Automatic Generation of Octree-based Three-Dimensional Discretizations for Partition of Unity Methods

Ottmar Klaas and Mark S. Shephard
Scientific Computation Research Center
Rensselear Polytechnic Institute
110 8th St., Troy, NY 12180
USA
email: oklaas@scorec.rpi.edu

Abstract: The Partition of Unity Method (PUM) can be used to numerically solve a set of differential equations on a domain Ω. The method is based on the definition of overlapping patches Ω_i comprising a cover $\{\Omega_i\}$ of the domain Ω. For an efficient implementation it is important that the interaction between the patches themselves, and between the patches and the boundary, is well understood and easily accessible during runtime of the program. We will show that an octree representation of the domain with a tetrahedral mesh at the boundary is an efficient means to provide the needed information. It subdivides an arbitrary domain into simply shaped topological objects (cubes, tetrahedrons) giving a non-overlapping discrete representation of the domain on which efficient numerical integration schemes can be employed. The octants serve as the basic unit to construct the overlapping partitions. The structure of the octree allows the efficient determination of patch interactions.

1. Introduction

Partition of Unity methods are capable of constructing conforming solution spaces used for the numerical solution of a set of differential equations on a domain Ω. They allow the inclusion of local properties of the solution into the constructed solution space. The basic idea is to create overlapping patches Ω_i comprising a cover $\{\Omega_i\}$ of the domain Ω with partitions of unity φ_i subordinate to the cover Ω_i. On each patch local function spaces V_i are set up reflecting the local

solution behavior. The global solution space V is then given by $V = \sum_i \varphi_i V_i$. Methods based on that principle were developed by Babuska et al. [1],[2] (Partition of Unity Finite Element Method), and Duarte and Oden [7],[8] (hp - clouds). Similar methods referred to as Element Free Galerkin Methods were developed by Belytschko et. al. [5], [11] or the Moving Least Square Reproducing Kernel Methods by Liu et al. [10].

One way to create the partition of unity is to start from an arbitrarily distributed set of nodes. No fixed connections between the nodes are required. The nodes are the centers of the overlapping patches Ω_i, which can have almost any shape with cubes and spheres among the most popular. The partition of unity is then created based on the moving least square scheme (see Lancaster and Salkauskas [9]). Advocators of this method praise the simplicity with which geometrically complex situations, like cracks, free surfaces etc., can be treated. The sometimes cumbersome task of meshing and remeshing of a valid finite element mesh can be avoided. However, due to the rather unstructured distribution of nodes over the domain other algorithmic issues arise. First, a discretization without structure does not allow determination of the patches that contribute to a certain integration point without performing a search. Second, the partition of unity based on the moving least square method creates shape functions that are expensive to integrate with the common integration rules, like Gauss quadrature formulas. Last, but not least, the treatment of the boundary conditions, and the interaction with the geometric boundary in general, becomes very difficult. Recently, Duarte, Babuska and Oden [7] proposed to use a Finite Element mesh to overcome the problems associated with an arbitrarily scattered set of nodes. The finite element mesh is used for the purpose of creating the partition of unity. The main difference to a mesh that would be suited for a finite element analysis is the lack of h-adaptation for singularities or steep gradients, which simplifies the process of mesh generation slightly. Instead, the singularities are captured by shape

functions enriched by the asymptotic expansions of the elasticity solution in the neighborhood of the singularity. Since the partition of unity is built on linear shape functions defined by the tetrahedral elements the numerical integration is easier in terms of needed integration points to reach a reasonable accuracy. Furthermore, since the usual finite element mesh topology is used no searching for shape functions contributing to a certain integration point has to be performed.

Both approaches, building the partition of unity from an unstructured set of nodes or from a finite element mesh, seem to be the extremal solutions lying on different sides of the optimal solution. From an implementation point of view it is important that the patches are clearly defined. The interaction between the patches themselves, and between the patches and the boundary, has to be well understood and easily accessible during the runtime of the program. An implementation based only on an arbitrary set of points with patches associated to them is inefficient since the code would have to perform expensive global searches to determine the interactions of the patches. On the other hand, a complete finite element mesh may represent more information than is necessary, leading to a memory inefficient program.

An alternative approach is to employ a structure to construct the patches which would provide a priori information with respect to the size and interactions of the patches to ensure a valid partition which could be generated and operated upon more efficiently than a complete mesh. To be effective, such a structure must efficiently provide all required neighbor information, while being flexible enough to provide the type of gradations of the patches necessary to obtain optimal convergence. Furthermore, it should allow the use of the moving least square method to construct the partition of unity as well as a partition of unity based on lagrangian type shape functions as they

are used in finite elements. The structure also needs to define the integration cells that have to be defined to integrate the governing equations.

In this paper we will show that an octree representation of the domain with a tetrahedral mesh at the boundary will serve the needed purposes. It subdivides an arbitrary domain into simply shaped topological objects (cubes, tetrahedrons) giving a non-overlapping discrete representation of the domain on which efficient numerical integration schemes can be employed. The octants serve as the basic unit to construct the patches, and allow the efficient determination of patch interactions. Given an adjusted octree, only a small number of possible octant neighbor constellations can arise, allowing templates to be constructed for the partition of unity functions on the octants. Last but not least, the structure of the octree reduces the memory consumption compared to a finite element mesh. For the same spatial discretization size the finite element mesh structure [4] needs about four times as much memory than the octree structure [14].

The paper is organized as follows: in section 2 we introduce the octree structure used as the basic building block for the partition of unity and the integration cells. Sections 3 and 4 describe how the open cover and the integration cells are constructed based on the octree structure. In section 5 we present two examples demonstrating the partition of unity discretization for complex three dimensional geometries.

2. The octree structure as a representation of an arbitrary domain Ω

Various types of spatially-based tree structures have been found to be effective in supporting searching operations [12] and domain discretization operations such as automatic mesh generation [16], [15]. The specific tree structure investigated here is an octree structure. Enhanced by storing octant neighbor information, the octree is capable of providing neighbor information in

constant time (instead of the standard O(log n) traversal time per determination) if the level difference between neighbored octants is controlled [14]. Furthermore, the simple shape of an octant allows an easy application of well known efficient integration rules.

An octree structure can be defined by enclosing the domain of interest Ω in a cube which represents the root of the octree, and then subdividing the cube into the eight octants of the root by bisection in all three directions. Those octants are then recursively subdivided to whatever levels are desired. The terminal octants of that subdivision process represent the basic units referred to by the application using the octree. See Fig. 1 for a picture of an octree and its corresponding data representation as a tree. We define the set O as the set of octants describing a given octree rooted by O_o. We define the $Level(O_o) = 0$, and derive

$$Level(O_i) = Level(Parent(O_i)) + 1 . \qquad (1)$$

For the ease of notation we assume a cubical root octree. The size of the root octant $size(O_o)$ is then defined as the maximum length in x,y or z direction of the geometric model underlying the discretization process. The size of a child octant O_i can be stated as

$$size(O_i) = size\left(\frac{Parent(O_i)}{2}\right) \qquad (2)$$

A single octant O_i can be identified uniquely by its parent and its child id. We will make use of an operator $Term(O_j)$ that will traverse the subtree rooted by O_j and return the set of terminal octants in that subtree.

To identify elements on the closure of an octant, a superscript is used to indicate the dimension of an octant topological entity, e.g. $\{O_i^3\}$ indicates the region of space associated with Octant O_i,

$\{O_i^2\}$ is the set of six faces bounding octant O_i, etc. With this notation in hand we introduce the operators returning the face, edge, and vertex neighbors of an octant O_i

$$FN(O_i) = \{O_j \in O \,|\, (\{O_j^2\} \cap \{O_j^2\} \neq \varnothing \wedge (O_j \neq descendent(O_i)))\} \quad (3)$$

$$EN(O_i) = \{O_j \in O \,|\, (\{O_j^1\} \cap \{O_j^1\} \neq \varnothing \wedge (O_j \neq descendent(O_i)))\} \quad (4)$$

$$VN(O_i) = \{O_j \in O \,|\, (\{O_j^0\} \cap \{O_j^0\} \neq \varnothing \wedge (O_j \neq descendent(O_i)))\}, \quad (5)$$

and we refer to the operator returning the set of all neighbors of an octant O_i as

$$N(O_i) = \{FN(O_i) \cup EN(O_i) \cup VN(O_i)\}. \quad (6)$$

We will assume that the sets are ordered, and a specific entity *j* can be picked by e.g. $VN_j(O_i)$.

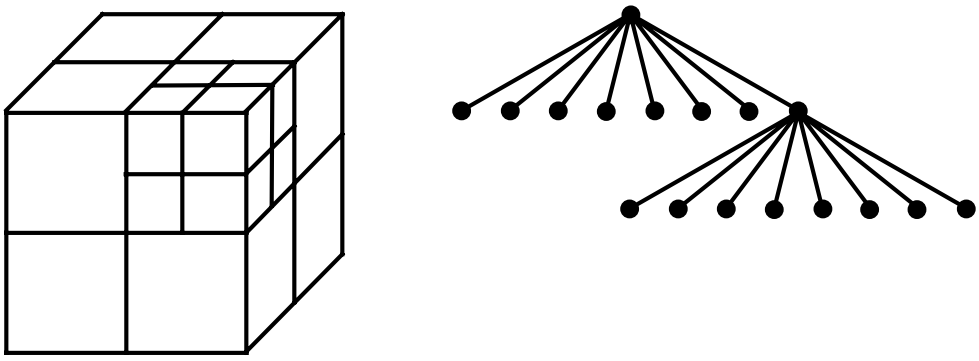

Fig. 1: The octree and its corresponding data representation.

We will define a one level adjusted octree as an octree where the level difference of all terminal octants and their face and edge neighbors is no more than one. Octants can be classified as interior octants, exterior octants, and boundary octants. Interior octants are octants that are fully embedded in the interior of the geometric domain Ω. Exterior octants are octants that are located outside

of the geometric domain Ω. Consequently, boundary octants are octants that are intersected by the boundary of the geometric domain Ω. See Fig. 2 for a simplified 2D graphical illustration.

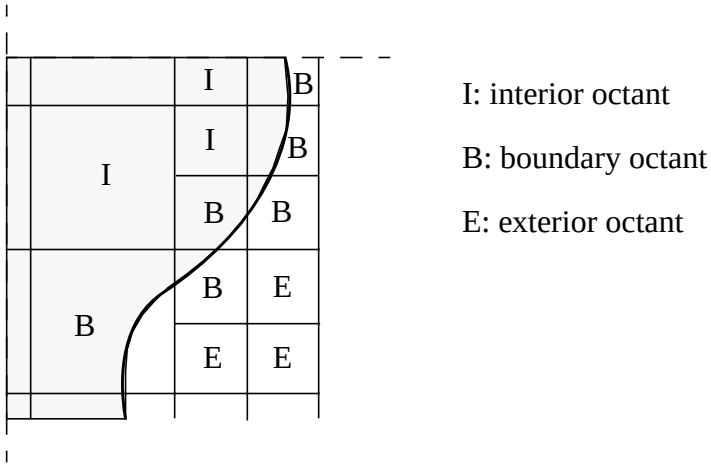

Fig. 2: Definition of interior, exterior, and boundary octants.

We introduce the operators $Interior(O_i)$, $Exterior(O_i)$, and $Boundary(O_i)$ that traverse the subtree rooted by O_i and return the set of interior, exterior, and boundary octants found in the subtree. Exterior octants are of no further use in terms of the computation. A set of interior and boundary octants will be used to define the open cover. For the ease of description we will introduce the notation O_{oc} to describe that set. We will require that the geometric model to be discretized is defined in the closure of O_{oc}. According to that definition, we allow octants, that are elements of the set, to overlap. All terminal interior and boundary octants could be used to form a valid set O_{oc}, or, as it was done in our computations, the set O_{oc} is given as

$$O_{OC} = \{Parent(Term(O_o)) \cap (Interior(O_o) \cup Boundary(O_o))\} \qquad (7)$$

The generation of the partition of unity discretization is implemented as a two step process. The first step being the generation of the octree structure, while the second step sets up the open cover and integration cells based on the octree structure. To generate an octree of appropriate size for the specific analysis discretization control information has to be applied. The information is usu-

ally applied to the topological entities of the geometric model in the form of absolute measures (e.g. maximum size of terminal octants) or relative measures (e.g. percentage error in describing curved boundaries). In terms of the to-be-constructed partition of unity, the information controls the size of the integration cells, which will be the terminal interior octants and tetrahedral elements generated in the boundary octants. After the discretization control information has been applied a root octant is determined such that the geometric model is contained within the closure of that octant. Recursively subdividing that octant and its children to the level dictated by the size control information, while maintaining the type of the newly created octants (interior, exterior, boundary), yields the octree fulfilling the control information. As a last step a tetrahedral mesh is generated in the boundary octants. References [16] and [6] discuss issues associated with the construction of the octrees as used in automatic mesh generation. The generation of the octree used here follows the same basic steps with the exception that the octants interior to the model need not be decomposed into finite elements. Given an octree structure created based on any form is octant size control, the algorithm for defining the one level difference between face and edge octants can be performed in $O(n\log n)$ time where n is the number of original octants using the approach given in reference [16]. The process of creating the tetrahedral elements to fill the portion of the boundary octants interior to the domain uses exactly the same procedures used to mesh the boundary octants in the parallel mesh generator described given in reference [6].

The performance of h-refinement of interior octants is supported by the straightforward application of the recursive refinement procedure used to construct the original octree. This must be followed by an updating of the one-level difference criteria. Since the tree at that start of the refinement step already satisfies the one-level difference requirement, the face neighbor pointers can be used to examine the neighbors to see if they need refinement in $O(1)$ time [13], [14].

Regaining the one-level difference can take from $O(1)$ time if propagation is limited, and in the worst case is to $O(n)$ if each and every octant needs to be refined [13]. When a boundary octant is refined the new boundary octants can be meshed using local remeshing procedures similar to those given in reference [3].

The sections, which follow describing the construction of the open cover and the integration cells, will assume that an octree, constructed as described, is available on which the partition of unity discretization can be built.

3. Constructing an open cover of Ω based on the octree structure

Overlapping patches Ω_i comprise the open cover $\{\Omega_i\}$ of Ω. The patches serve as the building blocks to form the partition of unity. For the partition of unity to be valid the patches can not be completely arbitrary. They have to fulfill certain requirements. First, it has to be guaranteed that each point in the domain of interest is covered by at least N_{min} patches, where N_{min} depends on the degree of the partition of unity. Second, the simplex formed by the center of the N_{min} patches must not degenerate. We refer to the Appendix for a detailed discussion of this issue. It is obvious that a method based on an arbitrary set of scattered nodes has to perform expensive computations for each integration point to guarantee the validity of the discretization and ultimately the reliable solution based on the given set of nodes. Duarte and Oden [8] describe an algorithm of order O(NlogN) that checks the requirements for the partition of unity discretization based on moving least square methods of order 0 and 1. The 2d-algorithm searches for the N_{min} patches by collecting a list of patches whose center falls into a bounding square around the point of interest, sorting the list with respect to distance, and checking whether the center of at least three patches are not aligned. The geometrical check for an alignment is not sufficient for higher order partition of

unity, e.g. like they appear in the element free galerkin method. More complex algorithms have to be employed. The Appendix describes in detail the requirements that have to be fulfilled for a partition of unity to be valid. As a last step the interaction between arbitrarily distributed patches has to be determined. This is again a searching problem where the best known algorithms are of order O(NlogN).

From the arguments made it becomes clear that the efficiency of partition of unity methods can be enhanced if the open cover is based on a suitable underlying structure. Suitable means that the chosen structure should support the method solving or at least reducing the implementation problems associated with the flexibility of an arbitrary scattered set of patches of the open cover, while not restricting the features of the methods gained from exactly that flexibility. One main argument for partition of unity methods is they do not necessarily need a mesh and avoid mesh generation. This argument translates into the demand for a structure that is easier and cheaper to generate than a classical Finite Element Mesh. Another important argument is the ability of partition of unity methods to support h-refinement by “throwing in” new patches wherever they are needed. Despite the fact that the new patches can not be completely arbitrary, but have to follow certain rules to guarantee the validity of the open cover, the goal should be to create a structure that supports local h-refinement. A third feature of meshless methods, the ability to easily adapt the local solution spaces to features of the actual solution, is not affected by the underlying structure as long as those features (e.g. singularities) can be spatially resolved, which brings us back to the need for a support of local h-refinement.

An open cover based on an octree can provide the structure needed to simplify the algorithmic problems discussed above. The validity of the open cover based on an octree can be guaranteed a

priori, i.e. no validity checks are necessary during runtime. We refer to the Appendix for a detailed discussion. The generation of an octree is more efficient than a finite element mesh, and h-refinement is easily possible since a valid adjusted octree has to follow far less rules than a finite element mesh. Furthermore, the octree has very good localization capabilities allowing refinement of the discretization in areas of singularities if necessary. Last but not least, the memory consumption is about four times smaller compared to a finite element structure [4] since the high structure of the octree allows it to calculate needed information fast rather than storing it, e.g. the coordinates of the corners of an octant can be computed from the coordinate of the center and the size of the octant.

For methods constructing the partition of unity based on the moving least square scheme the discretization represented by an octree cannot directly be an open cover $\{\Omega_i\}$ of Ω since the spaces represented by the cells of an octree are closed and do not overlap. However, there are two simple possibilities to create a valid open cover based on the given octree structure distinguished by whether the patches are centered around the center or around the corners of the octants. We introduce the following two definitions describing the creation of the open cover.

Definition 1: Let $O_i \in O_{oc}$. A member Ω_i of the cover $\{\Omega_i\}$ will be a cube of size $\alpha \cdot size(O_i)$ centered around the center of O_i.

Definition 2: Let $O_i \in O_{oc}$. A member Ω_i of the cover $\{\Omega_i\}$ will be a cube of size $\alpha \cdot \min(size(VN_j(O_i)))$ centered around corner j of O_i.

The value α has to be chosen such that an admissible open cover is created. Whether an open cover is admissible depends on the actual mesh free method that is used. E.g., for the element free Galerkin Method (Belytschko et al. [5]) or for the Moving Least Square Reproducing Kernel Galerkin Method (Liu et al. [10]), the value of α depends on the polynomial degree of the shape functions. The hp-clouds method (Duarte, Oden [8]) can be constructed based on a fixed degree for the partition of unity, while the approximation quality is increased by local tensor product spaces. This allows picking an admissible value for α once without the need to readjust if the polynomial degree of the shape functions is increased. The Appendix shows how the value for α has to be selected for different degrees of the partition of unity. It shows also that another criteria for an admissible open cover, the non-degeneration of the open cover, is automatically fulfilled for an open cover based on the definitions 1 or 2, and does not need to be checked during runtime.

See Fig. 3 and Fig. 4 for a 2 dimensional example of the open cover $\{\Omega_i\}$ created based on the given definitions. Note that the octants defining the open cover do not have to be terminal octants, any set O_{OC} can be used. This will allow the definition of the integration cells based also on the octree, but independent of the open cover $\{\Omega_i\}$. A low order integration with more integration cells can be performed as well as a higher integration order with fewer integration cells. Depending on the actual partition of unity, one method might be more cost effective than the other.

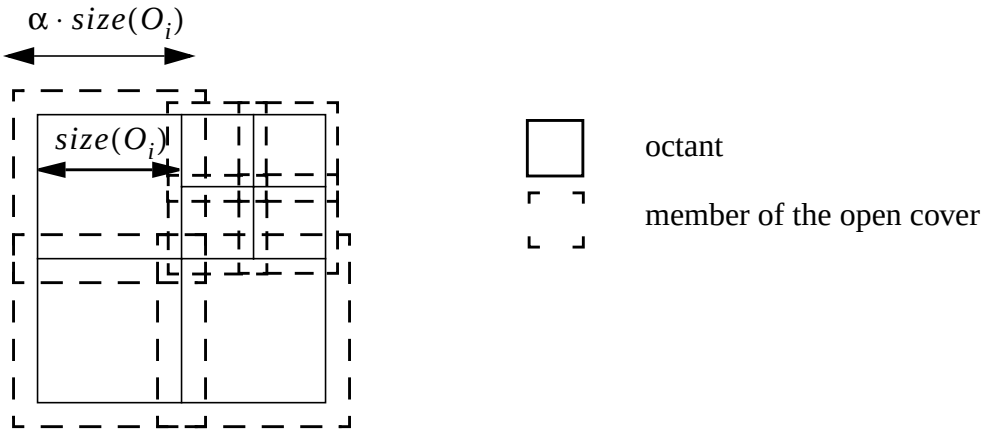

Fig. 3: The definition of the open cover $\{\Omega_i\}$ based on Definition 1.

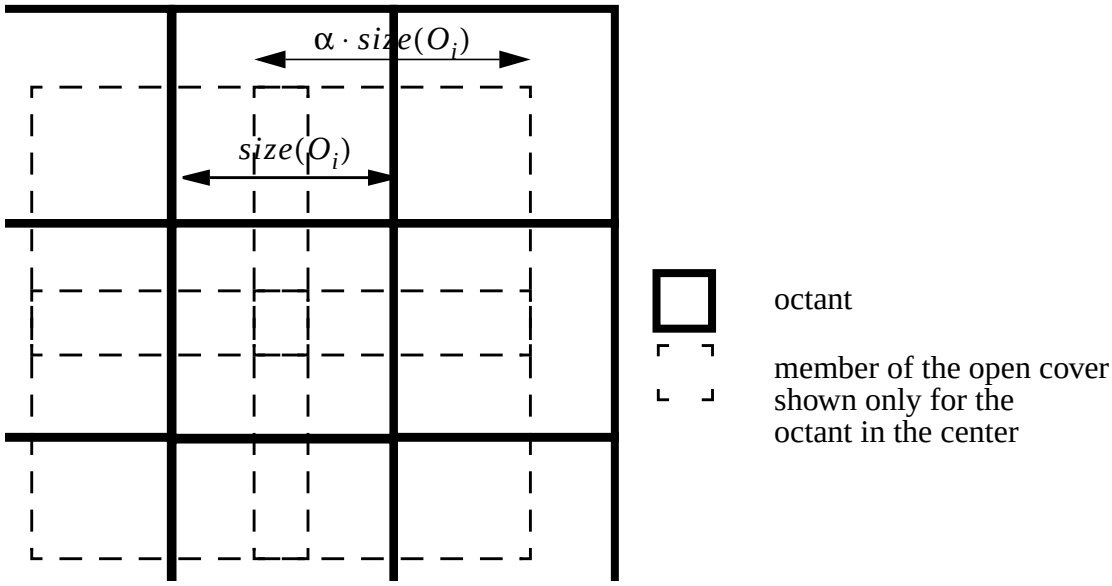

Fig. 4: The definition of the open cover $\{\Omega_i\}$ based on Definition 2.

Remark:

- The concept of creating the open cover will be the same for interior and boundary octants. That might lead to the fact that certain patches are located partly outside of the domain Ω (see Fig. 5). That does no harm as long as a reasonably large part of the associated patch intersects with the domain Ω to avoid nearly singular system matrices. We have found that the needed minimum value for α to construct a valid partition of unity creates an overlapping of the

patches with the domain that is large enough.

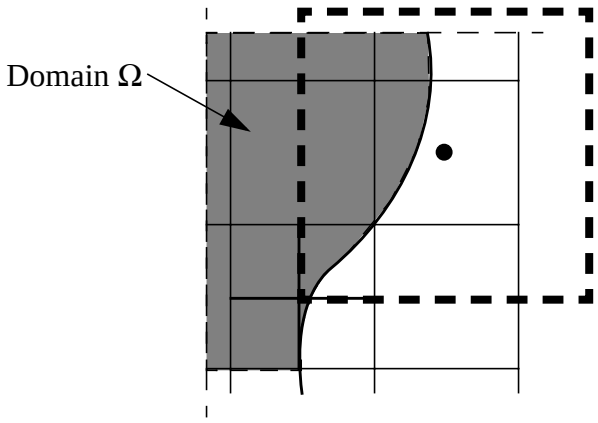

Fig. 5: Patch Ω_i located partly outside of the domain Ω.

4. Constructing integration cells based on the octree structure

For the numerical integration of the governing equations, numerical quadrature formulas have to be employed. The most efficient methods, e.g. the Gauss quadrature formula, are based on an integration over a unit domain for which the weights and integration points are tabulated. To make use of those methods a discretization consisting of simply shaped topological objects is of advantage since it facilitates the definition of the necessary mapping function. Since the spaces represented by the terminal octants are closed and do not overlap we can use interior terminal octants as integration cells for the numerical integration scheme. The cubical shape of an octant provides an easy way to map the integration domain onto a unit cube where standard integration rules are applied.

For arbitrary domains we face the problem that boundary octants are cut arbitrarily by the boundary of the problem domain (see Fig. 2). The domain of integration for that cell is consequently only the portion of the terminal cell that is interior to Ω. To perform the integration in terminal

cells that intersect the boundary we break those cells up into simplices (tetrahedrons in 3-D or triangles in 2D) over which the integration can be done with sufficient accuracy.

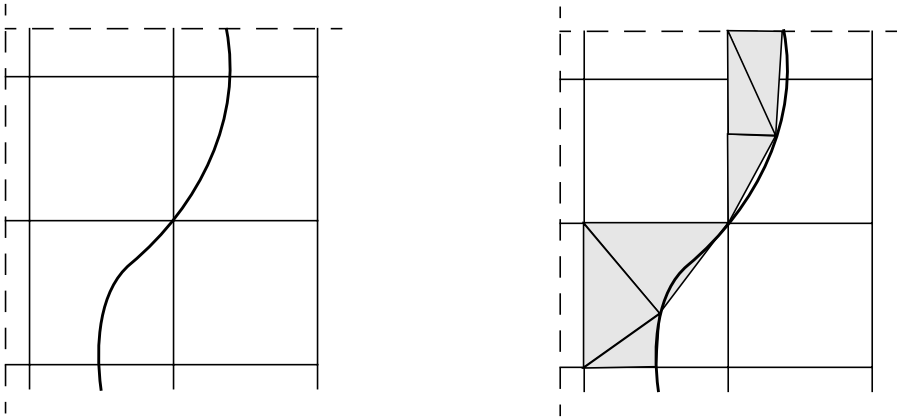

Fig. 6: Triangulations of boundary octants.

Fig. 6 gives an example for resolving the boundary octants into simplices. Note that it is not necessary that the closure of the simplices coincide with the boundary of two neighbored boundary octants. Any convenient subdivision of the boundary octants into simplices can be used as long as the simplices cover the space of the geometric domain Ω without gaps in between them.

The creation of the open cover and the integration cells as described eliminates any global searching for members of the open cover during integration. With the knowledge of the value α and utilizing direct face neighbor links [14] all patches covering a point $x \in \Omega$ can be found in O(1). Since the integration point procedure should be as fast as possible a list is set up for each integration cell that contains all patches covering a point $x \in \Omega$. This allows it to loop over the list at the time of integration without searching at all. The list is set up using the algorithm given in Fig. 7, where we make use of the operator *covered*(Ω_i, O_j) that returns -1 if the member of the open cover Ω_i does not cover the octant O_j at all, 0 if it covers the octant O_j at least partly. We assume that the octree was already created, and that the terminal octants serve as integration cells. Fur-

thermore, we assume that the set of octants O_{OC} was chosen as described in section 2., and that a valid open cover $\{\Omega_i\}$ was built based on O_{OC}.

1 for each octant $O_i \in O_{OC}$ do
2 get the set of neighbored elements $O_n = N(O_i) \cup \{O_i\}$
3 for each $O_j \in O_n$ do
4 done = FALSE
5 get the set of integration cells:
$O_{IC} = \{((Interior(O_j) \cup Boundary(O_j)))\} \cap Term(O_j)$
6 for each $O_{ic} \in O_{IC}$ do
7 if (not*covered*(Ω_i, O_{ic}))
8 add O_j to the list of octants attached to O_{ic}
9 else done = TRUE
10 end
11 if (not done) then $O_n = O_n \cup N(O_j)$
12 end
13 end

Fig. 7: Algorithm to find the patches contributing to the integration cells.

Note that the algorithm given in Fig. 7 does not consider tetrahedral elements. The tetrahedral elements are used as integration cells, but for each integration point the terminal octant containing that point is computed. The information about patches covering this integration point is then available since the algorithm (see line 5 in Fig. 7) considers terminal boundary octants, and those contain the tetrahedral elements. We would like to point out that the terminal octant containing a specific integration point is easily computable since the relation between tetrahedral elements and boundary octants is available during the generation of the tetrahedral boundary mesh and can be stored.

It can be seen that the algorithm depicted in Fig. 7 is of order O(N) where N is the number of members of the open cover $\{\Omega_i\}$. The steps 2 to 12 are local operations on an octant collecting

neighbor information using direct neighbor links. Those operations are performed in constant time assuming an adjusted octree [14].

Fig. 8 shows that for certain choices of the value α, depending on the level difference between the octants defining the open cover and the integration cells, the boundary of the patches Ω_i coincides with the boundary of the octants on level n in the interior of the domain. This avoids the need to check for each integration point if the contribution of a member of the open cover that is stored in the list of potentially contributing shape functions is non zero. Fig. 8 shows the situation for an open cover constructed from octants on level n-1, and integration cells constructed from the terminal octants (=level n). α is set to 3.0 for this example. It can be seen that the boundary of the patches of the open cover (defined on octants on level n-1) coincides with the boundary of the integration cells, which are the terminal octants on level n. This creates a situation comparable to the finite element method where all integration cells inside an element get contributions from the same shape functions, and allows the computation of an “integration cell stiffness matrix” which can then be assembled into the global system of equations yielding a faster computation of the global stiffness matrix. As explained in section 4. tetrahedral elements cross the boundary of octants, and therefore the described simplification does not apply for tetrahedral integration cells.

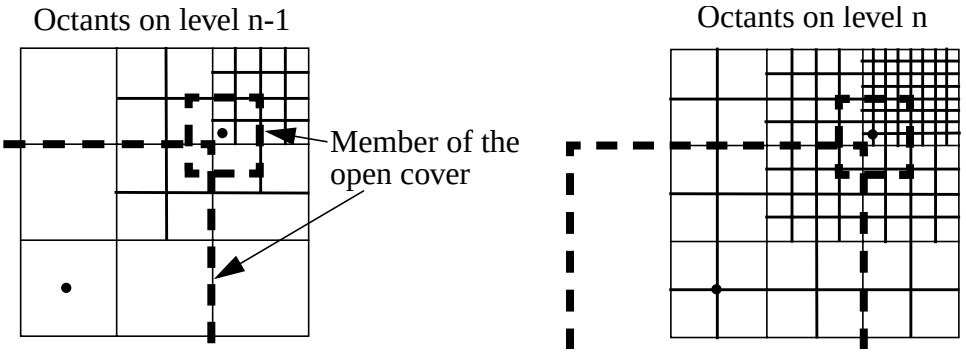

Fig. 8: Interaction between patches and integration cells.

5. Examples

The focus of this work was on the development of the octree-based discretization technology that can be used with the various partition of unity methods (like hp-clouds, Element Free Galerking Methods etc.). Simple numerical examples have been done to be sure that the structures will work, and we are currently working on implementing a partition of unity method that takes advantage of the discretization not only for the purpose of integration and identifying contributing shape functions, but also as a means to construct the shape functions themselves.

In the following we present two examples that show the discretization, i.e. the integration cells and the open cover created by the procedures worked out in this paper. As the first example we present a screwdriver. Fig. 9 shows the geometric model. In Fig. 10 the integration cells are depicted. It can be clearly seen that the interior integration cells are octants while all integration cells at the boundary are made of tetrahedral elements to capture the curvature of the boundary. Fig. 11 shows the overlapping patches that make up the open cover.

Fig. 9:Screwdriver: Geometric model.

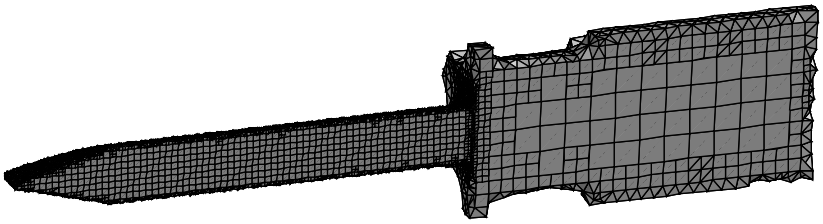

Fig. 10:Screwdriver (cut through): Integration cells.

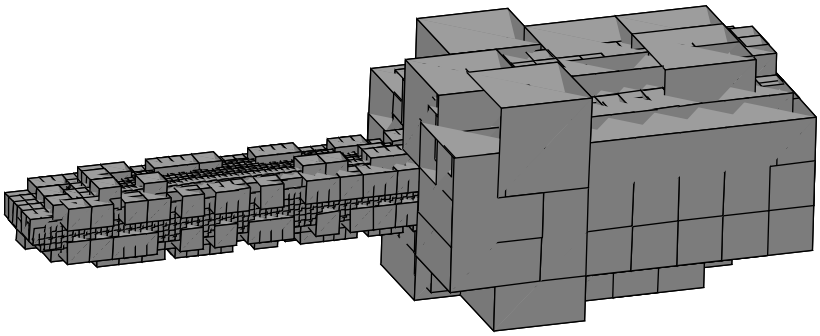

Fig. 11:Screwdriver: Overlapping patches.

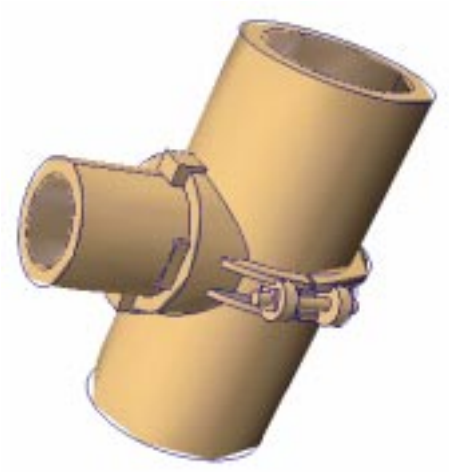

Fig. 12: Intersection of two pipes: Geometric model.

Fig. 13 and Fig. 14 show an example of the discretization of the intersection of two pipes for the model given in Fig. 12. Fig. 13 shows the integration cells for half the model, i.e. octants in the interior of the model and a tetrahedral mesh on the boundary to describe the curved boundary. The patches Ω_i defining the open cover $\{\Omega_i\}$ are given in Fig. 14. Fig. 14 a) gives a 3D view of the overlapping patches while Fig. 14 b) shows a projection of the octant boundaries to illustrate the overlapping more clearly.

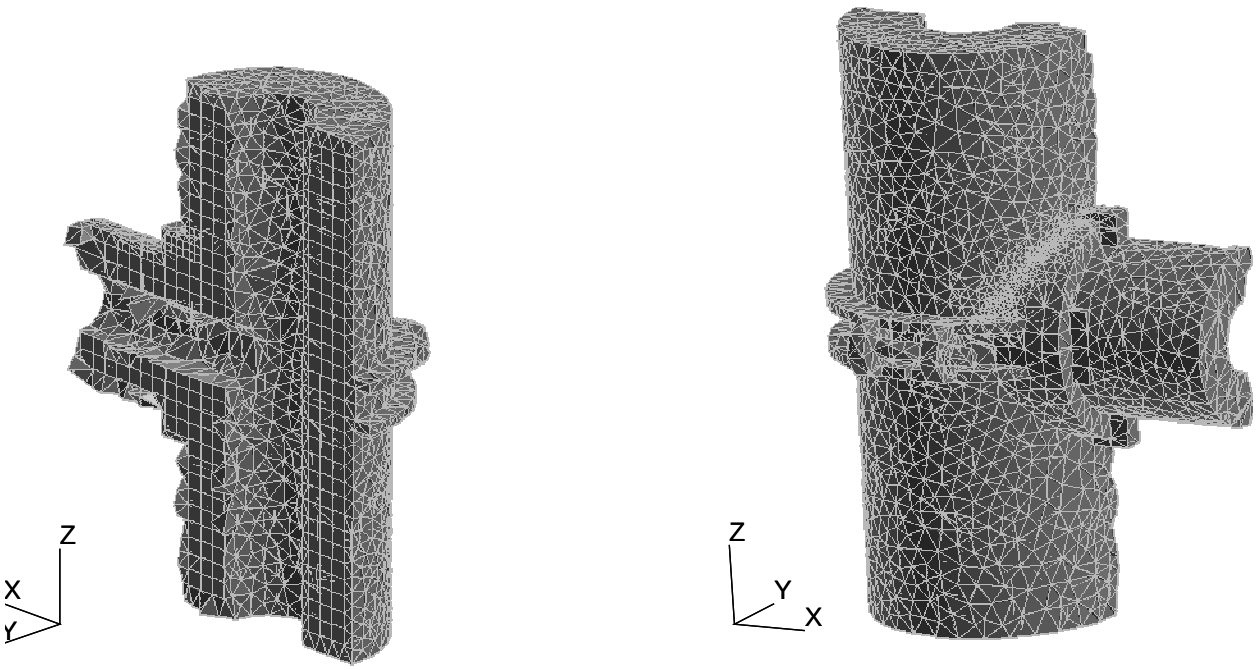

Fig. 13: Two pipes: Integration cells.

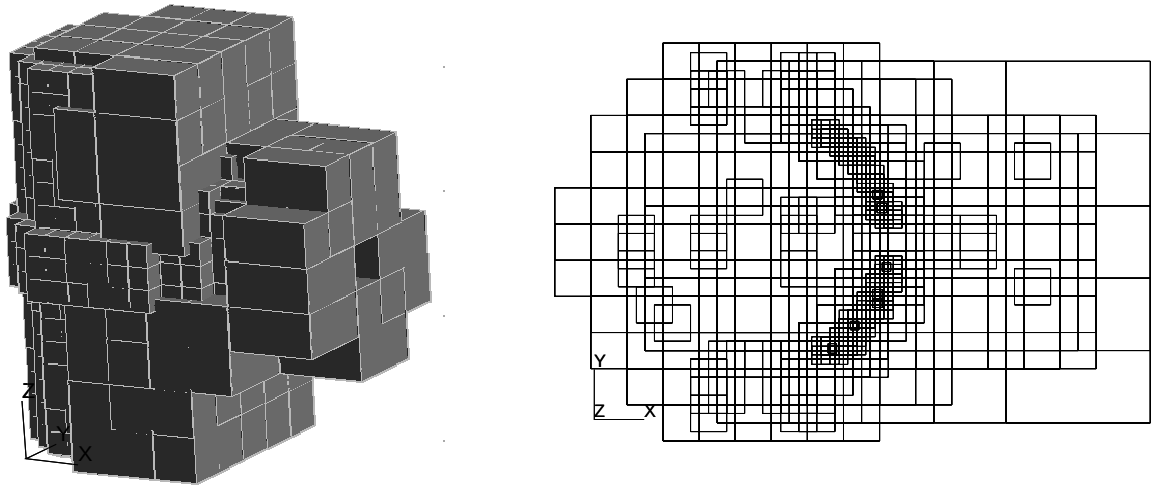

Fig. 14: Two pipes: Overlapping patches.

6. Conclusion

The automatic generation of three-dimensional discretizations for partition of unity methods was presented. The discretization is based on an octree structure. The octants are used to create the open cover as well as the integration cells. The center or the corners of the octants are used as the centers for the patches comprising the open cover. The structure of the octree allows the a priori determination of the size of the patches needed to form a valid open cover, which avoids expensive runtime validity checks. Furthermore, the search for patches contributing to an integration point reduces to a local operation of O(1) if face neighbor links are used. In order to capture the curvature of the boundary, the boundary octants are subdivided into tetrahedral elements serving as integration cells besides the interior octants. Two examples show the applicability of the presented methods to generate partition of unity discretizations for complex geometries.

7. Acknowledgment

The authors would like to gratefully acknowledge the support of the National Science Foundation under grant ASC-9704969.

8. References

[1] Babuska, I; Caloz, G; Osborn, JE (1994): Special finite element methods for a class of second order elliptic problems with rough coefficients. SIAM J. Numerical Analysis, 31(4), 745 - 981.

[2] Babuska, I; Melenk, JM (1997): The Partition of Unity Method. Int. J. Numer. Meth. Eng. 40, 727-758.

[3] Baehmann, PL; Shephard, MS (1989): Adaptive multiple level h-refinement in automated finite element analyses, Engng. with Computers 5(3/4),235-247.

[4] Beall, MW; Shephard, MS (1997): A General Topology-Based Mesh Data Structure. Int. J. Numer. Meth. Eng. 40, 1573 - 1596.

[5] Belytschko, T; Lu, YY; Gu, L (1994): Element-Free Galerkin Methods. Int. J. Numer. Methods Eng. 37, 229-256.

[6] deCougny, HL; Shephard, MS (1999): Parallel volume meshing using face removals and hierarchical repartitioning. Comput. Methods Appl. Mech. Eng. 174(3-4),275-298.

[7] Duarte, CA; Babuska, I; Oden, JT (1998): Generalized Finite Element Methods for Three Dimensional Structural Mechanics Problems. Proceedings of the International Conference on Computational Engineering Science, Atlanta, October 1998.

[8] Duarte, CA.; Oden, JT (1995): Hp Clouds - A Meshless Method to Solve Boundary-Value Problems. TICAM Report 95-05, Texas Institute for Computational and Applied Mathematics. The University of Texas at Austin.

[9] Lancaster, P; Salkauskas, K (1981): Surfaces Generated by Moving Least Squares Methods. Math. of Comp. Vol 37, Number 155, 141 - 158.

[10] Liu, WK; Li, S; Belytschko, T (1995): Moving Least Square Reproducing Kernel Methods. (I) Methodology and Convergence. Technical Report No. Tech-ME-95-3-XX, Department of Mechanical Engineering, Northwestern University.

[11] Lu, YY; Belytschko, T; Gu, L (1994): A new implementation of the element free Galerkin method. Comput. Methods Appl. Mech. Eng. 113, 397 - 414.

[12] Samet, H (1990): Applications of Spacial Data Structures. Addison-Wesley Publishing Company.

[13] Simone, ML (1988): A Distributed Octree Structure and Algorithms for Parallel Mesh Generation. Ph.D., Thesis, Scientific Computation Research Center, Report #13-98, Rensselaer Polytechnic Institute, Troy, NY.

[14] Simone, ML; Loy, RM; Shephard, MS; Flaherty, JE (1998): A Distributed Octree Structure and Algorithms for Parallel Mesh Generation. SCOREC Report #23 1996, Scientific Computation Reserach Center, Rensselaer Polytechnic Institute, Troy, NY 12180. Submitted to: J. of Parallel and Distributed Computing.

[15] Shephard, MS; de Cougny, HL; O'Bara, RM; Beall, MW (1999): Automatic Grid Generation Using Spatially-Based Trees. CRC Handbook of Grid Generation. Boca Raton, 1501 - 1522.
[16] Yerry, MA; Shephard, MS (1984): Automatic Three-Dimensional Mesh Generation by the Modified-Octree Technique. Int. J. Numer. Meth. Eng. 20, 1965 - 1990.

9. Appendix A

The partition of unity can be constructed by determining the best approximation $g(x) \in G_n$ of a function $f(x) \in V$, where V is a Hilbert space with scalar product $(\bullet, \bullet)$ and the corresponding norm $\|\bullet\| = \sqrt{(\bullet, \bullet)}$, and $G_n \subseteq V$ is a *n*-dimensional space with the same norm.

Now let $\{h_1, h_2, \ldots, h_n\}$ be a basis of G_n. Introducing the notation $\mathrm{u} = (u(x_1), \ldots, u(x_N))^T$ for a vector containing data at the *N* data points a scalar product $(u, v)_{\hat{x}} = \mathrm{u}^T \mathrm{W}(\hat{x})\mathrm{v}$ can be defined. $\mathrm{W}(\hat{x}) = \mathrm{diag}(w^1(\hat{x}), \ldots, w^N(\hat{x}))$ is a square *N*x*N* matrix with positive diagonal elements. Following standard arguments we calculate the best approximation $g(x)$

$$g(x) = \sum_{i=1}^{n} a_i(\hat{x}) h_i(x) \tag{8}$$

with

$$\sum_{i=1}^{n} a_i(\hat{x})(h_i, h_j)_{\hat{x}} = (f, h_j)_{\hat{x}},\ j = 1, 2, \ldots, n \tag{9}$$

If $\mathrm{W}(\hat{x})$ is a constant matrix, (8) is a classical, weighted least squares approximation. Otherwise a new set of constants $a_i(\hat{x})$ has to be computed for each new point where the value of the approximation is needed.

By introducing the matrix A with its elements $A_{ij} = (h_i, h_j)_{\hat{x}}$ we find the partition of unity by plugging the solution for the coefficients $a_i(\hat{x})$

$$a_i(\hat{x}) = \sum_{j=1}^{n} A_{ij}^{-1}(f, h_j)_{\hat{x}} \tag{10}$$

into (8)

$$\begin{aligned} g(x) = \sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}^{-1}(f, h_j)_{\hat{x}} h_i(x) &= \sum_{\alpha=1}^{N}\sum_{i=1}^{n}\sum_{j=1}^{n} h_i(x) A_{ij}^{-1} w^{\alpha}(\hat{x}) h_j(\hat{x}) f_{\alpha} \\ &= \sum_{\alpha=1}^{N} \varphi_{\alpha}(x) f_{\alpha} \end{aligned} \tag{11}$$

The partition of unity as it is developed here is used by Belytschko et. al. [5] in their Element-Free Galerkin Method (EFGM). They use the functions φ_i directly as the shape functions to discretize the variational form. The polynomial degree of the shape functions is increased by increasing the number of basis functions h_i used to calculate the partition of unity. At this point we would like to point out that a necessary condition for the matrix A to be invertible is

$$\forall x \in \Omega \quad \text{card}\{i | x \in \Omega_i\} > m \tag{12}$$

if m is the degree of the partition of unity. This basically compels us to increase the size of the patches for a p-extension if a fixed number of patches is given. To be able to compute the partition of unity from the moving least square method the Matrix $A_{ij} = (h_i, h_j)_{\hat{x}}$ has to be inverted. Therefore it is necessary that the matrix is not singular. We will point out some implications on implementation issues that arise from that condition.

Let us write the matrix $A_{ij} = (h_i, h_j)_{\hat{x}}$ explicitly for n basis functions and N points:

$$\begin{bmatrix} h_1(x_1) & h_1(x_2) & \dots & h_1(x_N) \\ h_2(x_1) & h_2(x_2) & \dots & h_2(x_N) \\ \dots & \dots & \dots & \dots \\ h_n(x_1) & h_n(x_2) & \dots & h_n(x_N) \end{bmatrix} \cdot \begin{bmatrix} w^1(\hat{x}) & 0 & \dots & 0 \\ 0 & w^2(\hat{x}) & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & w^N(\hat{x}) \end{bmatrix} \cdot \begin{bmatrix} h_1(x_1) & h_2(x_1) & \dots & h_n(x_1) \\ h_1(x_2) & h_2(x_2) & \dots & h_n(x_2) \\ \dots & \dots & \dots & \dots \\ h_1(x_N) & h_2(x_N) & \dots & h_n(x_N) \end{bmatrix}$$

$$= FWF^T = A \tag{13}$$

A necessary condition for the product FWF^T not to be singular is that $N \geq n$. Let us investigate the most critical case $N = n$. With the theorem (Sylvester Law of Inertia),

if $A \in \Re^{n \times n}$ is symmetric and $X \in \Re^{n \times n}$ is nonsingular, then A and $X^T AX$ have the same inertia, i.e. they have the same number of negative, zero and positive eigenvalues.

We conclude that A is not singular if $F \in \Re^{n \times n}$ is not singular, i.e. $\det F \neq 0$.

We will now discuss how the regular structure of the center of the patches defined by the octree guarantees a priori the non singularity of the product FWF^T. To simplify the discussion we will look at the 2-dimensional case; an extension to three dimensions is straight forward.

For a constant basis {1} FWF^T is positive based on the assumption that $W(x)$ is positive definite. For a linear basis {1,x,y}, i.e. n = 3 any point in the domain will be covered by at least 4 patches if α is chosen appropriately for each patch. α has to be at least 2; it has to be 3 for patches that are defined on an octant O_i if $\{\exists O_j \in N(O_i) \big| (Level(O_j) < Level(O_i))\}$. The necessary condition $N \geq n$ is then fulfilled. Since the center of the 4 patches are given as the corners of a square there are always 3 points defining a 2-dimensional simplex, i.e. they are not aligned yielding a positive definite matrix FWF^T. For a quadratic basis {1,x,y,x*x,x*y,y*y} α has to be chosen as 3 or 5, respectively. This will guarantee the coverage of any point in the domain by at least 9 patches. This fulfills the necessary condition $N \geq n$ with n =6 and N = 9. The matrix F would be singular if for any subset of 6 points coefficients $\lambda_1, \ldots, \lambda_6$ can be found, not all of them equal to zero with

$$\lambda_1 + \lambda_2 x_i + \lambda_3 y_i + \lambda_4 x_i^2 + \lambda_5 x_i y_i + \lambda_6 y_i^2 = 0 \qquad \forall i = 1 \ldots 6 \tag{14}$$

Points (x_i, y_i) that would do so have to lie on a conic section (including a straight line if the cone is degenerated into a cylinder). Obviously, the points given by the octree are representing a 9 point stencil pattern, and do not lie on a conic section.

Similar ideas can be employed for a larger basis or a basis in three dimensions to show that an α can always be chosen that guarantees a non-singular matrix A a priori. Costly computations during runtime to perform the check numerically are not necessary.