Lossless Geometry Compression for Floating-Point Data in Steady-State and Time-Varying Fields over Irregular Grids

Dan Chen, Yi-Jen Chiang, *Member, IEEE,* Nasir Memon, *Member, IEEE,* and Xiaolin Wu, *Member, IEEE*

(Revised May 2007)

***Abstract*— In this paper we investigate the problem of lossless geometry compression of irregular-grid volume data represented as a tetrahedral mesh. We propose a novel lossless compression technique that effectively predicts, models, and encodes geometry data for both steady-state (i.e., with only a single time step) and time-varying datasets. Our geometry coder applies to *floating-point* data without requiring an initial quantization step and is *truly lossless*. However, the technique works equally well even if quantization is performed. Moreover, it does *not* need any connectivity information, and can be easily integrated with a class of the best existing connectivity compression techniques for tetrahedral meshes with a small amount of overhead information. We present experimental results which show that our technique achieves superior compression ratios.**

***Index Terms*— graphics compression, lossless geometry compression, irregular grids, steady-state and time-varying fields.**

1. INTRODUCTION

In recent years, new challenges for scientific visualization have emerged as the size of data generated from simulations has grown exponentially [2]. The emerging demand for efficiently storing, transmitting, and visualizing such data in networked environments has motivated research in graphics compression for 3D polygonal models and volumetric datasets. The most general class of volumetric data is *irregular-grid* volume data represented as a *tetrahedral mesh*. It has been proposed as an effective means of representing disparate field data that arise in a broad spectrum of scientific applications.

Although there has been a significant amount of research done on tetrahedral mesh compression, most techniques reported in the literature have mainly focused on compressing *connectivity* information, rather than *geometry* information which consists of *vertex-coordinates* and *data attributes* (such as *scalar values* in our case). As a result, while connectivity compression achieves an impressive compression rate of 1–2 bits per triangle for triangle meshes [38], [34], [1], [39] and 2.04–2.31 bits per tetrahedron for tetrahedral meshes [13], [40], progress made in geometry compression has not been equally impressive. For a tetrahedral mesh, typically about 30 bits per vertex are required after compression [13] (this only includes the x, y, z coordinates with *no* scalar values, where each coordinate is initially quantized to 16 bits, i.e., 48 bits/vertex (b/v) before compression), and we do not know of any reported results on compressing time-varying fields over *irregular grids* (see Section 2). Given that the number of tetrahedra in a tetrahedral mesh is typically about 4.5 times the number of vertices and that connectivity compression results in about 2 bits per tetrahedron, it is clear that geometry compression is the bottleneck in the overall graphics compression. The situation gets worse for time-varying datasets where each vertex can have hundreds or even thousands of time-step scalar values.

The disparity between bit rates needed for representing connectivity and the geometry gets further amplified when *lossless* compression is required. Interestingly, whereas almost all connectivity compression techniques are lossless, geometry compression results in the literature almost always include a quantization step which makes them *lossy* (the only exceptions are the recent techniques of [23], [28]). While lossy compression might be acceptable for usual graphics models to produce an approximate visual effect to "fool the eyes," it is often not acceptable for scientific applications where lossless compression is desired. This is especially true for irregular-grid meshes which represent disparate field data, with more points sampled in regions containing more features. Applying any quantization often results in the collapse between neighboring points which are densely sampled, causing a loss of potentially important features in the data. Moreover, as pointed out in [23], scientists usually do not like their data to be altered in a process outside their control, and hence often avoid using any lossy compression.

In this paper we develop a truly lossless geometry compression techniques for tetrahedral volume data, for both steady-state and time-varying fields. By truly lossless we mean a technique where quantization of vertex coordinates/scalar values is not required. However, the technique works effectively even if quantization is performed. We take a novel direction in that our geometry coder is *independent* of connectivity coder (albeit they can be easily incorporated) and typically re-orders the vertex list *differently* from connectivity-coder traversal. This incurs an additional overhead for recording the permutation of the vertices in the mesh when incorporated with a connectivity coder. Our rationale for taking this direction is twofold. First, since geometry compression is the dominating bottleneck, we want to see how far we can push if the geometry coder is not "burdened" by connectivity compression. Secondly, such a geometry coder is interesting in its own right for compressing datasets not involving any connectivity information, such as *point clouds*, which are becoming increasingly important.

• A conference version of this paper appeared in *Proc. Eurographics/IEEE Symposium on Visualization (EuroVis '06)*, pp. 275–282, May, 2006 ([7]).

• Research supported by NSF Grant CCF-0118915; research of the second author is also supported by NSF CAREER Grant CCF-0093373 and NSF Grant CCF-0541255.

• The authors are with Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201, USA. Author E-mail: dchen@cis.poly.edu; {yjc, memon,xwu}@poly.edu.

It turns out that our geometry coder achieves nice improvements over the state-of-the-art techniques even after paying the extra cost for representing vertex ordering. In addition, we show that the vertex ordering can be efficiently encoded, resulting in even more improvements in overall compression rates.

Our method makes use of several key ideas. We formulate the problem of optimally re-ordering the vertex list as a *new* combinatorial optimization problem, namely, the *minimum-entropy Hamiltonian path problem*, and prove it to be NP-hard. By making some simplifying assumption, we reduce the problem to the *traveling salesperson problem* (TSP), and solve it with heuristic algorithms. To obtain better computation efficiency, we first perform a kd-tree-like partitioning/clustering, and then re-order the vertices within each cluster by solving the TSP problem. Our coding technique is a *two-layer modified arithmetic/Huffman code* based on an *optimized alphabet partitioning* approach using a greedy heuristic. Our geometry compression can also be easily integrated with the best *connectivity* compression techniques for tetrahedral meshes [13], [40] with a small amount of overhead.

Experiments show that our technique achieves superior compression ratios with reasonable encoding times. To the best of our knowledge the results reported in this paper represent the most competitive ones for lossless compression of geometry data for tetrahedral meshes both in terms of entropy of prediction error as well as final coded bit-rate. As for quantized (lossy) compression, compared with the state-of-the-art *flipping* algorithm (extended from the one for triangle meshes [39] to tetrahedral meshes), when both integrated with the same state-of-the-art connectivity coder [40], our approach also achieves big improvements of up to 62.09 b/v (67.2%) for steady-state data, and up to 61.35 b/v (23.6%) for time-varying data.

2. Previous Related Work

There has been a large amount of work on compressing polygonal meshes (e.g., [10], [15], [24], [27]). Much of this work has mainly focused on compressing *connectivity* information. Compression techniques for polyhedral volume meshes have also been widely studied [37], [13], [31], [40]; again the main focus has been on connectivity compression. These techniques achieve an impressive compression performance of 1–2 bits per triangle for triangle meshes [38], [34], [1], and 2.04–2.31 bits per tetrahedron for tetrahedral meshes [13], [40].

There are relatively few results that focus on geometry compression. Lee *et. al.* [27] proposed the *angle analyzer* approach for traversing and encoding polygonal meshes consisting of triangles and quads. Devillers and Gandoin [11], [12] proposed techniques that are driven by the geometry information, for both triangle meshes and tetrahedral meshes. They only consider compressing the vertex coordinates but not the scalar values for the case of tetrahedral meshes. Also, their techniques are for *multi-resolution* compression, rather than for *single-resolution* compression as we consider in this paper. (We refer to [12], [16], [25] and references therein for other multi-resolution methods.)

The most popular technique for geometry compression of polygonal meshes is the *flipping* algorithm using the *parallelogram rule* introduced by Touma and Gotsman [39]. Isenburg and Alliez [20] extended the parallelogram rule so that it works well for polygonal surfaces beyond triangle meshes. Isenburg and Gumhold [21] applied the parallelogram rule in their out-of-core compression algorithm for polygonal meshes larger than main memory. Other extensions of the flipping approach for polygonal meshes include the work in [26], [6]: In [26], Kronrod and Gotsman formulated the problem of optimally traversing triangles and predicting the vertices via flippings as a *cover tree* problem on the dual graph of the triangle mesh and solved it with a heuristic, and in [6] we improved the results of [26] by formulating the problem as that of finding a *constrained minimum spanning tree* on a different graph and solving it by a new heuristic algorithm. For volume compression, Isenburg and Alliez [19] extended the flipping idea to hexahedral volume meshes. Very recently, around the same time of the conference version of this paper [7], Isenburg *et al.* [22] gave a *streaming* compression technique for tetrahedral meshes larger than main memory, where the geometry coder used a predictor that is refined from flipping and the *midpoint rule* of [13]. Still, the flipping approach extended from triangle meshes [39] to tetrahedral meshes, combined with the best connectivity coder [13], [40], is widely considered as a state-of-the-art, benchmark geometry compression technique for tetrahedral meshes. We show in Section 4 that our new algorithm achieves significant improvements over this approach.

We remark that all the previous approaches mentioned above perform vertex-coordinate quantization (to 12 bits per coordinate typically) as the first step and hence are *lossy*. Although they can be arguably used without the initial quantization, these techniques do not have an appropriate and efficient mechanism for encoding high precision floating-point numbers that we have prior to quantization, and cannot be used directly for truly lossless compression. Recently, Isenburg *et. al.* [23] focused on removing the need of the initial quantization, for polygonal mesh compression using the flipping prediction. Also, slightly after the publication of the conference version of this paper [7], Lindstrom and Isenburg [28] proposed lossless compression algorithms for floating-point data in polygonal meshes, tetrahedral meshes, and point clouds. In our experiments, we compared with some variations of their lossless point-cloud and tetrahedral-mesh approaches that compress slightly better than those in [28], and show that our compression ratios compare favorably with the results of such variations (see Section 4 for details). However, it should be noted that the strength of [28] also includes its fast encoding speed, whereas our approach achieves better compression ratios at the cost of an extra encoding time in performing the vertex re-ordering step.

Since our approach does not need the connectivity information, it is natural to look at the point-cloud compression literature. The state-of-the-art point-cloud compression algorithms are given in [14], [17], [35] and the references therein. However, these algorithms all perform an initial quantization as the first step and hence are *lossy*. In addition, when applied to mesh compression, the point-cloud approaches need an extra overhead of encoding the vertex permutation when integrated with connectivity coder (just as in our case). Moreover, their reported results are all for quantized compression of surface models, and typically their predictions in such results are not as good as those of flipping that uses the connectivity information to help the prediction. Therefore the point-cloud approaches are less competitive, and flipping is still widely considered state-of-the-art for compressing quantized mesh geometry. The point-cloud approach that is most interesting and closely related to our work is the lossless point-cloud compression technique given in [28], with which our algorithm compares favorably in terms of the compression ratios as mentioned above.

3. Our Approach

In this section we develop the main ideas behind our technique. In the following, we assume that each vertex v_i in the vertex list consists of a $t+3$ tuple $(x_i, y_i, z_i, f_{i1}, f_{i2}, \cdots, f_{it})$ where x_i, y_i, z_i are the coordinates and $f_{i1}, \cdots, f_{it}$ are the scalar values for t time steps, with steady-state data ($t = 1$) being just a special case. From now on, unless otherwise stated, when we say vertex we mean the entire $t + 3$ tuple of coordinate and scalar values. We also assume that each (tetrahedral) cell c_i in the cell list has four indices to the vertex list identifying the vertices of c_i.

3.1. Overview of Proposed Technique

We adopt a common two-step modeling process for lossless compression [33]. First, we capture the underlying structure in the data with a simple-to-describe *prediction model*. Then, we model and encode the prediction residuals.

For the first task of finding a prediction scheme, we observe that irregular-grid volume data is often created from randomly selected points, hence even the popular flipping technique presented in [39] that works well with surface data fails to do a good job when applied to irregular-grid data (see Section 4). However, these randomly selected points are typically densely packed in space. Every point has a few other points that are in its immediate spatial proximity. Hence one way to efficiently represent a vertex is by its difference with respect to some neighboring vertex. This is called *differential coding*.

If we are free from the vertex-traversal order imposed by connectivity coder, then clearly there is no need to use a fixed order of vertex list, and different orders will lead to different compression performance with differential coding. This is the main idea behind our compression technique. The idea of re-ordering to benefit compression has been explored before for lossless image compression [29]. However, its application to geometry coding is novel and as we show later, quite effective. The reason why it works is that any re-ordering technique typically involves the need for overhead information to specify the order. However, in the case of geometry compression, this overhead is a small fraction of the total cost and turns out to be well worth the effort.

We formulate the problem of how to re-order the vertex list to get maximum compression as a *new* combinatorial optimization problem, i.e., the *minimum-entropy Hamiltonian path problem*, and prove it to be NP-hard. By making some simplifying assumption, we reduce the problem to the *traveling salesperson problem* (TSP), which again is NP-hard, but we are able to solve it by proposing some simple heuristic algorithms. To speed up the computation, we partition the vertices into clusters and then re-order the vertices within each cluster. As mentioned before, vertex re-ordering causes some extra overhead information when we integrate with the connectivity coder, and we address the issue of reducing this overhead in Section 3.5.

For the second task of encoding the prediction errors/residuals, we observe that the 8-bit exponents only have relatively few distinct values and together with the sign bit (we call the 9-bit concatenation a "signed exponent" for convenience) they can be effectively compressed by a simple gzip. Our focus is thus on the mantissa. For this reason, the vertex partitioning and TSP re-ordering steps mentioned above will be solely based on the mantissa values[1] (see below). To encode the mantissa of the prediction errors, a simple approach would be to use an entropy coding such as an arithmetic or Huffman code. The problem is that such mantissa difference values are taken from a very large alphabet (23-bit numbers with many distinct values) and the corresponding Huffman table or probability table for static and semi-adaptive arithmetic coding would be so large that it would offset any gain made by the coding technique itself. The problem is more than just table size. Even if one were to use some universal coding technique like adaptive arithmetic coding, the sparsity of samples does not permit reliable probability estimation of different symbols, leading to poor coding efficiency. This problem is known as the "model cost" problem in the data compression literature. To address this problem the common approach applied in the literature has been to arithmetic/Huffman code individual bytes of the differences. However, individual bytes of a difference value are highly correlated and such a coding strategy fails to capture these correlations. We use a *two-layer modified arithmetic/Huffman code* to solve this problem.

In summary, our technique for lossless geometry compression consists of the following steps:

1) **Step 1. Take differences along time steps.** This step is done for time-varying data only. For each vertex, the mantissa of the scalar value at time step i is replaced by its difference with the mantissa of the scalar value at time step $i - 1$.
2) **Step 2. Partition vertices into clusters.** We partition the vertices (the coordinates and their scalar values) into disjoint clusters. This is done by using the *mantissa* values of the vertex *coordinates*, so that vertices in each cluster are in close spatial proximity (in terms of the coordinate mantissa values). Note that the entire vertex record (all the components, including their signed exponents) moves as a whole. We describe this step in more detail in Section 3.3.
3) **Step 3. Re-order the vertices by formulating and solving a TSP problem.** For each cluster, we re-order the vertex entries to minimize the total number of bits needed to encode the overall mantissa differences from adjacent entries. The re-ordering is done by using the *mantissa* values of the *coordinates and scalar values*, but the entire vertex record moves as a whole. This step is described in more detail in Section 3.2.
   We remark that after re-ordering the vertex list, we update the cell list so that its vertex indices reflect the vertex re-ordering. This ensures that we maintain the original connectivity information at no extra cost. (The connectivity information can later be compressed independently by existing connectivity compression algorithms; see Section 3.5 for details.)
4) **Step 4. Take mantissa differences.** The mantissa of each element in vertex v_i is replaced by its difference with the mantissa of the corresponding element in the previous

[1] Since a simple gzip can already compress the signed exponents very well, there is no need to take differences on them. In addition, differences are dependent on the order while using gzip on the original signed exponents is not. Since compressing the mantissa differences is the major challenge, our vertex partitioning and re-ordering are solely based on the mantissa values, not to be distracted by the signed exponents. Therefore we only gzip the original signed exponents rather than their differences, so that gzip essentially compresses equally well on the signed exponents no matter what order the mantissa-based partitioning and re-ordering result in.

vertex v_{i-1}.

5) **Step 5. Entropy code the mantissa differences.** We gather frequencies of each element in the mantissa difference tuples and construct a two-layer modified arithmetic or Huffman code. (See Section 3.4 for details.)
6) **Step 6. Compress the signed exponents.** Finally, we compress the signed exponents for all x-values, for all y-values, etc., and for all scalar values at time step i, $i = 1, 2, \cdots$, in that order, using gzip.

We remark that our approach can be easily modified if an initial quantization step (and thus a *lossy* compression) is desired: the quantized integers play the roles of the mantissa values (in Steps 1–5), and we do not need to worry about the exponents (i.e., no need for Step 6). As for the above lossless algorithm, Steps 1, 4 and 6 are trivial and do not need further details. We elaborate the other steps in Sections 3.2–3.4. (Instead of describing them strictly in the order in which they are performed, we describe them in logical order with one idea leading to another.) In Section 3.5 we describe how to integrate our geometry coder with a class of the best existing connectivity compression methods [13], [40].

3.2. Step 3: Vertex List Re-ordering

The efficiency of differential coding of the mantissa values of the vertex list depends on the the manner this list is ordered. Given an unordered list of vertices $\{v_1, v_2, \ldots, v_n\}$, we want to compute a permutation π such that the objective function $F = \sum_{i=2}^{n} C(v_{\pi(i)} - v_{\pi(i-1)})$ is minimized. Here, the function $C(\cdot)$ represents the cost in bits of *encoding* the mantissa difference of two adjacent vertices. We observe that each permutation π on the vertices corresponds to a *Hamiltonian path* (which visits every node *exactly once*) on a *complete* graph G defined by the vertices. Specifically, we form a complete weighted (undirected) graph G where the nodes are the vertices $v_1, v_2, \ldots, v_n$ and the edges connect all pairs of nodes. The edge weight in G between two nodes $v_i = (x_i, y_i, z_i, f_{i1}, f_{i2}, \cdots, f_{it})$ and $v_j = (x_j, y_j, z_j, f_{j1}, f_{j2}, \cdots, f_{jt})$ is defined to be $w = (|\bar{x}_i - \bar{x}_j|, |\bar{y}_i - \bar{y}_j|, |\bar{z}_i - \bar{z}_j|, |\bar{f}_{i1} - \bar{f}_{j1}|, \cdots, |\bar{f}_{it} - \bar{f}_{jt}|)$, where $\bar{a}$ means the mantissa value of a. Note that w is a $t + 3$ tuple and can be viewed as a symbol. Moreover, the cost function F depends on the *entropy* of the weights w of the edges appearing in the Hamiltonian path of the corresponding permutation π. We define such entropy formally as follows.

Definition 1 *Given a Hamiltonian path P of a complete weighted graph G, let t_i be the number of times a distinct edge weight w_i appears in the edge weights of P, and $\alpha_i = t_i / \sum_i t_i$. Then the* zero-order entropy *of the weights on the edges of P is defined by $H_0(P, G) = -\sum_i \alpha_i \lg \alpha_i$.*

Over all possible Hamiltonian paths of G, there exists a path P that minimizes the entropy $H_0(P, G)$:

Definition 2 *Given a complete weighted graph G, we define a minimum-entropy Hamiltonian path (ME Hamiltonian path) to be a Hamiltonian path P such that $H_0(P, G) \leq H_0(P', G)$ for all Hamiltonian paths P' of G.*

Now the problem is: How do we compute an ME Hamiltonian path of G? This is a new problem and we can prove that it is NP-hard:

Theorem 1 *The problem of finding an ME Hamiltonian path on a complete weighted graph G is NP-hard. The statement is true regardless of the edge weights of G being tuples or scalar values.*

Proof: First, we cast the problem to the following *decision* problem: Given G and a value k, is there a Hamiltonian path on G that has entropy k? Now, we prove that the decision problem is NP-hard by reducing from the *original* Hamiltonian path problem, which is NP-complete [9]: given a *general, unweighted* graph H, is there a Hamiltonian path on H that visits each node of H exactly once using only the edges of H? We convert H by extending it to a weighted complete graph, i.e., by extending it to an instance G_I of the graph G, where a Hamiltonian path of entropy $k = 0$ on G_I is a Hamiltonian path on H and vice versa, as follows. We take H and add the missing edges so that each pair of nodes has an edge; this makes the new graph G_I a complete graph. We also need to define edge weights in the new graph G_I: we assign the *same* symbol w_0 to *all* the original edges of H; for each of the newly added edges, we assign a *distinct* symbol which is also distinct from w_0. Here the 'symbols' can represent either tuples or scalar values, and thus our proof carries over regardless of the edge weights being tuples or scalar values. Now we claim that H has a Hamiltonian path if and only if G_I has a Hamiltonian path of entropy $k = 0$. Indeed, if H has a Hamiltonian path P, then P is a Hamiltonian path on G_I that only uses the edges of H, i.e., that only uses edges of weight w_0, and hence the entropy of P is 0. On the other hand, if H does not have a Hamiltonian path, then any Hamiltonian path P' on G_I must use some newly added edge(s); therefore, it is not possible to have a single symbol for all edge weights of P' and hence the entropy of P' is not 0. ■

One major difficulty in trying to optimize the objective function F with the above ME Hamiltonian path formulation is that the cost function $C(e)$ is *not fixed* for a given edge e in G (rather, $C(e)$ varies depending on other edges appearing in a given Hamiltonian path P; see Definition 1). To simplify the problem, we make a simplifying assumption that the cost of representing the value n is just proportional to $\log_2 |n|$. Note that this implicitly assumes that a difference (a prediction error) whose absolute value is smaller has a higher frequency, which is commonly assumed in designing predictors for compression. With these assumptions, we now define $C(e)$ for each edge $e = (v_i, v_j)$ of G to be $C(e) = \lg |\bar{x}_i - \bar{x}_j| + \lg |\bar{y}_i - \bar{y}_j| + \lg |\bar{z}_i - \bar{z}_j| + \lg |\bar{f}_{i1} - \bar{f}_{j1}| + \cdots + \lg |\bar{f}_{it} - \bar{f}_{jt}|$[2], where the coordinates and scalar values of v_i and v_j are as given before and again $\bar{a}$ means the mantissa value of a. Note that $C(e)$ is now *fixed*, independent of the choice of a Hamiltonian path, and thus we can directly assign $C(e)$ as the length of edge e. This allows us to restate the problem as the following *traveling salesperson problem* (TSP):

> Form a complete weighted graph G where the nodes are the vertex entries and the edges connect each pair of nodes. The length of each edge e is given by $C(e)$ defined above. Find a Hamiltonian path on G that visits every node exactly once so that the total path length is minimized.

Unfortunately TSP is still an NP-complete problem in general, but there are many well-known heuristics to get good solutions for a given instance. In our implementation we chose to use

[2] We treat the case when the argument is zero as special and simply return a zero for the lg value.

the *Simulated Annealing (SA)* based heuristics and the *Minimum Spanning Tree (MST)* based approximation algorithm. The MST-based algorithm first computes a minimum spanning tree of G, and then visits each node exactly once by a depth-first-search traversal of the tree. This algorithm produces a Hamiltonian path with no more than twice the optimal path length if the distances in G obey the triangle inequality [9]. Although the triangle inequality may not always hold in our case, we observed that this algorithm did produce comparable-quality solutions and ran much faster ($O(n^2)$ time for n vertices since G is a complete graph) than simulated annealing; see Section 4 for details.

Finally, we remark that the re-ordering process has to be done just once at compression time and hence the running-time cost is absorbed in the preprocessing stage. The decompression process does not have to perform this re-ordering and can remain computationally very efficient.

3.3. Step 2: Partitioning the Vertex List into Clusters

This step is solely for the purpose of reducing the computing time of the re-ordering process in Step 3. Although the re-ordering process occurs only once during encoding and never during decoding, the number n of vertices present in a typical tetrahedral mesh is very large (e.g., hundreds of thousands), making even the $O(n^2)$-time MST heuristic infeasible. In fact just computing the weights on the complete graph G will need $O(n^2)$ time. Hence in Step 2 we first partition the vertex list into small clusters and then in Step 3 we re-order the vertices *within each cluster* by solving the TSP problem, one for each cluster. Note that the decoder does *not* need to know about the clustering and the re-ordering.

To achieve this clustering, we propose the following simple and effective *kd-tree-like partition* scheme. Suppose we want to form K clusters of equal size. We sort all vertices by the mantissa of the x-values, and split them into $K^{1/3}$ groups of the same size. Then, for each group we sort the vertices by the mantissa of the y-values and again split them equally into $K^{1/3}$ groups. Finally we repeat the process by the mantissa of the z-values. Each resulting group is a cluster of the same size, and the vertices in the same group are spatially close (in terms of the mantissa values of the coordinates). In this way, the original running time of $O(n^2)$ for re-ordering is reduced to $O(K(n/K)^2) = O(n^2/K)$, a speed-up of factor K. Also, the overall clustering operation takes only $O(n \log n)$ time (due to sorting). It should be noted that, as mentioned in Section 3.1, in both the clustering and re-ordering steps, the *entire vertex record* (all the components, including their signed exponents) moves as a whole.

3.4. Step 5: Entropy Coding Mantissa Differences

As mentioned in Section 3.1, vertex mantissa differences have a very high dynamic range and cannot be efficiently encoded in a straightforward manner. One way to deal with the problem of entropy coding of large alphabets is to partition the alphabet into smaller sets and use a product code architecture, where a symbol is identified by a set number and then by the element number within the set. If the elements to be coded are integers, as in our case, then these sets would be intervals of integers and an element within the interval can be represented by an offset from the start of the interval. The integers are then typically represented by a Huffman code of the interval identifier and then by a fixed-length encoding of the offset. This strategy is called *modified Huffman coding* and has been employed in many data compression standards [32], [18], [30]. In [4], [8], we formulated the problem of optimal alphabet partitioning for the design of a two-layer modified Huffman code, and gave both an $O(N^3)$-time dynamic programming algorithm and an $O(N \log N)$-time greedy heuristic, where N is the number of distinct symbols in the source sequence. The greedy method gives slightly worse (and yet comparable) compression ratios, but runs much faster.

In this paper, to encode the vertex mantissa differences, we use the above greedy algorithm to construct a two-layer modified Huffman code, and also extend the greedy algorithm in a similar way to construct a *two-layer modified arithmetic code* where in the first layer we replace the Huffman code by a semi-adaptive arithmetic code. As is well known, the latter (arithmetic code) gives better compression ratios, at the cost of slower encoding and decoding times.

Since we encode multiple clusters and multiple vertex components, we can either use a single arithmetic/Huffman code for the mantissa difference of each vertex component (coordinate or scalar value) for each cluster, or we can use a single arithmetic/Huffman code for all the data values in all clusters. In practice, we found that the former approach had too many probability/Huffman tables and thus very expensive overall table cost, and that the latter approach had less coding efficiency since not all those data values were correlated. There are other possible options that are in between these two extreme approaches. We found that the vertex mantissa differences for the coordinate values had similar distributions and using a single arithmetic/Huffman code for all of them gave the best performance—achieving a very good coding efficiency while greatly reducing the overall table cost. The same was the case for the scalar values. Hence we only use two arithmetic/Huffman codes for steady-state data, one for all coordinates across all clusters, the other for all scalar values across all clusters. We call this the **Combined Greedy** (**CGreedy**) approach. For time-varying data, we again use one arithmetic/Huffman code for all coordinates across all clusters, and in addition we use one arithmetic/Huffman code for the scalar values of *every i time steps* across all clusters (where i is a fixed integer); this is the **CGreedy**i approach. In summary, our proposed coding methods are CGreedy for steady-state data, and CGreedy$i$ for time-varying data, using two-layer modified arithmetic/Huffman coding.

At this point, we remark that our geometry decoder does *not* need to perform any of vertex clustering, vertex re-ordering, or alphabet partitioning. All it needs to do is to look up the probability/Huffman table and perform a standard decoding for arithmetic/Huffman code, plus a fast decoding of the signed exponents by gunzip. Hence the decompression essentially takes linear time, and can be performed very efficiently.

3.5. Integration with Connectivity Compression

In this section, we show how our geometry coder can be easily integrated with a class of the best existing *connectivity* coders for tetrahedral meshes [13], [40], which are lossless, so that we can compress both geometry and connectivity losslessly. We remark that some extra cost is associated with this integration, as discussed at the end of this section. However, our compression results are still superior after offsetting such extra cost, as will be seen in Section 4.

The technique of [13] achieves an average bit rate for connectivity of about 2 bits per tetrahedron, which is still the best reported result so far; the technique of [40] simplifies that of [13], with a slightly higher bit rate. Both techniques are based on the same traversal and encoding strategy, reviewed as follows. Starting from some tetrahedron, adjacent tetrahedra are traversed. A queue, Q, called *partially visited triangle list*, is used to maintain the triangles whose tetrahedron on one side of the triangle has been visited but the one on the other side has not. When going from the current tetrahedron to the next tetrahedron sharing a common triangle t, the new tetrahedron is encoded by specifying its fourth vertex, say v. If v is contained in some triangles in Q, v is encoded by a small local index into an enumeration of a small number of candidate triangles in Q that are (necessarily) adjacent to the current triangle t. If v cannot be found from these candidate triangles, an attempt is tried to encode v by a global index into the set of all vertices visited so far (namely, using $\log_2 k$ bits if k vertices have been visited so far). Finally, if v has never been visited before, then v is recorded by its full geometry coordinates.

To integrate our geometry coder with the above scheme, after clustering and re-ordering the vertex list, we update the cell list so that its vertex indices are the new indices of the corresponding vertices in the new vertex list. Specifically, after the vertex re-ordering, we produce a list of tuples (VID_{old}, VID_{new}), meaning that the vertex with index VID_{old} in the original vertex list now has index VID_{new} in the new vertex list. We then go over the cell list and update the vertex indices accordingly. Note that we now still maintain the original connectivity information with the cell list, and can discard the list of tuples (VID_{old}, VID_{new}). After our geometry compression is complete, the above connectivity compression scheme can be performed in exactly the same way, except that for the last case, when v has never been visited before, we use the (new) index to the (new) global vertex list for v (using $\log_2 n$ bits if the mesh has n vertices), rather than the full geometry information of v. In this way, the connectivity compression operates in the same way, with the "base geometry data" being the indices to the vertex list, rather than the direct geometry information of the vertices. To decompress, we first decode our geometry code, and then the connectivity code. Given the vertex indices, the corresponding actual geometry information is obtained from our decoded vertex list. It is easy to see that this integration scheme works for time-varying datasets as well.

The above integration scheme causes an extra cost of at most $\log_2 n$ bits per vertex, because the vertex list, after re-ordering by our geometry coder, is fixed and may not be in the same order as that in which the vertices are first visited in the connectivity-coding traversal. Therefore, during such traversal, when we visit a vertex v for the first time, we use the index to the global vertex list to represent v, resulting in a cost of $\log_2 n$ bits per vertex for encoding such a "vertex permutation". Our next task is to reduce such extra cost by encoding the *permutation sequence* (the sequence of vertex indices produced by connectivity traversal). Our idea is that the connectivity traversal goes through local vertices, which should also be near neighbors in our TSP order; in other words, the two sequences should be somehow correlated, which we can explore. Our solution is a simple one: we perform a differential coding on the permutation sequence. Typically there are many distinct index differences; we again encode them by the two-layer modified Huffman code using the greedy heuristic for alphabet partitioning (see Section 3.4). We show in Section 4 that this approach encodes the permutation sequence quite efficiently.

Data	# cells	# vertices	vertex-list size (byte)
Spx	12936	20108	321,732
Blunt	187395	40960	655,364
Comb	215040	47025	752,404
Post	513375	109744	1,755,908
Delta	1005675	211680	3,386,884
Tpost10	615195	131072	6,815,752
Tpost20	615195	131072	12,058,632

Table 1. Statistics of our test datasets. The entries in "vertex-list size" show the uncompressed file sizes of the vertex lists, including all coordinates and scalar values, each a binary 32-bit floating-point number.

Entropy (b/v)	TSP-ann	TSP-MST	Sorting	Orig.
Comb 125	57.38	61.54	64.53	75.71
Comb 216	58.81	61.71	64.59	
Comb 343	60.03	62.06	64.67	
Comb 512	60.40	62.46	64.79	
Time (sec)	**TSP-ann**	**TSP-MST**	**Sorting**	**Orig.**
Comb 125	2150	23.11	0.8	0
Comb 216	1188	11.19	0.8	
Comb 343	665	6.72	0.8	
Comb 512	572	5.00	0.8	

Table 2. Re-ordering results in terms of 8-bit entropy (b/v), with different numbers of clusters shown in the first column. The upper table shows the resulting 8-bit entropy; the lower table shows the re-ordering time. "Orig." means no re-ordering. The re-ordering times of TSP-ann and of TSP-MST include the time of Sorting.

4. RESULTS

We have implemented our techniques in C++/C and run our experiments on a Dell Precision PC with two 3GHz Intel Xeon CPUs and 6GB of RAM, running under RedHat Enterprise 64bit Linux OS. The datasets we tested are listed in Table 1[3]. They are all given as tetrahedral meshes, where Tpost10 and Tpost20 are of the same mesh with 10 and 20 time steps respectively, and the remaining datasets are steady-state data. These are all well-known datasets obtained from real-world scientific applications: the Spx dataset is from Lawrence Livermore National Lab, the Blunt Fin (Blunt), the Liquid Oxygen Post (Post), the Delta Wing (Delta), and the Tpost datasets are from NASA. The Combustion Chamber (Comb) dataset is from Vtk [36]. We show their representative isosurfaces in Figure 1.

Re-ordering Vertex List

To evaluate the effectiveness of our approach of re-ordering vertex list, we show in Table 2 the resulting entropy after performing the re-ordering on a representative dataset. Specifically, we compare the following re-ordering approaches: (1) no re-ordering: using the original order in the input; (2) sorting: partitioning the vertices into clusters, which involves sorting the vertices; (3) TSP-ann: first partitioning the vertices into clusters, and then re-ordering each cluster by solving the TSP problem using simulated annealing; and (4) TSP-MST: the same as (3) but solving the TSP problem with the minimum-spanning-tree heuristic. After

[3]They are all available at http://cis.poly.edu/chiang/datasets.html.

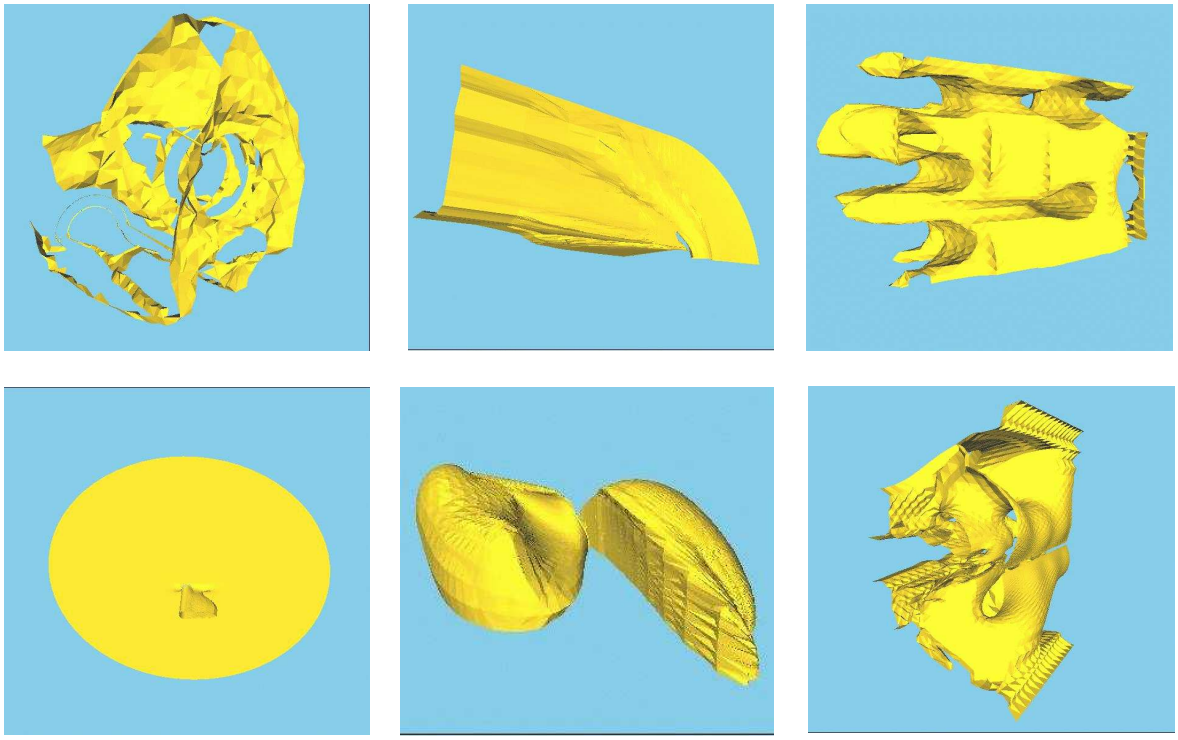

Fig. 1. Representative isosurfaces from our test datasets. Top row: left—Spx; middle—Blunt; right—Comb. Bottom row: left—Post; middle—Delta; right—Tpost10 (Tpost20).

re-ordering, we replace the mantissa of each vertex component by its difference from the corresponding mantissa of the previous vertex, and compute the *8-bit entropy* of the mantissa differences as follows: for a particular coordinate (or scalar value), a separate entropy is computed for the first byte, the second byte, and so on; we then sum the separate entropy of each byte. Table 2 also shows the corresponding re-ordering times.

It is easy to see from Table 2 that re-ordering clearly reduces the entropy, with TSP-ann giving the best entropy, followed by TSP-MST, and then sorting. We see that the running times were in reverse order, showing a nice trade-off between compression and speed. Observe that TSP-MST produces entropies comparable to those of TSP-ann, but the speed can be about 100 times faster. Also, it is interesting to see that as we partitioned into more clusters, the running times of TSP-MST and TSP-ann both reduced significantly, with similar entropy values. We found that a partition in which there were about 100–200 vertices per cluster typically gave the best balance between compression and speed—equally competitive compression with the best speed among all the choices that we tested. In summary, TSP-MST with a cluster size of 100–200 vertices (e.g., about 512 clusters for our datasets) is a right choice in terms of both compression ratios and computing speed.

As seen from Table 2, re-ordering (together with clustering) using TSP-MST and 512 clusters has a run-time speed of about 9.4K vertices/second. We remark that, as observed in our experiments, vertex re-ordering was the most time-consuming step in our compression process. This is also the additional-overhead step compared to other compression approaches. Our other compression steps such as arithmetic/Huffman coding and gzip are standard (except for alphabet partitioning using CGreedy/CGreedy$i$ that needs an additional sorting by quick sort, which is much faster than re-ordering), and their total running time was no more than the re-ordering time. The overall compression speed was typically at least 4.7K vertices/second, using our un-optimized implementation. This compression speed is roughly comparable to some recent techniques reported in the literature. For example, the compression speed of [14] is reported to be 5K vertices/second on a 2GHz PC.

Encoding for Steady-State Data

Recall from Section 3.4 that our encoding techniques are CGreedy using two-layer modified arithmetic or Huffman coding (A-Cg or H-Cg) for the mantissa differences, and gzip for the signed exponents. We show our compression results with *no quantization* (i.e., lossless compression) in Table 3, where we used TSP-MST and TSP-ann for vertex re-ordering. To evaluate the encoding effectiveness of A-Cg and H-Cg, we also compared them with the Greedy method using two-layer modified arithmetic or Huffman coding (A-g or H-g) where we had four probability-/Huffman-tables (one for each of x, y, z and f) for each cluster, as well as with 8-bit static and adaptive arithmetic coding (AC(S) and AC(A)), where for each vertex component we coded the first byte by an arithmetic code and then the next byte by another arithmetic code and so on, for each cluster (hence 16 arithmetic codes per cluster). We omit the results of AC(S) and AC(A) under TSP-MST as the scenarios were similar to those under TSP-ann.

It is clear from Table 3 that under the same vertex re-ordering method (TSP-MST or TSP-ann), CGreedy (A-Cg or H-Cg) is always much better than Greedy (A-g or H-g), which in turn is much better than arithmetic coding (AC(A) or AC(S)). This shows that our two-layer modified arithmetic/Huffman coding indeed solves very well the problem of large alphabet by sig-

	TSP-MST					TSP-ann						
Size (b/v)	H-Cg	H-g	A-Cg	A-g	Exp	H-Cg	H-g	A-Cg	A-g	AC(S)	AC(A)	Exp
Spx 216	91.38	108.38	89.18	105.43	14.09	89.17	107.22	86.43	104.06	348.80	111.11	14.07
Spx 512	90.70	120.80	88.10	114.03	13.92	89.12	119.46	86.03	112.59	381.52	110.89	13.93
Blunt 216	41.60	58.50	40.56	60.65	7.76	39.29	56.03	37.62	58.05	126.97	95.90	7.41
Blunt 512	41.49	64.29	40.33	66.75	7.61	39.62	62.47	37.99	64.86	155.30	104.71	7.36
Comb 216	68.20	78.06	67.23	76.65	5.66	65.82	76.07	64.80	74.63	284.59	96.05	5.27
Comb 512	68.88	86.07	68.05	83.28	5.85	66.35	83.81	65.27	80.98	251.15	101.54	5.38
Post 216	36.56	45.30	35.84	42.70	8.01	35.00	43.73	34.13	41.07	97.47	90.91	7.85
Post 512	37.84	49.05	37.24	46.55	8.68	28.99	47.92	35.51	45.16	118.80	94.23	8.46
Delta 216	74.65	81.78	70.07	81.07	11.57	70.12	78.22	66.43	77.42	153.72	89.22	10.90
Delta 512	74.03	82.85	70.27	83.00	10.99	70.15	79.93	66.44	80.06	210.15	93.29	10.56

Table 3. Compression results (b/v) with *no* quantization, for 216 and 512 clusters, using TSP-MST or TSP-ann for re-ordering the vertices. The bit rate before compression is 128 b/v. The cost of signed exponent is included in each entry; we also list this exponent cost separately (36 b/v before compression).

	TSP-MST			Original			Point Cloud
Size (b/v)	H-Cg	A-Cg	Exp	AC(A)	AC(S)	Gzip	ILS-point
Spx 216	91.4	89.2	14.1	107.2	116.1	112.7	99.5
Spx 512	90.7	88.1	13.9				
Blunt 216	41.6	40.6	7.8	108.6	92.9	25.9	59.7
Blunt 512	41.5	40.3	7.6				
Comb 216	68.2	67.2	5.7	98.5	102.6	100.7	76.3
Comb 512	68.9	68.1	5.9				
Post 216	36.6	35.8	8.0	105.3	99.0	28.1	65.5
Post 512	37.8	37.2	8.8				
Delta 216	74.7	70.1	11.6	100.9	104.1	126.4	77.4
Delta 512	74.0	70.3	11.0				

Table 4. Compression results (b/v) with *no* quantization, for 216 and 512 clusters. The bit rate before compression is 128 b/v. The cost of signed exponents is included in H-Cg and A-Cg; we also list this exponent cost separately (36 b/v before compression). We compare our results with Gzip, 8-bit adaptive and 8-bit static arithmetic coding (AC(A) and AC(S) respectively) on the original input data. We also compare our results with the lossless point-cloud method *ILS-point*.

nificantly improving over arithmetic coding, and that CGreedy further greatly improves over Greedy by saving the probability-/Huffman-table cost while retaining the compression efficiency of our two-layer modified coding. Moreover, under the same CGreedy or Greedy method, arithmetic coding compressed better than Huffman coding (i.e., A-Cg better than H-Cg, and A-g better than H-g) with longer encoding times, as expected. Therefore, A-Cg is our method of choice for achieving best compression efficiency. Comparing between the two vertex re-ordering methods, we see that TSP-MST resulted in compression ratios that were only slightly worse than those of TSP-ann, but TSP-MST ran significantly faster, similar to the situations of the results that we discussed in Table 2 (i.e., TSP-MST could be more than 100 times faster). Therefore TSP-MST is our method of choice. It is also interesting to see that the signed exponents were compressed well, and such compression results were similar between those under TSP-MST and those under TSP-ann. In summary, we choose TSP-MST for vertex re-ordering, A-Cg for encoding the mantissa differences, and gzip for encoding the signed exponents.

To evaluate our lossless compression approach, we would like to compare with the state-of-the-art lossless compression technique for floating-point data [28]. In [28], two approaches related to ours are proposed: (1) lossless point-cloud approach, where one uses the original order of the input vertex list and performs differential coding, i.e., using the previous item as the prediction and losslessly encoding the prediction error by their *range coding*; (2) lossless Flipping approach, where one extends the most popular flipping algorithm [39] from triangle meshes to tetrahedral meshes (i.e., one predicts the current vertex position by flipping from the opposite vertex of the face-adjacent tetrahedral cell through the center of the common face) using floating-point coordinates, and losslessly encodes the prediction error by their *range coding*. As shown in the experiments of [28], in both (1) and (2) the range coding can be replaced by the coding method (which we refer to as the *ILS code*) of their earlier work for compressing floating-point polygonal meshes [23]; we call these replacement methods *ILS-point* and *ILS-flip* respectively. The compression ratios of ILS-point and ILS-flip are actually (slightly) *better* than (1) and (2), although (1) and (2) are faster (see [28]). Since our focus is on compression ratios and the ILS code [23] is available from the authors[4], we compare our lossless compression approach with ILS-point and ILS-flip. Note that when applying to tetrahedral-mesh compression and integrated with connectivity coder (e.g., [40]), ILS-point has an extra overhead of encoding the vertex permutation sequence just as our approach, while ILS-flip does not have this overhead. Hence we need to add this

[4] `http://www.cs.unc.edu/~isenburg/`.

Size (b/v)	Exp	A-Cg	Z	Total	Gzip
Blunt 216	2.79	19.89	0.07	22.75	25.85
Blunt 512	2.67	19.62	0.07	22.37	25.85
Post 216	4.94	19.79	0.05	24.78	28.09
Post 512	5.27	19.94	0.05	25.26	28.09

Table 5. Compression results (b/v) with *no* quantization (128 b/v in raw data) and the special treatment of the z coordinate.

Entropy (b/v)	Flip	Original
Blunt	100.15	90.60
Comb	104.17	97.44
Post	108.22	102.99
Delta	103.03	97.33
Tpost10	299.87	257.44
Tpost 20	538.25	447.29

Table 6. The 8-bit entropy (b/v) of the prediction errors of *lossless* Flipping and the original input data. The raw-data bit rates are: 128 b/v for the first four datasets, 416 b/v for Tpost10, and 736 b/v for Tpost20.

Dataset	TSP-MST	$+\lg n$	+perm	ILS-flip
Blunt216	40.6 / 22.8	55.9 / 38.1	52.7 / 34.9	77.2
Blunt512	40.3 / 22.4	55.6 / 37.7	52.2 / 34.3	
Comb216	67.2	82.7	79.2	83.2
Comb512	68.1	83.6	79.9	
Post216	35.8 / 24.8	52.5 / 41.5	49.2 / 38.2	81.6
Post512	37.2 / 25.3	53.9 / 42.0	50.6 / 38.7	
Delta216	70.1	87.8	84.0	86.8
Delta512	70.3	88.0	83.8	

Table 7. Compression results (b/v) with *no* quantization (128 b/v in raw data), compared with lossless flipping ILS-flip. For Blunt and Post, we show both of our results *without* (left of "/") and *with* (right of "/") the special treatment of the z coordinate.

permutation overhead to our approach only when we compare with the Flipping method (either the lossless ILS-flip method or the lossy, quantized Flipping method).

In Table 4 we show our lossless compression results with *no quantization*, compared with gzip (Gzip) and 8-bit arithmetic coding on the original input data (i.e., *without* vertex re-ordering). For Gzip, we put all x-values together then all y-values together then all z-values and finally all scalar values, and compressed the resulting list by gzip. This gave the best compression ratios among other gzip options. In 8-bit arithmetic coding, as before we coded for each vertex component the first byte by an arithmetic code and then the next byte by another arithmetic code and so on (hence 16 arithmetic codes for the entire data). We tried both adaptive and static arithmetic codes; their results are listed as AC(A) and AC(S) respectively in Table 4. Finally, we also compare our results with ILS-point in Table 4.

As can be seen from Table 4, our approach and ILS-point are both always much better than arithmetic coding (no matter AC(A) or AC(S)), and even better than Gzip except for Blunt and Post. In addition, compared with ILS-point our results are always better, which we believe is mainly due to our vertex re-ordering. As for Gzip, it compressed amazingly well for Blunt and Post. With further investigation, we found that in these two datasets the z-coordinates only had relatively very few distinct values, i.e., the vertices lie on these relatively few z-planes, and Gzip was very good in encoding such repetitions (since it includes run-length coding as one of its coding options). For this, we slightly modify our algorithm and give the z-values a special treatment: all the partitioning/clustering and the TSP-MST vertex re-ordering steps are the same; only at the final encoding step, we code the entire z-values by gzip, while the x, y and scalar values are encoded as before (A-Cg for the mantissa differences and gzip for the signed exponents). We show the results in Table 5. Observe that gzip compressed the z-values from 32 b/v to 0.07 and 0.05 b/v! With our special treatment for z, our results are now better than Gzip for these two special datasets (see Table 5). In general, at the final encoding step, we can first try to gzip each of the entire x, y, z and scalar values to see if any of them deserves a special treatment, and then proceed to compress the remaining portions with our normal technique as above.

In addition to Gzip, arithmetic coding and ILS-point, it is natural to compare with Flipping, which is widely considered the state of the art when applied *after quantization*, and as mentioned above lossless Flipping methods (with *no* quantization) were recently given in [23], [28]. First, we computed the 8-bit entropy of the prediction errors of lossless Flipping, as well as the 8-bit entropy of the original input data, as shown in Table 6[5]. Interestingly, lossless Flipping actually *increases* the entropy for *all* our datasets, including steady-state and time-varying ones (such events have been observed in [23] for polygonal meshes, but only for very few datasets). Therefore, lossless Flipping is *not* a competitive predictor for our volume meshes.

Moreover, we compare our lossless compression results with those of ILS-flip, as shown in Table 7, where we also show our results after adding the *raw cost* ("$+\lg n$") and the *encoding cost* ("+perm") of vertex permutation when integrated with the connectivity coder [40]. It is interesting to see that ILS-flip compressed better than the 8-bit entropies shown in Table 6. This is due to the fact that the ILS code does not encode by units of bytes and is able to capture the dependencies between bytes, and therefore is not lower bounded by the 8-bit entropies. More importantly, from Table 7 we see that our final compression results (the entries in "+perm") are always better than those of ILS-flip, showing the efficacy of our approach.

Now, we want to see the compression performance of our methods when an initial quantization (i.e., *lossy* compression) is desired. To this end, we first quantized each vertex component (coordinate and scalar value) into a 32-bit integer[6], and then applied our algorithm as well as the state-of-the-art Flipping approach[7]. Note that now our method does not use gzip at all (since there is no exponent). To compare the prediction performance, we computed the 8-bit entropy of the prediction errors using our TSP-MST method and Flipping; the results are shown in Table 8. Again we also show our results after adding to the entropy the

[5]Surprisingly, Spx has only 2896 vertices (14.402%) that are referenced by the tetrahedral cells and thus the remaining vertices cannot be predicted by Flipping. Hence we do not list the results for Spx.

[6]We also tried 24-bit quantization, but Blunt, Post and Delta all resulted in some vertices collapsed with inconsistent scalar values. Hence we performed 32-bit quantization for the steady-state data.

[7]In [5] we improved the geometry compression over Flipping; however, recently we found that the connectivity-compression overhead in [5] seemed to offset the gains in geometry compression (we are still trying to reduce this overhead but the status is not finalized yet), and thus here we did not compare with [5].

Dataset	TSP-MST	$+\lg n$	+perm	Flip
Blunt216	42.28	57.60	48.72	87.65
Blunt512	41.87	57.19	48.23	
Comb216	77.85	93.37	87.29	90.02
Comb512	77.97	93.49	88.08	
Post216	41.28	58.02	46.21	85.06
Post512	41.72	58.46	46.64	
Delta216	64.97	82.66	76.42	83.26
Delta512	65.08	82.77	73.56	

Table 8. 8-bit entropy results (b/v) with 32-bit quantization (128 b/v in raw data).

raw cost ("$+\lg n$") and the *encoding cost* ("+perm") of vertex permutation. It can be seen that we encoded the permutation sequence quite efficiently, and the resulting entropies of our method (the entries in "+perm") were much better than those of Flipping. This shows that the prediction performance of our technique is much better than Flipping, without taking any coding technique into consideration.

In Table 9, we compare the 32-bit quantized compression results after the final encoding, where for Flipping we encoded the flipping errors by gzip (Gzip), 8-bit adaptive arithmetic coding (AC(A)), and 8-bit static arithmetic coding (AC(S)). Again we also list our results after adding the *raw cost* ("$+\lg n$") and the *encoding cost* ("+perm") of vertex permutation. Observe that at this time AC(S) is always better than AC(A) for encoding the flipping errors. Moreover, it is interesting to see that our encoding results (A-Cg) are better than their 8-bit entropies (TSP-MST) listed in Table 8 in all cases except for two (Comb 216 and Comb 512). This is due to the fact that our two-layer arithmetic coding technique is able to capture the dependencies between bytes whereas a simple 8-bit entropy calculation is not, therefore our encoding results are not lower bounded by the 8-bit entropies. Finally, we see clearly from Table 9 that our final results ("+perm") achieve significant improvements over Flipping (up to 62.09 b/v (67.2%) for "+perm" vs. AC(S) in Post 216), showing the efficacy of our technique despite the additional overhead cost of encoding the permutation sequence.

Encoding for Time-Varying Data

In Table 10 we show the results of applying our compression techniques to time-varying datasets, with *no* quantization. We found that as we increased i in CGreedy$i$, the compression ratio typically increased. Also, arithmetic coding (A-Cgi) compressed better than Huffman coding (H-Cgi) with longer encoding times, as expected. We also compared with Gzip, 8-bit adaptive and static arithmetic coding (AC(A) and AC(S)) on the original input data, as well as ILS-point. We can see that ILS-point is typically the best among the latter four, and our results are always much better than ILS-point, with the best one (A-Cg10 on Tpost20) 106.59 b/v (29.62%) more efficient. In Table 11, we compare our lossless compression results with the lossless Flipping approach ILS-flip. For our method, we also list the results of adding the extra cost of encoding the permutation sequence ("+perm"), which are our final results. As before, ILS-flip compressed much better than the 8-bit entropies shown in Table 6, showing that the ILS code is a nice coding technique. More importantly, we see from Table 11 that our final compression results (those in "+perm") are always much better than those of ILS-flip, with the best one (A-Cg10 on Tpost20) 104.14 b/v (28.09%) more efficient, despite paying the additional cost of encoding the vertex permutation. This shows that while lossless Flipping does not predict well as seen in Table 6, our technique, in particular the vertex re-ordering approach, is quite effective in achieving compression efficiencies.

In Table 12, we compare the results of applying our method as well as Flipping after an initial 32-bit quantization was performed. For Flipping, we show the results of using gzip, 8-bit adaptive arithmetic coding, and 8-bit static arithmetic coding to encode the flipping errors—we can see that this time the static arithmetic coding was always better than the adaptive one, and in fact better than gzip as well. For our method, again our final results need to add the extra cost of encoding the permutation sequence (shown as the entries in "+perm"). It is clear to see that our results are always much better than Flipping. However, comparing the A-CGi entries of Tpost10 and Tpost20 in Tables 11 and 12, we see that our *lossless* compression results are actually *better* than the (32-bit) quantized compression results, which is very surprising. To see how this is possible, recall that in lossless compression we entropy code the 23-bit mantissa differences and compress the 9-bit signed exponents by gzip, while for 32-bit quantization we entropy code the differences of 32-bit quantized integers. Observe that the entropy of 32-bit quantized integers can be *larger* than the entropy of 23-bit mantissa values: two floating-point values can have the *same* mantissa value but different exponents, resulting in two *distinct* integers (hence a larger entropy) after quantization if they are not quantized into the same value. Similarly, the entropy of the differences of 32-bit quantized integers can be larger than the entropy of 23-bit mantissa differences. Therefore the entropy code for the 23-bit mantissa differences can be better than the entropy code for the differences of 32-bit quantized integers, especially for such high-precision quantization. If gzip can code the signed exponents well, i.e., the extra overhead of encoding the signed exponents by gzip is still smaller than the savings from the entropy coding, then the lossless compression can be better than the quantized compression. Looking at Table 10, we see that the signed exponents are indeed compressed very well (from 117 b/v to 6.41 b/v in Tpost10, and from 207 b/v to 7.73 b/v in Tpost20). Note that for the steady-state datasets, our lossless compression results are better than the 32-bit quantized compression results only for Comb 216 and Comb 512 (see A-Cg in Tables 4 and 9), where Comb has the best compression ratios for the signed exponents among the datasets (see Table 4).

Finally, we repeated the same experiments as those in Table 12, except that we initially performed a 24-bit quantization; the results are shown in Table 13. This time our quantized compression results are better than the lossless ones due to lower-precision quantization, as expected. It is interesting to see that now the results of A-Cgi for different values of i are quite close to each other. Finally, it is clear from Table 13 that our compression results are always much better than Flipping, with the best improvement (A-Cg20 vs. F-AC(S) for Tpost20 +perm) up to 61.35 b/v (23.6%).

5. Conclusions

We have presented a novel lossless geometry compression technique for steady-state and time-varying fields over irregular grids represented as tetrahedral meshes. Our technique exhibits a nice trade-off between compression ratio and encoding speed. Among the options provided in various stages, the following version of

	TSP-MST			Flipping		
Size (b/v)	A-Cg	+lg n	+perm	Gzip	AC(A)	AC(S)
Blunt 216	22.36	37.69	30.13	69.67	106.27	97.23
Blunt 512	21.66	36.98	29.27			
Comb 216	78.66	94.18	84.32	105.71	110.11	97.51
Comb 512	78.87	94.39	84.72			
Post 216	22.30	39.05	30.32	95.12	100.93	92.41
Post 512	22.58	39.33	31.26			
Delta 216	55.33	73.02	66.90	75.31	99.44	91.32
Delta 512	55.23	72.92	66.22			

Table 9. Compression results (b/v) with 32-bit quantization (with *no* special treatment for z). The bit rate before compression is 128 b/v. The prediction errors in Flipping were encoded with gzip (Gzip), 8-bit adaptive arithmetic coding (AC(A)), and 8-bit static arithmetic coding (AC(S)).

	TSP-MST							Original			Point Cloud
	H-Cg1	H-Cg5	H-Cg10	A-Cg1	A-Cg5	A-Cg10	Exp	Gzip	AC(A)	AC(S)	ILS-point
Tpost10	165.97	157.58	155.40	149.70	140.00	137.63	6.41	204.07	262.09	269.54	205.01
Tpost20	333.71	320.93	323.36	278.35	258.10	253.28	7.73	406.16	453.05	455.62	359.87

Table 10. Compression results (b/v) with *no* quantization. The cost of signed exponents (Exp) is included in H-Cgi and A-Cgi. Before compression, Tpost10 is 416 b/v and Tpost20 is 736 b/v. We partitioned each dataset into 512 clusters and the TSP computation used TSP-MST. We compare our results with Gzip, 8-bit adaptive and 8-bit static arithmetic coding (AC(A) and AC(S) respectively) on the original input data. We also compare our results with the lossless point-cloud method *ILS-point*.

	A-Cg10	A-Cg5	A-Cg1	F-Gzip	F-AC(A)	F-AC(S)
Tpost10	145.83	155.18	162.51	240.49	207.17	191.65
+perm	152.25	161.61	168.94			
Tpost20	274.49	280.92	309.36	417.03	356.06	330.35
+perm	280.92	287.35	315.79			

Table 12. Compression results (b/v) with 32-bit quantization. Before compression, Tpost10 is 416 b/v and Tpost20 is 736 b/v. Our approaches used 512 clusters and TSP-MST. We compare with the results of Flipping where the flipping errors were encoded by Gzip (F-Gzip) and by 8-bit adaptive/static arithmetic coding (F-AC(A) and F-AC(S) respectively).

	A-Cg20	A-Cg10	A-Cg5	A-Cg1	F-Gzip	F-AC(A)	F-AC(S)
Tpost10	N/A	105.60	105.16	105.05	166.17	207.15	148.92
+perm	N/A	112.41	111.98	111.87			
Tpost20	192.24	195.78	195.15	194.16	289.03	384.95	260.39
+perm	199.04	202.60	201.95	200.96			

Table 13. Compression results (b/v) with 24-bit quantization. Before compression, Tpost10 is 312 b/v and Tpost20 is 552 b/v. Our approaches used 512 clusters and TSP-MST. We compare with the results of Flipping where the flipping errors were encoded by Gzip (F-Gzip) and by 8-bit adaptive/static arithmetic coding (F-AC(A) and F-AC(S) respectively).

	A-Cg10	A-Cg5	A-Cg1	ILS-flip
Tpost10	137.63	140.00	149.70	214.42
+perm	150.92	153.29	162.99	
Tpost20	253.28	258.10	278.35	370.71
+perm	266.57	271.39	291.64	

Table 11. Compression results (b/v) with *no* quantization. Before compression, Tpost10 is 416 b/v and Tpost20 is 736 b/v. Our approaches used 512 clusters and TSP-MST. We compare with the results of lossless flipping, ILS-flip.

our approach gives the best balance between compression ratios and compression speed: a partition with cluster size of about 100–200 vertices, TSP-MST for vertex re-ordering, and arithmetic CGreedy (A-Cg) and CGreedy$i$ (A-Cgi) respectively for steady-state and time-varying mantissa-difference encoding. Our technique achieves superior compression ratios with reasonable encoding times and fast (linear) decoding times. We also show how to integrate our geometry coder with the state-of-the-art connectivity coders, and how to reduce the integration overhead by compressing the permutation sequence.

One novel feature of our geometry coder is that it does not need any connectivity information. This makes it readily applicable to the compression of *point-cloud* data, which is becoming increasingly important recently. Our on-going work is to pursue this research direction; some preliminary results of this follow-up work are given in [3].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the valuable comments that resulted in a great improvement of this paper. We thank Martin Isenburg for the ILS code [23] that was used in ILS-point and ILS-flip for the experiments. We thank Cláudio Silva, Han-Wei Shen, Peter Williams and Prashant Chopra for providing the test data used in this paper. These datasets are also courtesy of Lawrence Livermore National Lab, NASA, and Vtk [36].

REFERENCES

[1] P. Alliez and M. Desbrun. Valence-driven connectivity encoding for 3D meshes. *Computer Graphics Forum*, 20(3):480–489, 2001. Special Issue for Eurographics '01.

[2] S. Bryson, D. Kenwright, and M. Cox. *Exploring gigabyte datasets in real time: algorithms, data management, and time-critical design*. ACM SIGGRAPH course note, 1997.

[3] D. Chen, Y.-J. Chiang, and N. Memon. Lossless compression of point-based 3D models. In *Proc. Pacifi c Graphics*, pages 124–126, 2005.

[4] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Optimal alphabet partitioning for semi-adaptive coding of sources with unknown sparse distributions. In *Proc. Data Compression*, pages 372–381, 2003.

[5] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Geometry compression of tetrahedral meshes using optimized prediction. In *Proc. European Conference on Signal Processing*, 2005.

[6] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Optimized prediction for geometry compression of triangle meshes. In *Proc. Data Compression*, pages 83–92, 2005.

[7] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Lossless geometry compression for steady-state and time-varying irregular grids. In *Proc. Eurographics/IEEE Symposium on Visualization (EuroVis '06)*, pages 275–282, May 2006.

[8] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Alphabet partitioning techniques for semi-adaptive Huffman coding of large alphabets. *IEEE Transactions on Communications*, 55(3):436–443, 2007.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.

[10] M. F. Deering. Geometry compression. In *Proc. ACM SIGGRAPH*, pages 13–20, 1995.

[11] O. Devillers and P.-M. Gandoin. Geometric compression for interactive transmission. In *Proc. Visualization '00*, pages 319–326, 2000.

[12] P.-M. Gandoin and O. Devillers. Progressive lossless compression of arbitrary simplicial complexes. *ACM Trans. Graphics*, 21(3):372–379, 2002. Special Issue for SIGGRAPH '02.

[13] S. Gumhold, S. Guthe, and W. Straser. Tetrahedral mesh compression with the cut-border machine. In *Proc. Visualization '99*, pages 51–58, 1999.

[14] S. Gumhold, Z. Karni, M. Isenburg, and H.-P. Seidel. Predictive point-cloud compression. In *ACM SIGGRAPH Sketches*, 2005.

[15] S. Gumhold and W. Straser. Real time compression of triangle mesh connectivity. In *Proc. ACM SIGGRAPH*, pages 133–140, 1998.

[16] H. Hoppe. Efficient implementation of progressive meshes. *Computers & Graphics*, 22(1):27–36, 1998.

[17] Y. Huang, J. Peng, C.-C. J. Kuo, and M. Gopi. Octree-based progressive geometry coding of point clouds. In *Proc. Eurographics Symposium on Point Based Graphics (SPBG '06)*, pages 103–110, 2006.

[18] R. Hunter and A. H. Robinson. International digital facsimile standards. *Proceedings of the IEEE*, 68(7):855–865, 1980.

[19] M. Isenburg and P. Alliez. Compressing hexahedral volume meshes. In *Proc. Pacifi c Graphics*, 2002.

[20] M. Isenburg and P. Alliez. Compressing polygon mesh geometry with parallelogram prediction. In *Proc. Visualization*, pages 141–146, 2002.

[21] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. *ACM Trans. Graphics*, 22(3):935–942, 2003. Special Issue for SIGGRAPH '03.

[22] M. Isenburg, P. Lindstrom, S. Gumhold, and J. R. Shewchuk. Streaming compression of tetrahedral volume meshes. In *Proc. Graphics Interface (GI '06)*, pages 115–121, June 2006.

[23] M. Isenburg, P. Lindstrom, and J. Snoeyink. Lossless compression of predicted floating-point geometry. *Computer-Aided Design*, 37(8):869–877, 2005.

[24] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. In *Proc. ACM SIGGRAPH*, pages 279–286, 2000.

[25] A. Khodakovsky, P. Schröder, and W. Sweldens. Progressive geometry compression. In *Proc. ACM SIGGRAPH*, pages 271–278, 2000.

[26] B. Kronrod and C. Gotsman. Optimized compression for triangle mesh geometry using prediction trees. In *Proc. Sympos. on 3D Data Processing, Visualization and Transmission*, pages 602–608, 2002.

[27] H. Lee, P. Alliez, and M. Desbrun. Angle-analyzer: A triangle-quad mesh codec. *Computer Graphics Forum*, 21(3):383–392, 2002. Special Issue for Eurographics '02.

[28] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics (Special Issue for Vis '06)*, 12(5):1245–1250, 2006.

[29] N. D. Memon, K. Sayood, and S. S. Magliveras. Lossless image compression with a codebook of block scans. *IEEE Journal on Selected Areas of Communications*, 13(1):24–31, January 1995.

[30] MPEG. Information technology—generic coding of moving pictures and associated audio information: video. MPEG: ISO/IEC 13818-2:1996(E), 1989.

[31] R. Pajarola, J. Rossignac, and A. Szymczak. Implant sprays: compression of progressive tetrahedral mesh connectivity. In *Proc. Visualization*, pages 299–306, 1999.

[32] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Rostrand Reinhold, 1993.

[33] J. J. Rissanen. Universal coding, information, prediction and estimation. *IEEE Transactions on Information Theory*, 30:629–636, 1984.

[34] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Trans. Visualization Computer Graphics*, 5(1):47–61, 1999.

[35] R. Schnabel and R. Klein. Octree-based point-cloud compression. In *Proc. Eurographics Symposium on Point Based Graphics (SPBG '06)*, pages 111–120, 2006.

[36] W. Schroeder, K. Martin, and W. Lorensen. *The Visualization Toolkit*. Prentice-Hall, 1996.

[37] A. Szymczak and J. Rossignac. Grow & fold: compression of tetrahedral meshes. In *Proc. Solid Modeling and Applications*, pages 54–64, 1999.

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

[39] C. Touma and C. Gotsman. Triangle mesh compression. In *Proc. Graphics Interface*, pages 26–34, 1998.

[40] C.-K. Yang, T. Mitra, and T.-C. Chiueh. On-the-fly rendering of losslessly compressed irregular volume data. In *Proc. Visualization '00*, pages 101–108, 2000.