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 gird, intersections, mixed-dimensional mesh, Plücker coordinates

1. Introduction

Structure of introduction:

- motivation for computing mesh intersections, other applications - basic overview of existing work, approaches - our contribution - new algorithms for element intersectins based on Plücker coordinates - improvement of the front tracking algorithm

[**?**] - Use *g*eometric predicates incircle and orientation. Use mesh of bounding boxes to search for intersection candidates. Only use point queries for all vertices of triangle/tetrahedron. Use compatible partitioning of elements by the crack elements.

[**?**] - Implementation of the Nitche method in Fenics. They need 2d-3d intersections, but only for distribution od quadrature points. So they do not store whole intersection data, but only the quad points. The boundary of one domain is partitioned into intersections with elements of the other domain. Use emph collision relations (pairs of faces of boundary and elmements of the background mesh) and *c*ollision maps (map face to all intersections and background element to all intersections, can be constructed from the collision relation), They cite books from computer graphics .. Mention geometric predicates. Present algorithms for: find intersection candidates, comput intersections, integrate over complex domains.

[**?**] Claim to provide robust algorithm covering all tiangle-trinagle degenerate cases. Basic idea is the same as our: use some tree (Binary SPace Tree) structure to find initial point on intersection curve and trace the curve in both

*Corresponding author.

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

directions. Quite clever data structure for the mesh. Adaptive precision geometric predicates. Store topological information about the intersection points (point on triangle, edge, vertex). Resolve various degenerate cases, dicuss mesh optimization after subdivision.

[**?**] Full algorithm for 2d=3D called PANG. Implemented in DUNE in 2010 (Bastian). Other software (Hecht, MpCCI). Triangle-Triangle and Tetra-Triangle intersections done similarly as in aur approach: set of points including vertices and then sort them countercolckwise. Local intersection algorithm do not provide neighbouring information so the traversal algorithm do not use it.

Review of algorithms: [**?**]

[**?**] Robust discretization of porous media. Complete error analysis with mortar non-compatible case. Seems they use normal fluxes accros fracture instead of pressure traces. This allows conforming discretization and error results.

[**?**] Application of mixed-dimensional approach in mechanics. Beam and shell elements ...

[**?**] Finite volume method, fully implicit two phase flow. Cite usage of FRAC3D generator.

[**?**] Space-time DG with adaptive mesh, need integration of product of functions on different meshes (in 2D). An algorithm for computing triangle-triangle intersections in 2D.

[**?**] DFN + MHFEM. Mortar method for intersections. Focused on formulation not on intersections.

[**?**] (Barbara) Mortad, non=matching, acustics. Theory, structured gird case. No complex intersections.

[**?**] Patent for curiosity.

[**?**] Immersed boundary. ??

[**?**] ??

[**?**]

[**?**] Original algorithm for line-thetrahedra intersections.

[**?**] Line-convex Polyhedra intersections using generalisation of line-clipping algorithm

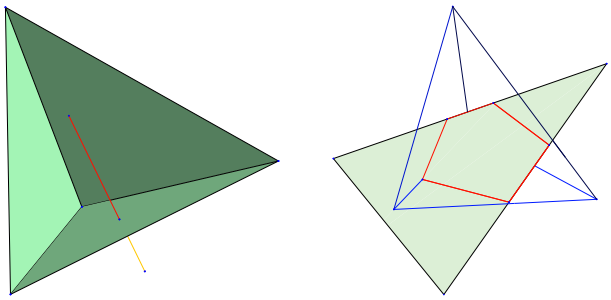
The initial problem that occurs when working with incompatible meshes of combined dimensions is the efficient computation of intersections of the mesh subdomains, in particular 1D-2D, 1D-3D, 2D-3D. There is a piece of code in Flow123d at the moment which uses means of analytic geometry. In particular, it computes an intersection of a line and a plane in 3D by solving Gaussian elimination. However it is not effiecient and it does not provide barycentric coordinates explicitly. In this section, we present our latest work on this problematics.

The topic is studied in the bachelor and master thesis of Viktor Friš [**?**], who was supervised by Jan Březina. They suggested a solution, based on [**?**] for the two aspects of the problem:

itemsep=-3pt an efficient identification of the intersecting pairs of elements,

iitemsep=-3pt an efficient computation of the actual intersection for the single element pair.

The algorithm is based on the methods used in the computer graphics, in particular for the task (b) Plücker coordinates are employed to efficiently compute 1D-2D intersection in three-dimensional space.



(a) 1D-3D (b) 2D-3D

For the 1D-3D and 2D-3D cases, the basic algorithm is applied on sides and edges of the simplices and all the topological information coming from the performed Plücker products is carefully collected to obtain final intersecting object. Further in 2D-3D case, an optimized tracing algorithm is suggested to directly obtain the correct order and orientation of the edges of the intersection polygon.

2. Element Intersections

In this section, we present algorithms for computing intersection of a pair of simpicial elements of different dimension in the 3D ambient space. In particular we are interested in intersection of 1D-2D, 1D-3D, 2D-2D, 2D-3D simplices. In general the intersection can be a point, a line segment or a polygon. All these three objects are represented as a vector of the vertices called *intersection points*. Basic idea is to compute 1D-2D case using the Plücker coordinates and reduce all other cases to that case.

The intersection point data structure keeps for each of the two intersection simplices:

- Barycentric coordinate of the intersection.
- Dimenstion of the most specific subsimplex (face, edge, vertex) where the point lies.
- Indice of the most specific subelement.

Further, keep sign of the permuted inner product we store In order to catch various degenerate cases, the intersec

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

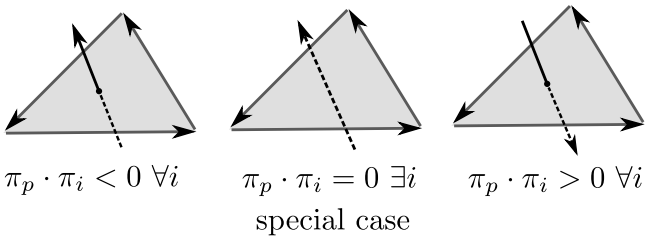


Figure 1: Sign of the permuted inner product is related to the relative position of two oriented lines. Dashed line symbolizes that the line is in the back, the lines intersect in the last case.

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

Let us consider a line p, with parametric equation $\boldsymbol{X} = \boldsymbol{A} + t\boldsymbol{u}$, $t \in (0, 1)$ and a triangle given by vertices $(\boldsymbol{V}_0, \boldsymbol{V}_1, \boldsymbol{V}_2)$ with oriented sides $s_i = (V_j, V_k)$, $j = (i+1) \bmod 3$, $k = (i+2) \bmod 3$. 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 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

$$\pi_p \odot \pi_{s_1} = \boldsymbol{u} \cdot (\boldsymbol{s}_1 \times V_2) + \boldsymbol{s}_1 \cdot (\boldsymbol{u} \times [\boldsymbol{V}_0 + w_1\boldsymbol{s}_2 - w_2\boldsymbol{s}_1]) = -w_1\boldsymbol{u} \cdot (\boldsymbol{s}_1 \times \boldsymbol{s}_2).$$

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:

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

For practical computation we use i with maximal $|u_i|$.

Next, we shell discuss the case when some of the permuted inner products is 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.

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 distinguis 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. Degenerated case of p laying on the triangle edge is also detected....

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

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

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 a polygon with 7 vertices at maximum. The sides of the polygon are coincident with either

sides of S_3 or with faces of the S_4. So a vertex of the polygon can arise from intersection of a side with a face or an edge with S_3 or from 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 consist of three parts. In the first part (Algorithm 1), we compute intersection points on baundary if S_3 using the line-tetrahedron intersection algorithm for every side. There are 0,1, or 2 intersection points on every side and the points are already sorted according to the side orientation, which match orientation of the intersection polygon. So, passing through the all sides and appending the points into I provides as with incomplete intersectionn polygon but in the correct order. Then we proceed to the second part (Algorithm 2). Using the line-triangle intersection algorithm for edges of S_4 we obtain points from interior of S_3 and need to insert them into I to correct place. To this end we exploit additional information about coincidence of points with faces of S_4. Every point p from the first part of the algorithm that is not a veertex of S_3 belongs to a face f of S_4, we know from the order of the points if the face comes before or after the point in the intersection polygon, in the first case we set $F[0, f] = p$ inthe second case we set $F[1, f] = p$. Every point p from the second part of the algorithm lies on an edge e which is between two faces f_0, f_1. The correct order of the faces on the intersection polygon is determined by the sign of the edge-triangle intersection. So, having the faces in the correct order we set $F[0, f_0] = p$ and $F[1, f_1] = p$.

Finally in the third part (Algorithm 3), the table F than allows us to modify array of successors P and get I in correct order as the list K.

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

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

```
1 I = ∅                                          // Intersection polygon.
2 F(:,:) = −1      // Unset entries of the face connection table.
3 for side s of S_3 do
4 |   L = intersection(s, S_4)                  // List of max. two points.
5 |   process point(L[0], 0)
6 |   process point(L[−1], 1)
7 if I[0] == I[−1] then remove I[−1]
```

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.

Algorithm 2: process point(p, i)

Input: intersection point p, face relation indicator i
Output: update of I and F

```
1  if p ≠ I[−1] then append p to I
2  set j to index of p in I
3  if p is on face f then F[i,f]=j
4  else if p is on edge e then
5  |  compute sign of intersection(e, S_3)
6  |  sort faces f_0, f_1 coincident with e
7  |  F[i, f_i]=j
8  else if p is on vertex v of S_4 then
9  |  for face f coincident with v do
10 |  |  if F[i, f] == −1 then F[i,f]=j
11 mark all edges of S_4 the p lies on
```

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

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

```
1  for unmarked edge e of S_4 do
2  |  p = intersection(e, S_3)
3  |  append p to I
4  |  set j to index of p in I
5  |  if p is inside of e then
6  |  |  sort faces f_0, f_1 coincident with e
7  |  |  F[0, f_0]=j
8  |  |  F[1, f_1]=j
9  |  else p on the vertex v of S_4
10 |  |  for face f coincident with v do
11 |  |  |  if F[i, f] == −1 then F[i,f]=j
12 |  |  mark all edges of S_4 coincident with v
```

Algorithm 4: 2d-3d intersection, finish sort of points

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

```
1  P = (1, ..., n − 1, 0)// array of sucessors
2  for f is face of S_4 do
3  |  if F[1, f] ≠ F[1, f] then P[F[1, f]] = F[0, f]
4  i = 0
5  for n = 0 To |I| − 1 do  K[n] = I[i] i = P[i]
```

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: *I suggest to denote IP, IPs intersection point and its plural respectively.*

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:*
component *is a set of connected elements of the same lower dimension (1D, 2D)*
bulk *is a set of 3D elements*
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..)*

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

2.6. Prolongation algorithm

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. 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.

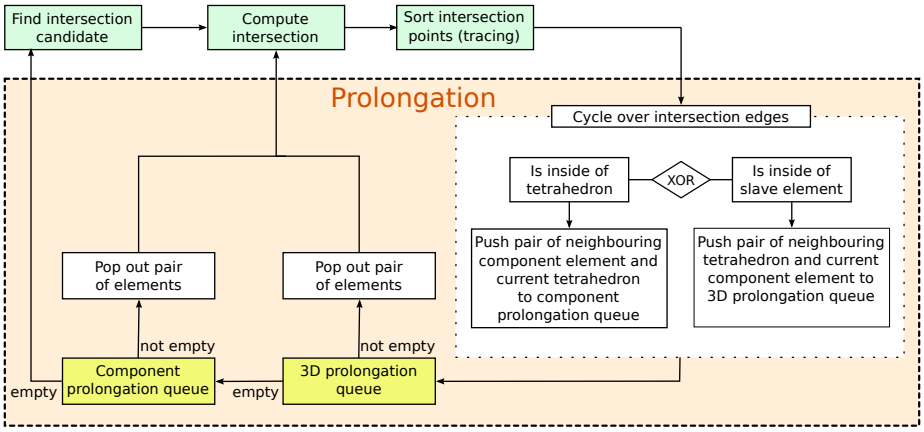


Figure 2: Prolongation algorithm for 1D-2D and 2D-3D intersections.

The prolongation algorithm is the same for both 1D and 2D components, see Figure 2. 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.

Prerequisites:

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

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

After computing the intersection of a pair of elements (line or triangle vs tetrahedron), we fill two queues with element pairs as candidates for further intersection. If the intersection edge (point of line in 1D, edge of polygon in 2D) is inside the tetrahedron, not on its surface, we get a neighbouring element of the component and push it back together with the current tetrahedron into *component prolongation queue*. If the intersection edge is inside the *slave* element (line or triangle), i.e. is on the surface of tetrahedron, we get a neighbouring element of the tetrahedron and push it back together with the current slave element into *3D prolongation queue*.

Then we empty the prolongation queues – the 3D prolongation queue at first, then the component prolongation queue. When both queues are empty, all intersections of a component have been found and we continue to look for another component.

The algorithm is now unified for 1D and 2D in contrast to [**?**], where the component prolongation queue is emptied at first.

A new candidate pair of elements is found during prologantion, based on the topological information of the intersection point. There are 3 possible cases (applies both for 1D-3D and 2D-3D):

- **IP lies on component element side and inside bulk element**
  We find all the sides of component element in which the IP lies (IP can be at node and connect more sides). We find the component neighboring elements over the sides and push a new candidate pair [component neighbor – current bulk element] into component queue.

- **IP lies on component element side and on the surface of 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 candidates pairs into the bulk queue: [current component element – bulk neighbor], [component neighbor – bulk neighbor]. If the candidates pair has been investigated already, we skip it.

- **IP lies inside component element (therefore must be on the surface of bulk element)**
  The same as before, but we push only [current component element – bulk neighbor] candidates pairs, since there is no component neighbor.

3. Front tracking algorithm

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.