Algorithm 932: PANG: Software for Nonmatching Grid Projections in 2D and 3D with Linear Complexity

MARTIN J. GANDER, Université de Genève
CAROLINE JAPHET, Université Paris XIII, INRIA, and University of Maryland

We design and analyze an algorithm with linear complexity to perform projections between 2D and 3D nonmatching grids. This algorithm, named the PANG algorithm, is based on an advancing front technique and neighboring information. Its implementation is surprisingly short, and we give the entire Matlab code. For computing the intersections, we use a direct and numerically robust approach. We show numerical experiments both for 2D and 3D grids, which illustrate the optimal complexity and negligible overhead of the algorithm. An outline of this algorithm has already been presented in a short proceedings paper of the 18th International Conference on Domain Decomposition Methods (see Gander and Japhet [2008]).

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation*; H.4.0 [**Information Systems Applications**]: General; I.7.2 [**Document and Text Processing**]: Document Preparation—*Languages and systems*; *Photocomposition and typesetting*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Advancing front algorithm, linear complexity, non-matching grid projections

ACM Reference Format:
Gander, M. J. and Japhet, C. 2013. Algorithm 932: PANG: Software for nonmatching grid projections in 2D and 3D with linear complexity. ACM Trans. Math. Softw. 40, 1, Article 6 (September 2013), 25 pages.
DOI: http://dx.doi.org/10.1145/2513109.2513115

1. INTRODUCTION

The need to project or interpolate grid functions from one grid onto grid functions of a nonmatching second grid is common in many areas of scientific computing. Examples of methods requiring such a grid transfer are the Chimera methods proposed by Steger et al. [1983] and analyzed in Brezzi et al. [2001], the Mortar methods first introduced in Bernardi et al. [1993, 1994] and extended in Ben Belgacem [1999], Belgacem and Maday [1999], and Braess and Dahmen [1998], the Patch methods by Glowinski et al. [2005] for local refinement, and extended in Picasso et al. [2008], and in Kamga and Pironneau [2007], and Optimized Schwarz waveform relaxation methods introduced in Gander [1997] and Gander et al. [1999] for linear parabolic and hyperbolic problems with constant coefficients, and extended in Martin [2005], Bennequin et al. [2009], Gander et al. [2007], and Halpern et al. [2010]. Chimera methods are domain

This work was partially supported by French ANR (COMMA) and GNR MoMaS.
Authors' addresses: M. J. Gander, Université de Genève, 2-4 rue du Lièvre, CP 64 CH-1211 Genève; email: Martin.Gander@unige.ch; C. Japhet, LAGA, Université Paris XIII, 99 Avenue J-B Clément, 93430 Villetaneuse, France, INRIA Paris-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, and CSCAMM, University of Maryland College Park, MD 20742; email: japhet@math.univ-paris13.fr.

DOI: http://dx.doi.org/10.1145/2513109.2513115

decomposition methods specialized for time dependent problems with moving parts, and thus naturally lead to nonmatching grids, to avoid regridding at each time step. The Mortar method is a domain decomposition method that permits both an entirely parallel generation of meshes, local adaptive meshes and fast and independent solvers in the subdomains. The Patch method, also known as "numerical zoom", is a multiscale method: a problem is solved on a computational domain with a coarse mesh, and then locally refined patches are applied in regions where more accuracy in the solution is required. The Schwarz waveform relaxation method is a domain decomposition method that computes in the subdomains on the whole time interval, exchanging space-time boundary data through optimized transmission operators. This method naturally applies to different space-time discretizations in subdomains, possibly nonconforming, and needs a very small number of iterations to converge.

All of these methods need a transfer of an approximate solution from one grid to a nonmatching neighboring grid. In the literature, this transfer problem has appeared under various names: intergrid communication problem [Meakin 1991], grid transfer problems [Plimpton et al. 1998], grid intersection problem [Lee et al. 2004], or overlaying surface meshes problem [Jiao 2001; Jiao and Heath 2004; Jain 2007]. Similar approaches for the interpolation of discrete approximations between nonmatching grids can be found in Löhner [2001, Chapter 13].

In order to obtain a robust and efficient projection algorithm, two main difficulties need to be addressed: a combinatorial one, in the sense that if n is the number of elements touching the interface, we search for a faster algorithm than the naive $O(n^2)$ algorithm, which just checks for every element in one grid, if it intersects with every other element of the other grid, see for example Flemisch et al. [2006]. The second difficulty is to numerically compute the intersection of two finite elements, which requires taking a numerical decision as to whether two segments (or surfaces) intersect or not, and whether one point is in an element or not. There is extensive literature on both problems, for a review, see Gander and Japhet [2008].

In this article, we present a compact algorithm with optimal complexity, which computes, for two nonmatching meshes, the associated mortar projection matrix (see Bernardi et al. [1993]), or any other similar quantity defined on the intersection of the elements. Thus we name this new algorithm Projection Algorithm for Nonmatching Grids (PANG).

This algorithm was introduced in Gander et al. [2005], Gander and Japhet [2008], and, after personal communication, implemented in Bastian et al. [2010] and DUNE [2010] for the corresponding software. Other software packages have been developed to handle nonmatching meshes, and see for example Hecht [2012] and MpCCI [2013].

In Section 2, we describe the projection algorithm from Gander and Japhet [2008] for irregular triangular grids. In Section 3, the projection algorithm is extended to nonmatching tetrahedral grids. We show in Section 4, numerical experiments both in 2D and 3D, which illustrate the optimal complexity and negligible overhead of the algorithm.

2. PROJECTION ALGORITHM FOR 2D MESHES

In this section, we present the algorithm that computes for two nonmatching 2D meshes representing the same planar geometry, the associated mortar projection matrix (see Bernardi et al. [1993]), or any other similar quantity defined on the intersection of the two meshes by `MortarInt` in the `Intersect` procedure. For mortar projection matrices, the 2D meshes are the interface meshes between two domains in 3D, which are nonconforming at the interface, and hence lead to the need for projection between nonmatching grids. An example that illustrates the complexity is given in Figure 1.

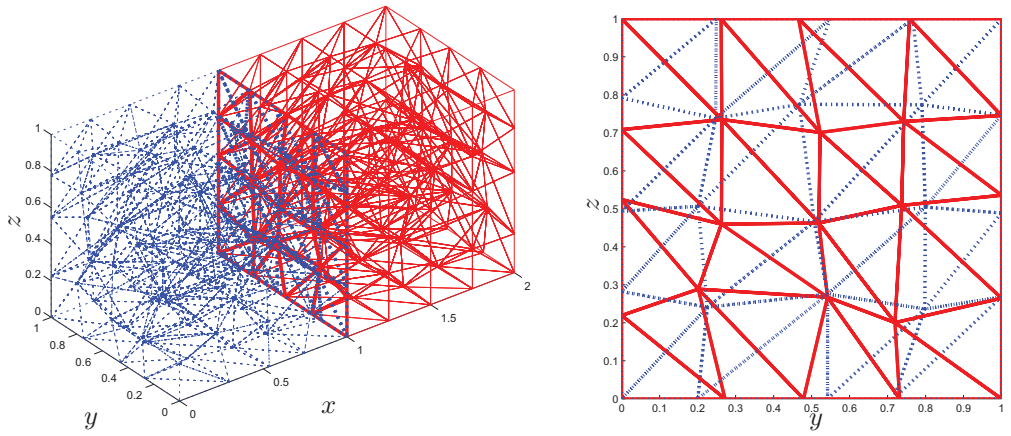

Fig. 1. Nonconforming tetrahedral discretization with two subdomains on the left, and the resulting triangular interface meshes on the right.

2.1. Algorithms for Computing the Intersection of Two Triangles

Elegant algorithms for computing intersections of polygons have been developed in the computer graphics community, and they are known under the name "polygon clipping algorithms", (see Weiler and Atherton [1977] and Greiner and Hormann [1998]). The basic idea of the algorithm is as follows: if the starting point on the one polygon is inside the other one, then one marches along the edges of the one polygon, and whenever an intersection is found, one switches the polygons and marches on the edges of the other one. As soon as one returns to a point already visited, the intersection polygon is complete. If the starting point on one polygon is outside of the other, one does not switch polygons for the first detected intersection, but starts from the second detected intersection. An example for the case of two triangles is shown in Figure 2, where we started with a corner of the blue (dashed) triangle outside of the intersection, and then find sequentially $P(1), P(2), \ldots, P(6)$, and then again $P(1)$, which closes the intersection polygon. This algorithm is widely used in computer graphics. In Greiner and Hormann [1998], a generalized version that can also handle self intersecting polygons is proposed, with the following quote:

> So far we have tacitly assumed that there are no degeneracies, i.e. each vertex of one polygon does not lie on an edge of the other polygon. Degeneracies can be detected in the intersect procedure [...] In this case we perturb the vertex slightly. If we take care that the perturbation is less than a pixel width, the output on the screen will be correct.

While for computer graphics, the pixel level is a natural scale for the perturbation, it is difficult to define acceptable perturbations for the nonmatching grid projection algorithm. In addition to the functioning of the algorithm, these perturbations are essential, as we illustrate in Figure 3. In the first frame, a node of the red (solid) triangle lies exactly on an edge of the blue (dashed) one. If, in the intersection algorithm, an intersection is calculated numerically, the algorithm is supposed to switch onto the blue (dashed) triangle, which is incorrect in the first frame, but correct in the second one. In the third example, two edges coincide, and in the fourth, two consecutive edges coincide—situations that appear typically in any mortar code (see Figure 1). Since we did not find a concise and robust solution in the polygon clipping algorithm for these problems in the context of the mortar application, we propose a completely different

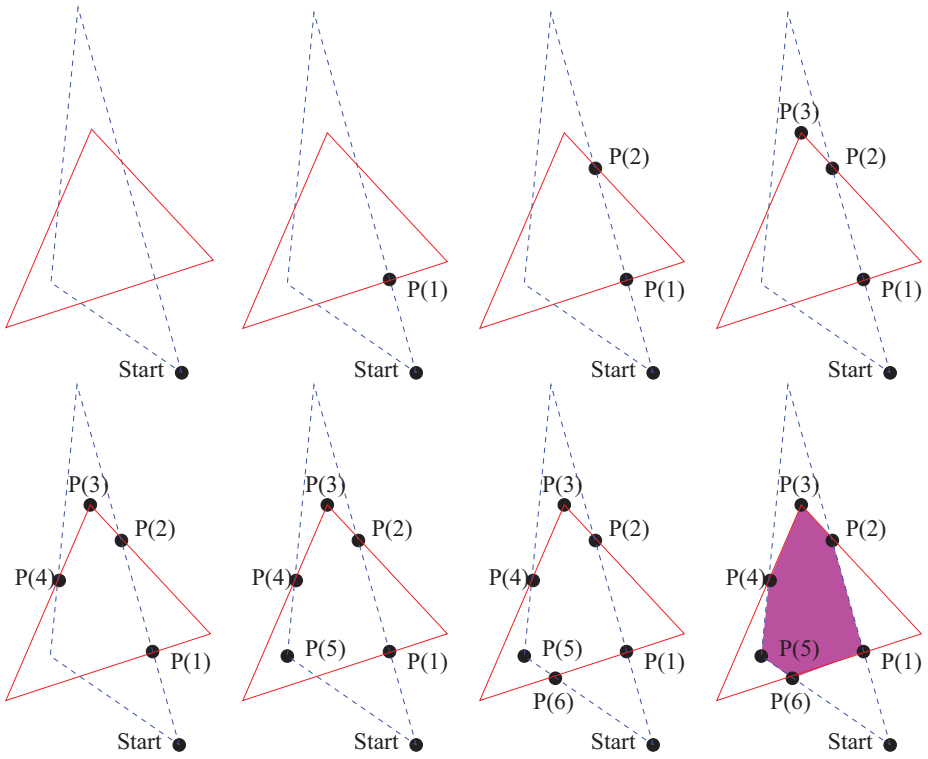

Fig. 2. Using the classical polygon clipping algorithm for computing the intersection of two triangles.

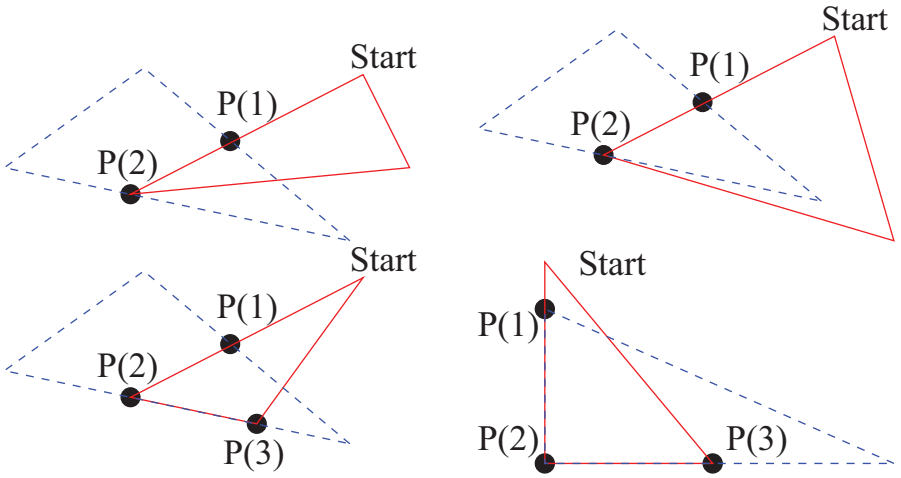

Fig. 3. Numerical difficulties in the classical polygon clipping algorithm for computing the intersection of two triangles.

algorithm. Another approach is given in Shewchuk [1997] who uses adaptive high precision floating point arithmetic to resolve difficult numerical decisions.

We present now our algorithm which computes the intersection polygon of two arbitrary triangles. Our algorithm turns out to be numerically robust and we show its

```
function [P,n,M]=Intersect(X,Y);
% INTERSECT intersection of two triangles and mortar contribution
%   [P,n,M]=Intersect(X,Y); computes for the two given triangles X
%   and Y the points P where they intersect, in n the indices of
%   which neighbors of X are also intersecting with Y, and the
%   local mortar matrix M of contributions of the element X on the
%   element Y.

[P,n]=EdgeIntersections(X,Y);
Q=PointsOfXInY(X,Y);
if size(Q,2)>1                          % if two or more interior
  n=[1 1 1];                            % points the triangle is
end                                     % candidate for all neighbors
P=[P Q];
P=[P PointsOfXInY(Y,X)];
P=SortAndRemoveDoubles(P);              % sort counter clock wise
M=zeros(3,3);
if size(P,2)>0
  for j=2:size(P,2)-1                   % compute interface matrix
    M=M+MortarInt(P(:,[1 j j+1]),X,Y);
  end;
%  patch(P(1,:),P(2,:),'m')             % draw intersection for illustration
%  pause(1)
end;
```

Fig. 4. Algorithm for computing the intersection of two triangles, and the integrals on this intersection of the products of finite element shape functions of the two triangles.

generalization to three dimensional interfaces in Section 3. It works as follows: we first compute all edge intersections, and for each triangle, all its corners that are contained in the other triangle. We then order the set of points obtained counterclockwise. This leads to the simple intersection algorithm shown in Figure 4. For two triangles `X` and `Y` with corner coordinates stored column-wise, the graphic primitive `EdgeIntersections(X,Y)` computes all intersections of edges of triangle `X` with edges of triangle `Y`, and the routine `PointsOfXInY(X,Y)` computes corners of triangle `X` in triangle `Y`. These two routines include borderline cases by using greater or equal in the decisions. The routine `SortAndRemoveDoubles(P)` sorts the points in `P` in counterclockwise order and removes duplicates, which turns out to make the algorithm numerically robust. A possible implementation of these basic routines is given in Appendix Section B.

The algorithm also returns the important information of which neighboring triangles of `X` will also intersect with `Y`, and the integrals on the intersection `P` of products of element shape functions of `X` with the ones of `Y`, computed by the routine `MortarInt` and gathered in the small matrix `M`. A possible implementation of this routine is also given in Appendix Section B.

2.2. Algorithms for Efficient Grid Traversals

An excellent overview for intergrid interpolation algorithms for nonstructured grids is given in Löhner [2001, Chapter 13] (see also the references therein). When many points need to be interpolated from an unstructured grid, the best algorithms that exhibit linear complexity, use neighboring information. One can use an advancing front technique that starts, for each new point at which one needs to interpolate data, a local search in the neighborhood of the element where the previous point was interpolated. Only if this search is not successful in less than a constant number of steps is a brute force search launched (see [Löhner 2001]). A related technique [Lee et al. 2004], with

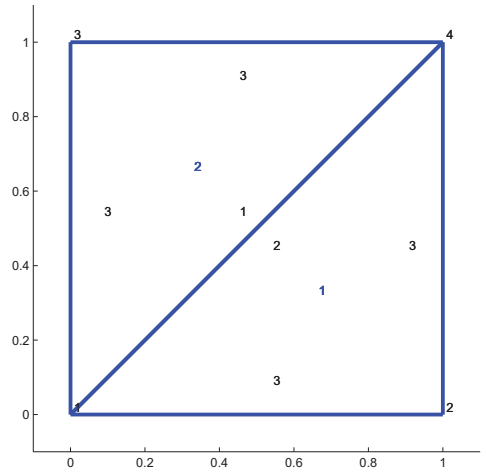

Fig. 5. Example of a simple mesh with neighboring information (see text).

approximately linear complexity, uses a self-avoiding walk, with a vicinity search, that can only fail after a boundary element had no intersection with any element of the other mesh; in this case a quad-tree search is employed. Further techniques for treating the boundary are given in Löhner [2001]. In Brune et al. [2013], local traversals for nodes-nested meshes are used to efficiently construct interpolation operators in the multigrid context.

We propose here an advancing front algorithm that uses the fact that each triangle knows its neighbors. For our algorithm, each triangle of a mesh is stored in a corresponding line of the mesh matrix `T`; the first three entries are indices into a table of nodal coordinates `N` stored column-wise, and the next three entries represent neighboring element indices, or the total number of triangles plus one to indicate a real boundary. For example, the unit square domain with two triangles shown in Figure 5 is given by the following node and mesh table.

```
N=[0 1 0 1
   0 0 1 1];
T=[1 2 4 3 3 2
   1 4 3 1 3 3];
```

Note that both nodes of the triangle and the neighboring triangle indices are ordered counterclockwise. The algorithm is defined as follows: given two mesh tables of triangular meshes `Ta` for mesh a, and `Tb` for mesh b, the algorithm starts with a pair of triangles that intersect, as shown in the first drawing in Figure 6 with the bold red (solid) triangle and the bold blue (dashed) triangle. Two triangles that intersect are often available at a corner, and in our algorithm we assume that these are the first triangles in `Ta` and `Tb`. Otherwise, one could also find two intersection triangles by one direct search, which does not alter the overall complexity of the algorithm. Then the algorithm computes the intersection of these first two triangles (step 2 in Figure 6). The next step (step 3 in Figure 6), consists of adding the neighbors of the triangle from mesh a, namely the bold blue (dashed) triangles, in a list `al`, as possible candidates that may intersect our triangle from mesh b. We continue intersecting with each triangle in the list `al`, and add untreated further neighbors to the list `al`, as shown in steps 4 to 9 in Figure 6. The shaded blue triangles with right-leaning diagonal lines are waiting in list `al` to be treated. We do not add neighbors of a triangle to the list `al` if

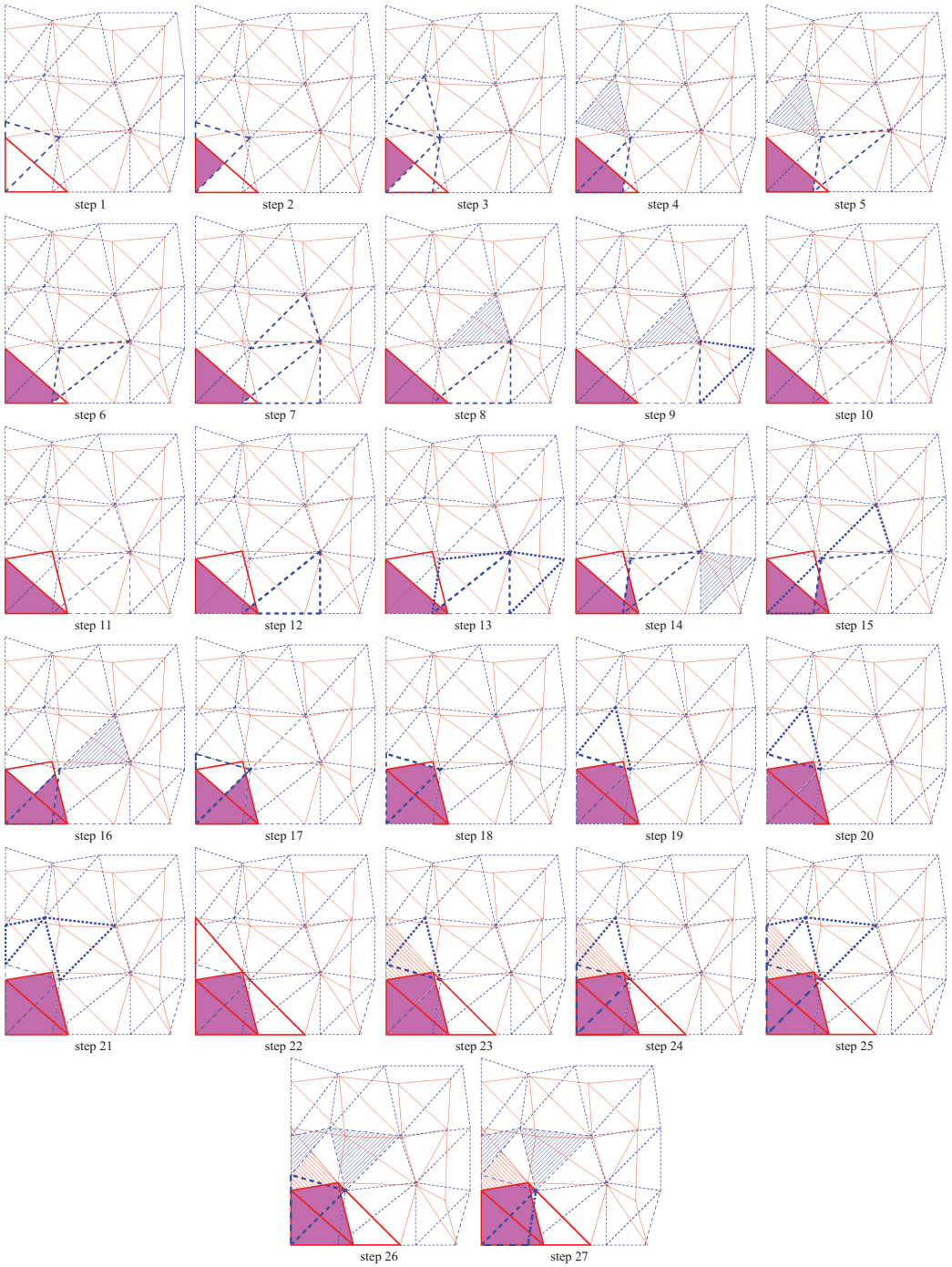

Fig. 6. First steps of an advancing front technique to compute the projection matrix for two arbitrary nonconforming triangular meshes.

```
function M=InterfaceMatrix(Na,Ta,Nb,Tb);
% INTERFACEMATRIX projection matrix for nonmatching triangular grids
%   M=InterfaceMatrix(Na,Ta,Nb,Tb); takes two triangular meshes Ta
%   and Tb with associated nodal coordinates in Na and Nb and
%   computes the interface projection matrix M

bl=[1];                          % bl: list of triangles of Tb to treat
bil=[1];                         % bil: list of triangles Ta to start with
bd=zeros(size(Tb,1)+1,1);        % bd: flag for triangles in Tb treated
bd(end)=1;                       % guard, to treat boundaries
bd(1)=1;                         % mark first triangle in b list
M=sparse(size(Nb,2),size(Na,2));
while length(bl)>0
  bc=bl(1); bl=bl(2:end);        % bc: current triangle of Tb
  al=bil(1); bil=bil(2:end);     % triangle of Ta to start with
  ad=zeros(size(Ta,1)+1,1);      % ad: flag for triangles in Ta treated
  ad(end)=1;                     % guard, to treat boundaries
  ad(al)=1;                      % mark first triangle in a list
  n=[0 0 0];                     % triangles intersecting with neighbors
  while length(al)>0
    ac=al(1); al=al(2:end);      % take next candidate
    [P,nc,Mc]=Intersect(Nb(:,Tb(bc,1:3)),Na(:,Ta(ac,1:3)));
    if ~isempty(P)               % intersection found
      M(Tb(bc,1:3),Ta(ac,1:3))=M(Tb(bc,1:3),Ta(ac,1:3))+Mc;
      t=Ta(ac,3+find(ad(Ta(ac,4:6))==0));
      al=[al t];                 % add neighbors
      ad(t)=1;                   % mark them as treated
      n(find(nc>0))=ac;          % ac is starting candidate for neighbor
    end
  end
  tmp=find(bd(Tb(bc,4:6))==0); % find non-treated neighbors
  idx=find(n(tmp)>0);            % take those which intersect with Ta
  t=Tb(bc,3+tmp(idx));
  bl=[bl t];                     % and add them
  bil=[bil n(tmp(idx))];         % with starting candidates Ta
  bd(t)=1;                       % mark them as treated
end
```

Fig. 7. Algorithm with linear complexity for computing the projection matrix of two nonmatching triangular grids.

the triangle did not intersect with our triangle from mesh b. We continue this process until list `al` becomes empty in step 10 of Figure 6. Now we know that the starting bold red triangle from mesh b cannot intersect any more triangles from mesh a, so we put all the neighbors of the starting triangle of mesh b into a list `bl`; this is only the bold red (solid) triangle in step 11 in Figure 6 in this case. Starting in step 12 with the bold blue (dashed) triangle, we kept as a starting candidate from the previous intersection computation, we now perform the same steps we used for the first red triangle on each of the red triangles in list `bl` until it becomes empty, and the algorithm terminates. Graphically, we denote by shaded red triangles with left-leaning diagonal lines, the ones that are currently on hold in list `bl`, and one can see in step 27 that the algorithm has not yet finished as there are still many triangles left in lists `al` and `bl` to be treated. The Matlab implementation of our algorithm is shown in Figure 7.

We now study the complexity of our projection algorithm: the key step is that we stored a starting candidate from list `al` for each of the triangles added to list `bl` in list `bil`. This information is obtained without extra calculation in the computation of the

intersection. Thus in step 12 of Figure 6, we know with which blue (dashed) triangle we have to start to compute intersections, and there is never a search needed for a candidate triangle of mesh a that could intersect with the currently treated triangle from mesh b. The algorithm treats triangles of mesh b one by one, and checks for each triangle, at most a constant number of triangles in mesh a, which shows that the expected complexity is linear. However, when every triangle of mesh a intersects every triangle of mesh b, then the constant equals the total number of triangles in mesh a and this leads to quadratic complexity and cannot be avoided in this case.

Note that this advancing front algorithm only uses the fact that each element has a certain number of neighbors—equal to 3 in the implementation shown. Thus the algorithm is independent of the number of dimensions. However, we assumed that the two meshes are simply connected, and also that the intersection of one element with the elements of the other mesh are simply connected, assumptions which generally hold for mortar methods. Otherwise the algorithm would need extra starting points in order to find the complete intersections.

An example of how to run the code is given in Appendix Section B.

3. PROJECTION ALGORITHM FOR 3D MESHES

In the context of mortar methods, three dimensional interfaces appear naturally when solving time dependent three dimensional problems. For patch methods, three dimensional interfaces appear naturally when solving three dimensional problems. The combinatorial algorithm for computing the mesh projection shown in Figure 7 is independent of the dimension, except for small changes in the number of indices, since the triangles have become tetrahedra now (or prisms for time dependent problems), and we give a precise implementation in Appendix Section C. The intersection algorithm shown in Figure 4 however needs to be generalized to three dimensional objects. To do so, we can proceed along the same lines as in our new 2D intersection algorithm: in the case of tetrahedral meshes, one first calculates all points were an edge of one tetrahedron traverses the face of the other, then all corners of one tetrahedron contained in the other, and finally orders the points face by face counter clockwise. The implementation of this algorithm is given in Appendix Section A.

The following example shows how the algorithm in 3D should be called, and also gives the structure of the input data. The intersection algorithm in 3D again uses several geometric primitives: `TriangleLineIntersection`, `InsertPoint`, `RemoveDuplicates`, `NewFace`, `OrderPoints`, and `PyramidVolume`, a possible implementation of which is given in Appendix Section C.

To test the algorithm, an example in Matlab is given in Appendix Section C.

Remark 3.1. In Matlab, one can replace the routines `RemoveDuplicates`, `PointsOfXinY` and `PointInTetrahedra` by the built-in Matlab routines `unique`, `inpolygon`, and `tsearchn` respectively. However, in this article, we prefer to give all the needed routines in detail so that the algorithm can be used in other languages such as Fortran, C or Python.

4. NUMERICAL EXPERIMENTS

We compare in Figure 8, our algorithm with brute force search, where for each element in the first mesh the intersection with every element in the second mesh is computed (see Appendix Section B for the code). We start with two random triangular meshes and present the average computing time for twenty projection calculations in two dimensions on the left, and for five projection calculations for the three dimensional case on the right, for the code see Appendix Section C. One can clearly see that the new algorithm scales linearly with the number of elements, while the brute force algorithm

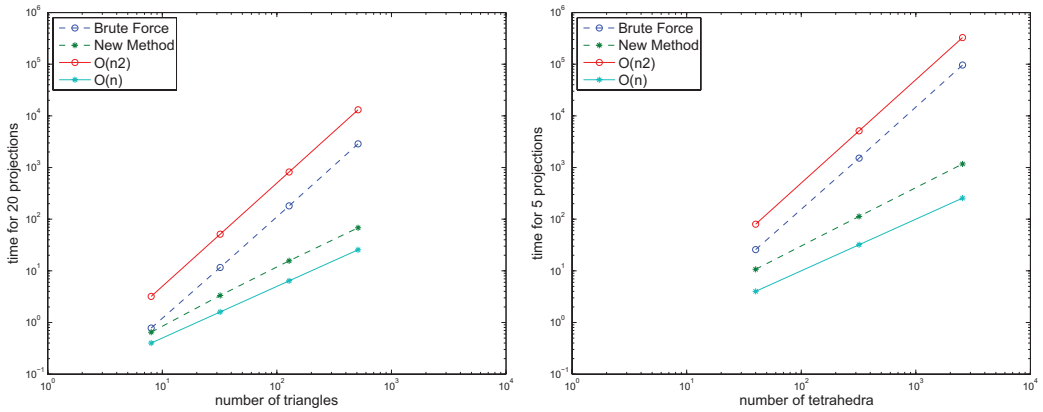

Fig. 8. Comparison in computing time for two-dimensional meshes on the left, and three-dimensional meshes on the right.

scales quadratically. We observe that in addition to asymptotic superiority, the new projection algorithm is already competitive for small meshes, that is, the algorithmic overhead is negligible.

5. CONCLUSIONS

We have presented intersection algorithms for nonmatching grids in two and three spatial dimensions. The algorithms all have linear complexity, and address the numerically difficult intersection process of geometric objects by a new, robust approach. Our numerical results show that the projection algorithm has a negligible algorithmic overhead, and thus it is already competitive for small meshes.

In the intersection algorithm we presented for two triangles/tetrahedra, before starting the computation of the intersection, one could use an inexact range test in order to quickly exclude nonintersecting triangles/tetrahedra and thus make the algorithm slightly faster.

The advancing front technique has already been extended to two nonconforming interfaces that do not quite lie in the same physical manifold (e.g. for contact problems), and thus an additional projection normal to the interface is necessary. It is currently adapted to the case of two-dimensional meshes that are immersed in a three-dimensional domain.

APPENDIXES

APPENDIX A. INTERSECTION3D ROUTINE

In this appendix, we give the intersection algorithm generalized to three-dimensional meshes.

```
function [P,nc,H,v]=Intersection3d(X,Y);
% INTERSECTION3D computes intersection polyhedron of two tetrahedra
%   [P,nc,H,v]=Intersection3d(X,Y); computes for the two given tetrahedra
%   X and Y (point coordinates are stored column-wise) the points P
%   where they intersect, in nc the bolean information of which
%   neighbors of X are also intersecting with Y, in H stored row-wise
%   the polygons of the faces of the intersection polyhedron, stored
%   counter-clock-wise looking from the outside, and in v the volume
%   of the intersection.
```

```
P=[];nc=[0 0 0 0];
sxk=zeros(4,1);                          % intersection node index
syk=zeros(4,1);                          % on surfaces of X and Y
l1=[1 2 3 4 1 2];                        % enumeration of lines
l2=[2 3 4 1 3 4];
s1=[1 1 2 3 1 2];                        % faces touching each line
s2=[4 2 3 4 3 4];
ni=[1 3 4; 1 2 4; 1 2 3; 2 3 4];         % faces touching each node
for i=1:6                                % find intersections of edges of
  for j=1:4                              % X with surfaces of Y
    p=TriangleLineIntersection([X(:,l1(i)) X(:,l2(i))],...
      [Y(:,j) Y(:,mod(j,4)+1) Y(:,mod(j+1,4)+1)]);
    if ~isempty(p)
      [k,P]=InsertPoint(p,P);            % insert point if new
      nc(s1(i))=1;nc(s2(i))=1;
      syk(j)=syk(j)+1;SY(j,syk(j))=k;  % remember to which surfaces
      sxk(s1(i))=sxk(s1(i))+1;SX(s1(i),sxk(s1(i)))=k;  % it belongs
      sxk(s2(i))=sxk(s2(i))+1;SX(s2(i),sxk(s2(i)))=k;
    end;
  end;
end;
for i=1:6                                % find intersections of edges of
  for j=1:4                              % Y with surfaces of X
    p=TriangleLineIntersection([Y(:,l1(i)) Y(:,l2(i))],...
      [X(:,j) X(:,mod(j,4)+1) X(:,mod(j+1,4)+1)]);
    if ~isempty(p)
      [k,P]=InsertPoint(p,P);
      nc(j)=1;
      sxk(j)=sxk(j)+1;SX(j,sxk(j))=k;
      syk(s1(i))=syk(s1(i))+1;SY(s1(i),syk(s1(i)))=k;
      syk(s2(i))=syk(s2(i))+1;SY(s2(i),syk(s2(i)))=k;
    end;
  end;
end;

fn=[0 0 0 0];
for i=1:4                                % find interior points of X in Y
  if PointInTetrahedron(X(:,i),Y)
    [k,P]=InsertPoint(X(:,i),P);
    sxk(ni(i,1))=sxk(ni(i,1))+1;SX(ni(i,1),sxk(ni(i,1)))=k;
    sxk(ni(i,2))=sxk(ni(i,2))+1;SX(ni(i,2),sxk(ni(i,2)))=k;
    sxk(ni(i,3))=sxk(ni(i,3))+1;SX(ni(i,3),sxk(ni(i,3)))=k;
    p=X(:,i); fn(i)=1;
  end;
  if PointInTetrahedron(Y(:,i),X)       % and vice versa
    [k,P]=InsertPoint(Y(:,i),P);
    syk(ni(i,1))=syk(ni(i,1))+1;SY(ni(i,1),syk(ni(i,1)))=k;
    syk(ni(i,2))=syk(ni(i,2))+1;SY(ni(i,2),syk(ni(i,2)))=k;
    syk(ni(i,3))=syk(ni(i,3))+1;SY(ni(i,3),syk(ni(i,3)))=k;
    p=Y(:,i);
  end;
end;
if sum(fn)>1                             % more than one point inside
  nc=[1 1 1 1];                          % means all neighbors intersect
end;
H=[];                                    % contains surfaces of the
n=0;                                     % intersection polyhedron
```

```
v=0;
if size(P,2)>3                          % construct intersection polyhedra,
  cm=sum(P')'/size(P,2);                % center of mass
  for i=1:4                             % scan surfaces of X and Y
    if sxk(i)>0
      no=RemoveDuplicates(SX(i,1:sxk(i)));
      if length(no)>2 & NewFace(H,no)  % don't compute degenerate polygons
        p=P(:,no);                      % and check if face is new
        [p,id]=OrderPoints(p,cm);       % order points counter-clock-wise
        n=n+1;
        H(n,1:length(id))=no(id);       % add surface
        v=v+PyramidVolume(p,cm);        % add volume
      end;
    end
    if syk(i)>0
      no=RemoveDuplicates(SY(i,1:syk(i)));
      if length(no)>2 & NewFace(H,no)
        p=P(:,no);
        [p,id]=OrderPoints(p,cm);
        n=n+1;
        H(n,1:length(id))=no(id);
        v=v+PyramidVolume(p,cm);
      end;
    end
  end
end;
```

APPENDIX B. REMAINING 2D ROUTINES

We show in this section a possible Matlab implementation of the complete remaining routines in 2D in order to test the code, and also give two examples of how to run it.

B.1. Remaining Geometric Primitives

Here is a possible implementation of the remaining geometric primitives `EdgeIntersections`, `PointsOfXInY` and `SortAndRemoveDoubles`, which are used by our intersection algorithm implementation in Figure 4:

```
function [P,n]=EdgeIntersections(X,Y)
% EDGEINTERSECTIONS computes edge intersections of two triangles
%   [P,n]=EdgeIntersections(X,Y) computes for the two given triangles X
%   and Y the points P where their edges intersect. In addition,
%   in n the indices of which neighbors of X are also intersecting
%   with Y are given.

P=[]; k=0; n=[0 0 0];
for i=1:3                               % find all intersections of edges
  for j=1:3
     b=Y(:,j)-X(:,i);
     A=[X(:,mod(i,3)+1)-X(:,i) -Y(:,mod(j,3)+1)+Y(:,j)];
     if rank(A)==2                      % edges not parallel
       r=A\b;
       if r(1)>=0 & r(1)<=1 & r(2)>=0 & r(2)<=1,  % intersection found
         k=k+1; P(:,k)=X(:,i)+r(1)*(X(:,mod(i,3)+1)-X(:,i)); n(i)=1;
       end;
     end;
  end;
end;
```

```
function P=PointsOfXInY(X,Y);
% POINTSOFXINY finds corners of one triangle within another one
%   P=PointsOfXInY(X,Y); computes for the two given triangles X
%   and Y (point coordinates are stored column-wise, in counter clock
%   order) the corners P of X which lie in the interior of Y.

k=0;P=[];
v0=Y(:,2)-Y(:,1); v1=Y(:,3)-Y(:,1);  % find interior points of X in Y
d00=v0'*v0; d01=v0'*v1; d11=v1'*v1;  % using baricentric coordinates
id=1/(d00*d11-d01*d01);
for i=1:3
  v2=X(:,i)-Y(:,1); d02=v0'*v2; d12=v1'*v2;
  u=(d11*d02-d01*d12)*id; v=(d00*d12-d01*d02)*id;
  if u>=0 & v>=0 & u+v<=1            % also include nodes on the boundary
    k=k+1; P(:,k)=X(:,i);
  end;
end;
```

Remark B.1. In all the tests over the past years, the preceding routine never failed to compute the correct intersection. There is however no proof using a detailed floating point analysis of this. In one case of a three-dimensional interface projected onto a two-dimensional surface, the routine `PointsOfXInY` failed, and we modified the decision line by adding a tolerance to guarantee that it also includes points that are numerically barely inside:

```
  if u>=0-eps & v>=0-eps & u+v<=1+eps % include nodes with guarantee
```

It remains open whether this is really necessary for the 2D grids we consider here. One could also extend the lines by `eps` in the routine **EdgeIntersections** to achieve a similar goal.

The third geometric primitive is:

```
function P=SortAndRemoveDoubles(P);
% SORTANDREMOVEDOUBLES sort points and remove duplicates
%   P=SortAndRemoveDoubles(P); orders polygon corners in P counter
%   clock wise and removes duplicates

ep=10*eps;                           % tolerance for identical nodes
m=size(P,2);
if m>0
  c=sum(P')'/m;                      % order polygon corners counter
  for i=1:m                          % clockwise
    d=P(:,i)-c; ao(i)=angle(d(1)+sqrt(-1)*d(2));
  end;
  [tmp,id]=sort(ao);
  P=P(:,id);
  i=1;j=2;                           % remove duplicates
  while j<=m
    if norm(P(:,i)-P(:,j))>ep
      i=i+1;P(:,i)=P(:,j);j=j+1;
    else
      j=j+1;
    end;
  end;
  P=P(:,1:i);
end;
```

B.2. Mortar Element Integration

Here is the implementation of the mortar integration routine `MortarInt`, useful when the mesh intersection algorithm is used for mortar methods:

```
function M=MortarInt(T,X,Y)
% MORTARINT computes mortar contributions
%   M=MortarInt(T,X,Y); computes for triangles X and Y the integrals
%   of all P1 element shape function combinations between triangles
%   X and Y on triangle T. The result is stored in the 3 by 3 matrix M

Jd=-T(1,1)*T(2,3)-T(1,2)*T(2,1)+T(1,2)*T(2,3)+T(1,1)*T(2,2)+...
   T(1,3)*T(2,1)-T(1,3)*T(2,2);

a=T(1,1);
d=T(2,1);
e=-T(2,1)+T(2,2);
b=-T(1,1)+T(1,2);
c=-T(1,1)+T(1,3);
f=-T(2,1)+T(2,3);

T1=1/2*(X(1,1)*(X(2,2)-X(2,3))-X(1,2)*(X(2,1)-X(2,3))+X(1,3)*(X(2,1)-X(2,2)));
T2=1/2*(Y(1,1)*(Y(2,2)-Y(2,3))-Y(1,2)*(Y(2,1)-Y(2,3))+Y(1,3)*(Y(2,1)-Y(2,2)));

A11=1/(2*abs(T1))*((X(2,2)-X(2,3))*b-(X(1,2)-X(1,3))*e);
B11=1/(2*abs(T1))*((X(2,2)-X(2,3))*c-(X(1,2)-X(1,3))*f);
C11=1/(2*abs(T1))*((X(2,2)-X(2,3))*a-(X(1,2)-X(1,3))*d ...
    +(X(1,2)*X(2,3)-X(2,2)*X(1,3)));
A12=1/(2*abs(T1))*((X(2,3)-X(2,1))*b+(X(1,1)-X(1,3))*e);
B12=1/(2*abs(T1))*((X(2,3)-X(2,1))*c+(X(1,1)-X(1,3))*f);
C12=1/(2*abs(T1))*((X(2,3)-X(2,1))*a+(X(1,1)-X(1,3))*d ...
    -(X(1,1)*X(2,3)-X(2,1)*X(1,3)));
A13=1/(2*abs(T1))*((X(2,1)-X(2,2))*b-(X(1,1)-X(1,2))*e);
B13=1/(2*abs(T1))*((X(2,1)-X(2,2))*c-(X(1,1)-X(1,2))*f);
C13=1/(2*abs(T1))*((X(2,1)-X(2,2))*a-(X(1,1)-X(1,2))*d ...
    +(X(1,1)*X(2,2)-X(2,1)*X(1,2)));
A21=1/(2*abs(T2))*((Y(2,2)-Y(2,3))*b-(Y(1,2)-Y(1,3))*e);
B21=1/(2*abs(T2))*((Y(2,2)-Y(2,3))*c-(Y(1,2)-Y(1,3))*f);
C21=1/(2*abs(T2))*((Y(2,2)-Y(2,3))*a-(Y(1,2)-Y(1,3))*d ...
    +(Y(1,2)*Y(2,3)-Y(2,2)*Y(1,3)));
A22=1/(2*abs(T2))*((Y(2,3)-Y(2,1))*b+(Y(1,1)-Y(1,3))*e);
B22=1/(2*abs(T2))*((Y(2,3)-Y(2,1))*c+(Y(1,1)-Y(1,3))*f);
C22=1/(2*abs(T2))*((Y(2,3)-Y(2,1))*a+(Y(1,1)-Y(1,3))*d ...
    -(Y(1,1)*Y(2,3)-Y(2,1)*Y(1,3)));
A23=1/(2*abs(T2))*((Y(2,1)-Y(2,2))*b-(Y(1,1)-Y(1,2))*e);
B23=1/(2*abs(T2))*((Y(2,1)-Y(2,2))*c-(Y(1,1)-Y(1,2))*f);
C23=1/(2*abs(T2))*((Y(2,1)-Y(2,2))*a-(Y(1,1)-Y(1,2))*d ...
    +(Y(1,1)*Y(2,2)-Y(2,1)*Y(1,2)));

M(1,1)=1/24*Jd*(2*A11*A21+B11*A21+A11*B21+2*B11*B21+4*C11*A21 ...
       +4*A11*C21+4*C11*B21+4*B11*C21+12*C11*C21);
M(1,2)=1/24*Jd*(2*A11*A22+B11*A22+A11*B22+2*B11*B22+4*C11*A22 ...
       +4*A11*C22+4*C11*B22+4*B11*C22+12*C11*C22);
M(1,3)=1/24*Jd*(2*A11*A23+B11*A23+A11*B23+2*B11*B23+4*C11*A23 ...
       +4*A11*C23+4*C11*B23+4*B11*C23+12*C11*C23);
M(2,1)=1/24*Jd*(2*A12*A21+B12*A21+A12*B21+2*B12*B21+4*C12*A21 ...
       +4*A12*C21+4*C12*B21+4*B12*C21+12*C12*C21);
```

```
M(2,2)=1/24*Jd*(2*A12*A22+B12*A22+A12*B22+2*B12*B22+4*C12*A22 ...
       +4*A12*C22+4*C12*B22+4*B12*C22+12*C12*C22);
M(2,3)=1/24*Jd*(2*A12*A23+B12*A23+A12*B23+2*B12*B23+4*C12*A23 ...
       +4*A12*C23+4*C12*B23+4*B12*C23+12*C12*C23);
M(3,1)=1/24*Jd*(2*A13*A21+B13*A21+A13*B21+2*B13*B21+4*C13*A21 ...
       +4*A13*C21+4*C13*B21+4*B13*C21+12*C13*C21);
M(3,2)=1/24*Jd*(2*A13*A22+B13*A22+A13*B22+2*B13*B22+4*C13*A22 ...
       +4*A13*C22+4*C13*B22+4*B13*C22+12*C13*C22);
M(3,3)=1/24*Jd*(2*A13*A23+B13*A23+A13*B23+2*B13*B23+4*C13*A23 ...
       +4*A13*C23+4*C13*B23+4*B13*C23+12*C13*C23);
```

B.3. Running an Example in 2D

The following example shows how the algorithm in 2D can be used for a simple model problem:

```
% 2D EXAMPLE: test the advancing front program for two dimensional meshes

[N1,T1]=Mesh2d(1); [N1,T1]=RefineMesh(N1,T1); [N1,T1]=RefineMesh(N1,T1);
[N2,T2]=Mesh2d(2); [N2,T2]=RefineMesh(N2,T2); [N2,T2]=RefineMesh(N2,T2);

fig=figure(1); clf; set(fig,'DoubleBuffer','on');
PlotMesh(N1,T1,'b'); PlotMesh(N2,T2,'r');
xlabel('x'); ylabel('y'); title('nonconforming grids in 2d');

M=InterfaceMatrix(N1,T1,N2,T2);
```

The example uses three basic routines to generate, refine, and plot meshes—namely `Mesh2d`, `RefineMesh` and `PlotMesh`, which are given below:

```
function [N,T]=Mesh2d(p)
% MESH2D generates simple 2d triangular meshes
%   [N,T]=Mesh2d(p); generates an initial coarse triangular mesh for
%   each value of p. The result is a list of triangles T which points
%   into the list of nodes N containing x and y coordinates.  The
%   triangle contains in entries 4 to 6 the neighboring triangle
%   indices, and as a guard # of triangles + 1 if there is no
%   neighbor.

if p==1
  N=[0 1 0 1
     0 0 1 1];
  T=[1 2 4 3 3 2
     1 4 3 1 3 3];
elseif p==2
  N=[0 1 0 1 0.5
     0 0 1 1  1];
  T=[1 2 5 4 2 3
     2 4 5 4 4 1
     1 5 3 1 4 4];
end
```

```
function [Nr,Tr]=RefineMesh(N,T);
% REFINEMESH refines the mesh by a factor of 4
%   [Nr,Tr]=RefineMesh(N,T); refines the mesh given by the nodes N and
%   the triangles T by cutting each triangle into four smaller ones.
%   The neighboring information in the triangles is also updated,
%   and the boundary guard is still number of trianlges plus one,
%   as in NewMesh.
```

```
Nr=N;                           % new node list starts with old one
nn=size(N,2);
Tr=[];                          % triangles start from scratch
nt=0;                           % number of new triangles
ep=10*eps;                      % to check identical nodes
for j=1:size(T,1),              % quadrisect all old triangles
  i=T(j,1:3); n=N(:,i);         % old nodes of current triangle
  n(:,4)=(n(:,1)+n(:,2))/2;     % 3 new nodes to be used
  n(:,5)=(n(:,1)+n(:,3))/2;
  n(:,6)=(n(:,2)+n(:,3))/2;
  for k=4:6,                    % insert new nodes in Nr if necessary
    l=find(Nr(1,:)==n(1,k));
    m=find(Nr(2,l)==n(2,k));
    if isempty(m),              % node not found, hence insert
      nn=nn+1;
      Nr(:,nn)=n(:,k);
      i(k)=nn;
    else
      i(k)=l(m);                % node was found
    end;
  end;
  Tr(nt+1,:)=[i(1) i(4) i(5) -1 nt+2 -1];
  Tr(nt+2,:)=[i(5) i(4) i(6) nt+1 nt+3 nt+4];
  Tr(nt+3,:)=[i(6) i(4) i(2) nt+2 -1 -1];
  Tr(nt+4,:)=[i(6) i(3) i(5) -1 -1 nt+2];
  nt=nt+4;
end;
for j=1:size(T,1),              % updating neighboring information
  i=T(j,1:3);
  ne=T(j,4:6);
  if ne(1)==size(T,1)+1         % real boundary
    Tr(4*(j-1)+1,4)=size(Tr,1)+1; Tr(4*(j-1)+3,5)=size(Tr,1)+1;
  else
    id=find(T(ne(1),1:3)==i(2));
    if id==1
      Tr(4*(j-1)+1,4)=4*(ne(1)-1)+3; Tr(4*(j-1)+3,5)=4*(ne(1)-1)+1;
    elseif id==2
      Tr(4*(j-1)+1,4)=4*(ne(1)-1)+4; Tr(4*(j-1)+3,5)=4*(ne(1)-1)+3;
    else
      Tr(4*(j-1)+1,4)=4*(ne(1)-1)+1; Tr(4*(j-1)+3,5)=4*(ne(1)-1)+4;
    end;
  end;
  if ne(2)==size(T,1)+1         % real boundary
    Tr(4*(j-1)+3,6)=size(Tr,1)+1; Tr(4*(j-1)+4,4)=size(Tr,1)+1;
  else
    id=find(T(ne(2),1:3)==i(3));
    if id==1
      Tr(4*(j-1)+3,6)=4*(ne(2)-1)+3; Tr(4*(j-1)+4,4)=4*(ne(2)-1)+1;
    elseif id==2
      Tr(4*(j-1)+3,6)=4*(ne(2)-1)+4; Tr(4*(j-1)+4,4)=4*(ne(2)-1)+3;
    else
      Tr(4*(j-1)+3,6)=4*(ne(2)-1)+1; Tr(4*(j-1)+4,4)=4*(ne(2)-1)+4;
    end;
  end;
  if ne(3)==size(T,1)+1         % real boundary
    Tr(4*(j-1)+4,5)=size(Tr,1)+1; Tr(4*(j-1)+1,6)=size(Tr,1)+1;
  else
```

```
    id=find(T(ne(3),1:3)==i(1));
    if id==1
      Tr(4*(j-1)+4,5)=4*(ne(3)-1)+3; Tr(4*(j-1)+1,6)=4*(ne(3)-1)+1;
    elseif id==2
      Tr(4*(j-1)+4,5)=4*(ne(3)-1)+4; Tr(4*(j-1)+1,6)=4*(ne(3)-1)+3;
    else
      Tr(4*(j-1)+4,5)=4*(ne(3)-1)+1; Tr(4*(j-1)+1,6)=4*(ne(3)-1)+4;
    end;
  end;
end;
```

```
function PlotMesh(N,T,col);
% PLOTMESH plots a triangluar mesh
%   PlotMesh(N,T); plots the mesh given by the nodes N and triangles
%   T. The real boundaries are drawn in bold and for small meshes
%   the node numbers are added as well.

axis('equal');
for i=1:size(T,1),
  for j=1:3,
    line([N(1,T(i,j)) N(1,T(i,mod(j,3)+1))], ...
      [N(2,T(i,j)) N(2,T(i,mod(j,3)+1))],'Color',col);
    bc=mean(N(:,T(i,1:3))')';
  end;
end;
m=size(N,2);
if m<10,
  for i=1:m,
    text(N(1,i)+.01,N(2,i)+.02,num2str(i));
  end;
end;
```

B.4. Testing the Complexity

We give here an example code to compare the computing time of the new method with a brute force algorithm, also given in the following:

```
% 2D COMPLEXITY compares the computing time for 2d projections

nr=3;                          % number of mesh refinements
np=20;                         % number of projection calculations
vtb=zeros(1,nr); vtn=zeros(1,nr); s=zeros(1,nr);
for k=1:nr
  [N1,T1]=Mesh2d(1); [N2,T2]=Mesh2d(2);
  for l=1:k
    [N1,T1]=RefineMesh(N1,T1); [N2,T2]=RefineMesh(N2,T2);
  end
  tb=0;tn=0;
  for i=1:np
    N1(:,5:end)=N1(:,5:end)+(0.1*rand(size(N1(:,5:end)))-0.05);
    N2(:,5:end)=N2(:,5:end)+(0.1*rand(size(N2(:,5:end)))-0.05);
    t0=clock; M=InterfaceMatrix(N1,T1,N2,T2); tn=tn+etime(clock,t0);
    t0=clock; Mb=InterfaceMatrixBruteForce(N1,T1,N2,T2); tb=tb+etime(clock,t0);
  end
  vtn(k)=tn; vtb(k)=tb; s(k)=size(T1,1);
end
c1=vtb(end)/s(end)^2; c2=vtn(end)/s(end); % align theoretical and numerical curves
loglog(s,vtb,'--o',s,vtn,'--*',s,c1*s.^2,'-o',s,c2*s,'-*');
```

```
legend('Brute Force','New Method','O(n^2)','O(n)',2);
xlabel('number of triangles'); ylabel('time for 20 projections');


function M=InterfaceMatrixBruteForce(Na,Ta,Nb,Tb);
% INTERFACEMATRIXBRUTEFORCE projection matrix for nonmatching grids
%   M=InterfaceMatrixBruteForce(Na,Ta,Nb,Tb); takes two triangular
%   meshes Ta and Tb with associated nodal coordinates in Na and Nb
%   and computes the interface projection matrix M using a brute
%   force approach

M=sparse(size(Nb,2),size(Na,2));
for i=1:size(Ta,1)
  for j=1:size(Tb,1)
    [P,nc,Mc]=Intersect(Nb(:,Tb(j,1:3)),Na(:,Ta(i,1:3)));
    if ~isempty(P)              % intersection found
      M(Tb(j,1:3),Ta(i,1:3))=M(Tb(j,1:3),Ta(i,1:3))+Mc;
    end
  end
end
```

APPENDIX C. REMAINING 3D ROUTINES

We show in this section a possible Matlab implementation of the remaining 3D routines from Section 3.

C.1. Precise Implementation of the 3D Advancing Front Algorithm

Here is the complete implementation of the algorithm in Figure 7 in 3D:

```
function InterfaceMatrix3d(Na,Ta,Nb,Tb);
% INTERFACEMATRIX3D projection matrix for nonmatching thetrahedra grids
%   M=InterfaceMatrix3d(Na,Ta,Nb,Tb); takes two tetrahedra meshes Ta
%   and Tb with associated nodal coordinates in Na and Nb and
%   computes the projection matrix M. As a precondition, the first
%   thetrahedra in Ta and Tb need to intersect.

bl=[1];                         % bl: list of tetrahedra of Tb to treat
bil=[1];                        % bil: list of tetrahedra Ta to start with
bd=zeros(size(Tb,1)+1,1);       % bd: flag for tetrahedra in Tb treated
bd(end)=1;                      % guard, to treat boundaries
bd(1)=1;                        % mark first tetrahedra in b list.
while length(bl)>0
  bc=bl(1); bl=bl(2:end);       % bc: current tetrahedra of Tb
  al=bil(1); bil=bil(2:end);    % tetrahedra of Ta to start with
  ad=zeros(size(Ta,1)+1,1);     % same as for bd
  ad(end)=1;
  ad(al)=1;
  n=[0 0 0 0];                  % tetrahedra intersecting with neighbors
  while length(al)>0
    ac=al(1); al=al(2:end);     % take next candidate
    [P,nc,H,v]=Intersection3d(Nb(:,Tb(bc,1:4)),Na(:,Ta(ac,1:4)));
    if ~isempty(P)              % intersection found
      t=Ta(ac,4+find(ad(Ta(ac,5:8))==0));
      al=[al t];                % add neighbors
      ad(t)=1;
      n(find(nc>0))=ac;         % ac is starting candidate for neighbor
    end
  end
```

```
  tmp=find(bd(Tb(bc,5:8))==0); % find non-treated neighbors
  idx=find(n(tmp)>0);          % only if intersecting with Ta
  t=Tb(bc,4+tmp(idx));
  bl=[bl t];                   % and add them
  bil=[bil n(tmp(idx))];       % with starting candidates Ta
  bd(t)=1;
end
```

C.2. Remaining Geometric Primitives

For the remaining geometric primitives `TriangleLineIntersection`, `InsertPoint`, `RemoveDuplicates`, `NewFace`, `OrderPoints`, and `PyramidVolume`, a possible implementation in Matlab is given in the following:

```
function p=TriangleLineIntersection(X,Y);
% TRIANGLELINEINTERSECTION intersection of a line and a triangle
%   p=TriangeLineIntersection(X,Y); computes for a given line X in 3d,
%   and a triangle Y in 3d, the point p of intersection in the
%   triangle, and otherwise returns p=[];

p=[];
b=Y(:,1)-X(:,1);
A=[X(:,2)-X(:,1) Y(:,1)-Y(:,2) Y(:,1)-Y(:,3)];
if rank(A)==3                        % edge and triangle not parallel
  r=A\b;
  if r(1)>=0 & r(1)<=1 & r(2)>=0 & r(3)>=0 & r(2)+r(3)<=1
    p=X(:,1)+r(1)*(X(:,2)-X(:,1));
  end;
end;
```

```
function p=PointInTetrahedron(X,Y);
% POINTINTETRAHEDRON check if a point is inside a tetrahedron
%   p=PointInTetrahedron(X,Y); checks if the point X is contained in
%   the tetrahedron Y.

D0=det([Y; ones(1,4)]);
D1=det([X Y(:,2:4); ones(1,4)]);
D2=det([Y(:,1) X Y(:,3:4); ones(1,4)]);
D3=det([Y(:,1:2) X Y(:,4); ones(1,4)]);
D4=det([Y(:,1:3) X; ones(1,4)]);
p=sign(D0)==sign(D1) & sign(D0)==sign(D2) & sign(D0)==sign(D3) & ...
  sign(D0)==sign(D4);
```

```
function [k,P]=InsertPoint(p,P);
% INSERTPOINT inserts a new intersection point
%   [k,P]=InsertPoint(p,P); inserts an intersection point p into
%   the list of intersection points P. If the point p is already
%   contained in P, the point is not inserted, and k is the index
%   of the point found in the list. Otherwise, the new point is
%   inserted, and k is its index in the list.

ep=10*eps;                           % tolerance for identical nodes
k=1;
while k<=size(P,2) & norm(p-P(:,k))>ep
  k=k+1;
end;
if k>size(P,2)                       % new node is not contained in P
  P(:,k)=p;
end;
```

```
function p=RemoveDuplicates(p)
% REMOVEDUPLICATES removes multiple entries
%   p=RemoveDuplicates(p) removes duplicate entries from the vector
%   p.

p=sort(p);
i=1;j=2;                          % remove duplicates
while j<=length(p)
  if p(i)<p(j)
    i=i+1;p(i)=p(j);j=j+1;
  else
    j=j+1;
  end;
end;
p=p(1:i);
```

```
function b=NewFace(H,id)
% NEWFACE checks if index set is contained already in H
%   b=NewFace(H,id) checks if a permutation of the vector id is
%   contained in a row of the matrix H.

n=size(H,1);
m=length(id);
A=zeros(n,m);
A(1:n,1:size(H,2))=H;
id=sort(id);
i=0;
b=1==1;                    % true
while b & i<n
  i=i+1;
  b=norm(sort(A(i,1:m))-id)>0;
end;
```

```
function [p,id]=OrderPoints(p,cm);
% ORDERPOINTS order points counterclockwise
%   [p,id]=OrderPoints(p,cm); orders the points in a plane in 3d
%   stored columnwise in the matrix p counterclockwise looking from
%   the side opposite to the point cm outside the plane. There must
%   be more than two points. p contains the reorderd points, and id
%   contains the index reordering.

d1=p(:,2)-p(:,1);               % two reference vectors
d2=p(:,3)-p(:,1);
if (cm-p(:,1))'*cross(d1,d2)<0  % good orientation ?
  dr=d1/norm(d1);               % 'x-axis' unit vector
  dn=d2-d2'*dr*dr;
  dn=dn/norm(dn);               % 'y-axis' unit vector
else
  dr=d2/norm(d2);               % 'x-axis' unit vector
  dn=d1-d1'*dr*dr;
  dn=dn/norm(dn);               % 'y-axis' unit vector
end;
for j=2:size(p,2)
  d=p(:,j)-p(:,1);
  ao(j-1)=angle(d'*dr+sqrt(-1)*d'*dn);
end;
[tmp,id]=sort(ao);
id=[1 id+1];
p=p(:,id);
```

```
function v=PyramidVolume(P,c)
% PYRAMIDVOLUME computes the volume of a pyramid
%   v=PyramidVolume(P,c) computes for the pyramid in 3d with base
%   polygone given by the nodes P stored columnwise and the top c
%   its volume v.

a=0;                            % area
for i=3:size(P,2)
  a=a+norm(cross(P(:,i)-P(:,1),P(:,i-1)-P(:,1)))/2;
end;
no=cross(P(:,3)-P(:,1),P(:,2)-P(:,1));
no=no/norm(no);
h=(c-P(:,1))'*no;
v=abs(a*h/3);
```

C.3. Running an Example in 3D

Here is how to run the intersection algorithm in 3D:

```
% 3D EXAMPLE test the advancing front program for three dimensional meshes

[N1,T1]=Mesh3d(1); [N2,T2]=Mesh3d(1); N2=(N2/1.5);

fig=figure(1); clf; set(fig,'DoubleBuffer','on');
PlotMesh3d(N1,T1,'b'); PlotMesh3d(N2,T2,'r');
xlabel('x'); ylabel('y'); zlabel('z'); title('nonconforming grids in 3d');

InterfaceMatrix3d(N1,T1,N2,T2);
```

The example uses basic routines to generate and plot 3D meshes, namely `Mesh3d` and `PlotMesh3d`, and one can also refine the mesh using `RefineMesh3d`. All these routines are given in the following, including the routine `FindFace` used internally by `RefineMesh3d`:

```
function [N,T]=Mesh3d(p)
% MESH3d generates simple tetrahedra meshes
%   [N,T]=Mesh3d(p); generates an initial coarse tetrahedra mesh for
%   each value of p. The result is a list of tetrahedra T which points
%   into the list of nodes N containing x and y coordinates.  The
%   tetrahedra contains in entries 5 to 8 the neighboring tetrahedra
%   indices, and as a guard # of tetrahedra + 1 if there is no
%   neighbor.

if p==1                     % a cube
  N=[0 1 0 0 1 1 0 1
     0 0 1 0 0 1 1 1
     0 0 0 1 1 0 1 1];
  T=[1 2 3 4 6 5 6 6
     2 6 3 8 6 6 5 6
     2 8 4 5 5 6 6 6
     3 4 8 7 5 6 6 6
     2 3 4 8 1 4 3 2];
elseif p==2                       % just one tetrahedra as default
  N=[0 1 0 0
     0 0 1 0
     0 0 0 1];
  T=[1 2 3 4 2 2 2 2];
end
```

```
function [Nr,Tr]=RefineMesh3d(N,T);
% REFINEMESH3D refines the mesh by a factor of 8
%   [Nr,Tr]=RefineMesh3d(N,T); refines the mesh given by the nodes N and
%   the tetrahedra T by cutting each tetrahedra into 8 smaller ones.
%   The neighboring information in the tetrahedra is also updated,
%   and the boundary guard is still the number of tetrahedra plus one,
%   as in NewMesh3d.

Nr=N;                              % new node list starts with old one
nn=size(N,2);
Tr=[];                             % tetrahedra start from scratch
nt=0;                              % number of new tetrahedra
ep=10*eps;                         % to check identical nodes
for j=1:size(T,1),                 % divide all old tetrahedra
  i=T(j,1:4); n=N(:,i);            % old nodes of current tetrahedra
  n(:,5)=(n(:,1)+n(:,2))/2;        % 6 new nodes to be used
  n(:,6)=(n(:,1)+n(:,3))/2;
  n(:,7)=(n(:,2)+n(:,3))/2;
  n(:,8)=(n(:,1)+n(:,4))/2;
  n(:,9)=(n(:,2)+n(:,4))/2;
  n(:,10)=(n(:,3)+n(:,4))/2;
  for k=5:10,                      % insert new nodes in Nr if necessary
    [i(k),Nr]=InsertPoint(n(:,k),Nr);
  end;                             % new tetrahedra with known neighborinfo
  Tr(nt+1,:)=[i(1) i(5) i(6)  i(8)  -1  nt+5  -1   -1 ]; % corner tetrahedra
  Tr(nt+2,:)=[i(5) i(2) i(7)  i(9)  -1   -1  nt+6  -1 ];
  Tr(nt+3,:)=[i(6) i(7) i(3)  i(10) -1   -1   -1  nt+7];
  Tr(nt+4,:)=[i(8) i(9) i(10) i(4) nt+8  -1   -1   -1 ];
  Tr(nt+5,:)=[i(5) i(7) i(6)  i(8)  -1  nt+7 nt+1 nt+6]; % interior tetrahedra
  Tr(nt+6,:)=[i(5) i(9) i(7)  i(8) nt+2 nt+8 nt+5  -1 ];
  Tr(nt+7,:)=[i(6) i(7) i(10) i(8) nt+3 nt+8  -1  nt+5];
  Tr(nt+8,:)=[i(8) i(9) i(7) i(10) nt+6  -1  nt+7 nt+4];
  nt=nt+8;
end;
nn=[1 2 3 5                        % new neighbor indices
    2 3 4 8
    3 4 1 7
    4 1 2 6];
for j=1:size(T,1),                 % updating remaining neighborinfo
  ne=T(j,5:8);
  if ne(1)==size(T,1)+1            % real boundary
    Tr(8*(j-1)+nn(1,:),5)=size(Tr,1)+1;
  else
    fn=FindFace(T(j,1:3),T(ne(1),1:4));
    Tr(8*(j-1)+nn(1,:),5)=8*(ne(1)-1)+fn;
  end;
  if ne(2)==size(T,1)+1            % real boundary
    Tr(8*(j-1)+nn(2,:),6)=size(Tr,1)+1;
  else
    fn=FindFace(T(j,2:4),T(ne(2),1:4));
    Tr(8*(j-1)+nn(2,:),6)=8*(ne(2)-1)+fn;
  end;
  if ne(3)==size(T,1)+1            % real boundary
    Tr(8*(j-1)+nn(3,:),7)=size(Tr,1)+1;
  else
    fn=FindFace(T(j,[3 4 1]),T(ne(3),1:4));
    Tr(8*(j-1)+nn(3,:),7)=8*(ne(3)-1)+fn;
```

```
  end;
  if ne(4)==size(T,1)+1          % real boundary
    Tr(8*(j-1)+nn(4,:),8)=size(Tr,1)+1;
  else
    fn=FindFace(T(j,[4 1 2]),T(ne(4),1:4));
    Tr(8*(j-1)+nn(4,:),8)=8*(ne(4)-1)+fn;
  end;
end;

function f=FindFace(X,Y)
% FINDFACE find face in tetrahedra
%   f=FindFace(X,Y) finds face X in thetrahedra Y, where the face and
%   tetrahedra are given by node indices.

f(1)=find(Y==X(1));
f(2)=find(Y==X(2));
f(3)=find(Y==X(3));
switch sum(f)
 case 6                 % face 1 2 3
  f(4)=5;
 case 9                 % face 2 3 4
  f(4)=8;
 case 8                 % face 3 4 1
  f(4)=7;
 case 7                % face 4 1 2
  f(4)=6;
end;
f=f';

function PlotMesh3d(N,T,col);
% PLOTMESH3D plots a tetrahedra mesh
%   PlotMesh3d(N,T,col); plots the mesh given by the nodes N and
%   tetrahedra T in color col. For small meshes the node numbers are
%   added as well.

axis('equal');
view(-37.5+180,30);
for i=1:size(T,1),
  PlotTetrahedron(N(:,T(i,1:4)),col);
end;
m=size(N,2);
de=0.05;
if m<10,
  for i=1:m,
    text(N(1,i)+de,N(2,i)+de,N(3,i)+de,num2str(i),'Color','b');
  end;
end;
```

C.4. Testing the Complexity

We give here an example code to compare the computing time of the new method with a brute force algorithm:

```
% 3D EXAMPLE3D compares in computing time for three dimensional projections

nr=3;                          % number of mesh refinements
np=5;                          % number of projection calculations
```

```
vtb=zeros(1,nr); vtn=zeros(1,nr); s=zeros(1,nr);
for k=1:nr
  [N1,T1]=Mesh3d(1); [N2,T2]=Mesh3d(1); N2=(N2/1.5);
  for l=1:k
    [N1,T1]=RefineMesh3d(N1,T1); [N2,T2]=RefineMesh3d(N2,T2);
  end
  tb=0;tn=0;
  for i=1:np
    N2(:,1:end)=N2(:,1:end)+(0.02*rand(size(N2(:,1:end))));
    t0=clock; InterfaceMatrix3d(N1,T1,N2,T2); tn=tn+etime(clock,t0);
    t0=clock; InterfaceMatrix3dBruteForce(N1,T1,N2,T2); tb=tb+etime(clock,t0);
  end;
  vtn(k)=tn; vtb(k)=tb; s(k)=size(T1,1);
end;
c1=vtb(end)/s(end)^2; c2=vtn(end)/s(end); % align theoretical and numerical curves
loglog(s,vtb,'--o',s,vtn,'--*',s,c1*s.^2,'-o',s,c2*s,'-*');
legend('Brute Force','New Method','O(n^2)','O(n)',2);
xlabel('number of triangles'); ylabel('time for 20 projections');
```

REFERENCES

BASTIAN, P., BUSE, G., AND SANDER, O. 2010. Infrastructure for the coupling of Dune grids. In *Proceedings of ENUMATH*, G. Kreiss, P. Lötstedt, A. Mølqvist, and M. Neytcheva, Eds., Springer, Berlin, 107–114.

BELGACEM, F. B. AND MADAY, Y. 1999. Coupling spectral and finite elements for second order elliptic three-dimensional equations. *SIAM J. Numer. Anal. 36*, 4, 1234–1263.

BELGACEM, F. B. 1999. The mortar finite element method with Lagrange multipliers. *Numer. Math. 84*, 2, 173–197.

BENNEQUIN, D., GANDER, M., AND HALPERN, L. 2009. A homographic best approximation problem with application to optimized Schwarz waveform relaxation. *Math. Comp. 78*, 185–223.

BERNARDI, C., MADAY, Y., AND PATERA, A. T. 1993. Domain decomposition by the mortar element method. In *Asymptotic and Numerical Methods for Partial Differential Equations with Critical Parameters*, H. K. and M. Garbey, Eds., Vol. 384, N.A.T.O. ASI, Kluwer Academic Publishers, Dordrecht, Netherlands, 269–286.

BERNARDI, C., MADAY, Y., AND PATERA, A. T. 1994. A new nonconforming approach to domain decomposition: The mortar element method. In Collège de France Seminar, H. Brezis and J.-L. Lions, Eds., Pitman, London, UK.

BRAESS, D. AND DAHMEN, W. 1998. Stability estimates of the mortar finite element method for 3-dimensional problems. *East-West J. Numer. Math. 6*, 4, 249–263.

BREZZI, F., LIONS, J.-L., AND PIRONNEAU, O. 2001. Analysis of a Chimera method. *C.R. Math. Acad. Sci. Paris 332,* 7, 655–660.

BRUNE, P., KNEPLEY, M., AND SCOT, L. 2013. Unstructured geometric multigrid in two and three dimensions on complex and graded meshes. *SIAM J. Sci. Comput.*

DUNE. 2010. Distributed and unified numerics environment. www.dune-project.org/modules/dune-grid-glue/.

FLEMISCH, B., KALTENBACHER, M., AND WOHLMUTH, B. I. 2006. Elasto-acoustic and acoustic-acoustic coupling on nonmatching grids. *Int. J. Num. Meth. Eng. 67*, 13, 1791–1810.

GANDER, M., HALPERN, L., AND KERN, M. 2007. Schwarz waveform relaxation method for advection–diffusion–reaction problems with discontinuous coefficients and non-matching grids. In *Decomposition Methods in Science and Engineering XVI*, O. Widlund and D. Keyes, Eds., Lecture Notes in Computational Science and Engineering, vol. 55, Springer, Berlin, Germany, 916–920.

GANDER, M. AND JAPHET, C. 2008. An algorithm for non-matching grid projections with linear complexity. In *Proceedings of the 18th International Domain Decomposition Conference,* M. Bercovier, M. J. Gander, D. Keyes, and O. Widlund, Eds., Lecture Notes in Computational Science and Engineering, Springer, Berlin, Germany.

GANDER, M., JAPHET, C., MADAY, Y., AND NATAF, F. 2005. A new cement to glue nonconforming grids with robin interface conditions: The finite element case. In *Proceedings of the 15th International Domain Decomposition Conference*. R. Kornhuber, R. H. W. Hoppe, J. Périaux, O. Pironneau, O. B. Widlund, and J. Xu, Eds., Lecture Notes in Computational Science and Engineering, vol. 40, Springer, Berlin, Germany, 259–266.

GANDER, M. J. 1997. Overlapping Schwarz for parabolic problems. In *Proceedings of Ninth International Conference on Domain Decomposition Methods*. P. E. Bjørstad, M. Espedal, and D. Keyes, Eds., 97–104.

GANDER, M. J., HALPERN, L., AND NATAF, F. 1999. Optimal convergence for overlapping and nonoverlapping Schwarz waveform relaxation. In *Proceedings of 11th International Conference of Domain Decomposition Methods*. C.-H. Lai, P. Bjørstad, M. Cross, and O. Widlund, Eds.

GLOWINSKI, R., HE, J., LOZINSKI, A., RAPPAZ, J., AND WAGNER, J. 2005. Finite element approximation of multi-scale elliptic problems using patches of elements. *Numer. Math. 101*, 4, 663–687.

GREINER, G. AND HORMANN, K. 1998. Efficient clipping of arbitrary polygons. *ACM Trans. Graph. 17*, 2, 71–83.

HALPERN, L., JAPHET, C., AND SZEFTEL, J. 2010. Discontinuous Galerkin and nonconforming in time optimized Schwarz waveform relaxation. In *Domain Decomposition Methods in Science and Engineering XIX*, Y. Huang, R. Kornhuber, O. Widlund, and J. Xu, Eds., Lecture Notes in Computational Science and Engineering, vol. 78, Springer, Berlin, Germany, 133–140.

HECHT, F. 2012. Freefem++. www.freefem.org/ff++/index.htm.

JAIN, A. 2007. Efficient parallel algorithm for overlaying surface meshes. Ph.D. thesis, College of Computing, Georgia Institute of Technology.

JIAO, X. AND HEATH, M. T. 2004. Common-refinement-based data transfer between non-matching meshes in multiphysics simulations. *Int. J. Num. Meths. Eng. 61*, 2402–2427.

JIAO, X. M. 2001. Data transfer and interface propagation in multicomponent simulations. Ph.D. thesis, University of Illinois at Urbana-Champaign.

KAMGA, J.-B. A. AND PIRONNEAU, O. 2007. Numerical zoom for multiscale problems with an application to nuclear waste disposal. *J. Comput. Phys. 224*, 403–413.

LEE, P., YANG, C.-H., AND YANG, J.-R. 2004. Fast algorithms for computing self-avoiding walks and mesh intersections over unstructured meshes. *Adv. Eng. Soft. 35*, 61–73.

LÖHNER, R. 2001. *Applied CFD Techniques: An Introduction Based on Finite Element Methods*. Wiley, Chichester, UK.

MARTIN, V. 2005. An optimized Schwarz waveform relaxation method for unsteady convection diffusion equation. *Appl. Numer. Math. 52*, 4, 401–428.

MEAKIN, R. L. 1991. A new method for establishing intergrid communication among systems of overset grids. In *Proceedings of the 10th AIAA Computational Fluid Dynamics Conference*. 662–671.

MPCCI. 2013. Metalmapper for integrated Design Workflows - Mapping of Manufacturing and CFD Simulations Results to Crash and NVH Calculations. www.mpcci.de/.

PICASSO, M., RAPPAZ, J., AND REZZONICO, V. 2008. Multiscale algorithm with patches of finite elements. *Comm. Num. Meth. Eng. 24*, 6, 477–491.

PLIMPTON, S., HENDRICKSON, B., AND STEWART, J. 1998. A parallel rendezvous algorithm for interpolation between multiple grids. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, Los Alamitos, CA, 8.

SHEWCHUK, J. R. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Comput. Geom. 18*, 3, 305–363.

STEGER, J. L., DOUGHERTY, F. C., AND BENEK, J. A. 1983. A chimera grid scheme, advances in grid generation. In *Advances in Grid Generation*, K. Ghia and U. Ghia, Eds., FED, vol. 5, ASME, New York, NY, 59–69.

WEILER, K. AND ATHERTON, P. 1977. Hidden surface removal using polygon area sorting. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, New York, NY, 214–222.

Received April 2012; revised January 2013; accepted March 2013