Computer Assisted Methods in Engineering and Science, **20**: 55–71, 2013.

TWENTY YEARS OF THE CAMES

Improved GETMe by adaptive mesh smoothing

Dimitris Vartziotis[1,2,3], Manolis Papadrakakis[1]
[1] *Institute of Structural Analysis & Antiseismic Research*
National Technical University Athens (NTUA)
Zografou Campus, 15780 Athens, Greece
[2] *NIKI Ltd. Digital Engineering, Research Center*
205 Ethnikis Antistasis Street, 45500 Katsika, Ioannina, Greece
e-mail: dimitris.vartziotis@nikitec.gr
[3] *TWT GmbH Science & Innovation, Department for Mathematical Research & Services*
Bernhäuser Straße 40–42, 73765 Neuhausen, Germany

Mesh smoothing improves mesh quality by node relocation without altering mesh topology. Such methods play a vital role in finite element mesh improvement with a direct consequence on the quality of the discretized solution. In this work, an improved version of the recently proposed geometric element transformation method (GETMe) for mesh smoothing is presented. Key feature is the introduction of adaptive concepts, which improve the resulting mesh quality, reduce the number of parameters, and enhance the parallelization capabilities. Implementational aspects are discussed and results of a more efficient version are presented, which demonstrate that GETMe adaptive smoothing yields high quality meshes, is particularly fast, and has a comparably low memory profile. Furthermore, results are compared to those of other state of the art smoothing methods.

Keywords: mesh smoothing, GETMe adaptive, parallel smoothing, finite element mesh, mesh quality, mesh generation.

1. Introduction

Mesh quality plays a key role in finite element simulations, since it has an impact on solution accuracy and efficiency [1–3]. Therefore, a lot of effort has been put into the development of methods for mesh quality improvement. These methods can either be based on mesh topology altering operations, like edge or face swapping, local subdivision, node insertion or deletion [4–8], or they are based on preserving mesh topology by applying node relocations only. These so-called smoothing techniques are of particular interest if mesh interfaces or boundaries have to be preserved. Among the most popular smoothing methods is the Laplacian smoothing, where nodes are iteratively replaced by the arithmetic mean of connected nodes [9, 10]. This method is popular due to its simple implementation and its fast convergence behavior. However, without modifications, mesh quality can deteriorate and invalid elements can be generated. Therefore, smart variants have been proposed, which incorporate a quality metric and apply node updates only in the case of quality improvements or involve alternative node averaging schemes [11, 12].

In contrast to the simple approach of smart Laplacian smoothing, new node positions can also be derived by solving local optimization problems [13–16]. In doing so, the evaluation of quality metrics and performing numerical optimization requires additional computational effort, but usually improves the resulting mesh quality if compared to smart Laplacian smoothing. Alternatively, instead of applying a comparably inexpensive local optimization scheme, global optimization-based methods improve overall mesh quality by solving an optimization problem involving all mesh entities [17–20].

This approach usually results in a significantly increased computational effort. Furthermore, the proper choice of quality metrics and objective functions plays a crucial role [21–23].

Due to the rapidly growing complexity of nowadays simulations, there is a great demand for fast mesh improvement methods providing high quality results. In this context, the computational effort of global optimization-based methods may become computationally too expensive. As an alternative, the geometric element transformation method (GETMe) has been proposed [24–26]. Instead of improving element shape based on solving optimization problems, GETMe achieves mesh improvement by applying specific geometric element transformations, which successively transform an arbitrary mesh element into its regular counterpart [27–29]. This is combined with a relaxation and weighted node averaging scheme involving mesh quality. In a first stage, regularizing transformations are applied to all elements in order to improve overall mesh quality. A subsequently applied second stage successively transforms the worst mesh elements only in order to improve the minimal element quality.

In this paper, an advanced version of GETMe smoothing, named GETMe adaptive, is proposed, which improves the former version with respect to the following aspects: enhanced applicability and flexibility with an adaptive smoothing control, unified approach by incorporating both smoothing stages within one main smoothing loop, the submesh smoothing instead of worst element smoothing further facilitating parallelization, and adaptive node relaxation instead of invalid element node resetting. Furthermore, from an algorithmic point of view smoothing control is simplified and the number of parameters is significantly reduced compared to GETMe smoothing. From an application point of view, the resulting mesh quality is improved whereas memory requirements and smoothing time are significantly reduced. Results of the new GETMe adaptive approach are compared to those of the straightforward implementation of GETMe smoothing in order to demonstrate the progress achieved with GETMe-based mesh smoothing. Furthermore, comparative results of smart Laplacian smoothing and a state of the art global optimization-based approach serve as additional benchmark tests. The evaluation is performed with respect to resulting mesh quality, smoothing run-time, memory requirements and parallel implementation performance.

The remainder of this paper is organized as follows. In Sec. 2, a brief overview of the geometric element transformation is given followed by the proposed GETMe adaptive approach and its implementational aspects. Section 3 presents numerical results for a tetrahedral as well as a hexahedral mesh and provides a comparison with the results obtained by other smoothing methods.

2. GETME ADAPTIVE

In this section basic principles of GETMe smoothing are summarized first. Then, GETMe is enhanced by concepts of adaptivity aiming at improving smoothing results, run-time behavior, memory requirements, and the reduction of the number of control parameters in order to simplify further the method.

2.1. Basic principles of GETMe smoothing

In order to assess element validity and regularity, the established mean ratio quality criterion [17, 21, 30] is described first. It is based on measuring the deviation of an arbitrary valid element from its regular counterpart. Let $E = (p_1, \ldots, p_{|E|})$ denote an element with $|E|$ nodes $p_1, \ldots, p_{|E|}$. In the following, an exemplaric description for tetrahedral elements (tet) and hexahedral elements (hex) is given using the element node numbering schemes according to Fig. 1. More general definitions of the mean ratio quality criterion also covering polygonal elements and other types of volumetric elements can be found in [28, 30].

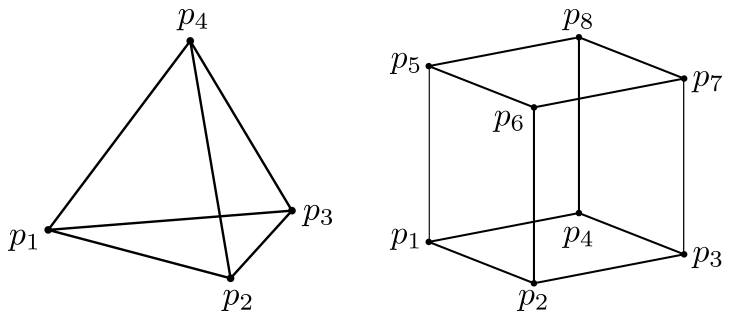

Fig. 1. Node numbering schemes for tetrahedral and hexahedral elements.

Each element node and its three emanating edges define a simplex. Node indices of such node simplices are given by N with

$$N^{\text{tet}} := \big((1,2,3,4)\big), \tag{1}$$

$$N^{\text{hex}} := \big((1,4,5,2),(2,1,6,3),(3,2,7,4),(4,3,8,1),\\ (5,8,6,1),(6,5,7,2),(7,6,8,3),(8,7,5,4)\big). \tag{2}$$

Here, the number of node simplices, i.e., the number of tuples in N, is denoted as $|N|$ and the k-th tuple of N represents the four node indices $N_{k,i}$, $i \in \{1,\dots,4\}$, of the k-th node simplex. It should be noticed that in the case of the tetrahedron all four node simplices represent the tetrahedron itself. Thus, it suffices to use only the node tetrahedron associated with p_1 resulting in $|N^{\text{tet}}| = 1$. An element is called valid if all node simplices have a positive volume, i.e., $\det D_k > 0$ for all $k \in \{1,\dots,|N|\}$ with the matrix of spanning edge vectors given by

$$D_k := \left(p_{N_{k,2}} - p_{N_{k,1}}, p_{N_{k,3}} - p_{N_{k,1}}, p_{N_{k,4}} - p_{N_{k,1}}\right). \tag{3}$$

If any $\det D_k \leq 0$ holds, the element is called invalid. The mean ratio quality number $q(E)$ of a valid element E is given by

$$q(E) := \frac{1}{|N|}\sum_{k=1}^{|N|} \frac{3\det(S_k)^{2/3}}{\operatorname{trace}(S_k^{\text{t}} S_k)}, \qquad S_k := D_k W^{-1}, \tag{4}$$

with an element type dependent target matrix W given by

$$W^{\text{tet}} := \begin{pmatrix} 1 & 1/2 & 1/2 \\ 0 & \sqrt{3}/2 & \sqrt{3}/6 \\ 0 & 0 & \sqrt{2/3} \end{pmatrix} \quad \text{and} \quad W^{\text{hex}} := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{5}$$

The columns of W represent the spanning edge vectors of a node simplex in the associated regular element. It holds that $q(E) \in [0,1]$ with small $q(E)$ denoting low quality elements and $q(E) = 1$ denoting ideal regular elements.

Quality improvement in GETMe smoothing is mainly based on regularizing mesh element transformations. These are geometric operations, which transform a given mesh element $E = (p_1,\dots,p_{|E|})$ into its transformed counterpart $E' = (p'_1,\dots,p'_{|E|})$. The transformation is chosen such that the element becomes more and more regular, if the transformation is applied iteratively. For planar polygonal elements, such transformations can be based on classic geometric operations, like erecting similar triangles on the sides of the polygons and taking the apices as transformed element nodes [27, 31]. Tetrahedral elements can be transformed by erecting the scaled normal of the opposing element face on each node [26], or by using the face normals of the dual element. The latter transformation scheme also applies to hexahedral, pyramidal and prismatic elements [28].

An example of consecutively applying the opposing face normals transformation to an initial tetrahedron is depicted in the upper row of Fig. 2. Here, the initial element and its mean ratio quality number is shown on the left. The following elements and quality numbers have been obtained by iteratively applying the transformation as given in [26]. Similarly, the lower row of Fig. 2 shows a sequence obtained by iteratively transforming a hexahedron using the dual element-based transformation [28].

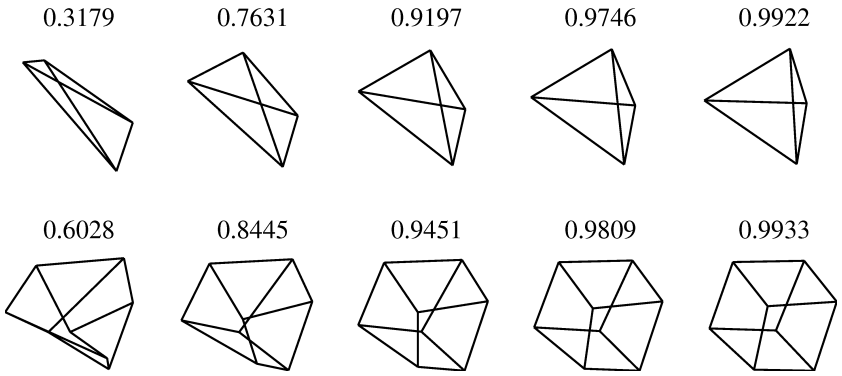

Fig. 2. Sequence of tetrahedra (upper) and hexahedra (lower), obtained by iteratively applying regularizing transformations to randomly chosen initial elements (left). The associated mean ratio quality number $q(E)$ is given above each element.

GETMe smoothing, as described for example in [28, 30], is based on a two-stage approach. The first, named GETMe simultaneous, is geared towards improving the mean mesh quality

$$q_{\text{mean}} := \frac{1}{n_E} \sum_{i=1}^{n_E} q(E_i), \tag{6}$$

which is the arithmetic mean of all n_E single element mean ratio numbers $q(E_i)$. First, all mesh elements are transformed to improve regularity, scaled to damp the element growth caused by the geometric operation, and relaxation is applied in order to mitigate the rapid change of shape. For each mesh node p_i, and an incident element E_j, this results in a new temporary node $p'_{i,j}$. New mesh node positions p'_i are derived as weighted averages of these temporary nodes according to

$$p'_i := \frac{\sum_{j \in J(i)} w_j p'_{i,j}}{\sum_{j \in J(i)} w_j}, \quad \text{with} \quad w_j := (1 - q(E_j))^\eta, \tag{7}$$

where $\eta > 0$ denotes a fixed amplification parameter and $J(i)$ denotes the index set of all elements incident with p_i.

As in the case of Laplacian smoothing, such a weighted node averaging approach can result in the generation of invalid elements. Therefore, nodes of invalid elements are iteratively reset to their old position until no invalid element remains in the mesh. This approach ensures the validity of the mesh. All steps of the scheme are applied iteratively until q_{mean} improvements obtained by two consecutive steps fall below a given threshold.

In many applications, such as finite element-based computations, the minimal element quality

$$q_{\text{min}} := \min_{i \in \{1,\ldots,n_E\}} q(E_i) \tag{8}$$

of the mesh also plays an essential role, since low q_{min} values might affect simulation accuracy and efficiency [30]. Therefore, the second stage, named GETMe sequential, focuses specifically on improving q_{min}. For this purpose, only the element with the lowest quality is selected, transformed, scaled, relaxed and new node positions of the element are set directly. This step is repeated until q_{min} cannot be further improved.

Depending on the application, boundary or even inner nodes might be fixed due to constraints. In that case, the minimal element quality number $q^*_{\min}$ of all elements with at least one free node is used instead of $q_{\min}$ as a selection criterion. Transforming the worst element and applying its new nodes directly might invalidate neighboring elements. Therefore, a similar node resetting technique is applied as in the case of the simultaneous approach. Here, an additional element quality penalty mechanism avoids infinite loops of picking and resetting the same element.

2.2. The GETMe adaptive approach

The GETMe approach described in the previous section has been further improved with the following aspects in mind: improving smoothing quality, reducing the number of control parameters, preserving simplicity, and increasing its amenability to parallel implementation.

Similar to the GETMe approach, smoothing by the new GETMe adaptive approach is performed in two stages. The first is geared towards improving q_{mean}, the second towards improving $q^*_{\min}$. However, in contrast to GETMe smoothing, these two stages are integrated into one smoothing loop. Furthermore, both stages use the same weighted node averaging scheme given by

$$p'_i := \frac{\sum_{j\in J(i)} w_j p'_{i,j}}{\sum_{j\in J(i)} w_j}, \quad \text{with} \quad w_j := \sqrt{\frac{\sum_{n\in N(j)} q(E_n)}{|N(j)|q(E_j)}}, \tag{9}$$

in order to compute new node positions. Here $N(j)$ denotes the index set of the neighbor elements of E_j, i.e., elements, which share at least one node with E_j. The number of such neighbor elements is given by $|N(j)|$. Compared to GETMe simultaneous smoothing using the averaging scheme according to Eq. (7), the first stage of GETMe adaptive smoothing differs in the choice of the weights w_j, as can be seen by Eq. (9), where the quality of a single element is related to the mean quality of its direct neighbors. This puts an emphasis on weights of lower quality elements. In GETMe sequential only the worst element is transformed and the resulting new element nodes are set directly, affecting all neighboring elements. In contrast, both stages of GETMe adaptive smoothing use the weighted node averaging scheme according to Eq. (9) in order to provide a more balanced result. The algorithmic description given in Fig. 3 provides an overview of the adaptive approach, which will be described in detail in the following.

In GETMe adaptive, only elements with a mean ratio quality number below a given threshold q_t are transformed. This threshold is set to one in line 1 of the algorithm, which enforces all elements to be transformed during the first stage. Furthermore, a state variable indicating the q_{mean} oriented stage and the associated table of node relaxation values are initialized. The role of the relaxation values will be discussed later and specific choices for the parameters involved will be given in Sec. 3.

The following sub-functions are applied:

- `ResetTemporaryNodesAndWeights:` For each node initialize a temporary node sum $\widehat{p}_i := (0,0,0)$ and weight sum $\widehat{w}_i := 0$.

- `AddTransformedElementNodesAndWeights:` For each mesh element E_j with $q(E_j) \le q_t$, compute the associated weight w_j according to Eq. (9), apply the geometric transformation to E_j using fixed transformation parameters, scale E'_j with respect to its centroid in order to preserve the sum of all element edge lengths and add the resulting weighted nodes $w_j p'_{i,j}$ to the temporary node sums $\widehat{p}_i$ and the weight w_j to the associated temporary weight sums $\widehat{w}_i$.

- `AddUntransformedElementNodesAndWeights:` Let I_T denote the index set of non-fixed nodes, which belong to at least one transformed element. For each untransformed element E_j with $q(E_j) > q_t$ and at least one node p_i with $i \in I_T$ compute the associated weight w_j according to Eq. (9) and add the weighted, but untransformed coordinates $w_j p_i$ of the nodes of E_j to the temporary node sums $\widehat{p}_i$ and the weight w_j to the associated temporary weight sums $\widehat{w}_i$.

```
Input  : Initial valid mesh
Output: Smoothed valid mesh
1  q_t := 1;
2  State := MeanCycleRunning;
3  SetNodeRelaxationValueTable(State);
4  for Iter := 1 to MaxIter do                          /* Main smoothing loop */
5  |  ResetTemporaryNodesAndWeights();
6  |  AddTransformedElementNodesAndWeights(q_t);
7  |  AddUntransformedElementNodesAndWeights(q_t);
8  |  ComputeNewNodes();
9  |  IterativeNodeRelaxation();
10 |  if State = MeanCycleRunning and Δq_mean < tol then
11 |  |  State := MinCycleStart;
12 |  |  SetNodeRelaxationValueTable(State);
13 |  end
14 |  if State = MinCycleRunning then
15 |  |  if no q*_min improvement in last iteration then
16 |  |  |  NoMinImproveCounter ++;
17 |  |  end
18 |  |  if NoMinImproveCounter > MaxNoMinImproveCounter then
19 |  |  |  State := MinCycleStart;
20 |  |  end
21 |  end
22 |  if State = MinCycleStart then
23 |  |  if no q*_min improvement in last min cycle then break;
24 |  |  State := MinCycleRunning;
25 |  |  NoMinImproveCounter := 0;
26 |  |  q_t := DetermineTransformationThreshold();
27 |  end
28 end
```

Fig. 3. Algorithmic description of the GETMe adaptive smoothing.

- `ComputeNewNodes`: For each $i \in I_T$ with $\widehat{w}_j > 0$ replace the contents of $\widehat{p}_i$ by the weighted coordinates $(1/\widehat{w}_i)\widehat{p}_i$ resulting in the new node coordinates p'_i according to Eq. (9). In the case of $\widehat{w}_j = 0$ use $\widehat{p}_i := p_i$.

- `IterativeNodeRelaxation`: Set the index into the table of relaxation values of each node to $r_i := 1$ and apply $p'_i := p_i$ for all $i \notin I_T$. For all $i \in I_T$ and a given table of relaxation values $R = (\varrho_1, \ldots, \varrho_k)$ compute the associated new coordinates as

$$p'_i := (1 - \varrho_{r_i})p_i + \varrho_{r_i}\widehat{p}_i. \tag{10}$$

  Subsequently, run the following iterative relaxation process until no invalid element remains in the mesh: For each invalid element mark the associated nodes and for each marked node increase the relaxation counter and recompute p'_i according to Eq. (10). Here, the last entry ϱ_k in the table R of descending relaxation values equals 0, and the increase of r_i is stopped if k is reached. This assures that if relaxation cannot avoid the generation of invalid elements, the nodes are reseted to their original valid position like in the case of the GETMe approach. After termination of the relaxation loop set the new mesh node coordinates to p'_i.

The subsequent lines 10 to 26 of the algorithm control the two smoothing stages. Here, the first if-statement of this block controls the termination of the q_{mean}-oriented first smoothing stage, in which all elements are transformed due to the choice $q_t := 1$. This stage, indicated by setting the state variable to `MeanCycleRunning`, is terminated in case the q_{mean}-improvement (denoted as Δq_{mean}) of two consecutive iterations drops below a prescribed threshold `tol`, which is usually set to 10^{-4}. If this is the case, the state is changed to `MinCycleStart` and an alternative table of relaxation values used in `IterativeNodeRelaxation` is set. Specific choices used for the tables of relaxation values are given in the numerical examples section.

The second stage, which is geared towards improving q^*_{min}, is divided into smoothing cycles consisting of an adaptive number of smoothing iterations. Here, at the beginning of each cycle (cf. line 22 of the algorithm), smoothing is terminated if the previous cycle did not result in an improvement of q^*_{min}. Subsequently, the no improvement counter is reset and a new element quality threshold q_t is determined. This is done by `DetermineTransformationThreshold` in which the quality numbers of all mesh elements are sorted in ascending order. After that, q_t is set to the quality value of a prescribed position in this sorted vector. Usually, this position is set to a fixed percentage of all elements times the number of elements.

After determining the new node positions within one q^*_{min}-oriented smoothing cycle, q^*_{min} is checked in lines 14f. If there is no improvement, then the no improvement counter is increased by one and the cycle is terminated as soon as this counter reaches a prescribed limit.

Due to their common approach of using geometric element transformations in combination with weighted transformed node averaging, GETMe smoothing as well as GETMe adaptive smoothing are generally applicable. As has been already shown by various examples of GETMe smoothing this includes structured and unstructured surface and volume meshes consisting of triangular, quadrilateral, tetrahedral, hexahedral, pyramidal, and prismatic elements [24–28, 32]. From an algorithmic point of view, GETMe adaptive differs from GETMe smoothing in the following points:

- Incorporation of two smoothing stages within one smoothing loop instead of applying two separate loops.
- Relaxation is applied to both previous and new node positions instead of involving previous and transformed elements. Furthermore, the relaxation parameter is adjusted on a nodal basis, which also replaces the iterative invalid element node resetting scheme.
- During the q^*_{min} oriented stage, GETMe adaptive transforms more than one element per iteration, which is more suitable for parallel mesh smoothing. Furthermore, both stages use the same node averaging approach. However, nodes of elements with $q(E) > q_t$ are directly used without transformation and scaling.
- Iterations of the second stage are organized in cycles. The transformation quality threshold q_t is updated at the beginning of each smoothing cycle.
- Fixed element transformation parameters are used instead of quality adaptive transformation parameters. The adjusted weights in the transformed element nodes averaging scheme are based on neighboring element quality ratios.
- The exponent η of the node weights w_j given in Eq. (7) and the penalty parameters of GETMe sequential are removed. This eliminates the need for a minimal element quality heap required by the sequential substep of GETMe.

2.3. Implementation and parallelization

Results of the GETMe approach have been published in [28, 30] using a straightforward C++ implementation [33]. This implementation incorporates an object-oriented data structure, which

provides enhanced topology information for nodes and elements leading to significantly increased memory requirements. Furthermore, the interchange of information between node and element objects leads to an additional runtime overhead. Nevertheless, GETMe smoothing turned out to be fast and effective if compared with other smoothing methods like smart Laplacian smoothing and a global optimization-based approach.

In order to make one step forward towards demonstrating the true efficiency of geometry-based smoothing approaches, the GETMe adaptive smoothing is implemented aiming at improving runtime and memory profile. Although such an implementation could have been based on C++, the C programming language [34] has been chosen instead, since it also builds a good foundation for future developments involving GPU-based computations using OpenCL [35], CUDA [36], or OpenACC [37], which are mainly C oriented. As has been recently shown, combined with domain decomposition methods such approaches open a new era in scientific computing [38]. In the following, some key aspects of this improved implementation will be discussed.

Mesh nodes and elements are stored in arrays containing the node coordinates and the node indices of the elements, respectively. Furthermore, since the weights w_j according to Eq. (9) involve quality numbers of neighbor elements, a neighbor element index table is also used. In addition to the current node coordinates p_i, GETMe adaptive smoothing also uses arrays in order to store the temporary node coordinate sums $\widehat{p}_i$, the new node coordinates p'_i, the weight sums w_i, the element quality numbers $q_j := q(E_j)$, and the indices r_i into the table of relaxation values for all $i \in \{1, \ldots, n_N\}$ and $j \in \{1, \ldots, n_E\}$, where n_N and n_E denote the number of mesh nodes and elements. Since the mean ratio quality criterion is expensive to evaluate, entries of the element quality array are only updated in the case of element node coordinate changes.

First, all values $\widehat{p}_i$, w_i are initialized to zero in `ResetTemporaryNodesAndWeights`. After that, in `AddTransformedElementNodesAndWeights` each element is transformed and scaled and the associated weighted new node coordinates and weights are successively added to the corresponding variables $\widehat{p}_i$, w_i. Here, the weights are determined using the tabulated element quality numbers q_j. Similarly, `AddUntransformedElementNodesAndWeights` adds untransformed, but weighted nodes and the corresponding weights to $\widehat{p}_i$ and w_i. Then, the unrelaxed new node coordinates according to Eq. (9) are computed by `ComputeNewNodes` and stored in $\widehat{p}_i$.

During the $q^*_{\min}$ oriented stage of the GETMe adaptive approach, these sub-functions operate only on a subset of the mesh, which is defined by the elements with a quality number below the quality threshold q_t. This also holds for `IterativeNodeRelaxation`, which iteratively determines the final new node coordinates p'_i. Since the relaxation step does not require the re-evaluation of the weights w_i, the element quality vector can already be filled with the quality numbers of the elements with respect to the new node coordinates p'_i. Here, non-positive entries, indicate invalid elements whose nodes require an additional node relaxation step using an adjusted individual relaxation parameter. The relaxation loop is terminated when all elements become valid. Then, all current mesh nodes p_i are set to the new coordinates p'_i.

As can be seen, the loops of all these sub-functions are amenable to parallelization. In the current C implementation of GETMe adaptive parallelization was realized by using the OpenMP API version 3.1 [39]. Here, simply adding `#pragma omp parallel for` directives suffice for the loops to be executed in parallel. In this context, reading and updating data like $\widehat{p}_i$ and w_i by simultaneous threads requires synchronization by the use of atomic operations. Results of the parallelized version of GETMe adaptive smoothing following this approach are given in Sec. 3.

Further potential runtime improvements can be achieved by avoiding thread synchronization caused by atomic operations using thread private dynamically allocated memory, which requires an additional implementational effort. However, for the current version, the authors choose not to optimize runtime behavior of GETMe adaptive by modifying data handling on an implementational level.

The OpenMP API supports shared memory multiprocessing programming, where concurrently accessing a large amount of memory by different threads on the same system often leads to a significant bottleneck. As an alternative, distributed computing approaches can be applied. Here, the mesh

is partitioned and smoothing of the submeshes is conducted on different systems. GETMe adaptive smoothing is also suitable for this type of parallelization, since its local smoothing approach, incorporating only information of direct neighbor elements, results in a low submesh interface communication overhead.

3. Numerical results

This section provides a detailed comparison for two generic meshes of smoothing results obtained by smart Laplace, a global optimization-based approach, GETMe, and GETMe adaptive. Default parameters for the GETMe adaptive smoothing approach presented in Subsec. 2.2, and a short description of the two alternative smoothing methods will be given first.

3.1. Test description

In the case of GETMe smoothing, the default parameters as given in [28, 30] have been used. For GETMe adaptive smoothing, the maximum number of iterations has been set to 1000 and the q_{mean} improvement tolerance to 10^{-4}. The tetrahedral mesh has been smoothed by using the opposite face normal transformation [26]. For the hexahedral mesh, the dual element-based transformation as defined in [32] has been applied. In both cases, the fixed element transformation parameter $\sigma = 3/2$ was used. The tables of relaxation values have been set to $R = (1, 1/4, 1/16, 0)$ and $R = (1/2, 1/4, 1/10, 1/100, 0)$ in the case of the q_{mean} and q^*_{min} oriented smoothing stage, respectively. Each q^*_{min} oriented smoothing cycle has been terminated after the NoMinImproveCounter reached five iterations.

Results of both GETMe variants are compared to those of smart Laplacian smoothing. In this approach, smoothing is applied iteratively by replacing each non-fixed inner node by the arithmetic mean of its neighboring nodes. However, a node update is only applied if this improves the average mean ratio quality of incident elements [11].

Smart Laplacian smoothing was implemented using the same data structures as the GETMe adaptive approach. Thus, it also benefits from the lean data management of the C implementation. In addition, results are provided for an OpenMP-based parallelized smart Laplacian smoothing approach. Here, testing if a node update would increase local mesh quality is applied in parallel for all nodes. In this case, nodes are not updated directly in order to assure reproducibility of results and the independence of the node numbering scheme. Instead, similar to GETMe adaptive, new node positions are stored in p'_i, which is interchanged with p_i at the end of the iteration. In the case of invalid element generation, nodes are iteratively reset to their original coordinates until the mesh is valid again. Smoothing is terminated, if the q_{mean} values of two consecutive iterations differ by less than 10^{-4}.

The quality of the results is compared to a state of the art global optimization-based approach. Here, the shape improvement wrapper of the mesh quality improvement toolkit Mesquite [40] has been applied iteratively until the q_{mean} improvements of two consecutive calls dropped below 10^{-4}. Within the shape improvement wrapper, a mesh untangling algorithm is applied first. Subsequently, mesh smoothing is accomplished by minimizing the sum of the inverse mean ratio values of all elements by a feasible Newton-based approach [41].

Mesquite also provides parallel mesh smoothing based on smoothing submeshes and synchronizing submesh interfaces using MPI [42]. However, for the given examples, the shape improvement wrapper failed due to the detection of termination criteria issues, which would have led to infinite loops. Therefore, as a substitute, optimal results have been assumed for the parallel global optimization-based smoothing approach. That is, the sequential runtime has been divided by the number of processor cores of the test system to provide an ideal case estimate of the parallel runtime. In practice, runtimes are expected to be significantly higher due to the usage of shared resources.

The GETMe approach and the Mesquite toolkit are implemented in C++. In contrast, GETMe adaptive and smart Laplacian smoothing are implemented in C. All programs have been compiled using the GNU Compiler Collection version 4.7.1 [43]. Computations have been accomplished on a personal computer with an Intel® Core™ i7-870 CPU (quad core, 8 MB cache, 2.93 GHz), 16 GB RAM, and a 64 bit Linux operating system. Here, hyper-threading has been deactivated and all four cores of the processor have been used for parallel computations. Thus, the ideal speedup factor for the parallel implementation is 4.

3.2. Tetrahedral mesh example

The first example considers the piston model depicted in Fig. 4. It was constructed by completing a partial model provided by the Drexel University Geometric & Intelligent Computing Laboratory model repository [44].

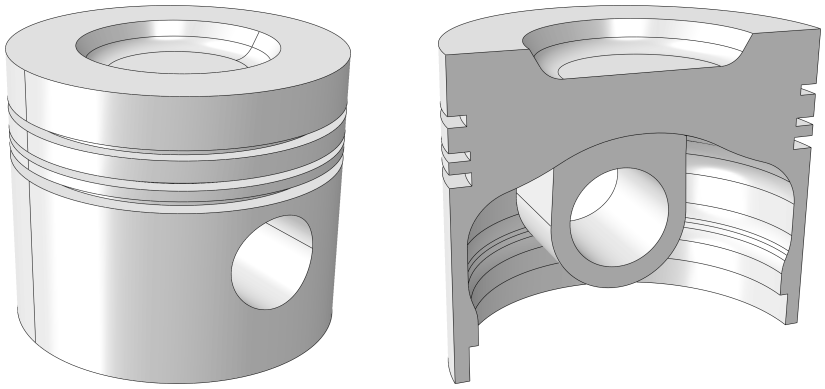

Fig. 4. Full piston model (left) and cross section (right).

A tetrahedral mesh consisting of 729 923 nodes and 4 129 608 elements has been generated by Delaunay tetrahedralization resulting in a highly unstructured mesh with the number of tetrahedra attached to individual nodes ranging from 2 to 48. This mesh was distorted by random element validity preserving node movements in order to increase the smoothing potential of the mesh. The resulting mesh has been smoothed with the C implementation of the smart Laplacian smoothing approach, the C++ based shape improvement wrapper of Mesquite, the C++ implementation of GETMe, and the new C implementation of GETMe adaptive, as described in the previous sections. Cross sections of the resulting meshes are depicted in Fig. 5. Here, each element is colored according to its mean ratio quality number, where reddish colors indicate low quality elements and bluish color high quality elements.

As can be seen in Fig. 5a, elements of the initial mesh are severely distorted to challenge all smoothing methods. At first sight it seems that the mesh quality obtained by smart Laplacian smoothing is comparable to those of the other smoothing methods. However, a closer look reveals elements of very low quality, which can also be verified by the worst element quality number $q^*_{\min} = 0.0001$ given in Table 1. In contrast, the corresponding quality numbers of the global optimization and GETMe-based methods are significantly better. For example, the numbers of elements E_j with $q^*_{\min} \leq q(E_j) \leq 0.4$ amount to 1 798 777 for the initial mesh and 4 463, 164, 514, 32 for its variants smoothed by smart Laplace, global optimization, GETMe, and GETMe adaptive, respectively. In the case of smart Laplace, the mean ratio quality of 465 elements is even below 0.1, which may lead to numerical instabilities in subsequent finite element computations. Nevertheless, the mesh is valid, which is not the case if classical Laplacian smoothing is used instead.

The increased number of elements with $q(E_j) \leq 0.4$ in GETMe smoothing is due to its approach of consecutively improving the worst elements, which leads to an accumulation of elements with

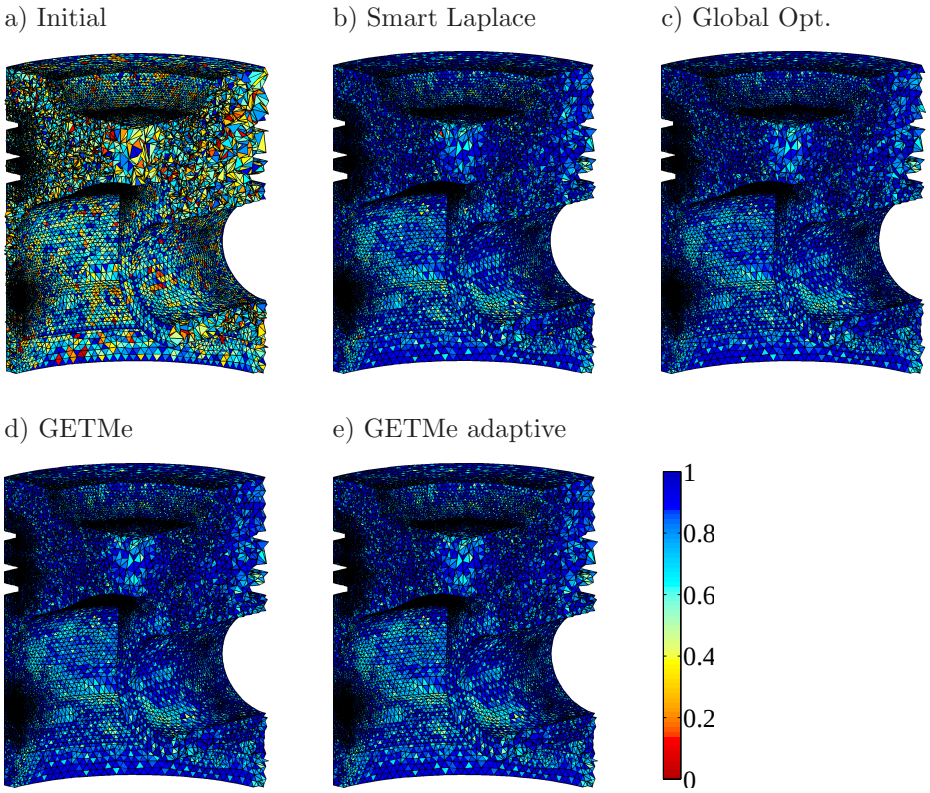

Fig. 5. Piston mesh cross sections with elements colored according to their mean ratio quality number.

a quality number slightly above $q^*_{\min}$. This effect is ameliorated by the adaptive approach, which also uses the weighted node averaging scheme of Eq. (9) during the $q^*_{\min}$-oriented smoothing stage instead of setting transformed nodes directly. Furthermore, both quality numbers obtained by GETMe adaptive smoothing are slightly better than those of GETMe smoothing.

Table 1 also provides the maximum memory size of the application measured in gibibytes (2^{30} bytes), the smoothing time in seconds, as well as the number of iterations. In the case of GETMe smoothing, the iteration number of the simultaneous and sequential substeps are given. Similarly, for GETMe adaptive iteration numbers are given for the q_{mean} and for the $q^*_{\min}$-oriented stage of the smoothing process.

Due to its simple approach, memory requirements of smart Laplacian smoothing are low. The large amount of memory used in the case of the C++ implementation of GETMe smoothing is not caused by the requirements of the algorithm, but by the use of general data structures storing

Table 1. Piston mesh smoothing results. (P) indicates parallel versions, (OP) denotes an estimate for optimal parallelization.

Name	Mem [GiB]	Time [s]	Iter	$q^*_{\min}$	q_{mean}
Initial	–	–	–	0.0000	0.4506
Smart Laplace	0.3	28.90	12	0.0001	0.8584
Global Opt.	2.0	791.42	149	0.2963	0.8654
GETMe	5.7	74.41	12/5500	0.3336	0.8636
GETMe adaptive	1.6	40.05	15/47	0.3421	0.8637
Smart Laplace (P)	0.4	9.18	12	0.0001	0.8584
Global Opt. (OP)	2.0	197.85	149	0.2963	0.8654
GETMe adaptive (P)	1.6	17.29	15/47	0.3421	0.8637

redundant topology and statistical informations. This can also be seen by the comparably low memory requirements of the GETMe adaptive approach implemented in C. Thus, an implementation of GETMe smoothing using similar lean data structures would result in a significantly lower memory profile comparable to that of GETMe adaptive. However, the results for the unmodified C++ implementation of GETMe smoothing are included for consistency with previous publications. Thus, improvements with respect to memory requirements and smoothing time demonstrated in this work could also be obtained for the examples given in previous publications.

On average, one iteration of smart Laplacian smoothing requires 2.41 s. It can be seen that smart Laplacian smoothing results in the lowest overall smoothing time. In contrast, the global optimization-based approach, due to the high average smoothing time of 5.31 s per iteration and the large number of iterations, takes 27.4 and 19.8 times longer if compared to smart Laplacian smoothing and GETMe adaptive smoothing, respectively. For the latter, the average time of 2.57 s per iteration during the first stage is only slightly larger than that of smart Laplacian smoothing with 2.41 s per iteration, despite the fact that the elements are transformed. This is due to the lower number of mean ratio quality evaluations. Furthermore, the average runtime per iteration of 0.03 s during the second smoothing stage of GETMe adaptive is low.

Mesh quality with respect to smoothing time is depicted in Fig. 6. As can be seen in the results for the minimal element quality number $q^*_{\min}$ given on the left, smart Laplacian smoothing leads to almost no improvement. In contrast, GETMe and GETMe adaptive lead to a sharp rise at the beginning of the q_{mean}-oriented stage, as well as during the complete $q^*_{\min}$-oriented stage. Improvements of the global optimization-based approach are achieved within the last quarter of its runtime, which prohibits a preliminary termination from an application point of view.

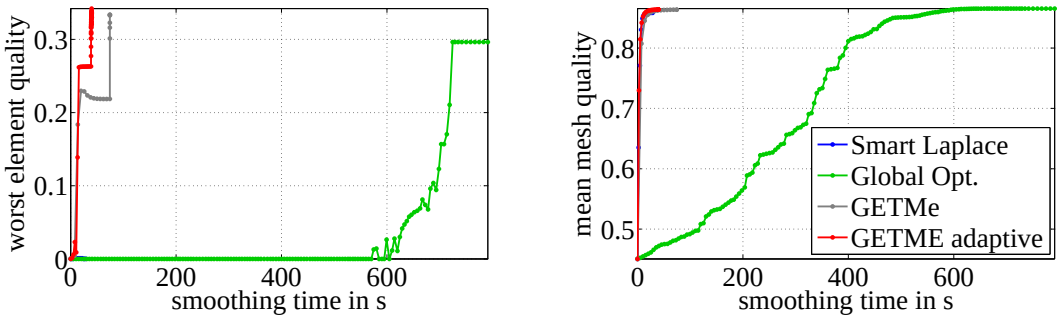

Fig. 6. Piston mesh $q^*_{\min}$ (left) and q_{mean} (right) with respect to smoothing time for sequential implementations.

The second part of Table 1 also provides results for the parallel versions. Again, it is pointed out that in the case of global optimization a lower bound for the runtime is given, by assuming that the method achieves the optimal speedup factor 4. The actual speedup factors of smart Laplacian smoothing and GETMe adaptive smoothing reach 3.1 and 2.3, respectively. As can be seen, the speedup of GETMe adaptive is inferior due to to the incorporation of atomic memory operations and a larger amount of data to be transferred.

3.3. Hexahedral mesh example

The second example considers the pump carter model depicted in Fig. 7, of which a STEP-file was provided by the courtesy of Rosalinda Ferrandes, Grenoble INP, by the AIM@SHAPE Shape Repository [45].

An all-hexahedral mesh consisting of 2 779 096 nodes and 2 646 976 elements has been generated by sweeping a quadrilateral surface mesh along the z-axis resulting in an almost structured hexahedral mesh, where 90% of the nodes are shared by eight hexahedra each. The number of

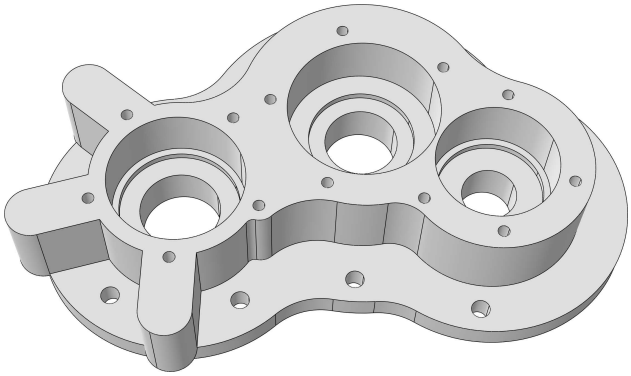

Fig. 7. Pump carter model.

attached elements of the remaining 10% of nodes ranges from two to ten. As in the case of the first example, this mesh was distorted by element validity preserving random node movements resulting in the initial mesh. The latter has been subsequently improved by all smoothing methods under consideration. Cross sections of the resulting meshes are depicted in Fig. 8. Compared to the case

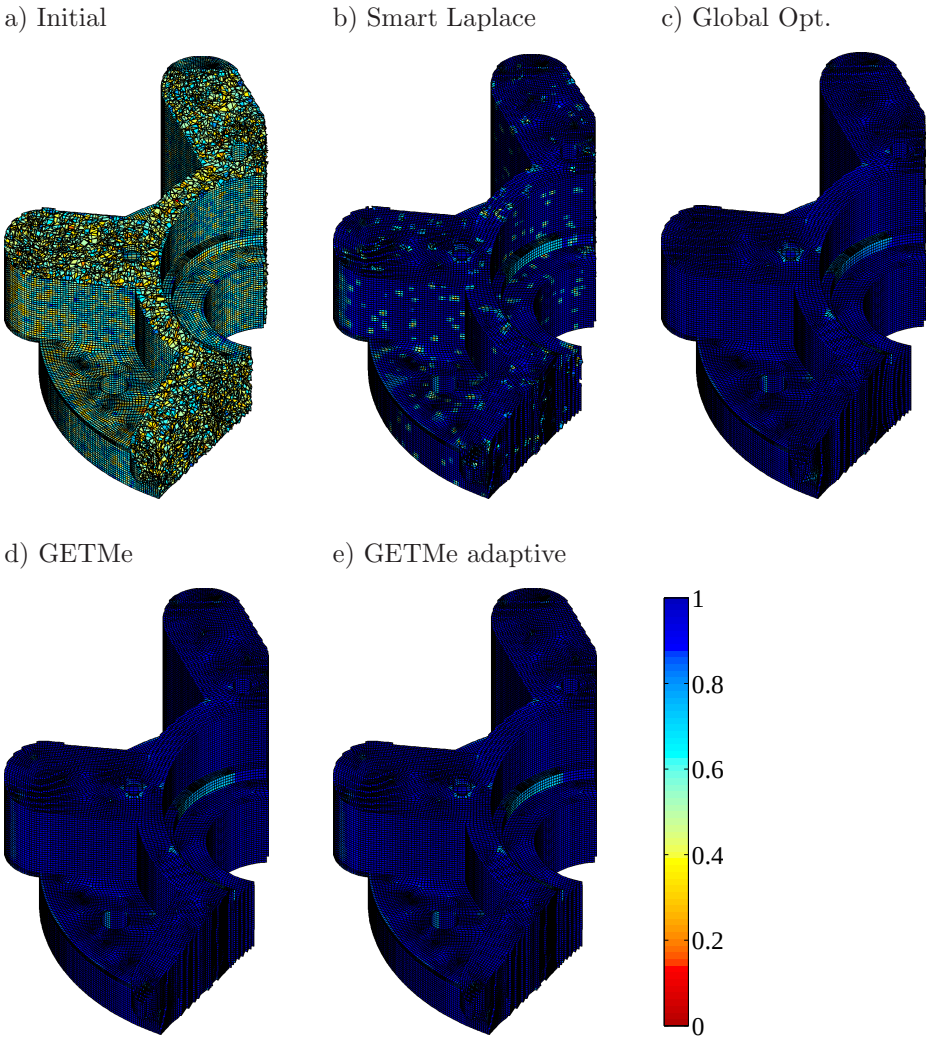

Fig. 8. Pump carter mesh cross sections with elements colored according to their mean ratio quality number.

of the tetrahedral mesh, the ratio of low quality elements is further increased in the case of smart Laplacian smoothing. For example, the numbers of elements E_j with $q^*_{\min} \leq q(E_j) \leq 0.5$ amounts to 1 586 570, 18 509, 23, 4, and 0 in the case of the initial mesh and those smoothed by smart Laplace, global optimization, GETMe, and GETMe adaptive, respectively. Here, the large number of low quality elements in the case of smart Laplacian smoothing is also reflected by the decreased mean mesh quality given in Table 2. As in the case of the first example, applying classical Laplacian smoothing invalidates the mesh.

Table 2. Pump carter mesh smoothing results. (P) indicates parallel versions, (OP) denotes an estimate for optimal parallelization.

Name	Mem [GiB]	Time [s]	Iter	$q^*_{\min}$	q_{mean}
Initial	–	–	–	0.0385	0.4709
Smart Laplace	0.7	344.62	30	0.0604	0.9393
Global Opt.	4.4	5241.92	269	0.4501	0.9766
GETMe	2.9	227.25	24/31800	0.4667	0.9720
GETMe adaptive	0.9	88.97	25/43	0.5442	0.9719
Smart Laplace (P)	0.9	103.84	30	0.0604	0.9393
Global Opt. (OP)	4.4	1310.48	269	0.4501	0.9766
GETMe adaptive (P)	0.9	33.81	25/43	0.5442	0.9719

Compared to global optimization and GETMe smoothing, GETMe adaptive smoothing leads to a further improvement of $q^*_{\min}$. In all three cases, mean mesh quality numbers are near the optimal value one, which is also reflected by the bluish element colors in the cross sections given in Fig. 8.

Again, the maximum memory size of smart Laplacian smoothing and GETMe adaptive is low, if compared to the C++ implementations of GETMe and global optimization. For example, global optimization requires five times more memory than GETMe adaptive. Furthermore, GETMe adaptive smoothing is 3.9, 58.9, and 2.6 times faster, compared to smart Laplacian smoothing, global optimization, and GETMe smoothing, respectively. Here, smart Laplacian smoothing is slower than both GETMe variants, due to the larger number of element quality evaluations and the fact that determining the mean ratio of hexahedra requires the computation of eight 3×3 determinants and Frobenius norms, each.

This increased speedup obtained by GETMe adaptive is also apparent in the quality with respect to smoothing time graphs depicted in Fig. 9. As can be seen, global optimization leads to a decrease of the worst element quality during the first iteration, which cannot be resolved during almost 4000 smoothing seconds. In contrast, the same method leads to a slow but steady improvement of the mean mesh quality within that time. The convergence behavior of smart Laplacian smoothing and both geometry-based approaches differs significantly. Here, the first few iterations lead to a sharp rise of mesh quality with respect to both, $q^*_{\min}$ and q_{mean}.

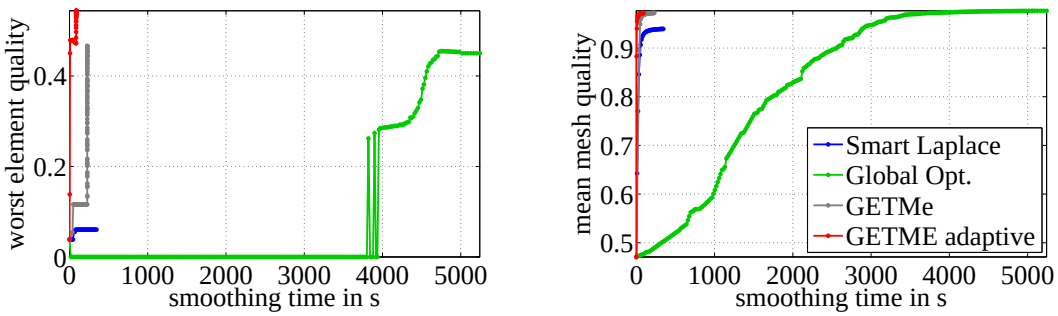

Fig. 9. Pump carter mesh $q^*_{\min}$ (left) and q_{mean} (right) with respect to smoothing time for sequential implementations.

Table 2 also provides results for the parallel versions of smart Laplacian smoothing and GETMe adaptive based on an OpenMP-approach. Here, the speed up by applying smart Laplacian smoothing in parallel is 3.3 compared to the sequential version. The speed up of parallel GETMe adaptive smoothing is 2.6. Furthermore, the parallel version of GETMe adaptive smoothing is 3.1 and 38.8 times faster compared to the parallel version of smart Laplacian smoothing and the ideal parallelization of the global optimization-based approach, respectively.

4. Conclusion

An advanced version of the geometric element transformation method for mesh smoothing has been presented. In contrast to the standard GETMe approach, GETMe adaptive involves concepts of adaptivity, implementing a quality controlled two-stage smoothing approach integrated into one main smoothing loop, an adaptive node relaxation scheme, in order to avoid the generation of invalid mesh elements and to accelerate the rate of mesh improvement, and a quality ratio-based weighting scheme for updating the nodes, which is now consistently applied in both smoothing stages. Furthermore, by the usage of fixed transformation parameters and by eliminating the element quality penalty parameters, GETMe adaptive further reduces the number of parameters if compared to the standard GETMe approach.

An improved code implementation has been presented in order to demonstrate that GETMe adaptive has both, a comparably low memory profile and short smoothing iteration runtimes. In combination with the powerful regularizing effect of the incorporated element transformations, this leads to an effective smoothing approach for meshes of various types. This has been shown by two examples also providing the comparison with smart Laplacian smoothing and a global optimization-based approach. From a quality point of view it turned out that GETMe adaptive smoothing can further improve the high quality results obtained by the standard GETMe approach. In both cases, quality results are at least comparable to those of a state of the art global optimization-based approach. In contrast, due to its simple approach, smart Laplacian smoothing fails to improve the minimal element quality.

Results of the new C implementation of GETMe adaptive smoothing, although not yet fully optimized, demonstrate the true potential of the method. For the given tetrahedral and hexahedral meshes consisting of a few million elements, GETMe adaptive smoothing was about 20 and 60 times faster compared to the results of a state of the art global optimization-based approach, requiring only 80% and 20% of the memory. Similar runtime improvements and memory reductions can be achieved by a corresponding C implementation of the standard GETMe approach.

A first parallel version of GETMe adaptive smoothing based on OpenMP resulted in a lower speedup factor of up to 2.6 compared to the smart Laplacian smoothing approach with a speedup factor of up to 3.3 on a system with a quad-core processor. Further efforts focusing on appropriate data structures and memory access optimization are expected to increase this speedup factor of GETMe adaptive smoothing. Results will also serve as a basis for massively parallel implementations of GPU-based GETMe variants, for which the local characteristics of the element transformation and node averaging approach should provide a favorable basis.

Acknowledgement

The authors would like to thank Joachim Wipper, TWT GmbH Science & Innovation, for providing and elaborating on the test examples used in this work.

References

[1] L.A. Freitag, C. Ollivier-Gooch. A cost/benefit analysis of simplicial mesh improvement techniques as measured by solution efficiency. *Int. J. Comput. Geom. Appl.*, **10**(4): 361–382, 2000.

[2] G. Strang, G. Fix. *An Analysis of the Finite Element Method*. Wellesley-Cambridge Press, second edition, 2008.
[3] J.R. Shewchuk. What is a good linear finite element? Interpolation, conditioning, anisotropy, and quality measures (preprint). *University of California at Berkeley*, 2002.
[4] L.A. Freitag, C. Ollivier-Gooch. Tetrahedral mesh improvement using swapping and smoothing. *Int. J. Numer. Meth. Eng.*, **40**(21): 3979–4002, 1997.
[5] B.M. Klingner, J.R. Shewchuk. Aggressive tetrahedral mesh improvement. In *Proceedings of the 16th International Meshing Roundtable*, 3–23, 2007.
[6] J.M. Escobar, R. Montenegro, E. Rodríguez, J.M. González-Yuste. Smoothing and local refinement techniques for improving tetrahedral mesh quality. *Comput. Struct.*, **83**(28–30): 2423–2430, 2005.
[7] M.-C. Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. *Int J Numer Methods Eng*, **40**(18): 3313–3324, 1997.
[8] P.J. Frey, P.-L. George. *Mesh Generation*. Wiley-ISTE, second edition, 2008.
[9] S.H. Lo. A new mesh generation scheme for arbitrary planar domains. *Int. J. Numer. Meth. Eng.*, **21**(8): 1403–1426, 1985.
[10] D.A. Field. Laplacian smoothing and Delaunay triangulations. *Commun. Appl. Numer. Methods*, **4**(6): 709–712, 1988.
[11] L.A. Freitag. On combining Laplacian and optimization-based mesh smoothing techniques. In *Trends in Unstructured Mesh Generation*, 37–43, 1997.
[12] M. Zhihong, M. Lizhuang, Z. Mingxi, L. Zhong. A modified Laplacian smoothing approach with mesh saliency. *Lect. Notes Comput. Sci.*, **4073**: 105–113, 2006.
[13] N. Amenta, M. Bern, D. Eppstein. Optimal point placement for mesh smoothing. *J. Algorithms*, **30**(2): 302–322, 1999.
[14] H. Xu, T.S. Newman. An angle-based optimization approach for 2D finite element mesh smoothing. *Finite Elem. Anal. Des.*, **42**(13): 1150–1164, 2006.
[15] Y. Zhang, C. Bajaj, G. Xu. Surface smoothing and quality improvement of quadrilateral/hexahedral meshes with geometric flow. *Commun. Numer. Methods Eng.*, **25**(1): 1–18, 2009.
[16] L. Chen. Mesh smoothing schemes based on optimal Delaunay triangulations. In *Proceedings of the 13th International Meshing Roundtable*, 109–120, 2004.
[17] L. Freitag Diachin, P. Knupp, T. Munson, S. Shontz. A comparison of two optimization methods for mesh quality improvement. *Eng. Comput.*, **22**(2): 61–74, 2006.
[18] A. Egemen Yilmaz, M. Kuzuoglu. A particle swarm optimization approach for hexahedral mesh smoothing. *Int. J. Numer. Meth. Fl.*, **60**(1): 55–78, 2009.
[19] S. Kulovec, L. Kos, J. Duhovnik. Mesh smoothing with global optimization under constraints. *Strojniški vestnik – J. Mech. Eng.*, **57**(7–8): 555–567, 2011.
[20] Y. Sirois, J. Dompierre, M.-G. Vallet, F. Guibault. Hybrid mesh smoothing based on Riemannian metric nonconformity minimization. *Finite Elem. Anal. Des.*, **46**(1–2): 47–60, 2010.
[21] P.M. Knupp. Algebraic mesh quality metrics. *SIAM J. Sci. Comput.*, **23**(1): 193–218, 2001.
[22] P.M. Knupp. Remarks on mesh quality. In *Proceedings of the 45th AIAA Aerospace Sciences Meeting and Exhibit*, 2007.
[23] X. Jiao, D. Wang, H. Zha. Simple and effective variational optimization of surface and volume triangulations. In *Proceedings of the 17th International Meshing Roundtable*, 315–332, 2008.
[24] D. Vartziotis, T. Athanasiadis, I. Goudas, J. Wipper. Mesh smoothing using the geometric element transformation method. *Comput. Methods Appl. Mech. Eng.*, **197**(45–48): 3760–3767, 2008.
[25] D. Vartziotis, J. Wipper. The geometric element transformation method for mixed mesh smoothing. *Eng. Comput.*, **25**(3): 287–301, 2009.
[26] D. Vartziotis, J. Wipper, B. Schwald. The geometric element transformation method for tetrahedral mesh smoothing. *Comput. Methods Appl. Mech. Eng.*, **199**(1–4): 169–182, 2009.
[27] D. Vartziotis, J. Wipper. Characteristic parameter sets and limits of circulant Hermitian polygon transformations. *Linear Algebra Appl.*, **433**(5): 945–955, 2010.
[28] D. Vartziotis, J. Wipper. Fast smoothing of mixed volume meshes based on the effective geometric element transformation method. *Comput. Methods Appl. Mech. Eng.*, **201–204**: 65–81, 2012.
[29] D. Vartziotis, S. Huggenberger. Iterative geometric triangle transformations. *Elemente der Mathematik*, **67**(2): 68–83, 2012.
[30] D. Vartziotis, J. Wipper, M. Papadrakakis. Improving mesh quality and finite element solution accuracy by GETMe smoothing in solving the Poisson equation. *Finite Elem. Anal. Des.*, **66:** 36–52, 2013.
[31] D. Vartziotis, J. Wipper. Classification of symmetry generating polygon-transformations and geometric prime algorithms. *Mathematica Pannonica*, **20**(2): 167–187, 2009.
[32] D. Vartziotis, J. Wipper. A dual element based geometric element transformation method for all-hexahedral mesh smoothing. *Comput. Methods Appl. Mech. Eng.*, **200**(9–12): 1186–1203, 2011.
[33] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.
[34] B.W. Kernighan, D.M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[35] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.2, Document Revision 15*. Khronos Group, 2011.
[36] *NVIDIA CUDA C Programming Guide, Version 4.2*. NVIDIA, 2012.
[37] *OpenACC Application Programming Interface, Version 1.0*. OpenACC-Standard.org, 2011.
[38] M. Papadrakakis, G. Stavroulakis, A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU-GPU architectures. *Comput. Methods Appl. Mech. Eng.*, **200**(13–16): 1490–1508, 2011.
[39] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.1*, 2011.
[40] Mesquite: mesh quality improvement toolkit, version 2.2.0. http://www.cs.sandia.gov/optimization/knupp/Mesquite.html, accessed June 20, 2012.
[41] M. Brewer, L. Freitag Diachin, P. Knupp, T. Leurent, D. Melander. The Mesquite Mesh Quality Improvement Toolkit. In *Proceedings of the 12th International Meshing Roundtable*, 239–250, 2003.
[42] *MPI: A Message-Passing Interface Standard, Version 2.2*. Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee, 2009.
[43] GNU compiler collection, version 4.7.1. http://gcc.gnu.org/, accessed July 4, 2012.
[44] Drexel University, Geometric & Intelligent Computing Laboratory model repository. http://edge.cs.drexel.edu/repository/, accessed July 3, 2012.
[45] Aim@Shape mesh repository. http://shapes.aim-at-shape.net/, accessed July 17, 2012.