):
        return None
    return Point3(L.p.x + u * L.v.x,
                  L.p.y + u * L.v.y,
                  L.p.z + u * L.v.z)

def _intersect_plane_plane(A, B):
    n1_m = A.n.magnitude_squared()
    n2_m = B.n.magnitude_squared()
    n1d2 = A.n.dot(B.n)
    det = n1_m * n2_m - n1d2 ** 2
    if det == 0:
        # Parallel
        return None
    c1 = (A.k * n2_m - B.k * n1d2) / det
    c2 = (B.k * n1_m - A.k * n1d2) / det
    return Line3(Point3(c1 * A.n.x + c2 * B.n.x,
                        c1 * A.n.y + c2 * B.n.y,
                        c1 * A.n.z + c2 * B.n.z),
                 A.n.cross(B.n))

class Point3(Vector3, Geometry):
    def __repr__(self):
        return 'Point3(%.2f, %.2f, %.2f)' % (self.x, self.y, self.z)

    def intersect(self, other):
        return other._intersect_point3(self)

    def _intersect_sphere(self, other):
        return _intersect_point3_sphere(self, other)

    def connect(self, other):
        return other._connect_point3(self)

    def _connect_point3(self, other):
        if self != other:
            return LineSegment3(other, self)
        return None

    def _connect_line3(self, other):
        c = _connect_point3_line3(self, other)
        if c:
            return c._swap()
```

```
    def _connect_sphere(self, other):
        c = _connect_point3_sphere(self, other)
        if c:
            return c._swap()

    def _connect_plane(self, other):
        c = _connect_point3_plane(self, other)
        if c:
            return c._swap()

class Line3:
    __slots__ = ['p', 'v']

    def __init__(self, *args):
        if len(args) == 3:
            assert isinstance(args[0], Point3) and \
                   isinstance(args[1], Vector3) and \
                   type(args[2]) == float
            self.p = args[0].copy()
            self.v = args[1] * args[2] / abs(args[1])
        elif len(args) == 2:
            if isinstance(args[0], Point3) and isinstance(args[1], Point3):
                self.p = args[0].copy()
                self.v = args[1] - args[0]
            elif isinstance(args[0], Point3) and isinstance(args[1], Vector3):
                self.p = args[0].copy()
                self.v = args[1].copy()
            else:
                raise AttributeError( '%r' % (args,))
        elif len(args) == 1:
            if isinstance(args[0], Line3):
                self.p = args[0].p.copy()
                self.v = args[0].v.copy()
            else:
                raise AttributeError( '%r' % (args,))
        else:
            raise AttributeError( '%r' % (args,))

        # XXX This is annoying.
        #if not self.v:
        #    raise AttributeError, 'Line has zero-length vector'

    def __copy__(self):
        return self.__class__(self.p, self.v)
```

```
copy = __copy__

def __repr__(self):
    return 'Line3(<%.2f, %.2f, %.2f> + u<%.2f, %.2f, %.2f>)' % \
        (self.p.x, self.p.y, self.p.z, self.v.x, self.v.y, self.v.z)

p1 = property(lambda self: self.p)
p2 = property(lambda self: Point3(self.p.x + self.v.x,
                                  self.p.y + self.v.y,
                                  self.p.z + self.v.z))

def _apply_transform(self, t):
    self.p = t * self.p
    self.v = t * self.v

def _u_in(self, u):
    return True

def intersect(self, other):
    return other._intersect_line3(self)

def _intersect_sphere(self, other):
    return _intersect_line3_sphere(self, other)

def _intersect_plane(self, other):
    return _intersect_line3_plane(self, other)

def connect(self, other):
    return other._connect_line3(self)

def _connect_point3(self, other):
    return _connect_point3_line3(other, self)

def _connect_line3(self, other):
    return _connect_line3_line3(other, self)

def _connect_sphere(self, other):
    return _connect_sphere_line3(other, self)

def _connect_plane(self, other):
    c = _connect_line3_plane(self, other)
    if c:
        return c
```

```
class Ray3(Line3):
    def __repr__(self):
        return 'Ray3(<%.2f, %.2f, %.2f> + u<%.2f, %.2f, %.2f>)' % \
            (self.p.x, self.p.y, self.p.z, self.v.x, self.v.y, self.v.z)

    def _u_in(self, u):
        return u >= 0.0

class LineSegment3(Line3):
    def __repr__(self):
        return 'LineSegment3(<%.2f, %.2f, %.2f> to <%.2f, %.2f, %.2f>)' % \
            (self.p.x, self.p.y, self.p.z,
             self.p.x + self.v.x, self.p.y + self.v.y, self.p.z + self.v.z)

    def _u_in(self, u):
        return u >= 0.0 and u <= 1.0

    def __abs__(self):
        return abs(self.v)

    def magnitude_squared(self):
        return self.v.magnitude_squared()

    def _swap(self):
        # used by connect methods to switch order of points
        self.p = self.p2
        self.v *= -1
        return self

    length = property(lambda self: abs(self.v))

class Sphere:
    __slots__ = ['c', 'r']

    def __init__(self, center, radius):
        assert isinstance(center, Vector3) and type(radius) == float
        self.c = center.copy()
        self.r = radius

    def __copy__(self):
        return self.__class__(self.c, self.r)

    copy = __copy__

    def __repr__(self):
```

```
            return 'Sphere(<%.2f, %.2f, %.2f>, radius=%.2f)' % \
                (self.c.x, self.c.y, self.c.z, self.r)

        def _apply_transform(self, t):
            self.c = t * self.c

        def intersect(self, other):
            return other._intersect_sphere(self)

        def _intersect_point3(self, other):
            return _intersect_point3_sphere(other, self)

        def _intersect_line3(self, other):
            return _intersect_line3_sphere(other, self)

        def connect(self, other):
            return other._connect_sphere(self)

        def _connect_point3(self, other):
            return _connect_point3_sphere(other, self)

        def _connect_line3(self, other):
            c = _connect_sphere_line3(self, other)
            if c:
                return c._swap()

        def _connect_sphere(self, other):
            return _connect_sphere_sphere(other, self)

        def _connect_plane(self, other):
            c = _connect_sphere_plane(self, other)
            if c:
                return c

    class Plane:
        # n.p = k, where n is normal, p is point on plane, k is constant scalar
        __slots__ = ['n', 'k']

        def __init__(self, *args):
            if len(args) == 3:
                assert isinstance(args[0], Point3) and \
                       isinstance(args[1], Point3) and \
                       isinstance(args[2], Point3)
                self.n = (args[1] - args[0]).cross(args[2] - args[0])
                self.n.normalise()
```

```
            self.k = self.n.dot(args[0])
        elif len(args) == 2:
            if isinstance(args[0], Point3) and isinstance(args[1], Vector3):
                self.n = args[1].normalised()
                self.k = self.n.dot(args[0])
            elif isinstance(args[0], Vector3) and type(args[1]) == float:
                self.n = args[0].normalised()
                self.k = args[1]
            else:
                raise AttributeError( '%r' % (args,))

        else:
            raise AttributeError( '%r' % (args,))

        if not self.n:
            raise AttributeError( 'Points on plane are colinear')

    def __copy__(self):
        return self.__class__(self.n, self.k)

    copy = __copy__

    def __repr__(self):
        return 'Plane(<%.2f, %.2f, %.2f>.p = %.2f)' % \
            (self.n.x, self.n.y, self.n.z, self.k)

    def _get_point(self):
        # Return an arbitrary point on the plane
        if self.n.z:
            return Point3(0., 0., self.k / self.n.z)
        elif self.n.y:
            return Point3(0., self.k / self.n.y, 0.)
        else:
            return Point3(self.k / self.n.x, 0., 0.)

    def _apply_transform(self, t):
        p = t * self._get_point()
        self.n = t * self.n
        self.k = self.n.dot(p)

    def intersect(self, other):
        return other._intersect_plane(self)

    def _intersect_line3(self, other):
        return _intersect_line3_plane(other, self)
```

```
    def _intersect_plane(self, other):
        return _intersect_plane_plane(self, other)

    def connect(self, other):
        return other._connect_plane(self)

    def _connect_point3(self, other):
        return _connect_point3_plane(other, self)

    def _connect_line3(self, other):
        return _connect_line3_plane(other, self)

    def _connect_sphere(self, other):
        return _connect_sphere_plane(other, self)

    def _connect_plane(self, other):
        return _connect_plane_plane(other, self)

#------------------------------------------------------------------------------
#------------------------------------------------------------------------------

# This second part of the code was written by Claudia Caudai

import numpy as np
import math
import random
from io import StringIO
from scipy import linalg as LA
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt


# PARAMETERS
initemp=50000   # INITIAL TEMPERATURE (MONTE CARLO ALGORITHM)
numiterationskb=100000   # MAX NUMBER OF ITERATIONS FOR 100KB CHAIN
energytresholdkb=50   # MAX NUMBER OF ITERATION WITHOUT ENERGY DECREASE FOR
100KB CHAINS
energytresholdMb=50   # MAX NUMBER OF ITERATION WITHOUT ENERGY DECREASE FOR
LONG CHAIN (i.e. LOW RESOLUTION CHAIN)
numiterationsMb=100000     # MAX NUMBER OF ITERATIONS FOR LONG CHAIN
numiterationsTemp=200   # NUMBER OF ITERATIONS FOR TEMPERATURE INCREASE
epsilonchi=0.1   # ACCEPTANCE ERROR PARAMETER
incremtemp=1.5   # TEMPERATURE INCREASING RATE
dimrate=0.998   # TEMPERATURE DECREASING RATE
```

```
sparseMb=8   # SPARSITY FACTOR IN CONTACT MATRIX BINNED FOLLOWIN DIVISION IN
TOPOLOGICAL DOMAINS
sparsekb=1.5   # SPARSITY FACTOR IN 100KB MATRICES
diagskb=2   # NUMBER OF NEGLECTED DIAGONALS IN 100KB MATRICES
diagsMb=2   # NUMBER OF NEGLECTED DIAGONALS IN LONG CHAIN MATRIX
mindistkb=120   # COMPENETRATION DISTANCE IN 100KB CHAINS
maxdistkb=8000   # MAX DISTANCE (NUCLEUS DIAMETER)
#maxdistMb=8000   # MAX DISTANCE (NUCLEUS DIAMETER)
maxangle=100   # IN DEGREES  # MAX CURVATURE ANGLE
arad=math.radians(maxangle)   # IN RADIANS  # MAX CURVATURE ANGLE

# IMPORT CONTACT MATRIX AND TOPOLOGICAL DOMAINS
HM= np.loadtxt('100kb_chr1_short_txt.txt')
domin=np.loadtxt('dom_chr1.txt')

# NUMBER OF DOMAINS
nbinMb=len(domin[:,1])
print "The total number of TD is", nbinMb

# NUMBER OF SELECTED CONTACTS IN THE LOWER RESOLUTION MATRIX
numcontMb = math.trunc(nbinMb**2/(2*sparseMb))


# PROCEDURE 1
def procedure1(l):

    global fincoordkb
    global stepenkb

    n = int(domin[l,1]-domin[l,0]+1)
    print "il segmento", l+1 ,"ha", n ,"bin"

    numcontkb=int(n*sparsekb)
    # print "the segment", l+1 ,"has", numcontkb ,"contacts"

    # definition of functions used into procedure1
    def diam(s):
        return 500-(HM[domin[l,0]-1+s,domin[l,0]-1+s]*300)/1000
        #print "the diameter of bin",s+1, "is", diam

    def vect(s):
        return diam(s)/2+diam(s+1)/2

    #for s in range(n):
        #print vect(s),s
```

```
 vectiniz=[]
 vector=[]
 for s in range(n-1):
   vectiniz.append(vect(s))
   vector.append(vect(s))

 alpha=[]
 beta=[]
 for s in range(n-2):
     alpha.append(0)
     beta.append(0)

 pHM=[]
 for s in range(n-1):
     for t in range(n-1):
         if (t-s)>= diagskb:
             pHM.append(HM[domin[l,0]+s,domin[l,0]+t])
 pHM=sorted(pHM,reverse=True)
 minkb=pHM[numcontkb-1]
 #print pHM,len(pHM),minkb

 pHMM=[]
 for s in range(n-1):
     for t in range(n-1):
         if (HM[domin[l,0]+s,domin[l,0]+t]>=minkb and (t-s)>=diagskb):
             print "in the fragment", l+1,"there are",
 int(HM[domin[l,0]+s,domin[l,0]+t]),
"contacts between the bin", int(domin[l,0]+s+1),
"and the bin", int(domin[l,0]+t+1)
             pHMM.append([HM[domin[l,0]+s,domin[l,0]+t],s+1,t+1])
 pHMM.reverse()
 no=len(pHMM)


 contenergy=0
 contiter=0
 contacc=0
 chitemp=0
 gg=0
 temp=initemp
 s=0
 en=[]
 stepenkb=[]
 fincoordkb=[]
```

```
while (contenergy<energytresholdkb and s<=numiterationskb):

    contenergy=contenergy+1
    contiter=contiter+1
    chitemp=(contacc+1.0)/contiter

    if (contiter==numiterationsTemp and gg==0):
        if chitemp<(1-epsilonchi):
            print "after", contiter, "iterations, the rate between
transitions accepted and proposed is", chitemp
            contiter=0
            contacc=0
            contanergy=0
            temp=temp*incremtemp
            print "the temperature grows up till", temp
        else:
            print "at iteration", s+1, "the rate is", chitemp, "and temperature
starts decreasing with the value", temp
            contiter=0
            contacc=0
            contenergy=0
            gg=1

    if gg==1:
        temp=temp*dimrate

    al=[]
    be=[]
    vt=[]
    for t in range(n-1):
        randb=random.randint(-10,10)
        #b=1
        if vectiniz[t]-20<=vector[t]+randb<=vectiniz[t]+20:
            vt.append(vector[t]+randb)
        else:
            vt.append(vector[t]-randb)
        #print vt[t],t

    for t in range(n-2):
        randb=random.randint(-10,10)/100.0
        #b=1/10.0
        if -arad<=alpha[t]+randb<=arad:
            al.append(alpha[t]+randb)
        else:
            al.append(alpha[t]-randb)
```

```
            randb=random.randint(-10,10)/100.0
            #b=1/10.0
            be.append(beta[t]+randb)

        # PLANAR ANGLES
        alp=[]
        p_al=[]
        dtpl=[]
        dtpl.append(0)
        dtpl.append(0)
        p_al.append(np.array((0,0,0)))
        p_al.append(np.array((vt[0],0,0)))
        for t in range(2,n):
            alp.append(sum((al[x]) for x in range(t-1)))
            p_al.append(np.array((p_al[t-1][0]+(math.cos(alp[t-2])*vt[t-1]),
p_al[t-1][1]+(math.sin(alp[t-2])*vt[t-1]),0)))
            dtpl.append([LA.norm(p_al[t]-p_al[t-1])])
            #print alp[t-2],"alp",p_al[t],"pal",dtpl[t],"dist",t

        # DIHEDRAL ANGLES
        pr=[]
        pr.append(p_al[0])
        pr.append(p_al[1])
        qua=[]
        qua.append(Quaternion())
        qua.append(Quaternion())
        q1=[]
        q1.append(Quaternion())
        q1.append(Quaternion())
        v=[]
        v.append(np.array((1,0,0)))
        v1=[]
        v1.append(np.array((1,0,0)))
        for t in range(1,n-1):
            v.append((p_al[t+1]-p_al[t])/(vt[t]*1.0))
            v1.append((p_al[t+1]-p_al[t]))
            #print v[t],t
        dtdi=[]
        dtdi.append(0)
        dtdi.append([LA.norm(pr[1]-pr[0])])
        for t in range(2,n):
            qua.append(Quaternion.new_rotate_axis(be[t-2],
Vector3(v[t-2][0],v[t-2][1],v[t-2][2])))
            q1.append(q1[t-1]*qua[t].normalise())
            pr.append(q1[t]*Vector3(v1[t-1][0],v1[t-1][1],v1[t-1][2])+pr[t-1])
```

```
            dtdi.append([LA.norm(pr[t]-pr[t-1])])
            #print v1[t-1],"v1",pr[t],"pr",t
            #print dtdi[t-2],t-2
        #print pr

        # DISTANCE MATRIX
        M=np.zeros((n,n))
        for g in range(n):
            for t in range(n):
                M[g][t]=LA.norm(pr[g]-pr[t])
            #print M

        c=0
        cc=0
        collisionkb=[]
        mdistkb=[]
        for g in range(n):
            for t in range(g+1,n):
                if M[g][t]<=mindistkb:
                    c=c+1
                    #print M[g][t], "collision between bins",g+1,"and",t+1,
         #"of the fragment",l+1
                    collisionkb.append([l+1,g+1,t+1,s+1])
                elif M[g][t]>maxdistkb:
                    cc=cc+1
                     #print M[g][t], "max distance violated by bins",g+1,
#"and",t+1,"of the fragment",l+1
                    mdistkb.append([l+1,g+1,t+1,s+1])
        #print collisionkb,mdistkb

        en.append(sum(M[pHMM[s][1]][pHMM[s][2]]*HM[pHMM[s][1]][pHMM[s][2]]
        for s in range(no)))
        if s==0:
            print "the initial energy for the fragment",l+1,"is",en[0]
            inienergy=en[0]

        den=en[s]-inienergy
        pen=math.e**(-den/temp*1.0)
        if (s>0 and c==0 and cc==0):
            if den<=0:
                inienergy=en[s]
                for t in range(n-1):
                    vector[t]=vt[t]
                for t in range(n-2):
                    alpha[t]=al[t]
```

```
                    beta[t]=be[t]
                contacc=contacc+1
                contenergy=0
                if s%100==0:
                    print "energy decreases",den,inienergy, en[s-1]
                stepenkb.append([l+1,en[s],den,s+1])
                fincoordkb=pr
            elif (s==0 or pen>=abs(random.randint(-10,10)/10.0)):
            #elif (s==0 or pen>=0.5):
                inienergy=en[s]
                for t in range(n-1):
                    vector[t]=vt[t]
                for t in range(n-2):
                    alpha[t]=al[t]
                    beta[t]=be[t]
                contacc=contacc+1
                contenergy=0
                if s%100==0:
                    print "probability acceptance",den, inienergy, en[s-1]
                stepenkb.append([l+1,en[s],den,s+1])
                fincoordkb=pr

        s=s+1

    print "the simulation for the fragment",l+1,"stops at the step",
   contiter, "after", contenergy,"steps with stable energy"
    print "the final temperature is", temp, "the final energy is",inienergy

    #print pr
    fname1_template = "coords2.{i}.txt"
    np.savetxt(fname1_template.format(i=l+1), fincoordkb, delimiter='\t',
    newline='\r\n', fmt='%.7f')
    fname2_template = "energy.{i}.txt"
    np.savetxt(fname2_template.format(i=l+1), stepenkb, delimiter='\t',
    newline='\r\n', fmt='%.7f')



def main1():
    global Mcoord
    global Stepen
    l=0
    Mcoord=[]
    Stepen=[]
    for l in range(nbinMb):
```

```
        procedure1(l)
        Mcoord.append(fincoordkb)
        Stepen.append(stepenkb)
        l=l+1

main1()

# BARYCENTERS, PLANAR AND DIHEDRAL ANGLES

b=[]
vect1=[]
vect2=[]
vs1=np.array((0,0,0))
vs2=np.array((0,0,0))
vvect1=np.array((0,0,0))
vvect2=np.array((0,0,0))
vsintor=np.array((0,0,0))
sinsin=0
plan=[]
chi=[]
for l in range(nbinMb):
    b.append([sum(Mcoord[l][s][0] for s in range(len(Mcoord[l])))/
    (len(Mcoord[l])*1.0),sum(Mcoord[l][s][1] for s in range(len(Mcoord[l])))/
    (len(Mcoord[l])*1.0),sum(Mcoord[l][s][2] for s in range(len(Mcoord[l])))/
    (len(Mcoord[l])*1.0)])
    vect1.append(b[l])
    #print b[l],"b",l
    #print vect1[l],"vect1",l
    vect2.append(Mcoord[l][len(Mcoord[l])-1]-b[l])
    plan.append(math.copysign(1,random.randint(-10,10))*
    np.arccos(np.dot(vect1[l],
    vect2[l])/(np.linalg.norm(vect1[l])*np.linalg.norm(vect2[l]))*1.0))
    if l==0:
        chi.append(0)
    else:
        vs1=-Mcoord[l-1][0]+b[l-1]
        vs2=Mcoord[l][len(Mcoord[l])-1]-b[l-1]
        vvect1=np.cross(vs1,vs2)
        vvect2=np.cross(vect1[l],vect2[l])
        vsintor=np.cross(vvect2,vs1)
        sinsin=np.dot(vsintor,vvect1)
        if sinsin>=0:
            chi.append((np.arccos(np.dot(vvect1,vvect2)/(np.linalg.norm(vvect1)*
 np.linalg.norm(vvect2))*1.0)).real)
        else:
```

```
            chi.append(-(np.arccos(np.dot(vvect1,vvect2)/(np.linalg.norm(vvect1)*
 np.linalg.norm(vvect2))*1.0)).real)

    #print vect1[l],plan[l],chi[l],l

# CONTACT MATRIX WITH LOWER RESOLUTION

HMM=np.zeros((nbinMb,nbinMb))
for l in range(nbinMb):
    for s in range(nbinMb):
        HMM[l][s]=sum(HM[n][p] for p in range(int(domin[l][0]-1),int(domin[l][1]))
        for n in range(int(domin[s][0]-1),int(domin[s][1])))


# PROCEDURE 2
def procedure2():

    global fincoordMb
    global stepenMb

    n = nbinMb*2+1
    print "the long chain has", nbinMb ,"bins"
    #print "the long chain has", numcontMb ,"contacts"

    vector=[]
    for s in range(n-1):
        if s%2==0:
            vector.append(np.linalg.norm(vect1[int(s/2)]))
        else:
            vector.append(np.linalg.norm(vect2[int((s-1)/2)]))
    #print vector

    alpha=[]
    beta=[]
    for s in range(n-2):
        if s%2==0:
            alpha.append(plan[int(s/2)])
            beta.append(chi[int(s/2)])
        else:
            alpha.append(0)
            beta.append(0)
    #print alpha,beta

    pHM=[]
    for s in range(int((n-1)/2-1)):
```

```
    for t in range(int((n-1)/2-1)):
        if (t-s)>= diagsMb:
            pHM.append(HMM[1+s,1+t])
pHM=sorted(pHM,reverse=True)
minMb=pHM[numcontMb-1]
#print pHM,len(pHM),minMb

pHMM=[]
for s in range(int(nbinMb-1)):
    for t in range(int(nbinMb-1)):
        if (HMM[1+s,1+t]>=minMb and (t-s)>=diagsMb):
            print "in the long chain there are", int(HMM[1+s,1+t]),
          "contacts between the bin", int(s+2),"and the bin", int(t+2)
            pHMM.append([HMM[1+s,1+t],s+2,t+2])
pHMM.reverse()
no=len(pHMM)

contenergy=0
contiter=0
contacc=0
chitemp=0
gg=0
temp=initemp
s=0
en=[]
stepenMb=[]
fincoordMb=[]

while (contenergy<energytresholdMb and s<=numiterationsMb):

    contenergy=contenergy+1
    contiter=contiter+1
    chitemp=(contacc+1.0)/contiter

    if (contiter==numiterationsTemp and gg==0):
        if chitemp<(1-epsilonchi):
            print "after", contiter, "iterations, the rate between
           transitions accepted and proposed is", chitemp
            contiter=0
            contacc=0
            contanergy=0
            temp=temp*incremtemp
            print "the temperature grows up till", temp
        else:
            print "at iteration", s+1, "the rate is", chitemp,"and
```

```
                temperature starts decreasing with the value", temp
                 contiter=0
                 contacc=0
                 contenergy=0
                 gg=1

        if gg==1:
            temp=temp*dimrate

        al=[]
        be=[]

        for t in range(n-2):
            if t%2==0:
                randb=random.randint(-10,10)/100.0
                #b=1/10.0
                al.append(alpha[t])
                if t==0:
                    be.append(beta[t])
                else:
                    be.append(beta[t]+randb)
            else:
                randb=random.randint(-10,10)/100.0
                #b=1/10.0
                if -arad<=alpha[t]+randb<=arad:
                    al.append(alpha[t]+randb)
                else:
                    al.append(alpha[t]-randb)
                be.append(beta[t])
        #print "angles",alpha, beta, al, be

        # PLANAR ANGLES
        alp=[]
        p_al=[]
        dtpl=[]
        dtpl.append(0)
        dtpl.append(0)
        p_al.append(np.array((0,0,0)))
        p_al.append(np.array((vector[0],0,0)))
        for t in range(2,n):
            alp.append(sum((al[x]) for x in range(t-1)))
            p_al.append(np.array((p_al[t-1][0]+(math.cos(alp[t-2])*
vector[t-1]),p_al[t-1][1]+(math.sin(alp[t-2])*vector[t-1]),0)))
            dtpl.append([LA.norm(p_al[t]-p_al[t-1])])
            #print alp[t-2],"alp",p_al[t],"pal",dtpl[t],"dist",vector[t-1],t
```

```
        # DIHEDRAL ANGLES
        pr=[]
        pr.append(p_al[0])
        pr.append(p_al[1])
        qua=[]
        qua.append(Quaternion())
        qua.append(Quaternion())
        q1=[]
        q1.append(Quaternion())
        q1.append(Quaternion())
        v=[]
        v.append(np.array((1,0,0)))
        v1=[]
        v1.append(np.array((1,0,0)))
        for t in range(1,n-1):
            v.append((p_al[t+1]-p_al[t])/((vector[t])*1.0))
            v1.append((p_al[t+1]-p_al[t]))
            #print v[t],t
        dtdi=[]
        dtdi.append(0)
        dtdi.append([LA.norm(pr[1]-pr[0])])
        for t in range(2,n):
            qua.append(Quaternion.new_rotate_axis(be[t-2],
Vector3(v[t-2][0],v[t-2][1],v[t-2][2])))
            q1.append(q1[t-1]*qua[t].normalise())
            pr.append(q1[t]*Vector3(v1[t-1][0],v1[t-1][1],v1[t-1][2])+pr[t-1])
            dtdi.append([LA.norm(pr[t]-pr[t-1])])
            #print v1[t-1],"v1",pr[t],"pr",t
            #print dtdi[t-2],t-2
        #print pr

        # DISTANCE MATRIX
        M=np.zeros((n,n))
        for g in range(n):
            for t in range(n):
                M[g][t]=LA.norm(pr[g]-pr[t])
            #print M

        if s==0:
            Mmin=[]
            for g in range(n):
                for t in range(g+1,n):
                    Mmin.append(M[g][t])
```

```
            if min(Mmin)<=300:
                mindistMb=min(Mmin)-1
            else:
                mindistMb=300
            print mindistMb,"collision distance between bins of long chain"

        c=0
        cc=0
        collisionMb=[]
        #mdistMb=[]
        for g in range(n):
            for t in range(g+1,n):
                if M[g][t]<=mindistMb:
                    c=c+1
                     #print M[g][t], "collision between bins",g+1,"and",t+1,
#"of the long chain
                    collisionMb.append([l+1,g+1,t+1,s+1])
                #elif M[g][t]>maxdistMb:
                     #cc=cc+1
                     #print M[g][t], "max distance violated by bins",g+1,"and",
#t+1,"of the long chain
                     #mdistMb.append([l+1,g+1,t+1,s+1])
        #print collisionMb,mdistMb

        en.append(sum(M[pHMM[t][1]*2-1][pHMM[t][2]*2-1]*HMM[pHMM[t][1]-1]
        [pHMM[t][2]-1] for t in range(no)))
        if s==0:
            print "initial energy for the long chain is",en[0]
            inienergy=en[0]

        den=en[s]-inienergy
        pen=math.e**(-den/temp*1.0)
        if (s>0 and c==0 and cc==0):
            if den<=0:
                inienergy=en[s]
                for t in range(n-2):
                    alpha[t]=al[t]
                    beta[t]=be[t]
                contacc=contacc+1
                #print "energy decreases",contacc,contiter,den
                contenergy=0
                stepenMb.append([l+1,en[s],den,s+1])
                fincoordMb=pr
            elif (s==0 or pen>=abs(random.randint(-10,10)/10.0)):
            #elif (s==0 or pen>=0.5):
```

```
                inienergy=en[s]
                for t in range(n-2):
                    alpha[t]=al[t]
                    beta[t]=be[t]
                contacc=contacc+1
                #print "probability acceptance",contacc,contiter,den
                contenergy=0
                stepenMb.append([l+1,en[s],den,s+1])
                fincoordMb=pr

        s=s+1

    print "the simulation for the long chain stops at the step", contiter,
  "after", contenergy,"steps with stable energy"
    print "the final temperature is", temp, "the final energy is",inienergy

    #print pr
    np.savetxt("finalcoords2.txt", fincoordMb, delimiter='\t',
    newline='\r\n', fmt='%.7f')
    np.savetxt("finalenergy.txt", stepenMb, delimiter='\t',
    newline='\r\n', fmt='%.7f')

procedure2()

# CALCULATION OF COORDINATES OF 100Kb BIN

pfin=[]
for l in range(nbinMb):
    nb=int(domin[l,1]-domin[l,0]+1)
    if l==0:
        vb0=b[0]
        vf0=Mcoord[0][nb-1]-b[0]
        cpv0=np.cross(vb0,vf0)
        wb0=fincoordMb[1]
        wf0=fincoordMb[2]-fincoordMb[1]
        cpw0=np.cross(wb0,wf0)
        Mv=np.matrix([vb0,vf0,cpv0])
        Mvi=np.matrix(LA.inv(Mv))
        Mw=np.matrix([wb0,wf0,cpw0])
        L=Mvi*Mw
        for t in range(nb):
            pfin.append([np.array(Mcoord[0][t]).dot(np.array(L))])
    else:
        vb0=b[l]
        vf0=Mcoord[l][nb-1]-b[l]
```

```
        cpv0=np.cross(vb0,vf0)
        wb0=fincoordMb[l*2+1]-fincoordMb[l*2]
        wf0=fincoordMb[l*2+2]-fincoordMb[l*2+1]
        cpw0=np.cross(wb0,wf0)
        Mv=np.matrix([vb0,vf0,cpv0])
        Mvi=np.matrix(LA.inv(Mv))
        Mw=np.matrix([wb0,wf0,cpw0])
        L=Mvi*Mw
        for t in range(1,nb):
                pfin.append([np.array(Mcoord[l][t]).dot(np.array(L))+
     pfin[int(domin[l-1,1]-1)]][0])

Mfin=[]
for l in range(len(HM)):
    #print Vector3(pfin[l][0][0],pfin[l][0][1],pfin[l][0][2]),l
    Mfin.append(Vector3(pfin[l][0][0],pfin[l][0][1],pfin[l][0][2]))

for l in range(len(HM)-1):
    print LA.norm(Vector3(pfin[l+1][0][0],pfin[l+1][0][1],pfin[l+1][0][2])-
    Vector3(pfin[l][0][0],pfin[l][0][1],pfin[l][0][2])), l


np.savetxt("Totalfinalcoords.txt", Mfin, delimiter='\t', newline='\r\n', fmt='%.7f')

print "The simulation stopped. A possible conformation of chromatin,
         starting from contact matrix data, is reached."

# PLOT OF FINAL CONFIGURATION

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for c,m,ms in [('b','o',400)]:
    for i in range(len(HM)):
        xs=Mfin[i][0]
        ys=Mfin[i][1]
        zs=Mfin[i][2]
        ax.scatter(xs, ys, zs, c=c, marker=m, s=ms)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

plt.show()
```