Efficient Algorithms for Mapping Cell Quantities between Overlapped 3-D Unstructured Meshes *

Andrew Kuprat† Stewart J. Mosso‡

Abstract

We develop fast algorithms for mapping cell fields between unstructured meshes that cohabit the same three-dimensional geometric domain. To do this, we compute a sparse matrix of intersection volumes V_{ij} between the j'th element of the "source" mesh and the i'th element of the "destination" mesh in $N \log N$ time, where N is the number of nonzero V_{ij}. This need only be done once, and then a sparse matrix-vector multiply can be rapidly performed once for each field to be mapped. The option exists to make the mappings conservative to round-off error. The algorithms are presented in detail and concrete numerical examples are presented.

1 Introduction

When performing a computational physics simulation in a given physical domain, it may be that two independent unstructured grids are defined on this domain and it may be necessary to map quantities between these meshes.

One situation occurs when "multi-physics" is present. That is, if there are multiple meshes in use for simulation of different kinds of physical processes occuring at the same time, it may be necessary to map quantities between the various meshes. This case is encountered in the Truchas casting simulation [1], where a simplicial (i.e., tetrahedral) mesh is required to simulate electromagnetism with the resulting computed inductive heating quantities (defined on the simplicial mesh) having to be mapped in a conservative fashion onto an unstructured hexahedral mesh that is used for computation of heat-transfer, phase change, and thermomechanical effects.

Another situation occurs during "restarts". If a physics simulation is run over a time interval (say $[T_1, T_2]$), it may have to be restarted on a different mesh in order to simulate the next time interval of interest (i.e., $[T_2, T_3]$). It may be that the mesh was deformed over the first time interval and was no longer usable, or that the physical phenomena occuring in the two time intervals are different enough in character that a new mesh is called for. This again happens in the Truchas code where a temperature field is passed between subsequent stages of simulation which describe different stages of the casting process.

Previous research has involved mappings between a 3-D unstructured mesh and a 3-D Cartesian mesh [3] which is from an algorithmic perspective a much simpler task than the task of mapping

*This is Los Alamos National Laboratory Report LA-UR-05-5084. This research is supported by the Department of Energy under contract W-7405-ENG-36.

†Theoretical Division, Group T-1, Mailstop B-221, Los Alamos National Laboratory, Los Alamos, NM 87545 (kuprat@lanl.gov).

‡Sandia National Laboratory (sjmosso@sandia.gov).

between two fully 3-D unstructured meshes. In [4], techniques are given for mapping quantities between a pair of unstructured 2-D surface meshes. An excellent discussion is given in that reference of the quandary that arises when mesh geometries do not match exactly and it is impossible to both preserve constant fields and be exactly conservative in the mapping process.

In this paper, we present an algorithm for rapidly and accurately mapping cell-based quantities from a source mesh potentially consisting of hexahedra ("hexes"), prisms, pyramids, or tetrahedra ("tets") to a destination mesh occupying the same domain, which also consists of hexes, prisms, pyramids, or tets.

2 Theory

Our task is to map a cell-based function defined on unstructured elements of 'Mesh A' to a cell-based function defined on unstructured 'Mesh B' quickly and in a conservative or near-conservative fashion. We assume the elements of both meshes come from the usual 'zoo' of 3-D elements: hexahedra, prisms, pyramids, and tetrahedra. Let $f^B(\mathbf{x})$ be the function defined by the n_B cell values for the B mesh and $f^A(\mathbf{x})$ be our function defined by the n_A cell values for the A mesh. Here $\mathbf{x} \in \Omega_B = \bigcup_{i=1}^{n_B} B_i$, the computational domain defined by the elements of the mesh B. This domain is assumed to be face-connected. I.e., we assume that between any two elements in mesh B, say $B_\alpha$and B_β, there is a path

$$B_\alpha = B_{i_1}, B_{i_2}, B_{i_3}, \ldots, B_{i_m} = B_\beta$$

such that B_{i_k} and $B_{i_{k+1}}$ share a (triangular or quadrilateral) face for all $1 \le k \le m-1$. Similarly, we assume the function $f^A(\mathbf{x})$ is defined on a domain $\Omega_A = \bigcup_{j=1}^{n_A} A_j$ which is the union of face-connected elements that comprise mesh A. For purposes of our mapping algorithm, we assume $\Omega_A \subseteq \Omega_B$. This condition will be slightly relaxed later.

Let f_i^B be the cell values of f^B on the mesh B, and f_j^A be the cell values of f^A on the mesh A. Let χ_{B_i} be the characteristic function for element B_i which is defined by

$$\chi_{B_i}(\mathbf{x}) = \begin{cases} 1, & \mathbf{x} \in B_i \\ 0, & \mathbf{x} \notin B_i \end{cases} .$$

Similarly, let χ_{A_j} be the characteristic function for element A_j. In this notation, $f^B(\mathbf{x}) = \sum_i f_i^B \chi_{B_i}(\mathbf{x})$ and $f^A(\mathbf{x}) = \sum_j f_j^A \chi_{A_j}(\mathbf{x})$. (This conveniently defines $f^B(\mathbf{x}) \equiv 0,\ \mathbf{x} \notin \Omega_B$ and $f^A(\mathbf{x}) \equiv 0,\ \mathbf{x} \notin \Omega_A$.) For our map to be exactly conservative,

$$\int_{\Omega_B} f^B(\mathbf{x})\, dV = \int_{\Omega_A} f^A(\mathbf{x})\, dV. \tag{1}$$

Obviously if $f^B(\mathbf{x}) = f^A(\mathbf{x})\ \forall \mathbf{x}$, this would be satisfied. Of course, this is not possible in general, but since we have n_B undetermined cell values f_i^B to work with, we can force $f^B = f^A$ in the "weak sense" by requiring

$$\int_{\Omega_B} f^B \chi_{B_i}\, dV = \int_{\Omega_A} f^A \chi_{B_i}\, dV, \qquad 1 \le i \le n_B. \tag{2}$$

These n_B equations will fix the n_B unknown coefficients f_i^B. Indeed from (2) we have

$$\int_{\Omega_B} \sum_k f_k^B \chi_{B_k} \chi_{B_i}\, dV = \int_{\Omega_A} \sum_j f_j^A \chi_{A_j} \chi_{B_i}\, dV.$$

So

$$|B_i| f_i^B = \sum_j |B_i \cap A_j| f_j^A,$$

where $|B_i|$ denotes the volume of B_i and $|B_i \cap A_j|$ denotes the volume of the intersection of element B_i with element A_j. This implies that we should set

$$f_i^B = \frac{1}{|B_i|} \sum_j |B_i \cap A_j| f_j^A. \tag{3}$$

Thus, with this choice for f_i^B, we have (2) obeyed $\forall B_i$, and summing over $1 \le i \le n_B$, we obtain the desired conservation condition (1).

Notice now that if $\Omega_A = \Omega_B$, then $\sum_j |B_i \cap A_j| = |B_i|$, so that (3) defines f_i^B as a *volume-weighted average of the* f_j^A. This weighted average has two desirable properties: (i) it is *local*, so that the f_i^B do not unphysically depend on f_j^A values that are not in the neighborhood of B_i. (ii) it is an average with non-negative weights, so that no new unphysical maxima and minima are created in any neighborhood. That is,

$$\min_{A_j \cap B_i \neq \emptyset} f_j^A \le f_i^B \le \max_{A_j \cap B_i \neq \emptyset} f_j^A, \quad 1 \le i \le n_B. \tag{4}$$

In practice, our algorithm often only computes approximate values for the intersection

$$V_{ij} \approx |B_i \cap A_j|. \tag{5}$$

We have three choices then. First, we could use (3) and perform the replacement (5) to obtain

$$f_i^B = \frac{1}{|B_i|} \sum_j V_{ij} f_j^A. \tag{6}$$

This produces a cell field $\{f_i^B\}$ that is neither conservatively mapped (1) or a weighted average, but is what we call a "raw" mapping. A second choice is to enforce conservation by multiplying the V_{ij} by a suitable factor. In fact, if we replace V_{ij} by $V_{ij} \frac{|A_j|}{\sum_m V_{mj}}$ in (6), to obtain

$$f_i^B = \frac{1}{|B_i|} \sum_j V_{ij} \frac{|A_j|}{\sum_m V_{mj}} f_j^A, \tag{7}$$

then we have that conservation has been restored:

$$\begin{aligned}
\int_{\Omega_B} f^B \, dV &= \sum_i |B_i| f_i^B \\
&= \sum_i |B_i| \left(\frac{1}{|B_i|} \sum_j V_{ij} \frac{|A_j|}{\sum_m V_{mj}} f_j^A \right) \\
&= \sum_j |A_j| \frac{\sum_i V_{ij}}{\sum_m V_{mj}} f_j^A \\
&= \int_{\Omega_A} f^A \, dV.
\end{aligned}$$

The third option is to keep our mapping's "weighted average" property that preserves maxima and minima. Thus we replace (6) by

$$f_i^B = \frac{1}{\sum_k V_{ik}} \sum_j V_{ij} f_j^A. \tag{8}$$

With this choice, assuming that our approximate V_{ij} are all non-negative and are only positive for those j where the corresponding A_j intersect B_i, we have preserved the desirable local minima / local maxima preserving property (4).

Our algorithm allows the user to specify all three choices (raw, conservative, and weighted average) and in practice we have found weighted average is the most-used choice.

Our strategy is to compute the values V_{ij} exactly when both A_j and B_i are convex planar polyhedra using the robust algorithm described in Section 3.4. If however A_j or B_i have some nonplanar faces (the chief case being when the faces are bilinear as part of a trilinear hexahedral element), the computation of the exact volume would be prohibitively expensive. Instead we 'planarize' any nonplanar faces (i.e., approximate the nonplanar facets with planar ones) and then compute the exact intersection volumes $V_{ij} \equiv |B_i' \cap A_j'|$, where B_i' and A_j' are the 'planarized' versions of B_i and A_j.

3 Algorithms

An efficient algorithm to map cell-based functions between unstructured meshes requires efficient evaluation of the approximate volumes in (5). This task is naturally broken up into two parts:

Find intersections: Find out for which i, j we have $V_{ij} \neq 0$. (Here we assume the approximations V_{ij} will only be nonzero if the exact volumes $|B_i \cap A_j|$ are nonzero.) V_{ij} is a sparse matrix: it has far fewer than $n_B \cdot n_A$ nonzero entries. Looping over all $n_B \cdot n_A$ entries would be fatally inefficient and is unnecessary. Our approach has complexity $\mathcal{O}(N \log N)$ where N is the number of nonzero V_{ij}.

Compute Intersections: For nonzero $\widetilde{V_{ij}}$, compute the intersection volumes exactly if the elements are planar and approximately if they are nonplanar.

3.1 Finding Intersections

The idea of our algorithm is to use the assumption of face-connectedness of both unstructured meshes. This allows us to traverse the A and B meshes simultaneously, finding the nonzero V_{ij} values along the way. We first start by putting A_1 (i.e. element number one from mesh A) onto a stack. Now we proceed by popping off the stack the element on top of the stack (call it A_j) and we say that we have "visited" this element; we then place back on the stack all unvisited face-neighbors of A_j. We deal with A_j by finding all B_i such that $B_i \cap A_j \neq \emptyset$ and computing the corresponding approximate intersection volumes V_{ij}. Initially, for A_1, we start with B_1 and walk along the B mesh using the B mesh connectivity until we find a B_i such that $A_1 \cap B_i \neq \emptyset$. Then we do some more walking in the B mesh in the neighborhood of B_i until we have discovered all the B_i such that $A_1 \cap B_i \neq \emptyset$. We continue by popping another element A_j off the stack. We then walk towards this new element on the B mesh starting not from B_1 but from the last element B_i that had a nonzero intersection with the previously considered element from the A mesh.

We thus are effectively performing a coordinated walk on the A and B meshes, and this should have a complexity $\mathcal{O}(N)$, where N is the number of nonzero V_{ij}. After all the elements of mesh

A have been visited, we reorder the volume contributions V_{ij} so that they can be put in row-packed sparse matrix form. This reordering process employs a heapsort [2] and takes $\mathcal{O}(N \log N)$ operations.

Algorithm 1 gives the 'outer' algorithm where a 'walk' is performed on the face-connected A mesh.

The subroutine `walk_mesh_pt` (Algorithm 2) accomplishes the task of "walking" from a starting element B along a path of face-connected elements and ending at a new element B that contains the point $\mathbf{x}$.

The idea of this algorithm is that given a current element B, the point is tested against all the faces of B. (Note: in the case of nonplanar bilinear faces, we actually test $\mathbf{x}$ against the plane that passes through the midpoints of the 4 linear edges bounding the face. It is easily shown that the 4 midpoints are coplanar.) If $\mathbf{x}$ is on the wrong side of face k, then B doesn't contain $\mathbf{x}$ and we must move to a new element which becomes the "new" B. We usually move towards $\mathbf{x}$ by choosing the neighboring element across face k. Since nonconvex domains may necessitate somewhat circuitous walking paths, we prove the following lemma.

Algorithm 1: Compute_IntVols

`Compute_IntVols(Mesh_A,Mesh_B,IntVols)`
[Output `IntVols` is a sparse matrix representation of the
approximate overlap volumes $V_{ij} \approx \text{volume}(B_i \cap A_j)$ for elements
B_i in `Mesh_B` and A_j in `Mesh_A`]

`vol_list`$\leftarrow \emptyset$
[`vol_list` will be a list of 3-tuples (i, j, V_{ij})
which will grow as intersection volumes are computed]
`OnStackA`$(A_j) \leftarrow$`.false.` for all elements A_j in `Mesh_A`
Stack $S_A \leftarrow \emptyset$
`SeedEltB`$\leftarrow B_1$ [first element of `Mesh_B`]
Put Element A_1 [first element of `Mesh_A`] on stack S_A
`OnStackA`$(A_1) \leftarrow$ `.true.`
Do while stack S_A nonempty
    Pop Element A_j off stack S_A
    Loop over the faces k of Element A_j
        If there is an element E_{opp} sharing face k with A_j then
        If (`.not.OnStackA`(E_{opp})) then
            Put E_{opp} on stack S_A
            `OnStackA`$(E_{\text{opp}}) \leftarrow$ `.true.`
    $\mathbf{x}_A \leftarrow$ centroid of vertices of Element A_j
    Call `walk_mesh_pt`(`SeedEltB`, `Mesh_B`, $\mathbf{x}_A$)
    [Walk on `Mesh_B`, updating `SeedEltB`, until $\mathbf{x}_A \in$ `SeedEltB`]
    Call `get_vols_around_elt`(A_j, `SeedEltB`, `Mesh_B`, `vol_list`)
    [Append to `vol_list` 3-tuples (i, j, V_{ij}) where $V_{ij} \approx \text{volume}(B_i \cap A_j)$ for
    $B_i =$`SeedEltB` and other B_i in the neighborhood of `SeedEltB` that intersect A_j]
Sort `vol_list` into sparse row-packed matrix `IntVols`
[Use heapsort to sort (i, j, V_{ij}) so that i's are in increasing order (and
if equal i, then j's are in increasing order).]
Return

Algorithm 2: Walk_mesh_pt [Walking on `Mesh` to point x]

`Walk_mesh_pt` (E,`Mesh`,$\mathbf{x}$)

Initially place E on stack S
Do while S nonempty
    Pop E off stack S
    `viable` $\leftarrow$ `.true.`
    $E_{\text{top}} \leftarrow 0$
    Loop over the faces k of Element E
        If (`viable`) then
            If there is an element E_{opp} sharing face k with E then
                If $\mathbf{x}$ on "wrong" side of face k (so that $\mathbf{x} \notin E$) then
                    `viable` $\leftarrow$ `.false.`
                    If E_{opp} was never put on stack S then
                        $E_{\text{top}} \leftarrow E_{\text{opp}}$ [E_{top} will be neighbor of E put on S *last* so it will be the *next* element considered]
                Else
                    If E_{opp} was never put on S then
                        Put E_{opp} on S
            Else
                If $\mathbf{x}$ on wrong side of face i then
                    `viable` $\leftarrow$ `.false`
        Else
            If there is an element E_{opp} sharing face k with E then
                If E_{opp} was never put on S then
                    Put E_{opp} on S
    If `viable` then
        Return [success: $\mathbf{x} \in E$]
    Else
        If $E_{\text{top}} \neq 0$ then
            Put E_{top} on S
$E \leftarrow 0$ [failure]
Return

Algorithm 3: Get_vols_around_elt [Walking on `Mesh_B` to find intersection volumes with element A_j from `Mesh_A`]

`Get_vols_around_elt`(A_j,`SeedEltB`,`Mesh_B`,`vol_list`)

$\mathcal{P} \leftarrow$ `planarization`(A_j)
[Replace curved faces of A_j with interpolating planes]
Initially place `SeedEltB` on stack S_B
Do while stack S_B nonempty
    Pop element B_i off stack S_B
    Loop over `nfac` faces k of B_i
        Approximate face k with plane P_k
    [Now $B_i \approx \bigcap_k^{\texttt{nfac}} D_k$ where D_k is the halfspace bounded by P_k]
    `PlaneCuts`(k) $\leftarrow$ `.false.`, $\quad 1 \le k \le$ `nfac`
    `emptyintersection` $\leftarrow$ `.false.`
    $\mathcal{P}_{\text{int}} \leftarrow \mathcal{P}$
    Do $k = 1$, `nfac`
        If $D_k \cap A_j = A_j$ then
            cycle [immediately start next iteration of 'do loop']
        Else if $D_k \cap A_j = \emptyset$ then
            `emptyintersection` $\leftarrow$ `.true.`
            exit [exit 'do loop']
        Else
            `PlaneCuts`(k) $\leftarrow$ `.true.`
            Call `Plane_Poly_Int3D`($P_k, \mathcal{P}_{\text{int}}, \mathcal{P}_{\text{tmp}}$)
            [Computes intersection polyhedron $\mathcal{P}_{\text{tmp}} = \mathcal{P}_{\text{int}} \cap D_k$]
            $\mathcal{P}_{\text{int}} \leftarrow \mathcal{P}_{\text{tmp}}$
            If $\mathcal{P}_{\text{int}} = \emptyset$ then
                `emptyintersection` $\leftarrow$ `.true.`
                exit [exit 'do loop']
    If `emptyintersection` = `.false.` then
        `NewVol`$\leftarrow$ `Volm_Poly3D`($\mathcal{P}_{\text{int}}$) [Volume of $\mathcal{P}_{\text{int}}$]
        Append 3-tuple ($i, j,$ `NewVol`) to `Vol_list`
        Do $k = 1,$ `nfac`
            If (`PlaneCuts(`k`)`) then
                If there is an element sharing face k with B_i then
                    $E_{\text{opp}} \leftarrow$ element opposite from face k of B_i
                    If E_{opp} never before placed on stack S_B then
                        Place E_{opp} on stack S_B
Return

Algorithm 4: Map_cell_field [Map field f^A to field f^B]

`Map_cell_field` (f^A, f^B, `IntVols`, `exactly_conservative`, `preserve_constants`)

[The sparse matrix `IntVols` is assumed stored as follows:
There are 3 arrays: `rowoff, col,` and `vol`.
Intersection volumes pertaining to B_i (row i) are in
array subset `vol(rowoff(i):rowoff(i+1)-1)`
For k'th element of `vol`, we have that `col(k)` gives the
column number $j =$ `col(k)`, pertaining to A_j, so that
`vol(k)`$= V_{ij} \approx \text{volume}(B_i \cap A_j)$.]

[We assume the following quantities are available:
$|A_j| = \text{volume}(A_j)$ $1 \le j \le n_A$
$|B_i| = \text{volume}(B_i)$ $1 \le i \le n_B$
`RowSum`$(i) = \sum_{j=1}^{n_A} V_{ij}$ $\quad 1 \le i \le n_B$
`ColSum`$(j) = \sum_{i=1}^{n_B} V_{ij}$ $\quad 1 \le j \le n_A$]

[We assume `exactly_conservative` and `preserve_constants` are not both `.true.`]

Do $i = 1, n_B$
  $f_i^B \leftarrow 0$
  Do $k =$ `rowoff`(i), `rowoff`$(i+1) - 1$
    $j \leftarrow$ `col`(k)
    If (`exactly_conservative`) then
      $f_i^B \leftarrow f_i^B +$ `vol`$(k) \times \frac{|A_j|}{\texttt{ColSum}(j)} \times f_j^A$
    Else
      $f_i^B \leftarrow f_i^B +$ `vol`$(k) \times f_j^A$
  If (`preserve_constants`) then
    $f_i^B \leftarrow f_i^B /$`RowSum`(i)
  Else
    $f_i^B \leftarrow f_i^B / |B_i|$
Return

Lemma 1. In Algorithm 2, if $\mathbf{x}$ is in some element, we are given a valid starting element, and there is an element containing $\mathbf{x}$, Algorithm 2 will successfully find the element containing $\mathbf{x}$.
Proof: Since we assume the mesh is face-connected, there is a face-connected path

$$E = E_0, E_1, E_2, \ldots, E_n \ni \mathbf{x} \tag{9}$$

from the starting element E to E_n which should be the new E returned by the algorithm. Algorithm 2 uses a stack S and initially E_0 is placed on the stack. Every time an element E is popped off the stack, it is checked for the property $\mathbf{x} \in E$, and if true, we are done. Otherwise, all the neighbors of E that have never been on the stack before are put on the stack. Since an element can only be placed at most once on the stack, the only way the algorithm can fail is that eventually the stack is empty and E_n has not been found. In this case, there is a finite set of elements that were visited before the algorithm failed. This set is

$$\mathcal{E}_{\text{visited}} = \{E \mid E \text{ was popped off } S\}.$$

Let E_i be the element in the sequence in (9) that belongs to $\mathcal{E}_{\text{visited}}$ and has maximal index in the sequence in (9). When E_i was popped off the stack, E_{i+1} was either placed on the stack because E_i and E_{i+1} are face-neighbors or it wasn't because it had previously been placed on the stack. Either way, this contradicts the assumption that $E_i \in \mathcal{E}_{\text{visited}}$ has maximal index. Q.E.D.

It is similarly proven that in Algorithm 1, the traversal of Ω_A using the stack S_A successfully visits all elements in mesh A once and terminates.

We call Algorithm 3 with `SeedEltB` equal to the element B_i that was found to contain element A_j by Algorithm 2. For the case that A_j and all B_i are planar and convex, we prove the following lemma and defer discussion of the nonplanar case to section 3.3.

Lemma 2. In Algorithm 3, for the case that A_j and all B_i are planar, convex polyhedra, and assuming that $A_j \cap$ `SeedEltB` $\neq \emptyset$, we have that all intersections of A_j with Mesh B will be found and deposited into `vol_list`.
Proof: Since we assume A_j is planar and convex, the `planarization` operation does nothing, so that $\mathcal{P} = A_j$. `SeedEltB` is initially placed on stack S_B. The algorithm works by popping an element B_i from Mesh B off the stack, computing the intersection volume $V_{ij} = |B_i \cap A_j|$ and then placing this (i, j, V_{ij}) information into `Vol_list` if $V_{ij} > 0$. The element B_i is stored as the intersection of `nfac` half-spaces. That is, $B_i = \cap_{k=1}^{\texttt{nfac}} D_k$, where D_k is the half-space bounded by P_k which is the plane containing the k'th boundary face. The intersection is performed using routine `Plane_Poly_Int3D` which intersects a half-space with a planar convex polyhedron. B_i is assigned to $\mathcal{P}_{int}$, the "intersection polyhedron", and this polyhedron is then intersected in order with $D_1, D_2, \ldots, D_{\texttt{nfac}}$, each intersection resulting in a potentially smaller $\mathcal{P}_{int}$. At the end of this process, if $\mathcal{P}_{int} \neq \emptyset$, routine `Volm_Poly3D` computes the volume of $\mathcal{P}_{int} = B_i \cap A_j$ by decomposition into tetrahedra. If $V_{ij} > 0$, then for each face k of B_i, we place the adjacent element E_{opp} on S_B if and only if (i) the element E_{opp} actually exists and has never been placed on the stack before, and (ii) the plane P_k containing the common boundary facet k between B_i and E_{opp} nontrivially divides A_j. (Condition (ii) is checked by seeing if all the nodes of A_j lie on either side of P_k.) Clearly, if P_k does not intersect A_j, then it is impossible for both $B_i \cap A_j \neq \emptyset$ and $E_{opp} \cap A_j \neq \emptyset$. Since $B_i \cap A_j \neq \emptyset$, we have that $E_{opp} \cap A_j = \emptyset$, and we are justified in not putting E_{opp} onto the stack. Proceeding in this fashion, it is clear that this algorithm will find all intersections $V_{ij} = |B_i \cap A_j|$ if the set

$$\mathcal{B}_{A_j} = \{B_i \mid |B_i \cap A_j| > 0\}$$

is face-connected. Now if B_{i_1} and B_{i_2} are in $\mathcal{B}_{A_j}$, let $\mathbf{x}_1 \in \text{interior}(B_{i_1} \cap A_j)$ and $\mathbf{x}_2 \in \text{interior}(B_{i_2} \cap A_j)$. Then since A_j is connected, we can connect $\mathbf{x}_1$ to $\mathbf{x}_2$ by a curve Γ lying entirely within interior(A_j). Since $A_j \subseteq \cup_{i=1}^{n_B} B_i$, Γ traces a path from $\mathbf{x}_1$ to $\mathbf{x}_2$ entirely within Mesh B as well. If necessary, we can perturb Γ so that it doesn't intersect any of the vertices or edges in Mesh B and "cleanly" intersects the facets in Mesh B (i.e., so that the curve does not run tangentially within any facet for any positive distance). Since the original Γ lies entirely within the interior of A_j, and the necessary perturbations (to avoid vertex and edge intersections and to assure clean face intersections) can be arbitrarily small, the perturbed Γ can be assumed to lie entirely within the interior of A_j as well. In this case, all the B_i encountered by Γ when travelling from $\mathbf{x}_{i_1}$ to $\mathbf{x}_{i_2}$ form a face-connected sequence and all are members of $\mathcal{B}_{A_j}$. This shows that $\mathcal{B}_{A_j}$ is face-connected. Q.E.D.

Algorithm 4 is the "payoff" algorithm where we take our intersection volumes that have been previously computed in Algorithms 1-3 and use them to map a cell field f^A on mesh A to a cell field f^B on mesh B. At the end of Algorithm 1, the overlap volumes (i, j, V_{ij}) stored in `vol_list` were sorted into a row-packed sparse matrix `IntVols` which is actually three arrays `rowoff`, `col`, and `vol` which represent a standard row-packed sparse matrix data structure. We also assume that the volumes of the elements in both meshes and the column and row sums of the V_{ij} have been computed and are available. Algorithm 4 takes the input cell field f^A, along with `IntVols`, and the two logical flags `exactly_conservative`, and `preserve_constants` and then maps f^A to f^B by sparse matrix multiplication. If neither boolean flag is true, the "raw" mapping (6) is used. If `exactly_conservative` is `.true.` then we use the exactly conservative mapping (7). If `preserve_constants` is `.true.`, we use the weighted average mapping (8). (We call the flag `preserve_constants` because (8) preserves constants in the sense that if $f^A \equiv k$, then $f^B \equiv k$.)

It is not legal for both `exactly_conservative` and `preserve_constants` to be `.true.` Indeed, in order to obtain both of those properties simultaneously, an iteration would be necessary and in fact both those properties are inconsistent with each other unless Volume(Ω_A) = Volume(Ω_B) exactly.

3.2 Practical Geometry Considerations

In practice we cannot have $\Omega_A = \Omega_B$ exactly. For example, if we alternately discretize some sphere S using tetrahedra on mesh A and hexahedra on mesh B, we will not usually have $\Omega_A = \Omega_B$ and certainly not $\Omega_A = S$ or $\Omega_B = S$. This is assuming the tetrahedra have planar facets and the hexahedra have bilinear facets. If the boundary nodes of the hex and tet grids reside on ∂S, then vol$(S) >$ vol(Ω_A) and vol$(S) >$ vol(Ω_B). In fact, even with a large number of elements, although

$$\text{vol}(S) \approx \text{vol}(\Omega_A) \approx \text{vol}(\Omega_B),$$

probably none of these volumes will be exactly equal. In this scenario, parts of the tet mesh might slightly poke out of Ω_B. When Algorithm 1 calls Algorithm 2, it is called with the centroids $\mathbf{x}$ of some elements in Ω_A. (Here the "centroid" is the cheap-to-compute average position of the vertices of the element.) Since the centroid of A_j is at least a distance ir(A_j) from points on the surface of A_j (where ir(A_j) denotes the radius of the largest sphere inscribed in A_j centered on the centroid), we have that the element A_j can stick out of Ω_B by at least ir(A_j) and the algorithm will still work. So our formal assumption $\Omega_A \subseteq \Omega_B$ is relaxed so that elements of mesh A can stick out of Ω_B to this small degree.

3.3 Treatment of nonplanar faces

The polyhedral zoo of elements processed by our algorithm all have facets which are either planar triangles or bilinear quadradrilatals. The elements possessing some or all bilinear quads for facets are hexahedra, pyramids, and triangular prisms. The key subroutine `Plane_Poly_Int3D` described in Section 3.4 performs intersections of half-spaces (defined by oriented planes) with planar polyhedra. Thus we need to "planarize" any nonplanar facets before calling `Plane_Poly_Int3D`.

We proceed as follows. First it is easy to prove that for an element E and a bilinear quad facet k, there exists a "best fit" plane P_k that passes through the midpoints of all four edges of the facet as well as through the centroid of the four vertices. In fact, if the vertices of the quad are labelled $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4$ in cyclical order, the normal of this plane is

$$\hat{\mathbf{n}}_k = \pm \frac{(\mathbf{x}_2 - \mathbf{x}_4) \times (\mathbf{x}_3 - \mathbf{x}_1)}{||(\mathbf{x}_2 - \mathbf{x}_4) \times (\mathbf{x}_3 - \mathbf{x}_1)||}. \tag{10}$$

(We choose the sign so that $\hat{\mathbf{n}}_k$ is an "inward" normal pointing towards the interior of the element E.) The equation of the plane P_k is thus

$$\hat{\mathbf{n}}_k \cdot (\mathbf{x} - \mathbf{x}_k^{\text{cen}}) = 0,$$

where $\mathbf{x}_k^{\text{cen}}$ is the centroid of the vertices of the face, $\mathbf{x}_k^{\text{cen}} = (\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + \mathbf{x}_4)/4$. Now let D_k be the half-space bounded by P_k with normal $\hat{\mathbf{n}}_k$ pointing towards the interior of D_k. (If instead k is a planar triangular face, then D_k is simply the half-space bounded by the (correctly oriented) plane P_k containing face k.) We say the *planarization* of E is given by

$$\mathit{planarization}(E) = \cap_{k=1}^{\texttt{nfac}} D_k, \tag{11}$$

where `nfac` is the number of faces of E. $\mathit{planarization}(E)$ is our stand-in for E when intersection volumes need to be computed in Algorithm 3. The element A_j is initially planarized, and then it is intersected with the "best fit" planes of the faces of element B_i in order to compute the approximation $V_{ij} \approx B_i \cap A_j$. If E is planar, convex, then $\mathit{planarization}(E) = E$ and no harm is done. If E has slightly nonplanar facets and the approximating planes do not intersect with dihedral angles greater than π, then $\mathit{planarization}(E)$ will be a reasonably good approximation to E. If E has highly nonplanar facets or if the approximating planes P_k have some dihedral angle intersections that exceed π, then $\mathit{planarization}(E)$ will be a poor approximation to E so that the approximation V_{ij} may not be close to the true intersection volume, but this is a situation that should not occur for many elements in well-constructed meshes.

Now regarding our treatment of Algorithm 2, suppose two elements B_{i_1} and B_{i_2} share the same quad face. Say that face is k_1 in an ordering of the faces of E_{i_1} and say that face is k_2 in an ordering of the faces of E_{i_2}. Then we have that P_{k_1}=P_{k_2} (the planarizations of the two adjacent elements agree on their common face), $\hat{\mathbf{n}}_{k_1} = -\hat{\mathbf{n}}_{k_2}$, and $\{D_{k_1}, D_{k_2}\}$ form a nonoverlapping partition of the space $\mathbb{R}^3$. However, it does *not* follow from this that

$$\{\mathit{planarization}(B_1), \mathit{planarization}(B_2), \ldots, \mathit{planarization}(B_{n_B})\} \tag{12}$$

would form a nonoverlapping polyhedral decomposition of the domain $\Omega_B = \cup_i B_i$. In fact, there can be "holes" in the way (12) covers Ω_B in the neighborhood of edges. (See Figure 1.) This means that in Algorithm 2, if we interpret the statement

If $\mathbf{x}$ on "wrong" side of face k...

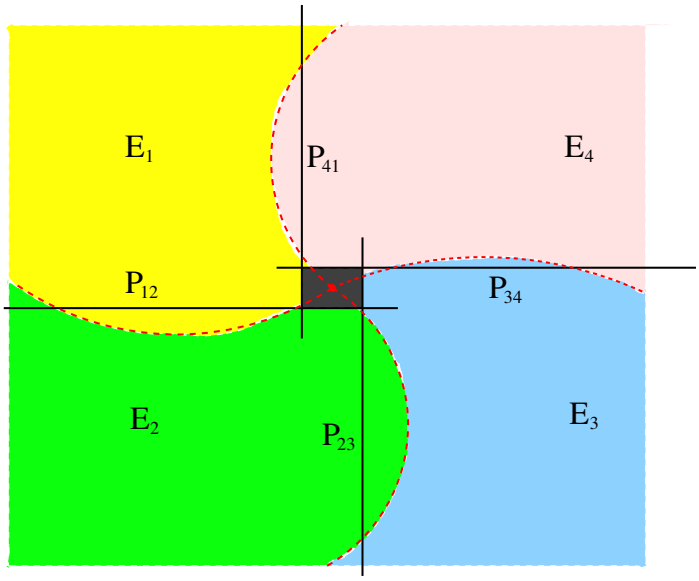

Figure 1: Hole created by planarization: Curved interface between elements E_1, E_2 is replaced by common "best fit" plane P_{12} and similarly for the other interfaces. This creates a hole—a region shown by the gray area that would not be considered to be within any element. (2-D schematic of the 3-D situation.)

to mean

$$\text{If } \mathbf{x} \text{ on "wrong" side of plane } P_k..., \tag{13}$$

then we have a rapid test for determining whether to traverse across a face, but it is conceivable that $\mathbf{x}$ in mesh A will not be locatable in mesh B, if it falls in one of the "holes", such as depicted in Figure 1. To prevent falling into holes, for the purposes of Algorithm 2, we use (13), but with respect to an element B and a face k, we move the position of P_k back so that all of face k is on the "correct" side of P_k. With this readjustment, the set (12) will form a cover of Ω_B with some overlaps. This means that in the neighborhood of a face, point $\mathbf{x}$ may be considered to be inside of more than one element. This is acceptable, since the first such element encountered will be returned by Algorithm 2 and will be used as the seed element for Algorithm 3.

3.4 Computing Intersections

This section describes our algorithm for computing the intersection volume $V_{ij} = |B_i \cap A_j|$ exactly in the case that both elements A_j from mesh A and B_i from mesh B are convex planar polyhedra. Two key algorithms are employed: `Plane_Poly_Int3D` intersects a half-space with a planar convex polyhedron and returns the intersection which, if nonempty, is itself a planar convex polyhedron. `Volm_Poly3D` computes the volume of a planar convex polyhedron by decomposing it into tetrahedra and then summing the volumes of the tetrahedra.

3.5 Weighted Average vs. Exactly Conservative

Since the "weighted average" mapping ((8) invoked with `preserve_constants` in Algorithm 4) has the desirable property of not creating any new local minima or maxima (4), it is the mapping we

have used most often in practice. A typical case for our Truchas code is that we have a set of "volume fraction" fields $f^{(1)}, f^{(2)}, \ldots, f^{(P)}$ that have the property

$$\sum_{p=1}^{P} f_i^{(p)} = 1, \quad \forall i. \tag{14}$$

Since "the sum of the weighted average" is the "weighted average of the sum", it is easy to prove that the mapped fields (when using the weighted average mapping (8)) will obey the "summing to 1" property (14) as well. This property will not be preserved by the "raw" or "exactly conservative" mappings.

Since it is tempting to use the non-conservative weighted average mapping in most situations, we compute a bound on the L_1 error between the conservative and weighted average mappings. Let $f(\mathbf{x})$ denote the source cell field on mesh A; we take this to be a density of some sort of substance and so make the assumption that $f(\mathbf{x}) \geq 0$. Let f^{cons} and f^{wa} be the conservative and weighted average fields derived over mesh B by using (7) and (8) respectively. Then

$$\begin{aligned}
\int_{\Omega_B} |f^{\text{cons}}(\mathbf{x}) - f^{\text{wa}}(\mathbf{x})| \, dV &= \sum_i |B_i| \left| f_i^{\text{cons}} - f_i^{\text{wa}} \right| \\
&= \sum_i |B_i| \left| \frac{1}{|B_i|} \sum_j V_{ij} \frac{|A_j|}{\sum_m V_{mj}} f_j - \frac{1}{\sum_k V_{ik}} \sum_j V_{ij} f_j \right| \\
&= \sum_i \sum_j \left| V_{ij} \left(\frac{|A_j|}{\sum_m V_{mj}} - \frac{|B_i|}{\sum_k V_{ik}} \right) f_j \right| \\
&\leq \sum_i \sum_j V_{ij} f_j \left(\left| \frac{|A_j|}{\sum_m V_{mj}} - 1 \right| + \left| \frac{|B_i|}{\sum_k V_{ik}} - 1 \right| \right) \\
&= \sum_j \left(f_j \sum_i V_{ij} \right) \left| \frac{|A_j|}{\sum_m V_{mj}} - 1 \right| + \sum_i \left(\sum_j V_{ij} f_j \right) \left| \frac{|B_i|}{\sum_k V_{ik}} - 1 \right| \\
&\approx \sum_j \left(\int_{A_j} f(\mathbf{x}) \, dV \right) \left| \frac{|A_j|}{\sum_m V_{mj}} - 1 \right| + \sum_i \left(\int_{B_i} f(\mathbf{x}) \, dV \right) \left| \frac{|B_i|}{\sum_k V_{ik}} - 1 \right| \\
&\equiv \mathbf{F}^A \cdot \mathbf{E}^A + \mathbf{F}^B \cdot \mathbf{E}^B
\end{aligned} \tag{15}$$

Here, we have defined

$$\mathbf{F}_j^A \equiv \int_{A_j} f(\mathbf{x}) \, dV \qquad \mathbf{F}_i^B \equiv \int_{B_i} f(\mathbf{x}) \, dV$$

as being, respectively, the integral of f over an element of Mesh A, and the integral of f over an element of Mesh B. We have defined

$$\mathbf{E}_j^A \equiv \left| \frac{|A_j|}{\sum_m V_{mj}} - 1 \right| \qquad \mathbf{E}_i^A \equiv \left| \frac{|B_i|}{\sum_k V_{ik}} - 1 \right|$$

as being, the relative volume errors caused by our decomposition of elements A_j in mesh A and elements B_i in mesh B, respectively.

Now since

$$\begin{aligned}
\left| \int_{\Omega_A} f(\mathbf{x}) \, dV - \int_{\Omega_B} f^{\text{wa}}(\mathbf{x}) \, dV \right| &= \left| \int_{\Omega_B} f^{\text{cons}}(\mathbf{x}) \, dV - \int_{\Omega_B} f^{\text{wa}}(\mathbf{x}) \, dV \right| \\
&\leq \int_{\Omega_B} |f^{\text{cons}}(\mathbf{x}) - f^{\text{wa}}(\mathbf{x})| \, dV,
\end{aligned}$$

we have that by (15), $\mathbf{F}^A \cdot \mathbf{E}^A + \mathbf{F}^B \cdot \mathbf{E}^B$ is also bound for the conservation error of f^{wa}.

So we can use f^{wa} and be sufficiently conserving if the relative volume error vectors $\mathbf{E^A}$ and $\mathbf{E^B}$ are sufficiently small. In fact, with our algorithm, the error vector entries will be identically zero for planar-faceted elements that are completely covered by planar-faceted elements from the other mesh. Errors creep in the more the elements are curved, and the more the elements from the two meshes fail to match up at the boundary. The worst case would be mapping a field f which is only nonzero near a curved boundary $\partial\Omega_A$ and where the characteristic element sizes in Ω_A, Ω_B differ greatly. In this case, the error vectors $\mathbf{E}^A$, $\mathbf{E}^B$ would be large (due to poor matching of the meshes at $\partial\Omega_A$) precisely on those elements where the entries of the integrated quantity vectors $\mathbf{F}^A$, $\mathbf{F}^B$ are nonzero. In fact in our Truchas code we have a field representing the creation of inductive heating near the surface of a conductor being mapped from a tet mesh to a hex mesh. Only for this field do we feel the need to use the conservative map f^{cons} rather than the weighted average f^{wa}.

4 Numerical Results

We show results of our mapping algorithm on a "curved pipe with slit" geometry. In Figure 2, we see a tet grid with 140651 elements on which a sample cell field $f((x, y, z)) = 1 + \sin(z)$ has been defined on the elements, where for each tet, the z value used to evaluate $f(z)$ for that tet is taken to be at the centroid of the tet. (In the figure, the geometry fits in the box $[0, 12] \times [0, 12] \times [-5, 5]$ and z is in the 'up' direction.) In Figure 3, the field f has been mapped conservatively to a cell field over a hex mesh of the same geometry. The hex mesh has 42496 elements, and the number of intersection volumes V_{ij} that are nonzero in the map between the two meshes is 1032910. Here we have used the `exactly_conservative` option, so the integral of the field is same 513.9857790995 on both meshes using double precision computation. Although both tet and hex meshes attempt to discretize the same geometry, the boundary curvature causes the volumes of the two meshes to be different: 529.68 on the tet-discretized region and 529.34 on the hex discretized region. Thus, to accomodate the mesh volume differences and still be exactly conservative, it is not surprising the the maximum value of the field is 2.0252 on the hex mesh, which represents just over a 1% overshoot of the maximum value 2.0000 on the tet mesh. We note that the minimum value on the hex mesh is 0.0006 which does not undershoot the minimum value of 0.0000 on the tet mesh. This is because by (7), the values of the mapped field using the `exactly_conservative` option are still a positive linear combination of local source field values, so that negative values cannot arise when mapping a non-negative field.

In Figure 4 we show the mapped field over the same hex mesh, where we have used the `preserve_constants` (weighted average) option. Here, the integral of the field over the destination mesh agrees only to 3 decimal places (it is 513.66); however, the global minima/maxima of the mapped field are 0.0006 and 1.9994 respectively, which are within the corresponding bounds of the source tet mesh.

The timings on an Apple G5 workstation with IBM XLF compiler are consistent with the claimed $N \log N$ scaling for the building of the sparse mapping matrix (Algorithms 1-3). Using the same curved pipe geometry we computed mappings between three pairs of grids, always mapping from an all-tet mesh to an all-hex mesh. The timings are given in Table 1. We see that the time for computing the sparse matrix V_{ij} is in fact growing linearly with N, taking about 25μs per overlap entry. We note that Algorithms 1-3 need only be called *once*. Then any number of fields may be mapped by calling Algorithm 4 for each field. Algorithm 4 is very cheap compared to Algorithms 1-3 and it in fact has complexity $\mathcal{O}(N)$. Comparing the second and third rows in Table 1, it appears that Algorithm 4 time complexity is superlinear. However, this simply results from

Tet Elements n_A	Hex Elements n_B	Nonzero V_{ij} N	Time Alg. 1-3	Time Alg. 4		
				raw	cons	wted avg
2450	792	16042	0.36s	0.14ms	0.37ms	0.13ms
11761	6500	106127	2.73s	1.27ms	3.50ms	1.27ms
140651	42496	1032910	25.33s	23.98ms	91.60ms	24.13ms

Table 1: Timings for 3 hexmesh-to-tetmesh mappings on curved pipe geometry

the fact that the vector of source field values over the tet mesh can completely fit in cache memory for the intermediate-sized case and cannot completely fit in cache memory for the largest-sized case on the Apple G5 architecture employed.

References

[1] D.A. Korzekwa et al. Truchas models multi-physics phenomena, especially casting of alloys. The grid mapping software used in this paper is part of the Truchas software package. For information or to request software, contact the project team at telluride-support@lanl.gov.

[2] "Numerical Recipes in Fortran, 2nd Ed." by W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, Cambridge University Press, 1992.

[3] V. Maronnier, M. Picasso, and J. Rappaz, "Numerical simulation of three-dimensional free surface flows," Int. J. Num. Meth. Fluids, Vol. 42, pp. 697–716, 2003.

[4] X. Jiao and M.T. Heath, "Common-refinement-based data transfer between non-matching meshes in multiphysics simulations," Int. J. Num. Meth. Eng., Vol. 61, pp. 2402–2427, 2004.

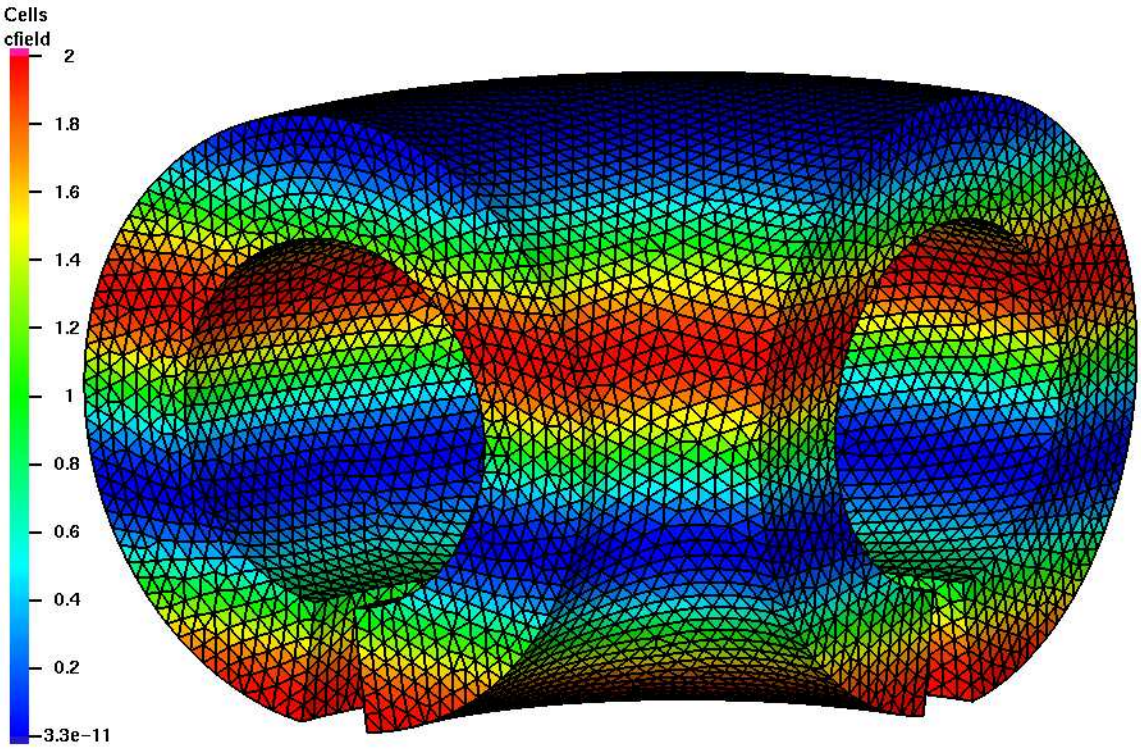

Figure 2: Tet mesh on slitted curved pipe geometry showing source field $f(\mathbf{x}) = 1 + \sin(z)$

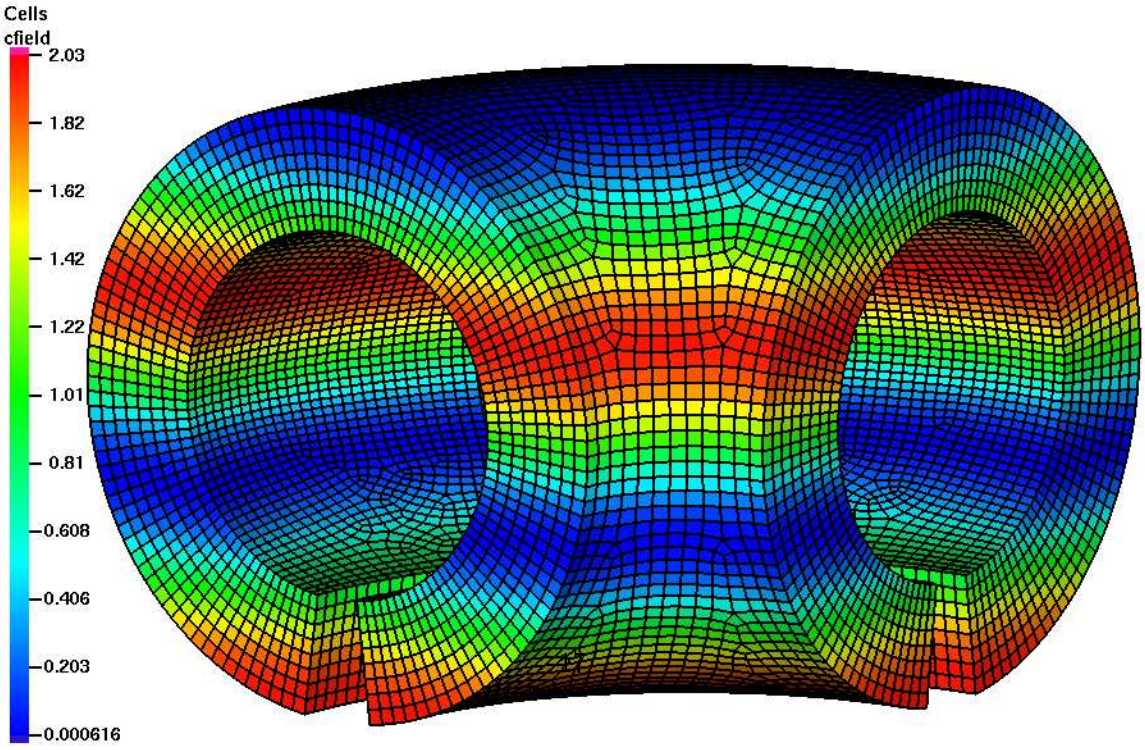

Figure 3: Hex mesh on same geometry showing mapped field with `exactly_conservative` option.

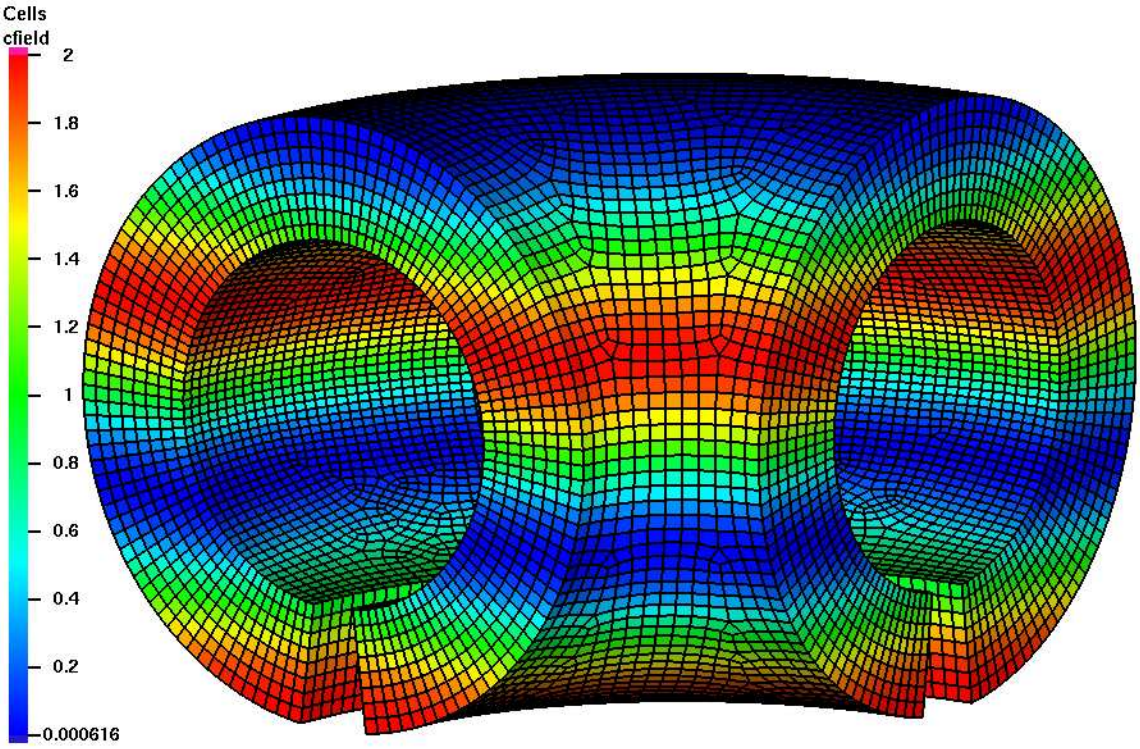

Figure 4: Hex mesh on same geometry showing mapped field with `preserve_constants` (weighted average) option.