Fast Algorithms for Intersection of Non-matching 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?* **JB:** *Non-matching grid seems to be the most common.*

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 [4] 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 [5] is also studied in the groundwater context, see e.g. [6], [7], [8] and already adopted by some groundwater simulation software, e.g FeFlow [9] and Flow123d [10]. Nevertheless as the complexity of the geometry increase (e.g. when lot of fractures are randomly generated) the compatible meshing becomes painful or even

*Corresponding author.

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

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 prerequisite 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.

According to our knowledge there are lot of works using non-matching grids yet only few of them discuss algorithms how to compute their intersections. Gander and Japhet [11] present the PANG algorithm for 2d-2d and 3d-3d intersections that can be used e.g. for mesh overlapping methods. They use the advancing front technique to get candidate pairs in linear time. The algorithm is part of the DUNE library [12]. Massing, Larson, and Logg [2] present an algorithm for 2d-3d intersections as part of their implementation of the Nitche method which is part of the Doplhin project [13]. They use axes aligned bounding boxes of elements (AABB) and bounding interval hierarchy (BIH) to get intersection candidate pairs of elements, the GTS library [14] is used for 2d-3d intersections. Finally, there is the work of Elsheikh and Elsheikh [15] presenting an algorithm for 2d-2d mesh union operation which includes calculation and imprinting of the intersection curves. They exploit binary space partitioning for search of initial intersection and the advancing front method for intersection curve tracking.

In this paper we present a new approach to calculation intersections of simplicial elements of different dimensions based on the Plücker coordinates further developing the algorithm of Platis and Theoharis [16] for ray-tetrahedron intersections. Element intersections based on Plücker coordinates are combined with the advancing front method which allows us to reuse Plücker coordinates and their products between neigbouring elements and reduce the number of arithmetic operations. The paper is organized as follows. In Section 2 the algorithms for 1d-2d, 1d-3d and 2d-3d intersections of simplices are described. In Section 2.5 we discuss our implementation of the advancing front technique and usage of AABB and BIH for its initialization. Finally, in Section 4, we provide benchmarks and comparison of individual algorithms.

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 address intersection algorithms for 1D-2D, 1D-3D, 2D-3D pairs of elements. We have implemented the case 2D-2D as well however the treatment of the special cases is quite technical and not fully completed yet. 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.

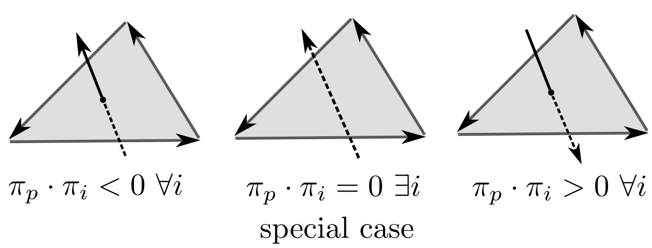


Figure 1: Sign of the permuted inner product is related to the relative position of the two oriented lines. Dashed line symbolizes that the line is in the back, the lines intersect in the middle case. **PE:** *circle dot permuted inner product*

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 [17] and used for 2d-2d mesh intersections in [15]. 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 [18] 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 [16]. 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.

```
I = {} for unmarked face f of S_3 do
    L = intersection(p, f)
    if L is none or degenerate then continue
    if L is inside the edge e then set τ_{S_4} = (1, e)
    mark faces coincident with e
    else if L is at the vertex v then
        set τ_{S_4} = (0, v)
        mark faces coincident with v
    append L to I if |I| = 2 then break
if |I| = 1 and I is outside of p then erase I
else if |I| = 2 then
    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 intersection polygon (IP), $n \leq 7$. 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. To get all ICs, 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 IP consists of three stages: calculation of side-tetrahedron ICs (Algorithm 2), calculation of edge-triangle ICs (Algorithm 3), oredering of ICs (Algorithm 5). The intersection corners appended to the list I as they are computed however their order on the polygon boundary is defined by the *connection tables* $F_g(:)$ and $F_p(:)$. Every side of the polygon that lies on n-face $x \in M_2 \cup M_3$ is followed by an IC given by $F_g[x]$ and every IC p is followed by the side that lies on $F_p[p] \in M_2 \cup M_3$ (see **JB:** *possible figure*). The vertices of the polygon are numbered in the same way as is the order and orientation of the sides of S_2, that is counterclockwise around the interiror with normal pointing to us.

Algorithm 2 computes all ICs on the boundary of S_2. It passes through every side s of the triangle S_2 and computes the line-tetrahedron intersection L. If the side is just touching S_3 and we get L with single IC. 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. To this end we save the IC into separate list J and skip filling of the connection tables. In the regular case, we process each of the two ICs in L (loop on line 2). The IC p is added to the list I, vertices of S_2 added twice are merged at final stage. Then (line 6) we identify the n-face $x \in M_3$ the point p lies on and set the tables F_g, F_p. For the IC at the vertex of S_2 we set x to that vertex regardless of its position on S_3. The backward temporary link on the line 13 is used later in Algorithm 3 to allow proper continuation of the IC if it lies on edge or at vertex of S_3. The condition at the same line deals with the case that two sides of S_3 intersect the same object $x \in M_3$, in such case the back-link is unnecessary and would overwrite the correct link set by previous IC.

Algorithm 3 uses the line-triangle intersection algorithm for the edges of S_3 (line 1). First, the intersection $L[e]$ is evaluated for every edge e. The loop

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**
  $L = \text{intersection}(s, S_3)$
  if $|L| = 0$ **then continue**
  if $|L| = 1$ **then** append p to J **continue**
  for p *in* L **do**
    p lies on $x \in M_3$
    if p *lies at vertex* v *of* S_3 **then** $x = v$
    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** $F_g[x] = p$

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

for *edge* e *of* S_3 **do** $L[e] = \text{intersection}(e, S_2)$
for *edge* e *of* S_3 *with non-empty* $L[e]$ **do**
  $p = L[e]$
  if p *is inside* e **then**
    $(f_0,\ f_1) =$ **edge faces** (e)
  else p at the vertex v of S_3
    $(f_0,\ f_1) =$ **vertex faces** (v, L) `// Algorithm 4`
  if p *is on boundary of* S_2 **then**
    p lies on edge or at vertex $x \in M_3$
    $q = F_g[x]$ `//` q `is already computed` p
    if $F_p[q] = x$ **then** $F_p[q] = f_1$
    else $F_g[f_0] = q$
    $F_g[x] = -1$ `// remove the backlink`
  else
    append p to I
    $F_g[f_0] = p,\ F_p[p] = f_1$ `// overwrite the backlink`

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. Then we pass through once again skipping edges with none or degenerate $L[e]$. For every IC $p = L[e]$ we first get its (generalized) faces that are before and after the IC on IP. For an IC inside the edge e the function *edge faces*, returns its adjacent faces f_0, f_1 (see Fig. 2). Their order is given by the sign permuted inner products in 1d-2d intersection. The order of faces match the order of sides of IP if the sign is negative. If the sign is positive the function *edge faces* returns face pair (f_1, f_0).

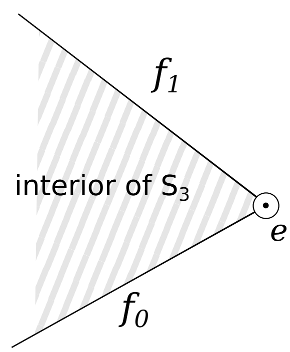


Figure 2: Order of faces adjacent to the oriented edge e pointing up.

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

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

(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 5), 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, order the points

Input: all points in I
Output: polygon in K in correct order

if $|I| < 3$ **then** return J
else return I sorted according to connectivity in F_g and F_p

JB: *Write correct algo with deduplication of vertices.*

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*

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

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, or 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,
- **Bounding Interval Hierarchy (BIH)** – we alternatively create BIH above AABB to fastly search created bounding boxes for two colliding 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 3 and see us moving from the green box in the left upper corner. If an intersection exists, we have just started a new component and we can procced to tracking the intersection. Otherwise, we select another 1D or 2D element and start over.

Let us now discuss the advancing front algorithm displayed by the scheme in Figure 3. The main idea is to compute intersections for a component element

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

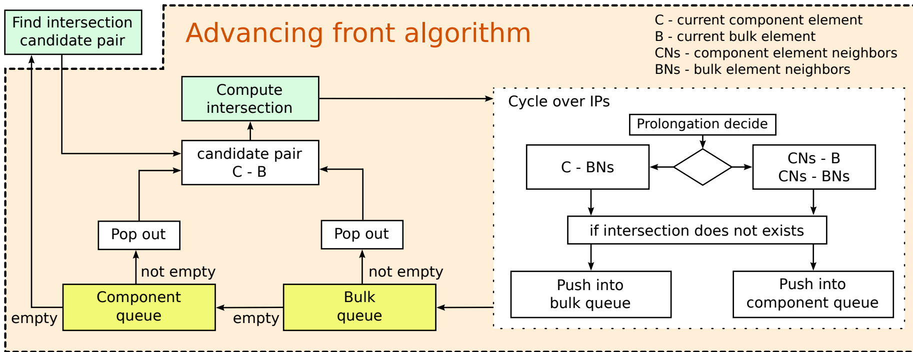


Figure 3: Advancing front algorithm for 1D-2D and 2D-3D intersections.

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).

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 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.

PE: *We can discuss further the covering/closing of the elements and component numbering which is not tested thoroughly at the moment. We can show in a figure the case in 'prolong_meshes_13d/prolongation_13d_04.msh', where actually 4 components are found (therefore bulk is defined as connected 3D elements).*

4. Benchmarks

In this section we present numerical results on several benchmark problems. At first we shall compare the effectivity of our fundamental algorithms for intersections with other approaches. **JB:** *I prefer to call them element intersection. Fundamental intersection seems to epic.* **JB:** *We compare just against NGH which we should briefly describe. Neither of the algorithms is optimized. The new one deals correctly with special cases and provides barycentric coordinates (so more work for less time).* **JB:** *Also write little about Flow123d here. The language etc.*

Next we shall compare our algorithms with different initialization phase (candidate pairs search), and using the advancing front method or not. We shall show the results both on a mesh of a real locality and an artificial mesh.

4.1. Fundamental Intersection Algorithms

The first benchmark focuses on the fundamental 1D-3D and 2D-3D intersections. We randomly generated 100000 element pairs inside a unit cube, from which approximately 65% have nonempty intersection. In some of the empty cases, the bounding boxes did not collide, so the actual intersection algorithms were skipped. A single element pair was computed 100 times to obtain reasonable computational time.

algorithm, 2D-3D	FLOPs estimate
parametric plane	585
normal plane (reuse)	765
Moller and Trumbore	783
Plücker (reuse)	306

Table 1: Estimation of floating point operations count and comparison of different approaches. Only 2D-3D case considered.

PE: *comment on different strategies* **JB:** *Rather put into text. I shall comment that.*

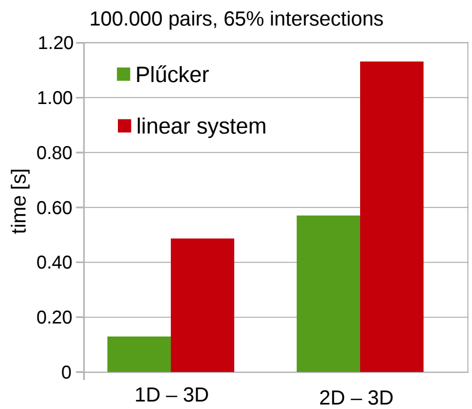


Figure 4: Efficiency of Plücker approach compared to the legacy NGH code.

We see the benchmark results in the Figure 4. Green values correspond to the presented algorithms using Plücker coordinates, red values **PE:** *****NGH. The gained speed up factor is approximately 5 in 1D-3D case and 2 in 2D-3D. Further we provide estimated count of floating point operations for different approaches in the Table 1.

4.2. General algorithms

So far, we have suggested an intersection algorithm that uses axes aligned bounding boxes, BIH to find initial intersection and advancing front method (shortly AFront) to prolong the intersection. From now on, we shall refer to the algorithm made of these components *BIHsearch*. In this section, we shall compare two additional algorithms *BBsearch* and *BIHonly* to see the effects of BIH and AFront (see Table 2). *BBsearch* does not use BIH, but searches for the initial intersection using only AABB, and then follows AFront. The version *BIHonly* computes AABB together with BIH, but does not use AFront.

BIHsearch	BBsearch	BIHonly
BIH(AABB)	AABB	BIH(AABB)
AFront	AFront	—

Table 2: Structure of the general algorithms. 'AFront' stands for the proposed advancing front method. **JB:** *Just in the text. Possibly use different name for variants.*

Let us now start with the problem on an artificial mesh.

Next, we study the performance of the intersection algorithms on a mesh of a real problem, see Figure 7. The mesh represents a mountain ridge in the Jizera mountains which includes a system of geological fractures (Figure 7a). We also used this model to create a less real mesh, in which the fractures extend through the bulk surface (Figure 7b). However, this situation might rather occur

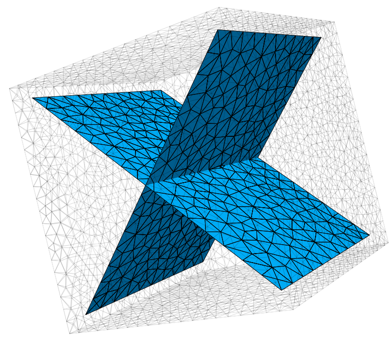


Figure 5: Artificial mesh – a cube with two perpendicular planes placed on the diagonals of the cube. The planes are also nonmatching, therefore can be seen as two independent components.

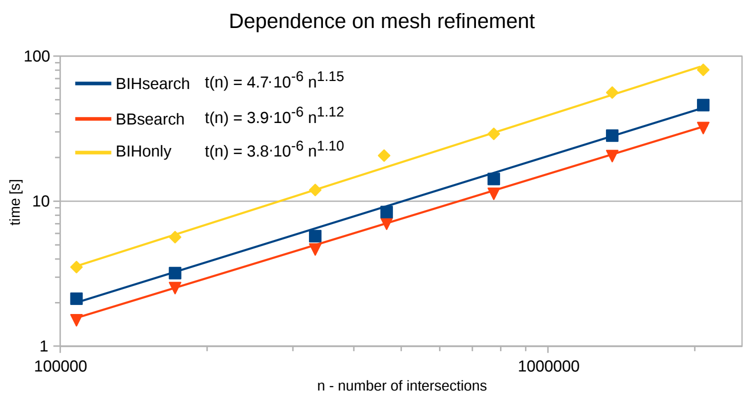


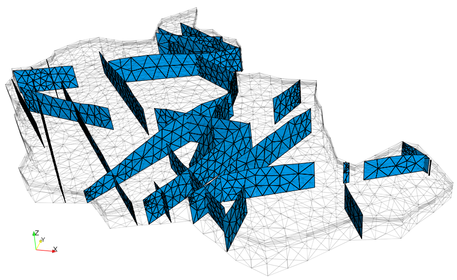
Figure 6: asdf

in another application than aporous media problem. In total, 28 fractures are in the mesh.

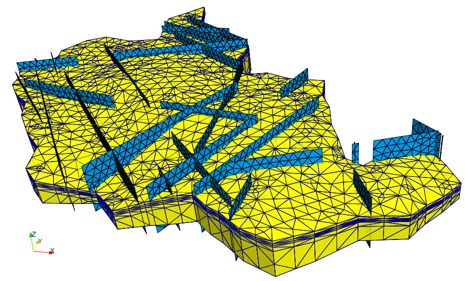
The results for both meshes can be seen in the Figure 8, pay attention to the different time scales in the graph. In the first case, we notice that algorithms using AFront are nearly twice as fast as *BIHonly*. The speed up corresponds to the time saved by using AFront instead of searching the whole mesh for candidate pairs. Computation of BIH in *BIHsearch* pays off and the algorithm performs better than *BBsearch*.

In the second case, we observe large blow up for *BBsearch*. It is caused by the exterior elements, for which all bulk elements bounding boxes are iterated, so we can be sure there is no intersection. This of course is much better performed using BIH.

In contrast to the artificial case on the fractured cube, BIH construction is worth due to the presence of more components. Having more fractures inside the bulk would result to even bigger difference.



(a) interior fractures



(b) extending fractures

Figure 7: A mesh of the real locality of Bedřichov in the Jizera mountains. We see fractures inside the bulk mesh in the left figure, fractures are extending the bulk mesh.

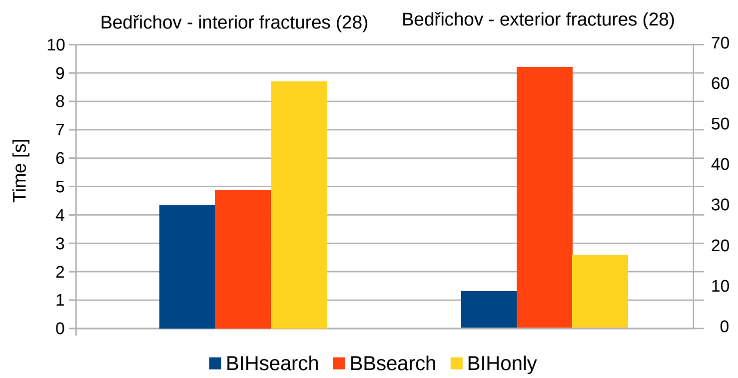


Figure 8: Comparison of the algorithms on meshes of Bedřichov locality – interior fractures on the left, extending fractures on the right.

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

[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] T.-P. Fries, T. Belytschko, The extended/generalized finite element method: An overview of the method and its applications 84 (3) 253–304. doi:10.1002/nme.2914.
URL http://onlinelibrary.wiley.com/doi/10.1002/nme.2914/abstract

[5] 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

[6] 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

[7] 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

[8] J. Březina, J. Stebel, Analysis of model error for a continuum-fracture model of porous media flow, in: 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

[9] 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

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

[11] 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

[12] P. Bastian, M. Droske, C. Engwer, R. Klfkorn, T. Neubauer, M. Ohlberger, M. Rumpf, Towards a unified framework for scientific computing, in: T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, T. Schlick, R. Kornhuber, R. Hoppe, J. Priaux, O. Pironneau, O. Widlund, J. Xu (Eds.), Domain Decomposition Methods in Science and Engineering, no. 40 in Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, pp. 167–174, DOI: 10.1007/3-540-26825-1_13.
URL http://link.springer.com/chapter/10.1007/3-540-26825-1_13

[13] A. Logg, G. N. Wells, J. Hake, Dolfin: a c++/python finite element library (2012).

[14] Gts, gnu triangulated surface library, software package., http://gts.sourceforge.net/.

[15] 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

[16] 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

[17] 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

[18] S. Fortune, Stable maintenance of point set triangulations in two dimensions, in: Proceedings of the 30th Annual Symposium on Foundations of Computer Science, SFCS '89, IEEE Computer Society, pp. 494–499. doi:10.1109/SFCS.1989.63524.
URL http://dx.doi.org/10.1109/SFCS.1989.63524