Fast Algorithms for Intersection of Nonmatching Grids Using Plücker coordinates.

Jan Březina[a,*], Pavel Exner[a]

[a] *Technical University of Liberec, Studentská 1402/2, 461 17 Liberec 1, Czech Republic*

Abstract

Keywords: non-matching grid, intersections, mixed-dimensional mesh, Plücker coordinates

1. Introduction

PE: *How did you decided for 'non-matching' term? Instead of e.g. incompatible or non-conforming?*

The grid intersection algorithms are crucial for several techniques that try to overcome some limitations of the classical finite element method. The Chimera method [1], also called overset grid, and similar Nitche method [2] allow solution of the problems with changing geometry as in the fluid-structure problems. The Mortar method [3] allows domain decomposition, independent meshing of domains, and supports sliding boundaries. However our primal motivation is usage of XFEM methods and non-matching meshes of mixed dimension in groundwater models.

The realistic models of groundwater processes including the transport processes and geomechanics have to deal with a complex nature of geological formations including the fractures and wells. Although of small scale, these features may have significant impact on the global behavior of the system and their representation in the numerical model is imperative. One possible approach is to model fractures and wells as lower dimensional objects and introduce their coupling with the surrounding continuum. The discretization then leads to the meshes of mixed dimensions, i.e. composed of elements of different dimension. This approach called mixed-dimensional analysis in the mechanics [4] is also studied in the groundwater context, see e.g. [5], [6], [7] and already adopted by some groundwater simulation software, e.g FeFlow [8] and Flow123d [9]. Nevertheless as the complexity of the geometry increase (e.g. when lot of fractures are randomly generated) the compatible meshing becomes painful or even impossible. In order to avoid these difficulties we may discretize the continuum

*Corresponding author.

Email addresses: jan.brezina@tul.cz (Jan Březina), pavel.exner@tul.cz (Pavel Exner)

and every fracture and well independently getting a non-matching (or incompatible) mesh of mixed dimensions and then apply XFEM to represent jumps of the solution on the fractures or singularities around the wells. The prerequisity for such approach is a fast and robust algorithm for calculating intersections of individual meshes.

We consider a composed mesh $\mathcal{T}$ consisting of simplicial meshes $\mathcal{T}_i$ of dimensions $d_i \in \{1, 2, 3\}$, $i = 1, \dots, N_{\mathcal{T}}$ in the 3d ambient space. We assume that every mesh $\mathcal{T}_i$ is a connected set with no self intersection. Further we assume only single 3d mesh $\mathcal{T}_1$. The mesh intersection problem is to find all pairs of elements $L \in \mathcal{T}_i$, $K \in \mathcal{T}_j$, $i \neq j$ that have non-empty intersection and to compute that intersection.

The mesh intersection problem consists of the two parts: First, generate a set of candidate pairs (K, L). Second, compute the intersection for particular pair. In order to get candidate pairs efficiently, the existing algorithms use either various space trees [2] or front tracing [10] or both [11] as in our approach. To compute the actual intersections, we use the Plücker coordinates for the line-triangle intersections and a modification of the Platis and Theoharis algorithm [12] for the line-tetrahedron intersections. These are used as the building blocks for the triangle-triangle and the triangle-tetrahedron cases.

JB: *Better overview of works, seems that most of works deals with simple intersections, can not find any example of 2d-3d.*

PE: *Unify the term 'front tracing' 'front tracking' 'advancing front'.* Our contribution is twofold: First, we use the front tracking algorithm both to minimize set of intersection candidates as well to reuse part of calculations made on neighbouring intersections. Second, we present family of efficient algorithms based on Plücker coordinates for computing 1d-2d, 1d-3d, 2d-2d, and 2d-3d intersections of simplicies. The paper is organized as follows ...

2. Element Intersections

In this section, we present algorithms for computing intersection of a pair of simplicial elements of a different dimension in the 3D ambient space. In particular we are interested in the intersections of 1D-2D, 1D-3D, 2D-2D, 2D-3D pairs of elements. The fundamental idea is to compute intersection of 1D-2D simplices using the Plücker coordinates and reduce all other cases to this one.

We denote S_i a simplicial element with $i+1$ vertices (of dimension i). We call vertices, edges, faces and simlices itself the n-faces and we denote M_i the set of all n-faces of the simplex S_i. In general, an intersection can be a point, a line segment or a polygon called *intersection polygon* (IP) in common. The intersection polygon is represented as a list of its corners called *intersection corners* (IC). The IP data structure keeps also reference to the intersecting simplices. A data structure of a single IC consists of:

- the barycentric coordinate $\boldsymbol{w}_K$ of IC on K,
- the dimension d_K of the most specific n-face the IC lies on,

- the local index i_K of that n-face on K,

for each intersecting element K of the pair. The pair $\tau_K = (d_K, i_K)$ we call the topological position of the IC on K.

2.1. Plücker Coordinates

Plücker coordinates represent a line in 3D space. Considering a line p, given by a point $\boldsymbol{A}$ and its directional vector $\boldsymbol{u}$, the Plücker coordinates of p are defined as

$$\pi_p = (\boldsymbol{u}_p, \boldsymbol{v}_p) = (\boldsymbol{u}_p, \boldsymbol{u}_p \times A).$$

Further we use a permuted inner product

$$\pi_p \odot \pi_q = \boldsymbol{u}_p \cdot \boldsymbol{v}_q + \boldsymbol{u}_q \cdot \boldsymbol{v}_p.$$

The sign of the permuted inner product gives us the relative position of the two lines, see Figure 1.

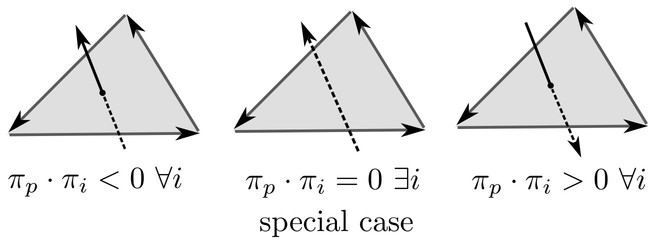
$\pi_p \cdot \pi_i < 0\ \forall i$
$\pi_p \cdot \pi_i = 0\ \exists i$
special case
$\pi_p \cdot \pi_i > 0\ \forall i$

Figure 1: Sign of the permuted inner product is related to the relative position of the two oriented lines. Dashed line symbolizes that the line is in the back, the lines intersect in the middle case. **PE:** *circle dot permuted inner product*

2.2. Intersection Line-Triangle (1D-2D)

Let us consider a line segment p with parametric equation

$$\boldsymbol{X} = \boldsymbol{A} + t\boldsymbol{u},\ \ t \in (0,1) \tag{1}$$

and a triangle T given by vertices $(\boldsymbol{V}_0, \boldsymbol{V}_1, \boldsymbol{V}_2)$ with oriented sides $s_i = (\boldsymbol{V}_j, \boldsymbol{V}_k)$, $j = (i+1) \bmod 3$, $k = (i+2) \bmod 3$.

Lemma 2.1. *The permuted inner products $\pi_p \odot \pi_{s_i}$, $i = 0, 1, 2$ have the same non-zero sign if and only if there is an intersection point X on the p and inside the triangle T. The barycentric coordinates of X on T are*

$$w_i = \frac{\pi_p \odot \pi_{s_i}}{\sum_{j=0}^{2} \pi_p \odot \pi_{s_i}}. \tag{2}$$

Proof. Using the barycentric coordinates the intersection point can be expressed as $\boldsymbol{X} = \boldsymbol{V}_0 + w_1\boldsymbol{s}_2 - w_2\boldsymbol{s}_1$. The line p have Plücker coordinates $(\boldsymbol{u}, \boldsymbol{u} \times \boldsymbol{X})$ since these are invariant to change of the initial point. Combining these two expressions we get

$$\pi_p \odot \pi_{s_1} = \boldsymbol{u} \cdot (\boldsymbol{s}_1 \times \boldsymbol{V}_2) + \boldsymbol{s}_1 \cdot (\boldsymbol{u} \times [\boldsymbol{V}_0 + w_1\boldsymbol{s}_2 - w_2\boldsymbol{s}_1]) = -w_1\boldsymbol{u} \cdot (\boldsymbol{s}_1 \times \boldsymbol{s}_2).$$

Since $\boldsymbol{s}_0 + \boldsymbol{s}_1 + \boldsymbol{s}_2 = 0$ we have $\boldsymbol{s}_1 \times \boldsymbol{s}_2 = \boldsymbol{s}_2 \times \boldsymbol{s}_0 = \boldsymbol{s}_0 \times \boldsymbol{s}_1$ and thus

$$\pi_p \odot \pi_{s_i} = -w_i\boldsymbol{u} \cdot (\boldsymbol{v}_1 \times \boldsymbol{v}_2).$$

The point X is inside of T if and only if $w_i > 0$ for all $i = 0, 1, 2$. □

Having the barycentric coordinates of X on T, we can compute also its local coordinate on p from its parametric form:

$$X_i = A_i + tu_i, \text{ for } i = 1, 2, 3 \tag{3}$$

We use i with maximal $|u_i|$ for practical computation.

The calculation of the intersection proceeds as follows:

1. Compute or reuse Plücker coordinates and permuted inner products: π_p, π_i, $\pi_s \odot \pi_i$, for $i = 1, 2, 3$.
2. Compute barycentric coordinates $w_i,\ i = 1, 2, 3$ using (2).
3. If any w_i is less then ϵ there is no intersection, return empty IP.
4. If all w_i are greater then ϵ, we set $\tau_T = (2, 0)$ for the IC.
5. If one w_i is less then ϵ, intersection on edge s_i, we set $\tau_T = (1, i)$.
6. If two w_i are less then ϵ, intersection in the vertex V_i, we set $\tau_T = (0, , i)$.
7. If all w_i are less then ϵ, the line is coplanar with the triangle, both objects are projected to the plane $x_i = 0$ where i is the index of maximal component of the triangle's normal vector. Every pair p, s_i is checked for an intersection on T boundary either inside s_i or in a vertex V_i setting the topological info τ_T to $(1, i)$ or $(0, i)$ respectively. At most two ICs are obtained.
8. For each IC the barycentric coordinates $(1-t, t)$ on the line p are computed according to (3).
9. If $t \in (-\epsilon, \epsilon)$ or $t \in (1 - \epsilon, 1 + \epsilon)$, we set $\tau_p = (0, 0)$ or $\tau_p = (0, 1)$, respectively.
10. If $t \notin (-\epsilon, 1 + \epsilon)$, the IC is eliminated.

The check for the same sign of the inner products can be viewed as a geometric predicate for the presence of the intersection and orientation of the line with respect to the triangle. Adaptive-precision evaluation of the geometric predicates was designed by Schewchuk [13] and used for 2d-2d mesh intersections in [11]. However, we rather apply a fixed tolerance check for the zero barycentric coordinates and consistently keep the topological positions in this and related algorithms. **JB:** *Can we make the algorithm* parsimonious *in the spirit of the*

Fortune [?] quoted by Schewchuk? Seems that our problem is more local than the line example that was proven to be NP-hard.

The algorithms for 1d-3d and 2d-3d intersections use simpler version of the 1d-2d intersection algorithm, in particular the search for ICs in the coplanar case (item 7) is not necessary and the test in the last point is not performed. and degenerate cases higher dimensional cases.

2.3. Intersection Line-Teraherdon (1D-3D)

In this section we consider intersection of a line segment p given by the parametric equation (1) with a tetrahedron S_3. The used algorithm is based on 1d-2d algorithm and closely follows [12]. Our modification takes into account intersection with a line segment and consistently propagates topological position of ICs.

Algorithm 1: 1d-3d intersection

```
   Input: Tetrahedron $S_3$, line segment $p$.
   Output: List of ICs on sorted along $p$.
1  $I = \{\}$ for unmarked face $f$ of $S_3$ do
2      $L$ = intersection($p$, $f$)
3      if $L$ is none or degenerate then continue
4      if $L$ is inside the edge $e$ then set $\tau_{S_4} = (1, e)$
5      mark faces coincident with $e$
6      else if $L$ is at the vertex $v$ then
7          set $\tau_{S_4} = (0, v)$
8          mark faces coincident with $v$
9      append $L$ to $I$ if $|I| = 2$ then break
10 if $|I| = 1$ and $I$ is outside of $p$ then erase $I$
11 else if $|I| = 2$ then
12     trim intersection with respect to the line segment $p$
```

Algorithm 1 first compute line-face intersections for every face of S_4 avoiding duplicate computation of ICs on edges and at vertices and skipping remaining faces once two ICs are found. Tetrahedron has six edges, so 7 Plücker coordinates and 6 inner products are computed at most. Precomputed coordinates and products are passed the 1D-2D algorithm which is performed fro the whole line p. After collecting line-tetrahedron ICs, we do the line segment trimming 12. If both ICs are out of the line segment p we eliminate both of them. If one of the ICs if out of p we use the closest end point of the line segment instead and interpolate barycentric coordinates of the IC on S_4. The topological positions are updated as well. The result of the algorithm are zero up to two ICs sorted by the parameter t of the line p.

2.4. Intersection Triangle-Tetrahedron (2D-3D)

The intersection of a triangle S_2 and a tetrahedron S_3 is an n-side polygon, $n \leq 7$. The sides of the polygon lie either on sides of S_2 or on faces of S_3. Thus each vertex (IC) of the polygon can arise either from side-face intersection, or from edge-triangle intersection, or be a vertex of S_2. So we have to compute at most 12 side-face intersections and at most 6 edge-triangle intersections. However, to this end we only need to compute 9 Plücker coordinates (3 sides, 6 edges) and 18 permuted inner produts, one for every side-edge pair. Computation of the intersection polygon consists of two parts: calculation of side-tetrahedron ICs (Algorithm 2) provides all ICs on the boundary of S_2, calculation of edge-triangle ICs (Algorithm 3). **PE:** *The following sentence is somehow weird.. I would say that we store ICs I; to sort I during the computation, we use the two tables; at the end, I is ordered correctly.* Correct order of the ICs, stored in the list I, is defined by the *connection tables* $F_g(:)$ and $F_p(:)$. **PE:** *The following sentece is a copy of the second sentence is this subsection.* Every side of the intersection polygon lies either on a side of S_2, or on a face, or on an edge of S_3. Let us denote M_2 the set of sides of S_2 and M_3 the set of volume, faces, edges, and vertices of S_3. Every side of polygon that lies on $x \in M_2 \cup M_3$ is followed by an IC given by $F_g[x]$ and every IC p is followed by the side that lies on $F_p[p] \in M_2 \cup M_3$.

Algorithm 2 passes through every side s of the triangle S_2 and computes the line-tetrahedron intersection L. In the regular case we process each of the two ICs in L. The IC p is added to the list I unless the last point is the same. This condition is effective just for the first IC in L and merges ICs at the same vertex of S_2. Then we identify an object $x \in M_3$ the point p lies on (line 7). **PE:** *I am not sure, if the term 'object' is clear..*

PE: *Why calling it vertex v instead of IC p here? What is y?* If the vertex v is between sides s_1, s_2, we effectively set connections $F_g[s_1] = v$, $F_g[y] = v$, $F_p[v] = s_2$. The backward temporary link on the line 13 is used in Algorithm 3 to fix $F_p[p]$ for the second p lying on an edge or a vertex of S_3. The condition at the end deals with the case in which two sides intersect the same object $x \in M_3$.

For the regular intersection L, we pass through its ICs, as described above. **PE:** *For special cases*, we firstly compute ICs on the boundary of S_2 using the line-tetrahedron intersection algorithm for every side. We store single point intersections into separate list J and skip filling the connection tables. This happens when the side touches S_3 at its edge or vertex. These ICs will be rediscovered again in Algorithm 3 with better topological information, however this is not the case if the touched edge e of S_3 is coplanar with S_2 and the IC is inside of e. Therefore we keep a separate list J of ICs to deal with this case.

Algorithm 3 uses the line-triangle intersection algorithm for the edges of S_3 (line 1). The loop produces ICs in the interior of S_2 and possibly those ICs with special position on vertex or edge of S_3 already computed in Algorithm 2. Every edge e of S_3 is oriented so that the pair of adjacent faces f_0, f_1 appears in the same order on the intersection polygon when the IC on e has a negative sign (see Figure 1). The function *edge faces*, used on line 4 and later on, uses the sign

Algorithm 2: 2d-3d intersection, points on triangle boundary

```
Input: input data
Output: List of ICs on sorted output data
$F_g(:) = -1,\ F_p(:) = -1$                                    // Unset links.
for side $s$ of $S_2$ do
    $L$ = intersection$(s, S_3)$
    if $|L| = 0$ then continue
    if $|L| = 1$ then append $p$ to $J$ continue
    for $p$ in $L$ do
        if $p \neq I[-1]$ then append $p$ to $I$
        $p$ lies on $x \in M_3$
        if $p$ is first in $L$ then
            $F_g[x] = p,\ F_p[p] = s$
        else $p$ is the last in $L$
            $F_g[s] = p,\ F_p[p] = x$
            if $F_g[x] = -1$ then $F_g[x] = p$
if $I[-1] = I[0]$ then
    $x = F_p[I[-1]]$
    if $F_g[x]] = I[-1]$ then
        $F_g[x] = I[0]$
    $F_g[s_2] = I[0]$
    remove $I[-1]$
```

Algorithm 3: 2d-3d intersection, points in triangle interior

```
Input: $I$ with ICs on $S_2$ boundary, partially filled $F$
Output: all ICs in $I$, complete $F$
1  for edge $e$ of $S_3$ do $L[e]$ = intersection($e, S_2$)
2  for unmarked edge $e$ of $S_3$ do
3      $p = L[e]$
4      if $p$ is inside $e$ then
5          $(f_0,\ f_1)$ = edge faces $(e)$
6      else $p$ at the vertex $v$ of $S_3$
7          $(f_0,\ f_1)$ = vertex faces $(v,L)$                  // Algorithm 4
8          mark all edges coincident with $p$
9      if $p$ is on boundary of $S_2$ then
10         $p$ lies on edge or at vertex $x \in M_3$
11         $q = F_g[x]$                          // $q$ is already computed $p$
12         if $F_p[q] = x$ then $F_p[q] = f_1$
13         else $F_g[f_0] = q$
14         $F_g[x] = -1$                                 // remove the backlink
15     else
16         append $p$ to $I$
17         $F_g[f_0] = p$                             // overwrite the backlink
18         $F_p[p] = f_1$
19 if $|I| < 3$ then return $J$
20 else return $I$ sorted according to connectivity in $F_g$ and $F_p$
```

Algorithm 4: 2d-3d intersection, vertex faces

```
Input: vertex $v$ of $S_3$, $L[:]$ intersection results for edges of $S_3$
Output: $(x_1, x_2)$, $x_1, x_2 \in M_3$, coincident with $v$ and intersected by the
        plane of $S_2$
1  $e_0,\ e_1,\ e_2$ edges coincident with $v$ oriented out of $v$ $s[i] = L[e_i]$, for
     $i = 0, 1, 2,$
2  if $s[:]$ have 1 non-degenerate edge $e$ then
3      return pair of degenerate edges sorted according to edge faces $(e)$
4  else if  $s$ have 1 degenerate edge $e$ then
5      $f$ is face opposite to $e$ if other two edges $e_a$, $e_b$ have different sign
         then
6          $z =$ edge faces$(e_a)$
7          replace $g \in z,\ g \neq f$ with $e$ return $z$
8      else  append IC of $v$ to $J$ return anything
9
10 else if  $s$ have edge $e$ with sign oposite to other two then
11     return edge faces$(e)$
12 else $s$ have all signs same
13     append IC of $v$ to $J$ return anything
```

of the intersection to return the pair of faces ordered correctly. Similarly the function *vertex faces* (Algorithm 4, described later) returns a pair of generalized faces (face or edge) possibly adjacent to the IC $L[e]$ at the vertex v of S_3. On line 7 we mark all edges coincident with the vertex, since if they have any other intersection corner they are degenerate and not processed anyway.

2.4.1. Vertex Faces Algorithm

This function gets an IC p at vertex v of S_3 as a parameter. The IC is special vertex case of non-degenerate edge-triangle intersection. The function returns a pair of generalized faces of S_3 preceeding and succeeding p on the polygons boundary in the case that p is at interior of S_2. The basic idea is to use the signs of ICs of the three edges coincident with v. Possible cases are:

- **All ICs have the same sign.** (line 12) We return any pair of faces. S_2 is touching S_3 at the vertex v, the polygon degenerates into the single IC p, no connection information from table F is necessary.
- **Single IC has the opposite sign to the other two.** (line 10) Let e be the edge of the single IC with the different sign. The plane of S_2 separates e from the other two edges so it goes through the faces adjacent to e. The order is detemined by the function *edge faces.*
- **Single degenerated IC.** (line 4) Let us denote e the edge with degenerated IC and f the face between the other two edges. The other two

(non-degenerates) edges may have either the opposite sign (the plane is cutting S_3) or the same sign (the plane is touching S_3 at the edge e). In the first case, the call of edge faces for e returns (f_x, f) or (f, f_x), then the vertex faces function returns (e, f) or (f, e), respectively.

  The edge e lies in the the singel edge

- **Two degenerated ICs.** (line 2) A face of S_3 lies in the plane of S_2, single edge e have non-degenerate IC. We treat the two degenerate edges as special case of faces adjacent to e and return them sorted like the faces given by edge faces of edge e.

Finally in the third part (Algorithm 4), the table F allows us to modify array of successors P and get I in the correct order as the list K.

Algorithm 5: 2d-3d intersection, finish sort of points

```
Input: all points in $I$
Output: polygon in $K$ in correct order
1 $P = (1, \dots, n-1, 0)$// array of sucessors
2 for $f$ is face of $S_3$ do $P[F[1, f]] = F[0, f]$
3 $i = 0$
4 for $n = 0$ to $|I| - 1$ do $K[n] = I[i]$ $i = P[i]$
```

List $in[f]$ contains index of the intersection corner that follows after f on the boundary of traced polygon, similarly $out[f]$ stores index of the intersection corner that preeceeds the face f.

Possible cases for processing L:

1. Regular case, L consists of two intersections p, q sorted by orientation of s, laying inside of s.
   If p is on the edge e of S_3 compute sign of intersection(e, S_2), sort the faces f_0, f_1 coincident with e and set $in[f_0]$ to index of p in L. Similarly if q is on the edge, set $out[f_1]$ to index of q in L.
   If p is in vertex v of S_3, for every face f coincident with v set in[f] to index of p unless there is some index already set. So, we do not over ride entries comming from the edge intersections. Similarly set out[f] if q is in vertex of S_3.
   If p is on face f_0 of S_3, set $in[f_0]$ to index of p. Similarly, if q is on face f_1 of S_3, set $out[f_1]$ to index of q. laying on faces f_p, f_q of S_3.
2. L consists of a single intersection corner p (touching S_3)
   If p is on edge, compute sign of intersection(e, S_2), sort the faces f_0, f_1, set $in[f_0]$ and $out[f_1]$ to index of p.
   If p

How tracing works.

- If there are no intersections in vertex of S_3.

intersection polygon are found as intersection corners of either triangle side and tetrahedron or tetrahedron edge and triangle. Therefore we use both algorithms above for 1D-3D and 1D-2D, respectively. Data are again efficiently passed to lower dimensional problems, so

The array of intersection corners is generally not sorted. We use two so called *tracing* algorithms and we intend to orient the edges of the polygon in the same direction as the triangle is oriented. If one of the intersection corner is pathologic, a general convex hull method is applied using the Monotone chain[1] algorithm. The points are sorted using only their barycentric coordinates.

An optimized algorithm has been suggested for non-pathologic cases. At this moment all the collected topology data come into play. The algorithm takes advantage by using only the data already computed and also lowers the complexity to $O(N)$, compared with the Monotone chain complexity $O(N \log N)$ (N being number of intersection corners).

2.5. Tracking boundary of the intersection polygon

PE: *It seems to me, that we can really describe the prolongation in general, for 1D and 2D, refering to them as components. I tried to do so..*

PE: *We need proper definitions of terms we use:*
candidates pair *is a pair of a component element and a bulk element, that might intersect each other (due to intersection of their bounding boxes or prolongation result)* pathologic, special, degenerate case ??

PE: *Do you want to use American 'neighbor' or all other English 'neighbour'? (I prefer non-american.. to be unified at the end..)*

3. Advancing Front Method

add references...

Consider now a complex mesh of combined dimensions consisting of *components*, which are sets of connected elements of the same lower dimension (1D, 2D), in the space of connected 3D elements, which we shall call a *bulk*. Obtaining all of component-bulk intersections is done in two phases: firstly, we look for the first two elements intersecting each other (initialization); secondly, we prolong the intersection by investigating neighbouring elements (intersection tracking).

To construct the Advancing front algorithm, we shall need:

- **element connectivity** – we assume this data is available from mesh preprocessing,
- **Axes Aligned Bounding Boxes (AABB)** – we construct these in order to decide fastly whether to compute the actual intersection of two elements,

[1] Wikibooks, [online 2016-03-01], http://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain

- **Bounding Interval Hierarchy (BIH)** – we alternatively create BIH above AABB to fastly search created bounding boxes for two coinciding with each other and thus obtaining a candidate pair.

The intersection tracking itself can be also seen as a *breadth-first search* [2] algorithm over the BIH, following the component elements.

Initialization. We start with selecting an arbitrary 1D or 2D element. Then we search the bulk elements, checking for a collision of bounding boxes, to create a candidate pair. Using only AABB, we need to iterate over bulk elements in $O(n)$, n being the number of all elements in our case. Using BIH, we can speed up the search to $O(\log n)$ on average. In later case, we are paying the costs in the construction of BIH, which is a quicksort like algorithm running at $O(n \log n)$ on average.

Now that we have provided the first candidate pair, we can look at the scheme in Figure 2 and see us moving from the green box in the left upper corner. If there exists an intersection, we start a new component and we can procced to tracking the intersection. Otherwise, we select another 1D or 2D element and start over.

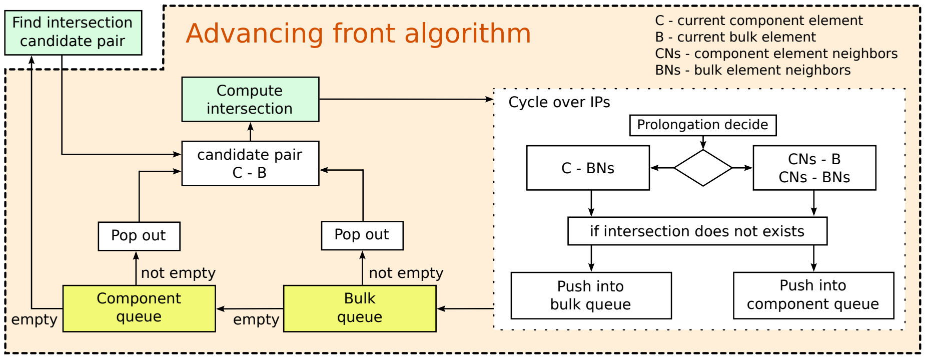

Figure 2: Advancing front algorithm for 1D-2D and 2D-3D intersections.

The advancing front algorithm works the same way for both 1D and 2D components and is displayed by the scheme in Figure 2. The main idea is to compute intersections for a component element with all possible bulk elements, and then move to a next neighboring component element. For this reason, we define two queues of candidate pairs: a *bulk queue* and a *component queue* (yellow boxes in Figure 2). We shall now describe the algorithm in detail.

On input we consider a candidate pair, for which a non-empty intersection is computed. Now we look for new candidate pairs among the neighboring elements (the block Prolongation decide). Therefore, we iterate over the intersection points and further exploit the topological information. There are 3

[2]Wiki, [online 2016-03-01], https://en.wikipedia.org/wiki/Breadth-first_search

possible cases (applies both for 1D-3D and 2D-3D), how the intersection might be prolonged:

- **IP lies on the component element side and inside the bulk element**
  We find all the sides of component element in which the IP lies (IP can be at node and connect more sides). Next, we find the component neighboring elements over the sides and push all new candidate pairs [component neighbor – current bulk element] into the component queue. Note, that there can be more than one neighbor on a side, if the component has branches.

- **IP lies on the component element side and on the surface of the bulk element**
  We find all the faces of bulk element in which the IP lies (1 face, or 2 faces (IP on an edge), or 3 faces (IP at a node)). We find the corresponding neighboring bulk elements over the faces and push the new candidate pairs [current component element – bulk neighbor] into the bulk queue and [component neighbor – bulk neighbor] into the component queue.

- **IP lies inside component element (therefore must be on the surface of bulk element)**
  We proceed as in previous case, but we push only [current component element – bulk neighbor] candidate pairs, since there is no component neighbor.

If the candidate pair has been found already, we skip it. We also see that the candidate pairs are of three types: [current component element – bulk neighbor], [component neighbor – current bulk element], [component neighbor – bulk neighbor], from which only the first one goes into the bulk queue, trying to cover the whole component element.

Then we empty the two queues. We pop out new candidate pairs from the *bulk queue* as long as it is not empty and for every new intersection computed, we repeat the previous part (means that we can further fill both queues). The *bulk queue* is empty when the component element is fully covered by bulk elements, or when there is no bulk neighbor to which we can advance. Then we can pop a new candidate pair from *component prolongation queue* and process it. When both queues are empty, all intersections of a component have been found and we start over by looking for the first intersection of another component.

4. Benchmarks

5. Conclusions

TODO: - line intersection tracking for accelerate 2D-2D intersections - better handling of special cases in paarticular in relation to prolongations - better calculation reuse (pass with prolongations) - optimisation of element intersection - skip unnecessary calculations

6. Acknowledgement

The paper was supported in part by the Project OP VaVpI Centre for Nanomaterials, Advanced Technologies and Innovations CZ.1.05/2.1.00/01.0005.

[1] F. Brezzi, J.-L. Lions, O. Pironneau, Analysis of a chimera method 332 (7) 655–660. doi:10.1016/S0764-4442(01)01904-8.
URL http://www.sciencedirect.com/science/article/pii/S0764444201019048

[2] A. Massing, M. G. Larson, A. Logg, Efficient implementation of finite element methods on nonmatching and overlapping meshes in three dimensions 35 (1) C23–C47. doi:10.1137/11085949X.
URL http://epubs.siam.org/doi/abs/10.1137/11085949X

[3] F. B. Belgacem, The mortar finite element method with lagrange multipliers 84 (2) 173–197. doi:10.1007/s002110050468.
URL http://link.springer.com/article/10.1007/s002110050468

[4] S. Bournival, J.-C. Cuillire, V. Franois, A mesh-geometry based approach for mixed-dimensional analysis, in: Proceedings of the 17th International Meshing Roundtable, Springer, pp. 299–313.
URL http://link.springer.com/chapter/10.1007/978-3-540-87921-3_18

[5] V. Martin, J. Jaffr, J. E. Roberts, Modeling fractures and barriers as interfaces for flow in porous media 26 (5) 1667. doi:10.1137/S1064827503429363.
URL http://link.aip.org/link/SJOCE3/v26/i5/p1667/s1&Agg=doi

[6] A. Fumagalli, A. Scotti, Numerical modelling of multiphase subsurface ow in the presence of fractures 3 (1). doi:10.1685/journal.caim.380.
URL http://openjournal.simai.eu/index.php/caim/article/view/380

[7] J. Bezina, J. Stebel, Analysis of model error for a continuum-fracture model of porous media flow, in: T. Kozubek, R. Blaheta, J. stek, M. Rozlonk, M. ermk (Eds.), High Performance Computing in Science and Engineering, no. 9611 in Lecture Notes in Computer Science, Springer International Publishing, pp. 152–160, DOI: 10.1007/978-3-319-40361-8_11.
URL http://link.springer.com/chapter/10.1007/978-3-319-40361-8_11

[8] M. G. Trefry, C. Muffels, FEFLOW: A finite-element ground water flow and transport modeling tool 45 (5) 525–528. doi:10.1111/j.1745-6584.2007.00358.x.
URL http://onlinelibrary.wiley.com/doi/10.1111/j.1745-6584.2007.00358.x/abstract

[9] J. Březina, J. Stebel, P. Exner, D. Flanderka, Flow123d, http://flow123d.github.com (2011–2015).

[10] M. J. Gander, C. Japhet, Algorithm 932: PANG: Software for nonmatching grid projections in 2d and 3d with linear complexity 40 (1) 1–25. doi:10.1145/2513109.2513115.
URL http://dl.acm.org/citation.cfm?doid=2513109.2513115

[11] A. H. Elsheikh, M. Elsheikh, A reliable triangular mesh intersection algorithm and its application in geological modelling 30 (1) 143–157. doi:10.1007/s00366-012-0297-3.
URL http://link.springer.com/article/10.1007/s00366-012-0297-3

[12] N. Platis, T. Theoharis, Fast ray-tetrahedron intersection using plucker coordinates 8 (4) 37–48.
URL http://www.tandfonline.com/doi/abs/10.1080/10867651.2003.10487593

[13] J. R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates 18 (3) 305–363.
URL https://www.researchgate.net/publication/279567628_Adaptive_precision_floating-point_arithmetic_and_fast_robust_geometric_predicates