Intersection of non-matching grids of different dimensions

Jan Březina[a,*], Pavel Exner[a]

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

Abstract

TODO: An 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

*Corresponding author.

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

are randomly generated) the compatible meshing becomes painful or even impossible. In order to avoid these difficulties we may discretize the continuum 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 $d_i \in \{1, 2, 3\}$, $i = 1, \ldots, N_{\mathcal{T}}$. We assume that the meshes $\mathcal{T}_i$ are connected, that is the graph having elements for vertices and their neighbouring for edges is connected. We also assume that no submesh intersects itself. Further we assume that only $\mathcal{T}_1$ has $d_1 = 3$ and $d_i < 3$ for $i > 1$. 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. Every (possibly degenerated) intersection polygon (IP) is determined by its corners called intersection corners (IC). For every IC we want to compute its barycentric coordinates on K and on L.

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 present family of efficient algorithms based on Plücker coordinates for computing 1d-2d, 1d-3d, 2d-2d, and 2d-3d intersections of simplicies. Second, we use the front tracking algorithm both to minimize set of intersection candidates as well to reuse part of calculations made on neighboring intersections. 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 in 1D-2D, 1D-3D, 2D-2D, 2D-3D cases. It will be suitable to denote S_i a simplicial element with i vertices (of dimension $i - 1$). In general, an intersection can be a point, a line segment or a polygon. All these three objects are represented as a vector of vertices called *intersection points* (later denoted IP, or IPs in plural).

The fundamental idea is to compute intersection of 1D-2D simplices using the Plücker coordinates and reduce all other cases to this one.

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.

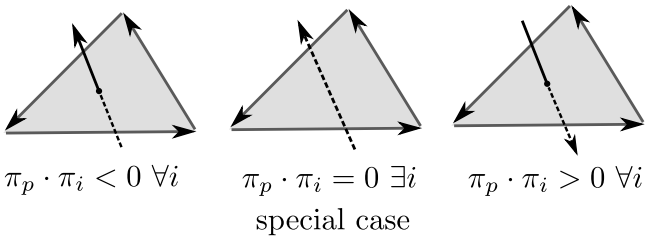

Figure 1: Sign of the permuted inner product is related to the relative position of the two oriented lines. Dashed line symbolizes that the line is in the back, the lines intersect in the middle case. **PE:** *circle dot permuted inner product*

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

PE: *In this paragraph we investigate real lines. Trimming comes later when tolerance is denoted, see next PE note.* Let us consider a line p with parametric equation $\boldsymbol{X} = \boldsymbol{A} + t\boldsymbol{u}$, $t \in \mathbf{R}$ and a triangle given by vertices $(\boldsymbol{V}_0, \boldsymbol{V}_1, \boldsymbol{V}_2)$ with oriented sides $s_i = (\boldsymbol{V}_j, \boldsymbol{V}_k)$, $j = (i+1) \bmod 3$, $k = (i+2) \bmod 3$. The permuted inner products $\pi_p \odot \pi_{s_i}$, $i = 0, 1, 2$ have the same sign if and only if there is an intersection point with the line p inside the triangle. In such a case, the inner products provides scaled barycentric coordinates of the intersection on the triangle. In particular for the barycentric coordinate w_i related to the vertex $\boldsymbol{V}_i$ we have

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

Indeed, 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 $(\boldsymbol{u}, \boldsymbol{u} \times \boldsymbol{X})$ (Plücker coordinates are invariant to change of 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

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

Having the barycentric coordinates of the intersection on the triangle we can compute also its local coordinate on p from its parametric form:

$$X_i = A_i + t u_i, \text{ for } i = 1, 2, 3$$

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

adaptive precision arithmetic [11] (based on edge-triangle intersections, use calculations that are similar to usage of Plücker coordinates), [13]

Next, we shall discuss the case when some of the permuted inner products are zero. In fact, we test if any barycentric coordinate is under given tolerance ($|w_i| \leq \epsilon$). Various *pathological* cases are detected and further topological information is atached to the intersection data. In the case of one zero coordinate (non-coplanar intersection on the edge) or two zero coordinates (intersection in the vertex), we add the local edge index or the local vertex index to the intersection, respectively. When all coordinates are (close to) zero, the line and the triangle are (nearly) coplanar. For every edge s_i we compute its intersection with the line segment p in terms of the local coordinates t_i and t_p. Only intersections with $t_i \in (-\epsilon, 1 + \epsilon)$ are accepted, for t_i close to 0 and 1 we append the vertex index to the intersection. Edges parallel to p or lying on the p are skipped.

Finally, we obtain zero, one, or two intersection points with additional topological information. The intersection point data structure keeps for each of the two intersecting elements:

- Barycentric coordinate of the intersection.
- Dimension of the most specific subelement (face, edge, vertex) where the point lies.
- Local index of the most specific subelement.

Further, we store the sign of the permuted inner product in order to detect various special and degenerate cases higher dimensional cases.

TODO: We should distinguish "special" case and "degenerate" case. The later one is the case when line lies in the plane of the triangle. The first case should be treated as usual only set aditional topology info about vertex or edge. The degenerate case is not necessary to investigate in the case of auxiliary intersection only in the case of final intersection. Need to verify it since it is not the case in current code.

We distinguish two variants of the desired result: the auxiliary intersection used as subprocedure of the other intersections and the final intersection. In the first case we just return the array of the intersection points with the line. **PE:** *In the first case, we do not trim the line (We need both IPs (if there are two), for 1D-3D). In the final case, we need to check that also* $t_p \in (-\epsilon, 1 + \epsilon)$. *That*

is the current state. Degenerated case of p laying on the triangle edge is also detected.... **JB:** *No, in 1D-3D we doesn't need result of the degenerate case (all products zero), which is the only case we can obtain two IPs. I'm going to do not deal with the two variants and focus more on correctness then on optimality of the described algorithm.*

Intersection with line segment ... (final 1D-2D intersection)

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

JB: *Has to be completed yet, however we consider that we obtain 0,1, or 2 distinguish IPs for all cases including various touch cases. Anyway every IP results from some non-degenerate 1d-2d intersection. We also assume that the IPs are in order according to the orientation of the side which is counterclockwise aroung the triangle.* We consider a line p and a tetraherdon... Tetrahedron has six edges, so six Plücker coordinates and inner products are computed at most. These data are passed to the 1D-2D algorithm described above, which is applied to the every face of the tetrahedron face.

There might be two intersection points maximally, which are finally checked that they belong o abscissas. If the line has its boundary point inside the tetrahedron, the intersection line must be cut and coordinates of intersection point are interpolated. Pathologic cases are again carefully processed so we obtain the most extensive topology information we can. See [**?**] for detailed description of all possible cases.

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

The intersection of a triangle S_3 and a tetrahedron S_4 is an n-side polygon, $n \leq 7$. The sides of the polygon lie either on sides of S_3 or on faces of S_4. Thus each vertex (IP) of the polygon can arise either from side-face intersection, or from edge-triangle intersection, or be a vertex of S_3. 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 IPs (Algorithm 1) provides all IPs on the boundary of S_3, calculation of edge-triangle IPs (Algorithm 2). **PE:** *The following sentence is somehow weird.. I would say that we store IPs I; to sort I during the computation, we use the two tables; at the end, I is ordered correctly.* Correct order of the IPs, 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_3, or on a face, or on an edge of S_4. Let us denote M_3 the set of sides of S_3 and M_4 the set of volume, faces, edges, and vertices of S_4. Every side of polygon that lies on $x \in M_3 \cup M_4$ is followed by an IP given by $F_g[x]$ and every IP p is followed by the side that lies on $F_p[p] \in M_3 \cup M_4$.

Algorithm 1 passes through every side s of the triangle S_3 and computes the line-tetrahedron intersection L. In the regular case we process each of the two

Algorithm 1: 2d-3d intersection, points on triangle boundary

Input: input data
Output: List of IPs on sorted output data

```
1  F_g(:) = −1,  F_p(:) = −1                                  // Unset links.
2  for side s of S_3 do
3      L = intersection(s, S_4)
4      if |L| = 0 then continue
5      if |L| = 1 then append p to J continue
6      for p in L do
7          if p ≠ I[−1] then append p to I
8          p lies on x ∈ M_4
9          if p is first in L then
10             F_g[x] = p, F_p[p] = s
11         else p is the last in L
12             F_g[s] = p,  F_p[p] = x
13             if F_g[x] = −1 then F_g[x] = p
14 if I[−1] = I[0] then
15     x = F_p[I[−1]]
16     if F_g[x]] = I[−1] then
17         F_g[x] = I[0]
18     F_g[s_2] = I[0]
19     remove I[−1]
```

IPs in L. The IP p is added to the list I unless the last point is the same. This condition is effective just for the first IP in L and merges IPs at the same vertex of S_3. Then we identify an object $x \in M_4$ 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 IP 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 2 to fix $F_p[p]$ for the second p lying on an edge or a vertex of S_4. The condition at the end deals with the case in which two sides intersect the same object $x \in M_4$.

For the regular intersection L, we pass through its IPs, as described above. **PE:** *For special cases*, we firstly compute IPs on the boundary of S_3 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_4 at its edge or vertex. These IPs will be rediscovered again in Algorithm 2 with better topological information, however this is not the case if the touched edge e of S_4 is coplanar with S_3 and the IP is inside of e. Therefore we keep a separate list J of IPs to deal with this case.

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

Input: I with IPs on S_3 boundary, partially filled F
Output: all IPs in I, complete F

1 **for** *edge e of S_4* **do** $L[e] =$ intersection(e, S_3)
2 **for** *unmarked edge e of S_4* **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_4
7 $(f_0,\ f_1) =$ **vertex faces** (v,L) `// Algorithm 3`
8 mark all edges coincident with p
9 **if** *p is on boundary of S_3* **then**
10 p lies on edge or at vertex $x \in M_4$
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 2 uses the line-triangle intersection algorithm for the edges of S_4 (line 1). The loop produces IPs in the interior of S_3 and possibly those IPs

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

Input: vertex v of S_4, $L[:]$ intersection results for edges of S_4
Output: (x_1, x_2), $x_1, x_2 \in M_4$, coincident with v and intersected by the plane of S_3

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 IP 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 IP of v to J **return** anything

with special position on vertex or edge of S_4 already computed in Algorithm 1. Every edge e of S_4 is oriented so that the pair of adjacent faces f_0, f_1 appears in the same order on the intersection polygon when the IP on e has a negative sign (see Figure 1). The function *edge faces*, used on line 4 and later on, uses the sign of the intersection to return the pair of faces ordered correctly. Similarly the function *vertex faces* (Algorithm 3, described later) returns a pair of generalized faces (face or edge) possibly adjacent to the IP $L[e]$ at the vertex v of S_4. On line 7 we mark all edges coincident with the vertex, since if they have any other intersection point they are degenerate and not processed anyway.

2.4.1. Vertex Faces Algorithm

This function gets an IP p at vertex v of S_4 as a parameter. The IP is special vertex case of non-degenerate edge-triangle intersection. The function returns a pair of generalized faces of S_4 preceeding and succeeding p on the polygons boundary in the case that p is at interior of S_3. The basic idea is to use the signs of IPs of the three edges coincident with v. Possible cases are:

- **All IPs have the same sign.** (line 12) We return any pair of faces. S_3 is touching S_4 at the vertex v, the polygon degenerates into the single IP p, no connection information from table F is necessary.
- **Single IP has the opposite sign to the other two.** (line 10) Let e be the edge of the single IP with the different sign. The plane of S_3 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 IP.** (line 4) Let us denote e the edge with degenerated IP 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_4) or the same sign (the plane is touching S_4 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 IPs.** (line 2) A face of S_4 lies in the plane of S_3, single edge e have non-degenerate IP. 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 4: 2d-3d intersection, finish sort of points

Input: all points in I
Output: polygon in K in correct order
1 $P = (1, \ldots, n-1, 0)$// `array of sucessors`
2 **for** f *is face of* S_4 **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 point that follows after f on the boundary of traced polygon, similarly $out[f]$ stores index of the intersection point 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_4 compute sign of intersection(e, S_3), 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_4, 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_4.
   If p is on face f_0 of S_4, set $in[f_0]$ to index of p. Similarly, if q is on face f_1 of S_4, set $out[f_1]$ to index of q. laying on faces f_p, f_q of S_4.
2. L consists of a single intersection point p (touching S_4)
   If p is on edge, compute sign of intersection(e, S_3), 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_4.

intersection polygon are found as intersection points 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 points 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 point 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 points).

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. Front tracking algorithm

Prerequisites:

- need information about element connectivity (mesh preprocessing)
- find the starting intersection – AABB (Axes Aligned Bounding Boxes)
- possibly create Bounding Interval Hierarchy (BIH)

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 3D elements, which we shall call a *bulk*. Obtaining all of the 1D-3D and 2D-3D intersections is based on finding the first two elements intersecting each other. Then we can prolongate the intersection by investigating neighbouring elements.

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

It can be seen as a *breadth-first search* [2] algorithm on the graph of component elements.

breadth first search algorithm:

1. Get next unprocessed component element k.
2. Find intersection candidates $\mathcal{K}$ in bulk mesh (3D elements).
3. For $K \in \mathcal{K}$ compute intersection (k, K).
4. Push the intersection neigbours into queues: $(k, L) \to Q_b$, $(l, K) \to Q_c$.
5. While $(l, L) \in Q_b$ check intersection (l, L). Append queues.
6. Pop pair from Q_c. (move to the next component element: go to 3).
7. If Q_c is empty, go to 1.

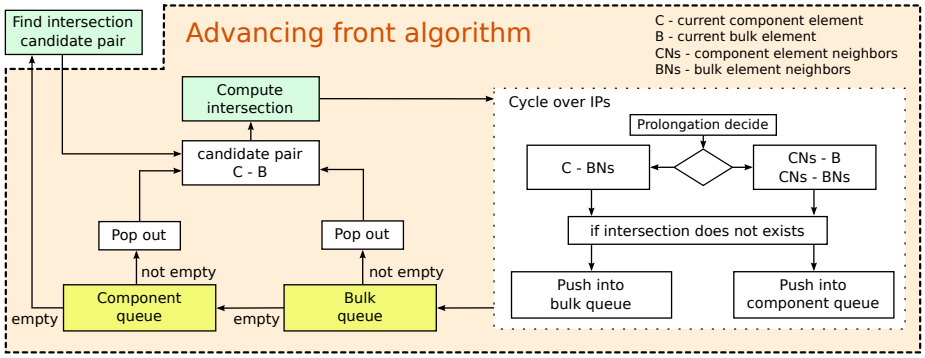

Figure 2: Prolongation algorithm for 1D-2D and 2D-3D intersections.

The advancing front algorithm works the same way for both 1D and 2D components and is displayed 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 their topological information. There are 3 possible cases (applies both for 1D-3D and 2D-3D):

- **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

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

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 following new candidate pairs into the bulk queue: [current component element – bulk neighbor], [component neighbor – bulk neighbor].

- **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