Unsupervised Learning Applied to Progressive Compression of Time-Dependent Geometry

Thomas Baby, Youngmin Kim, and Amitabh Varshney *

Department of Computer Science

University of Maryland

College Park

MD 20742, USA

Abstract

We propose a new approach to progressively compress time-dependent geometry. Our approach exploits correlations in motion vectors to achieve better compression. We use unsupervised learning techniques to detect good clusters of motion vectors. For each detected cluster, we build a hierarchy of motion vectors using pairwise agglomerative clustering, and succinctly encode the hierarchy using entropy encoding. We demonstrate our approach on a client-server system that we have built for downloading time-dependent geometry.

Key words: Distributed/Network graphics, Pattern Recognition, Clustering Algorithms

* Corresponding author. Fax. +1-301-405-6707 Email. varshney@cs.umd.edu

1 Introduction

In recent years, there has been a significant growth in e-commerce and entertainment over the Internet. Triangulated 3D geometric models are starting to appear on the World Wide Web as virtual shopping malls and virtual games become more common. Currently, most of the geometric data are static, i.e., they do not vary with time. However, if one were to look at the 2D world, where images (static 2D data) were followed by video, one can expect time-dependent geometry to become a more common form of data on the Internet of the future. Time-dependent geometry arises in simulations of many naturally occurring phenomena, e.g., water waves, plant growth, molecular dynamics, cloth animation, etc. In this paper, we describe how to *compress time-dependent geometry* for the purpose of transmission over the World Wide Web.

There has been a great deal of research on compressing static geometry [3, 7, 12, 18, 22, 23], but very little work has been done on compressing time-dependent 3D geometric data. The work by Lengyel [15] compresses dynamic meshes by solving for few-parameter deformation models and encoding the residuals. The prototype system considered in their work uses affine transform as the deformation model. However, the problem of determining the class of deformation for a vertex is still open, thus limiting the applicability of their algorithm. In our approach, we view motion as simple translation. In many examples of dynamic models (e.g., water, molecular simulation), the object is non-rigid and fluid. The components that comprise the object (e.g., atoms in a molecule, particles simulating waves on water) move in a seemingly independent manner, yet create concerted effects in the object. In such particle-like systems [17], it is sufficient to model motion as translation. In this paper, our

focus is on compressing the motion vectors.

Particles in a dynamic object move in such a way as to create concerted effects in the object. Atoms in a protein may move with different velocities, but their concerted movement can create a channel in the center of the protein [13]. These concerted movements typically imply interesting patterns in the translation vectors and hence low entropy (information content). This observation suggests that one should be able to achieve good compression by exploiting these patterns.

Since motion vectors lack connectivity, connectivity-driven compression algorithms cannot be used to compress them. The geometry-driven compression algorithm by Gandoin and Devillers [10] does not take advantage of motion correlations. Compression algorithms based on building a minimum Hamiltonian path through the motion vectors also do not take advantage of motion correlations. In this paper, we describe the step we have taken towards exploiting motion correlations for good compression of motion vectors. We use ideas from unsupervised learning theory to find good clusters of motion vectors. By encoding vectors within each cluster in its own local coordinate system, we achieve de-correlation of motion vectors. The main contributions of our paper are the following:

1. We show that good compression can be achieved by exploiting motion correlations.

2. We show that good clusters can be found using the expectation maximization algorithm from unsupervised learning theory.

3. We demonstrate a progressive compression scheme for motion vectors suitable for progressive downloads.

4. We demonstrate a system that supports progressive download of dynamic, geometric data at varying precisions.

2 Related Work

Compression of static geometry has been an active area of research among computer graphics researchers since the article by Deering on generalized triangle strips [7]. Most algorithms are connectivity-driven, i.e., they compress the connectivity of a 3D model first and encode the geometry (vertex coordinates) in terms of the connectivity [3, 12, 18, 22, 23]. Connectivity determines the order of enumeration of vertices and provides useful hints for coordinate prediction. Geometry compression is achieved by encoding residuals after coordinate prediction.

Progressive compression schemes create a sequence of meshes of increasing detail and attempt to encode the refinement from one mesh to its successor in the sequence using few bits [2, 4, 6, 10, 14, 16, 21]. Naturally, these schemes have their origins in surface simplification algorithms. The various schemes differ in their compression rates, quality of intermediate meshes, and the fineness of the granularity allowed in each refinement step. Some schemes achieve good compression rates by sacrificing the quality of the intermediate meshes [6, 10] while some others sacrifice the granularity allowed by each refinement [6, 16, 21]. The algorithm by [2] allows fine grained refinements and

produces good quality intermediate meshes, while maintaining good compression ratios, compressing the connectivity of nearly manifold meshes to 3.7 bits per vertex on average. The best single-rate algorithm compresses connectivity information to around 2 bits per vertex [23].

Work by Lengyel [15] on compressing time-dependent geometry classifies vertices of a mesh into clusters — a vertex is classified based on the class of deformation its neighborhood undergoes over time and the parameters of the deformation. Vertices that belong to the same cluster are encoded in local coordinates. The deformation model and the model parameters of a cluster are used to predict the position at a given time of a vertex that belongs to the cluster, and the residual is encoded.

Alexa *et al.* [1] represent time-dependent geometry based on principal components analysis. They use the singular value decomposition (SVD) technique to find principal components of the animation sequence which has temporal and spatial coherence across the key frames. They can achieve 1:10 to 1:100 compression ratio with hardly any visual degradation. Since the order of base objects computed by SVD naturally determines the progressive animation compression, their work also suggests a level-of-detail formulation for time-dependent geometry. Shamir and Pascucci [19] have presented a multi-resolution approach for time-dependent meshes. Their approach analyzes the input sequence of meshes and separates the low-frequency global affine transformations in the temporal domain from the high frequency local vertex deformations. This idea of separable approximations in frequency domain allows their approach to efficiently reconstruct higher detail representations from coarse representations in both space and time.

3 Clustering Motion Vectors

As mentioned earlier, particles in a dynamic object move to create concerted effects in the object, i.e., their motion vectors exhibit correlations. In Figure 1, we show two orthogonal views of motion vectors of atoms in a molecular simulation, where each motion vector is plotted as a point. These motion vectors were obtained from biological experiments on the opening of a channel in an *E. Coli* bacterium. In this example, the vectors form a five-sided structure resembling a badminton shuttlecock. Detecting the five faces enables one to align them along their principal directions, which has the effect of de-correlating them, thus lowering their entropy and improving their compressibility.

Our algorithm for compressing motion vectors is as follows. First, we detect clusters using finite mixture models (Sections 3.1 and 3.2). Then, we transform each motion vector to a normalized coordinate using the principal components of its cluster. Finally, we encode the normalized motion vectors hierarchically (Section 4).

3.1 Finite Mixture Models

Clustering or unsupervised learning is a well-known problem in statistical pattern recognition [8]. Finite mixtures offer a formal, probabilistic model-based approach to cluster statistical observations. Observations are assumed to have been produced by one of a set of R alternative random sources, where each source is a probability density function and the ith source has probability p_i of being picked. Gaussian mixtures, which are widely used in pattern recognition, computer vision, signal and image analysis, and machine learning, are

examples of finite mixtures. In a Gaussian mixture, each random source $\mathbf{c}_i$ is assumed to be a Gaussian distribution with mean μ_i and covariance Σ_i.

We use Gaussian mixture models to cluster motion vectors (i.e., our observations). Identifying which source produced each of the observations leads to a clustering of the observations. Consider a density function $p(\mathbf{x}|\Theta)$ that is governed by a set of parameters Θ (e.g., in a Gaussian mixture, Θ would be the probabilities (p_i), means (μ_i) and covariances (Σ_i) of the Gaussian sources). Assume that we have a set of N observations drawn from the distribution p, i.e., $\mathcal{X} = \{\mathbf{x}_1, ..., \mathbf{x}_N\}$, where the observations are independently and identically distributed with distribution p. The probability of observing the data set is

$$p(\mathcal{X}|\Theta) = \prod_{i=1}^{N} p(\mathbf{x}_i|\Theta) = \mathcal{L}(\Theta|\mathcal{X}) \quad (1)$$

The function $\mathcal{L}(\Theta|\mathcal{X})$ is called the likelihood of the parameters given the data. The standard algorithm for fitting finite mixture models to a set of observations is the Expectation-Maximization (EM) algorithm. It finds the Θ that maximizes $\mathcal{L}$, provided it does not converge to a local maxima due to improper initialization.

The output of a model fitting algorithm such as EM is a set of R source parameters (p_i, μ_i, and Σ_i), which can be plugged in to Bayes' rule to compute the conditional probability of source $\mathbf{c}_j$ given observation $\mathbf{x}_i$. We assign each observation to the source that has the greatest conditional probability given the observation.

3.2 Expectation-Maximization Algorithm

As mentioned before, Expectation-Maximization (EM) [9], also known as "Fuzzy k-means", is a popular algorithm to find a good estimate of the parameters of a mixture model. The estimate returned by EM is the one that maximizes the likelihood (See Equation (1)) of the parameters given the data.

EM is an iterative algorithm. The tth iteration begins with an estimate Θ^t of the parameters of the model, and ends with an improved estimate Θ^{t+1}. In each iteration, EM computes the extent to which a data point $\mathbf{x}_i$ belongs to the cluster associated with source $\mathbf{c}_j$, which is given by $w_{ij} = P(\mathbf{c}_j|\mathbf{x}_i, \Theta^t)$.

$$\begin{aligned} a_{ij} &= P(\mathbf{x}_i|\mathbf{c}_j, \Theta^t) = (2\pi \parallel \Sigma_j \parallel)^{-\frac{1}{2}} e^{-\frac{1}{2}(\mathbf{x}_i-\mu_j)^T \Sigma_j^{-1}(\mathbf{x}_i-\mu_j)} \\ w_{ij} &= P(\mathbf{c}_j|\mathbf{x}_i, \Theta^t) = a_{ij}p_j / \Sigma_{k=1}^{R} a_{ik}p_k \text{ (Bayes' Rule)} \end{aligned}$$

In the new model Θ^{t+1}, the mean μ_j of the jth source is the weighted mean of all data values, where the weights are $\{w_{1j}, w_{2j}, ...w_{Nj}\}$. Similarly, we compute new estimates for source probabilities p_j and source Gaussian covariances Σ_j as follows:

$$\begin{aligned} p_j &\leftarrow \frac{\Sigma_{i=1}^{N} w_{ij}}{N}, \\ \mu_j &\leftarrow \frac{1}{\Sigma_{i=1}^{N} w_{ij}} \sum_{i=1}^{N} w_{ij}\mathbf{x}_i, \\ \Sigma_j &\leftarrow \frac{1}{\Sigma_{i=1}^{N} w_{ij}} \sum_{i=1}^{N} w_{ij}(\mathbf{x}_i - \mu_j)(\mathbf{x}_i - \mu_j)^T \end{aligned}$$

As mentioned before, EM needs proper initialization so that it does not converge to a local maxima. We provide the clustering results of a simple k-means process as the initial values of the EM algorithm. In Figure 2, we show the

k-means-generated clusters of the motion vectors shown in Figure 1. We run EM until the change in each p_j between successive iterations is less than a user-specified threshold. In Figure 3, we show the EM-generated clusters of the motion vectors shown in Figure 1.

4 Progressive Compression of Motion Vectors

Recall that the goal of our work is the compression of time-dependent geometry for the purpose of transmission. Therefore, as mentioned in Section 1, we compress motion vectors in a progressive manner, which enables their transmission from coarse to fine detail. The various steps of our algorithm are shown in Figure 4. The EM algorithm, described in the previous section, is the first step of our algorithm. It outputs the most likelihood estimate of the probabilities, means, and covariances of the R random sources. As mentioned earlier, we use these model parameters to assign each observation to a source (or cluster) , i.e., we assign an observation to the source that has the largest conditional probability given the observation ($\underset{j}{\text{argmax}} \; w_{ij}$, using notation from Section 3.2). These assignments determine the clustering. Each cluster identified by EM is called an *EM-cluster*. In this section, we describe the subsequent steps.

Approach Overview Our strategy for compressing motion vectors is to find a good clustering using EM. We then perform Principal Component Analysis (PCA) on each EM-cluster to normalize the motion vectors in the EM-cluster and improve their compressibility. For each EM-cluster, we then build a hierarchy of clusters using the normalized motion vectors by pairwise agglomerative clustering. Each EM-cluster is at the root node of one such hierarchy. We then

simplify this hierarchy to remove redundant information. A careful traversal of this hierarchy is needed to reconstruct the coarse to fine progressive transmission of detail.

4.1 Principal Component Analysis

For each EM-cluster, we perform Principal Component Analysis (PCA) to compute the mean and principal directions of the cluster. PCA is a process of analyzing the distribution of a cloud of points. Specifically, PCA gives us the orientation, location, and the scaling of the best-fitting ellipsoid to a set of points. An easy way to do this is by using the singular value decomposition (SVD) of the covariance matrix [11]. For a set of points $\mathbf{x}_i$ $(i = 1, \ldots, N)$, we can compute their mean ($\mu_{\mathbf{x}}$) and the covariance matrix (M) as follows:

$$\mu_{\mathbf{x}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i$$
$$\mathbf{M} = \frac{1}{N-1} \sum_{i=1}^{N} (\mathbf{x}_i - \mu_{\mathbf{x}}) \cdot (\mathbf{x}_i - \mu_{\mathbf{x}})^T$$

A singular value decomposition of the covariance matrix ($\mathbf{M}$) will result in:

$$\mathbf{M} = \mathbf{V}^T \mathbf{D} \mathbf{V},$$

where the matrix $\mathbf{D}$ is diagonal and $\mathbf{V}$ is a rotation matrix, such that $\mathbf{V}^T = \mathbf{V}^{-1}$. We transform the motion vectors of a cluster into a new coordinate frame whose axes are aligned with the principal directions of the cluster and whose origin is the mean of the cluster. The axes for each cluster of motion vectors in Figure 1 are depicted in Figure 5. This normalization step de-correlates the motion vectors, thereby lowering their information content and increasing

their compressibility. The resulting normalized motion vectors are used in the later stages of the algorithm.

4.2 Pairwise Agglomerative Clustering

Since the goal of our compression is to progressively encode motion vectors, we use the normalized motion vectors in each EM-cluster to build a hierarchy of clusters by *pairwise agglomerative clustering*. The cluster at the root of such a hierarchy is an EM-cluster. At deeper levels of the hierarchy, the clusters become progressively smaller, until at the lowest level, a cluster contains a single normalized motion vector. Such a hierarchy is called a *cluster tree*, and is depicted in Figure 6.

As the name implies, pairwise agglomerative clustering builds the hierarchy in a bottom-up fashion. Initially, each normalized motion vector that belongs to an EM-cluster is in its own cluster. In each step, the two closest clusters are merged to create a new cluster. The process is repeated until a single cluster is left. Each node of the cluster tree represents a cluster. Each non-leaf node has two children, which represent the two clusters whose merge created the non-leaf node's cluster. Each node stores a representative point of the cluster (mean) and the dimensions of the bounding box of the cluster (bbox). The mean of a leaf node is the normalized motion vector stored at the leaf, and the mean of a non-leaf node is the arithmetic mean of the means of its children. The bounding box of a leaf-node (colored green) is a box of zero edge length. The bounding box of a non-leaf node (colored blue) is defined as the smallest box that is centered at its mean and large enough to contain the bounding boxes of its child clusters.

Notation We introduce the following notation for ease of exposition. P_{ct} denotes any node in a cluster tree. $Lt(P_{ct})$ and $Rt(P_{ct})$ denote its left and right children respectively. R_{ct} denotes the root node of a cluster tree. Using the above notation, the mean and bbox for a non-leaf node are computed using the following formulae:

$$P_{ct} \rightarrow mean \;= (Rt(P_{ct}) \rightarrow mean + Lt(P_{ct}) \rightarrow mean)/2$$

$$P_{ct} \rightarrow bbox[i] =$$

$$|Lt(P_{ct}) \rightarrow mean[i] - Rt(P_{ct}) \rightarrow mean[i]|$$

$$+ \max(Lt(P_{ct}) \rightarrow bbox[i], Rt(P_{ct}) \rightarrow bbox[i])$$

Each node stores an identifier (id), which is the value of a monotonically increasing counter at the time of creation of the node. If a cluster is created in the ith iteration of pairwise agglomerative clustering, it is given a larger id value than a cluster created in the $(i-1)^{th}$ iteration. Thus, the root node has the largest value of id. Each node also stores a list of particle identifiers (particle_id_list), which contains the identifiers of particles whose normalized motion vectors belong to the node's cluster. Thus, each leaf-node in the cluster tree stores only a single particle identifier which is the identifier of the particle whose normalized motion vector is stored at the leaf-node. The particle_id_list of a non-leaf node is obtained by appending the particle_id_list of its right child to the particle_id_list of its left child. The particle_id_list of a root node contains identifiers of all particles whose normalized motion vectors belong to the same EM-cluster.

The bbox field at a node helps in deciding which nodes to transmit when

a user requests motion vectors to a specified accuracy. Only nodes whose bounding boxes are larger than the specified accuracy need to be transmitted. Having decided which nodes to transmit, the id field values in those nodes provide a coarse to fine detail ordering of those nodes, consistent with pairwise agglomerative clustering. Nodes need to be transmitted in descending order of id values to obtain a coarse to fine progression. The mean field at a node is the normalized motion vector for all particles in the node's list of particles (particle_id_list).

4.3 Delta Tree

The cluster tree provides a progressive transmission scheme, however the hierarchy can be further simplified. Let us focus on the mean field for the moment. Consider a non-leaf node P_{ct}. Since $P_{ct} \rightarrow mean$ is the arithmetic mean of $Lt(P_{ct}) \rightarrow mean$ and $Rt(P_{ct}) \rightarrow mean$, it is sufficient to store (and transmit) $P_{ct} \rightarrow mean$ and its delta difference from one of the child mean values (e.g., $Rt(P_{ct}) \rightarrow mean - P_{ct} \rightarrow mean$), using which the client is able to reconstruct all three values. By applying this simplification in a recursive manner, we construct a simplified tree, called the *delta tree*, which has the same number of nodes and connectivity as the cluster tree. Thus, each node P_{dt} in the delta tree has a counterpart P_{ct} in the cluster tree. P_{dt} stores a delta difference

vector field (delta) which is defined as follows:

$$P_{dt} \rightarrow delta = \begin{cases} Rt(P_{ct}) \rightarrow mean - P_{ct} \rightarrow mean \\ \text{(if } P_{ct} \text{ is a non-leaf)} \\ \\ \overrightarrow{0} \\ \text{(if } P_{ct} \text{ is a leaf)} \end{cases}$$

Since the leaf nodes in the delta tree have the zero vector as the delta difference vector, they can be deleted with no loss of information. Given the mean value of the root node of the cluster tree, and the delta tree, one can reconstruct all the mean values in the original cluster tree. Figure 7 depicts the delta tree corresponding to the cluster tree in Figure 6.

As shown in Figure 7, each node of the delta tree stores three other fields. The id field of a delta tree node is the same as the id field value of its counterpart node in the cluster tree. The bbox_size field in a delta tree node is the length of the diagonal of the bounding box of the counterpart node in the cluster tree. As before, based on a user-specified accuracy threshold, the value of the bbox_size field is used to determine if a node is to be transmitted. Having selected the delta tree nodes to be transmitted, the id field gives an ordering of them to provide coarse to fine progressive transmission.

Recall that the particle_id_list field in a cluster tree node stores all particles whose normalized motion vectors belong to that node's cluster. Thus, the particle_id_list field in the root node of the cluster tree has all identifiers of

all particles whose motion vectors belong to the same EM-cluster. The order of particle identifiers in the list is determined by the structure of the tree since each non-leaf node simply concatenates the particle_id_list of its right child to that of its left child. We reduce the amount of storage required for particle_id_list by reassigning particle identifiers in the following manner. Let p_i be the particle identifier in the ith position of particle_id_list of a root cluster tree node. We assign to particle identifier p_i the new identifier i. To make this reassignment work in the presence of multiple EM-clusters, we add an offset determined by the EM-cluster to all the new identifiers of that EM-cluster. The advantage of this reassignment is that all particles that fall within a cluster will have contiguous particle identifiers irrespective of the level of the cluster in the cluster tree. Thus, in place of storing a list at each node, we can store two numbers, i.e., the smallest reassigned particle identifier (pid_{small}) and the number of particles in the list (pid_{count}) .

A further reduction in storage is possible by the following observation. Let R_{ct} be the root node of a cluster tree. Let $left_subtree_size$ be the number of elements in $Lt(R_{ct}) \rightarrow particle_id_list$. Note that this number is also equal to the number of leaf nodes in the subtree rooted at $Lt(R_{ct})$. Then, the following formulae hold:

$$
\begin{aligned}
Lt(R_{ct}) &\rightarrow pid_{small} = R_{ct} \rightarrow pid_{small} \\
Rt(R_{ct}) &\rightarrow pid_{small} = R_{ct} \rightarrow pid_{small} + left_subtree_size \\
Lt(R_{ct}) &\rightarrow pid_{count} = left_subtree_size \\
Rt(R_{ct}) &\rightarrow pid_{count} = R_{ct} \rightarrow pid_{count} - left_subtree_size
\end{aligned}
$$

Thus, if we knew pid_{small} and pid_{count} for R_{ct}, then from the single number $left_subtree_size$, we can obtain pid_{small} and pid_{count} values for both its children. Applying this idea recursively, we see that at each node P_{ct}, we need

to store only a single number, namely the number of leaf nodes in the subtree rooted at $Lt(P_{ct})$. Given the values of pid_{small} and pid_{count} for the root node of a cluster tree, and the $left_subtree_size$ values at each node, we can reconstruct all particle lists. This simplification is reflected in the delta tree as well. The field $left_subtree_size$ in a delta tree node P_{dt} is the value of $left_subtree_size$ in its counterpart node P_{ct} in the cluster tree.

The advantage of the delta tree is that it is significantly more lightweight than its corresponding cluster tree. Yet, given $mean$, pid_{small}, and pid_{count} of the root node of a cluster tree, we can reconstruct the original cluster tree from the delta tree. Recall that our algorithm constructs as many delta trees as the number of EM-clusters.

4.4 Linearization

Now, we describe the algorithm for selecting and ordering nodes of a single delta tree for transmission. We initialize a max-heap with the root node of the delta tree. In each iteration, we extract the element in the heap with the maximum value of id. If this delta tree node has a bbox_size value greater than a user-specified threshold accuracy, we append this node to the output buffer and insert its children (if they exist) to the heap. If, on the other hand, the bbox_size value is smaller or equal to the user-specified accuracy, we append the node to the output buffer, but do not insert its children to the heap. This algorithm ensures that delta tree nodes are transmitted in a coarse to fine progression of detail.

4.4.1 Interpretation of Transmitted Data

Recall that in order to reconstruct the normalized motion vectors of a single EM-cluster, we need its delta tree, $R_{ct} \rightarrow mean$, $R_{ct} \rightarrow pid_{small}$, and $R_{ct} \rightarrow pid_{count}$, where R_{ct} is the root node of the corresponding cluster tree. Initially, when information from no delta tree node is available, $R_{ct} \rightarrow mean$ is a representative for a cluster of normalized motion vectors, namely, the normalized motion vectors that belong to the EM-cluster. All particles whose normalized motion vectors belong to the EM-cluster use this representative as a coarse approximation of their motion. These particles have reassigned particle identifiers in the range $[R_{ct} \rightarrow pid_{small}, R_{ct} \rightarrow pid_{small} + R_{ct} \rightarrow pid_{count} - 1]$.

Notation $\mathcal{R}(i, j)$ denotes the set of particles whose reassigned particle identifiers falls in the range $[i, i + j - 1]$. $\mathcal{C}(i, j)$ denotes the cluster of normalized motion vectors of particles in $\mathcal{R}(i, j)$. $\mathcal{R}EP(i, j)$ denotes the representative normalized motion vector for the cluster $\mathcal{C}(i, j)$. Using this notation, we can say the following:

$$\mathcal{R}EP(R_{ct} \rightarrow pid_{small}, R_{ct} \rightarrow pid_{count}) = R_{ct} \rightarrow mean$$

In general, after information from the first n nodes $(P_{dt}^0, P_{dt}^1, \cdots, P_{dt}^{n-1})$ of a linearized delta tree has been received, the normalized motion vectors within its EM-cluster fall into $n + 1$ clusters $\{\mathcal{C}(i_0, j_0), \mathcal{C}(i_1, j_1), \cdots \mathcal{C}(i_n, j_n)\}$. Each cluster $\mathcal{C}(i_k, j_k)$ has a corresponding representative $\mathcal{R}EP(i_k, j_k)$ and set of particles $\mathcal{R}(i_k, j_k)$. When information from the $(n + 1)^{th}$ delta tree node P_{dt}^n is received, one of the $n + 1$ clusters, say $\mathcal{C}(i_k, j_k)$, is split into two smaller clusters and its corresponding set of particles $\mathcal{R}(i_k, j_k)$ is subdivided into two subsets. This process is repeated until all delta nodes are processed. Thus, the

information in a delta tree node encodes a split operation. In column 1 of Table 1, we give the formulae for the clusters formed by splitting cluster $\mathcal{C}(i_k, j_k)$ using the delta node P_{dt}^n. In column 2, we give the formulae for computing the corresponding cluster representatives.

Although the information in each node of the linearized delta tree contains information on how to perform a cluster split, it does not tell us which of the existing clusters to split. This information is present in the parent-child relationships of the delta tree, but is lost during linearization. Therefore, along with $delta$ and $left_subtree_size$ values for each delta tree node, we also need to send pid_{small}. The cluster $\mathcal{C}(i_k, j_k)$ for which i_k equals pid_{small} is the one that has to be split. pid_{small} can be generated for each delta tree node during linearization by applying the following formulae recursively:

$$R_{dt} \rightarrow pid_{small} = R_{ct} \rightarrow pid_{small} \ (Root\ Node)$$

$$Lt(P_{dt}) \rightarrow pid_{small} = P_{dt} \rightarrow pid_{small}$$

$$Rt(P_{dt}) \rightarrow pid_{small} = P_{dt} \rightarrow pid_{small} + P_{dt} \rightarrow left_subtree_size$$

The id and bbox_size fields are used for the selection and ordering of nodes for transmission. They are not transmitted to the client.

4.5 Interleaving and Encoding

We have described how motion vectors within a single EM-cluster are normalized, hierarchically clustered, and the delta tree extracted and linearized.

However, when multiple EM-clusters are present, we interleave the information from the delta tree nodes of the different delta trees before transmission. Thus the root nodes of the different delta trees have their information sent first, before information from any other nodes are sent. Huffman encoding is done on this interleaved array.

4.6 Retrieving Motion Vectors

$R_{ct} \rightarrow mean$ is a coarse approximation to the normalized motion vector for all particles in an EM-cluster. As more delta nodes are received, the approximation becomes more accurate. The original motion vectors can be computed from the normalized motion vectors of an EM-cluster by a coordinate transformation using the Gaussian mean and principal directions of the EM-cluster.

5 Handling Multiple Frames

In the discussion so far, we have concentrated on the progressive compression of motion vectors, without mentioning how they are obtained. In general, motion vectors can be computed as the difference in particle positions in the ith and $(i+k)^{th}$ frames. When $k = 1$, the resulting motion vectors represent particle motion in fine detail. As k is increased, the resulting motion vectors are coarse approximations of particle motion. In our approach, we build a hierarchy of coarse to fine approximations by constraining k to be of the form 2^l for some non-negative integer l. Each value of l defines an approximation level as shown in Figure 8. At a given level l, motion vectors are computed between frames $m * 2^l$ and $(m+1) * 2^l$ only, for $m \geq 0$. Our approach allows a

user to mix motion vectors from different approximation levels, allowing fine detailed motion in the region of interest and coarse approximations elsewhere, as illustrated by the blue bars in Figure 8.

6 Progressive Transmission of Time-Dependent Geometry

We now describe a system that we have built for the progressive download of time-dependent geometry. In Figure 9, we show the user interface at the client. The vertical bar can be used to control how motion vectors are to approximate the motion. The intervals along this bar indicate how motion vectors are to be computed. The frames that are the end points of an interval are used to compute the motion vectors for that interval. For example, in Figure 9, motion is approximated using three sets of motion vectors. Frames 0 and 2 are used to compute the first set, frames 2 and 3 to compute the second, and frames 3 and 4 to compute the third. By clicking on the bottom end (i.e., the red square) of an interval on this bar, a user can control the length of the interval, which controls the approximation level (Section 5).

The horizontal bar in Figure 9 is a slider, using which a user can specify a *threshold accuracy* for the current interval (or the current set of motion vectors), which is denoted by a black arrowhead at its bottom end (the interval $[0, 2]$ in Figure 9). The threshold accuracy is used to determine which delta tree nodes are to be sent to the client.

A typical user interaction consists of the following steps. First, the vertical bar is used to specify how motion is to be approximated across frames. Then, for each interval, the horizontal slider is used to specify a threshold accuracy.

Finally, time-dependent geometry can be viewed using the specified parameters. The specified parameters can be modified at any time to dynamically adjust the display of the time-dependent geometry.

7 Results

In this section, we present the results of running our algorithm on data obtained from a molecular dynamics simulation. The simulation consists of the opening of a channel in an Escherichia coli (E. Coli) bacterium molecule [20]. The data consists of five frames (Frames $0-4$), and each frame consists of the 3D coordinates of 10585 atoms (Figure 10).

With each of the $x-$, $y-$, $z-$ coordinates quantized to 16 bits, our algorithm compresses the 10585 motion vectors between frames 0 and 1 to 35.82 bits per atom. The algorithm by Gandoin *et al.* [10] compresses the same data to 40.48 bits per atom. Our algorithm achieves superior compression rates because atoms exhibit correlations in their motion. We expect similar improvements to hold for motion vectors between other pairs of frames too, since atoms move in a correlated fashion between every successive pair of frames. Our results are superior for other quantization levels too, as seen in Table 2.

Since we use a vector quantization scheme for each frame independently, errors can accumulate. This is a known problem in vector quantization techniques. To address the problem of error accumulation, we encode frame i with respect to a decompressed frame $i-1$. In other words, the motion vectors between frame $i-1$ and frame i are the difference between the real data at frame i and the decompressed data at frame $i-1$ [5]. This is shown in Table 3.

The encoding as a whole took 248 seconds per frame on a Pentium IV 1.5GHz processor as follows: 4.5 seconds for initial k-means, 236.5 seconds for the EM Algorithm, and 7 seconds for PCA, Pairwise Agglomerative Clustering, Simplification, and Linearization. The decoding time, however, was never more than 30 milliseconds. Decoding just involves reversing quantization and simple transformations that include a rotation of the principal axes, a scaling for de-normalization, and a translation from the cluster's origin.

The primary goal of our encoding scheme is increasing the compressibility by lowering the entropy of motion vectors. To do this, we need the results of clustering to be as accurate as possible. Our experiments show that the EM algorithm results in better clustering, especially around the boundaries of two (or more) clusters. For example, for the motion vectors between frames 0 and 1, the EM algorithm results in much better clusters (shown in Figure 3) compared to the results of a simple k-means approach shown in Figure 2. This encouraged us to use the EM algorithm notwithstanding a 20-fold increase in encoding time. Usually the decoding time is more critical in most web-based downloading applications. Hence our system with a real-time decoding performance is well-suited for such applications. However, if the encoding time is critical one can just skip the EM and use simple k-means-generated clusters in the later stages of our algorithm.

8 Conclusion

In this paper, we have described an algorithm to compress time-dependent geometry by treating motion as simple translation. We have proposed a scheme that takes advantage of correlations in motion by detecting interesting clusters

using Expectation-Maximization. Currently, our algorithm requires the user to specify the number of clusters. In future work, we plan to automatically determine the best number of clusters. We also plan to extend our work to compress 3D geometric models.

Acknowledgments

We would like to acknowledge the anonymous referees for their careful reviews for this paper that have led to a much better presentation of our results. This work has been supported in part by the NSF grants: IIS 00-81847, CCF 04-29753, and CNS 04-03313.

References

[1] M. Alexa and W. Müller. Representing animations by principal components. *Computer Graphics Forum*, 19(3):411–418, 2000.

[2] P. Alliez and M. Desbrun. Progressive compression for lossless transmission of triangle meshes. *ACM Transactions on Graphics*, pages 195–202, 2001. SIGGRAPH 2001 Conference Proceedings.

[3] P. Alliez and M. Desbrun. Valence-driven connectivity encoding for 3d meshes. In *Eurographics 2001 Conference Proceedings*, pages 480–489, 2001.

[4] C. L. Bajaj, V. Pascucci, and G. Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *IEEE Visualization '99 Conference Proceedings*, pages 307–316, 1999.

[5] P. H. Chou and T. H. Meng. Vertex data compression through vector

quantization. *IEEE Transactions on Visualization and Computer Graphics*, 8(4):373–382, October/December 2002.

[6] D. Cohen-Or, D. Levin, and O. Remez. Progressive compression of arbitrary triangular meshes. In *IEEE Visualization '99 Conference Proceedings*, pages 67–72, 1999.

[7] M. Deering. Geometry compression. *Proceedings of ACM SIGGRAPH 1995*, pages 13–20, 1995.

[8] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, 1973.

[9] M. A. T. Figueiredo and A. K. Jain. Unsupervised learning of finite mixture models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(3):381–396, Mar 2002.

[10] P. Gandoin and O. Devillers. Progressive lossless compression of arbitrary simplicial complexes. *ACM Transactions on Graphics*, 21:372–379, 2002. SIGGRAPH 2002 Conference Proceedings.

[11] G. H. Golub and C. F. Van Loan. *Matrix Computation.* North Oxford Academic, Oxford, England, 1983.

[12] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. *ACM Transactions on Graphics*, pages 133–140, 1998. SIGGRAPH 1998 Conference Proceedings.

[13] H. Huitema and R. Van Liere. Interactive visualization of protein dynamics. *Proceedings of IEEE Visualization 2000*, pages 465–468, oct 2000.

[14] A. Khodakovsky, P. Schröder, and W. Sweldens. Progressive geometry compression. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 271–278. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[15] J. E. Lengyel. Compression of time-dependent geometry. In *Proceedings*

of the 1999 symposium on Interactive 3D graphics, pages 89–95. ACM Press, 1999.

[16] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, / 2000.

[17] W. T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *Computer Graphics*, 17(3):359–376, 1983.

[18] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, /1999.

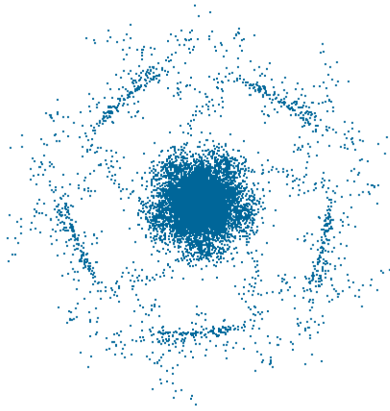
[19] A. Shamir and V. Pascucci. Temporal and spatial level of details for dynamic meshes. In Chris Shaw and Wenping Wang, editors, *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST-01)*, pages 77–84, New York, November 15–17 2001. ACM Press.

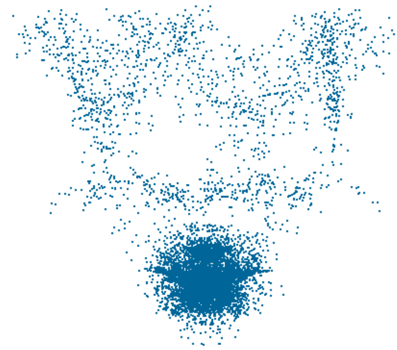
[20] S. Sukharev, S. R. Durell, and H. R. Guy. Structural models of the mscl gating mechanism. *J. Biophys*, 81(2):917–936, 2001.

[21] G. Taubin, A. Gueziec, W. Horn, and F. Lazarus. Progressive forest split compression. *Computer Graphics*, 32(Annual Conference Series):123–132, 1998.

[22] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[23] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface '98 Conference Proceedings*, pages 26–34, 1998.

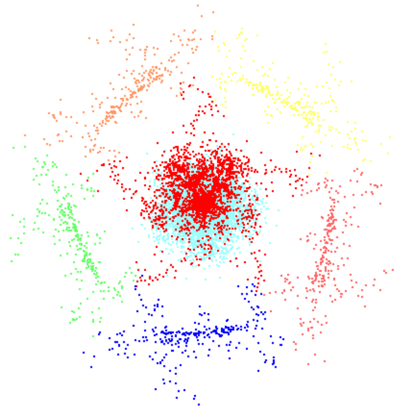




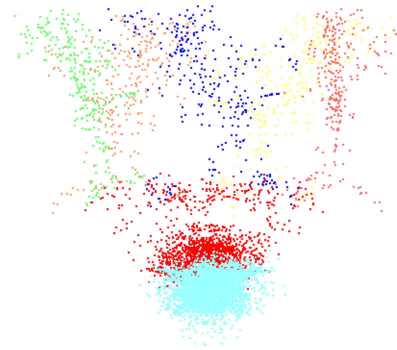
Top View Side View

Fig. 1. Orthogonal views of motion vectors between frames 0 & 1 of a molecular dynamics simulating the opening and closing of the mechanosensitive ion-channel in the E. Coli bacterium's cell membrane

Figure 1
Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney



Top View

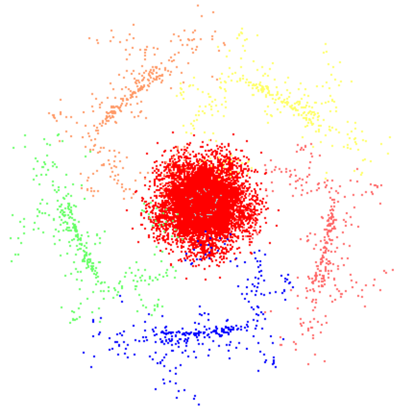


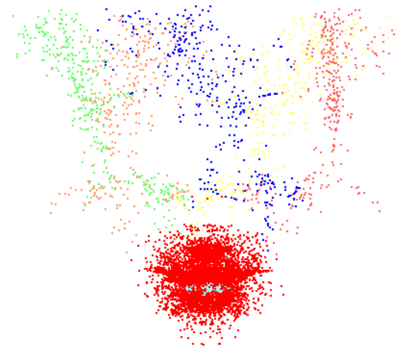
Side View

Fig. 2. Orthogonal views of motion vectors from Figure 1 color-coded by their k-means clusters

Figure 2

Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney





Top View　　　　Side View

Fig. 3. Orthogonal views of motion vectors from Figure 1 color-coded by their EM clusters

Figure 3
Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney

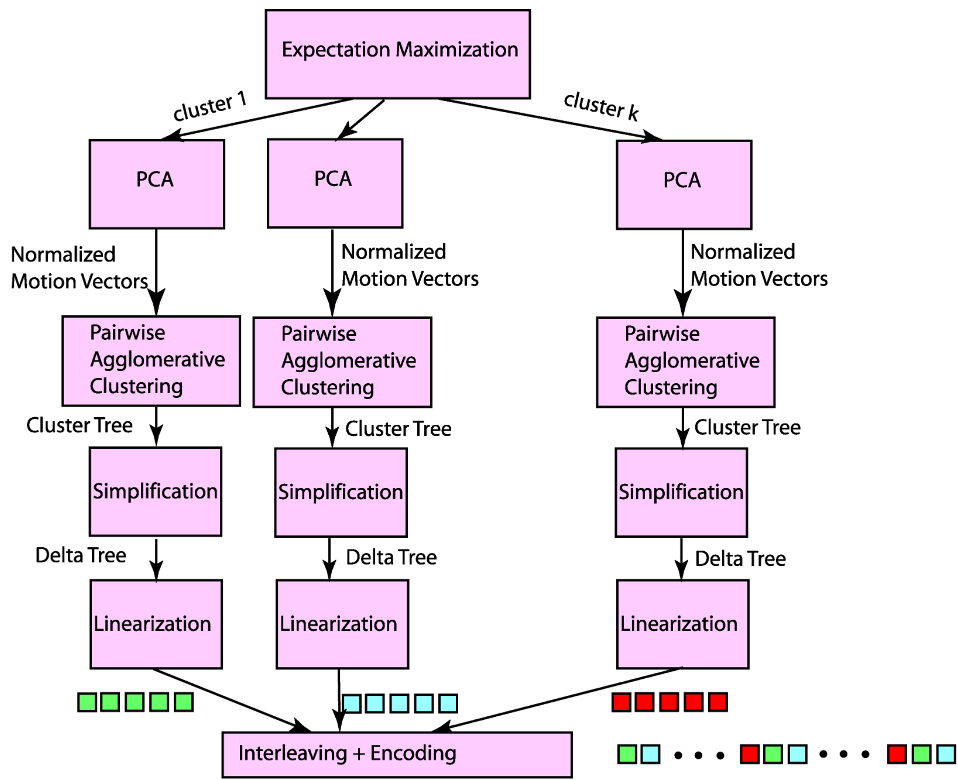
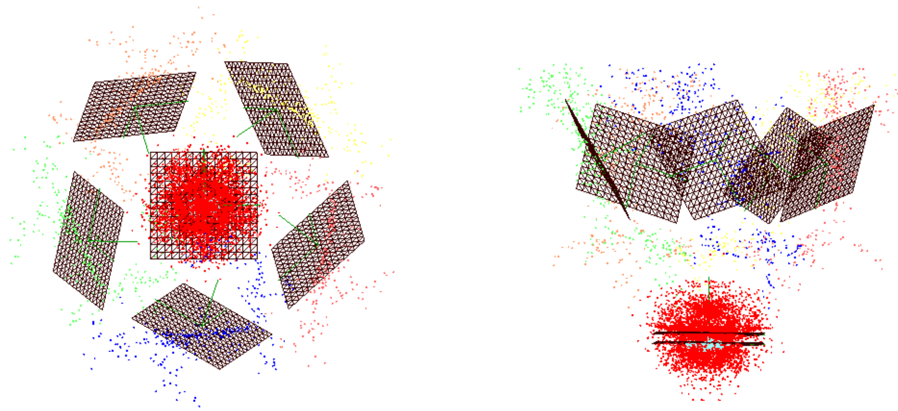


Fig. 4. Overview of our algorithm

Figure 4
Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney



Top View Side View

Fig. 5. Local frames of each cluster centered at their respective means and aligned with their respective principal direction

Figure 5
Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney

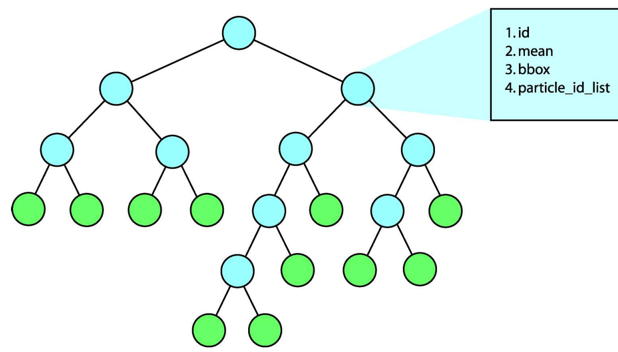


Fig. 6. Cluster Tree

Figure 6
Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney

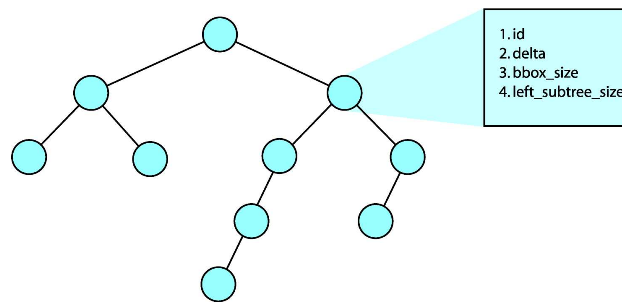


Fig. 7. Delta Tree corresponding to the cluster tree in Figure 6. Note that the green nodes in the cluster tree have no counterpart nodes in the delta tree.

Figure 7

Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney

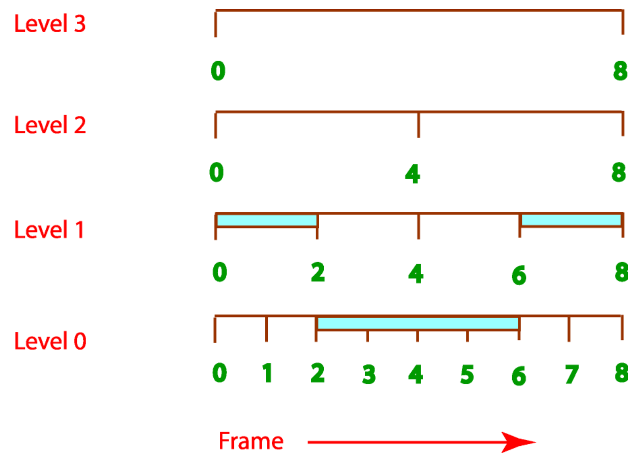


Fig. 8. Hierarchy of approximations to particle motion. The blue bars indicate coarse motion approximations between frames $[0, 2]$ and $[6, 8]$, and fine motion approximation between frames $[2, 6]$.

Figure 8
Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney

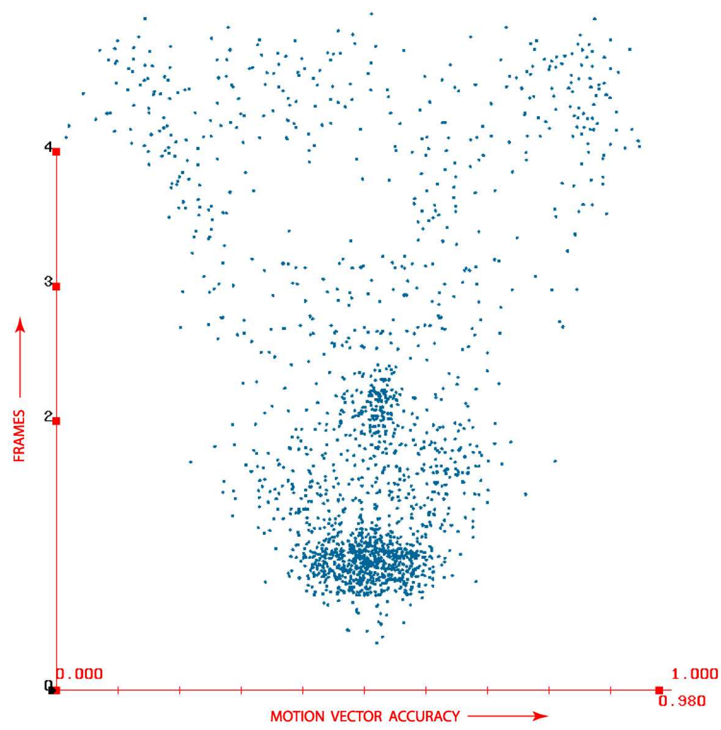


Fig. 9. Interface to our system.

Figure 9
Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney

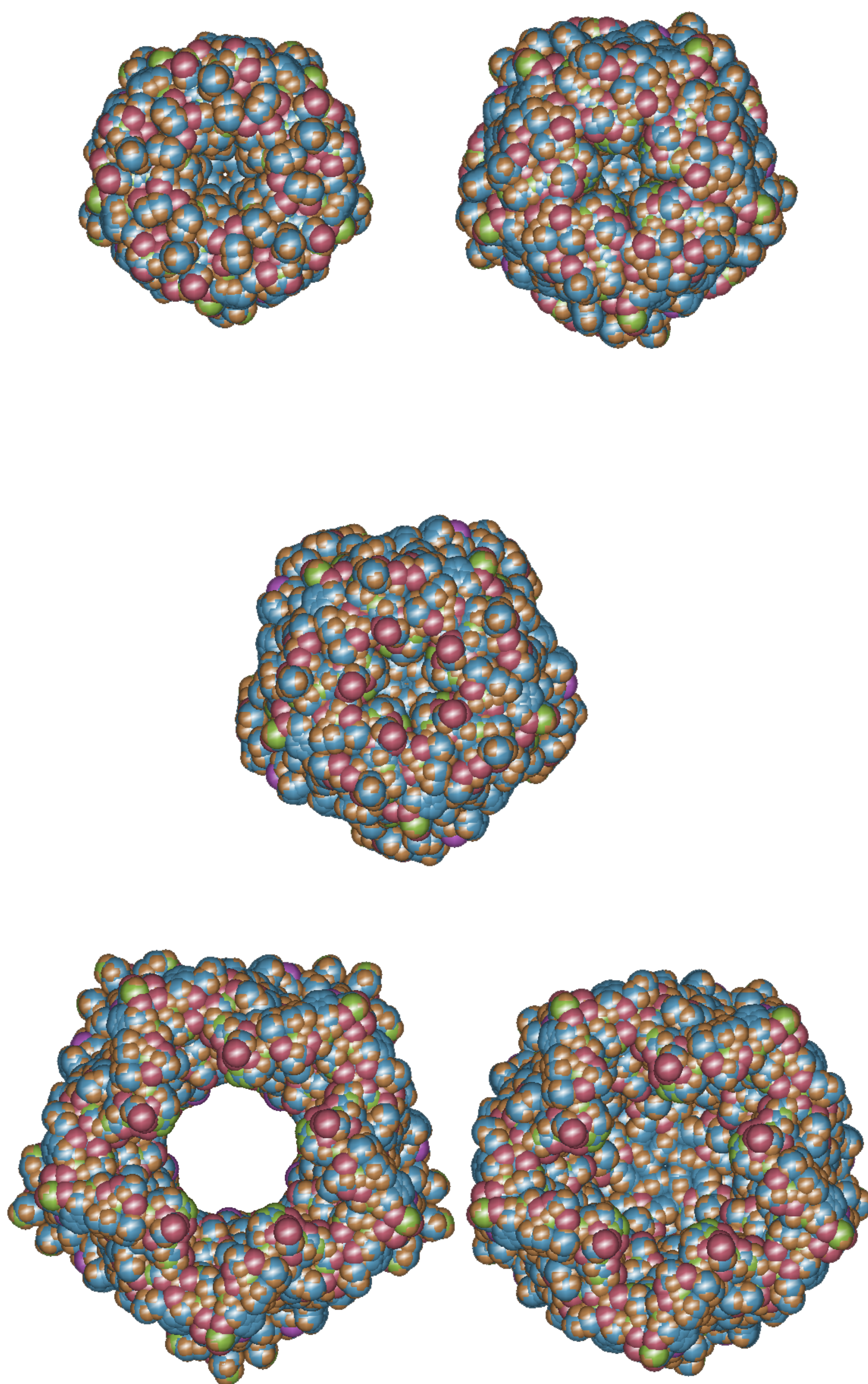


Fig. 10. Frames from the molecular dynamics simulation data (Top Row = Frames 0 & 1, Middle Row = Frame 2, Bottom Row = Frames 3 & 4).

Figure 10
Authors: Thomas Baby, Youngmin Kim, Amitabh Varshney

Figure Captions

Figure 1. **Orthogonal views of motion vectors between frames 0 & 1 of a molecular dynamics simulating the opening and closing of the mechanosensitive ion-channel in the E. Coli bacterium's cell membrane.**

Figure 2. **Orthogonal views of motion vectors from Figure 1 color-coded by their k-means clusters.**

Figure 3. **Orthogonal views of motion vectors from Figure 1 color-coded by their EM clusters.**

Figure 4. **Overview of our algorithm.**

Figure 5. **Local frames of each cluster centered at their respective means and aligned with their respective principal direction.**

Figure 6. **Cluster Tree.**

Figure 7. **Delta Tree corresponding to the cluster tree in Figure 4. Note that the green nodes in the cluster tree have no counterpart nodes in the delta tree.**

Figure 8. **Hierarchy of approximations to particle motion. The blue bars indicate coarse motion approximations between frames** $[0, 2]$ **and** $[6, 8]$**, and fine motion approximation between frames** $[2, 6]$**.**

Figure 9. **Interface to our system.**

Figure 10. **Frames from the molecular dynamics simulation data (Top**

Row = Frames 0 & 1, Middle Row = Frame 2, Bottom Row = Frames 3 & 4).

$\mathcal{C}(i,j)$	$\mathcal{R}EP(i,j)$
$\mathcal{C}(i_k, P_{dt}^n \rightarrow left_subtree_size)$	$\mathcal{R}EP(i_k, j_k) - P_{dt}^n \rightarrow delta$
$\mathcal{C}(i_k + P_{dt}^n \rightarrow left_subtree_size, j_k - P_{dt}^n \rightarrow left_subtree_size)$	$\mathcal{R}EP(i_k, j_k) + P_{dt}^n \rightarrow delta$

Table 1

Formulae for a cluster split

Model (#Particles)	Quantization	Gandoin (bits/particle)	Our Algo. (bits/particle)
Molecular Dynamics Simulation (10585)	12-bit	22.48	19.25
	16-bit	40.48	35.82

Table 2

Compression results on molecular dynamics simulation data in number of bits per particle.

	Errors without Incremental Encoding	Errors with Incremental Encoding
Frame 1	0.0996	0.0996
Frame 2	0.1544	0.1123
Frame 3	0.1819	0.1160
Frame 4	0.1854	0.0869

Table 3

Error Accumulation without and with incremental frame encoding. Each entry shows the root-mean-squared error between the actual and the reconstructed atom positions. The atom positions across all the frames have been normalized to lie in a unit cube.