INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2008; **57**:1753–1770
Published online 5 December 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/fld.1685

CartGen: Robust, efficient and easy to implement uniform/octree/embedded boundary Cartesian grid generator

Rohallah Tavakoli*, †

Department of Material Science and Engineering, Sharif University of Technology, Tehran, Iran

SUMMARY

An efficient and easy to implement method to generate Cartesian grids is presented. The presented method generates various kinds of Cartesian grids such as uniform, octree and embedded boundary grids. It supports the variation of grid size along each spatial direction as well as anisotropic and non-graded refinements. The efficiency and ease of implementation are the main benefits of the presented method in contrast to the alternative methods. Regarding octree grid generation, applying a simple and efficient data compression method permits to store all grid levels without considerable memory overhead. The presented method generates octree grids up to a 13-level refinement (8192^3 grids on the finest level) from a complicated geometry in a few minutes on the traditional desktop computers. The FORTRAN 90 implementation of the presented method is freely available under the terms of the GNU general public license. Copyright © 2007 John Wiley & Sons, Ltd.

Received 15 July 2006; Revised 7 October 2007; Accepted 14 October 2007

KEY WORDS: Cartesian grid generation; embedded boundary; hierarchical grid; immersed boundary; octree generation; STL file; voxelization

1. INTRODUCTION

Several numerical methods in the computational mechanics such as finite element method, finite volume method (FVM) and finite difference method need to subdivide the physical domain into sub-domains, which is called the spatial computational grid. In recent years, automatic grid generation is one of the focus areas in the research of computational mechanics and computational geometry [1].

*Correspondence to: Rohallah Tavakoli, Department of Material Science and Engineering, Sharif University of Technology, P.O. Box 11365-9466, Tehran, Iran.
†E-mail: tav@mehr.sharif.edu, rohtav@gmail.com

The numerical grid generation generally can be classified into body-fitted and non-body-fitted categories. The body-fitted methods can be divided into structured and unstructured grid generation methods. Although the structured and unstructured body-fitted methods have been successful in handling complex geometries, they are usually very time consuming to generate grids for complex geometries. Furthermore, in the case of complex geometries, these methods do not have enough robustness to generate a suitable grid, fully automatically without any human effort.

In contrast to the body-fitted structured or unstructured grid generation methods, Cartesian grids are inherently non-body-fitted; i.e. the volume mesh structure is independent of the surface discretization and topology. This characteristic greatly simplifies the grid generation procedure and promotes extensive automation. The Cartesian grids can be divided into three categories: uniform, octree (quadtree in 2D) and embedded boundary grids.

In the uniform Cartesian grid, the curved boundaries are approximated in a staircase manner. This limitation might decrease the accuracy or efficiency (since it enforces the use of a fine grid) of the numerical solution particularly for complex geometries. In spite of this weakness, in some engineering applications, staircase boundary approximation is reasonable and sufficient. For example, in the field of computer-aided thermal or structural topology and shape optimization, several researchers have used the uniform Cartesian grid to discretize the spatial domain [2–10]. Also in some applications the body-fitted and uniform Cartesian grids are applied simultaneously to balance between the accuracy and efficiency of simulation [11–15].

Recently, the embedded boundary Cartesian approach is proposed to overcome the mentioned limitation of the uniform Cartesian grid [16–22]. In this approach, the physical domain is embedded completely within a larger Cartesian grid. The bulk of data underlying an embedded boundary discretization utilizes rectangular indexing and regular treatment, and only a small number of cells near the embedded boundary require special treatment.

The tree-based Cartesian grid, i.e. octree (or quadtree in 2D) [23–27] is another treatment to fade the mentioned limitation of the uniform Cartesian grid. In this approach, the grid resolution is increased near the high-curvature boundaries. This method is usually combined with the embedded boundary methods to compete with the body-fitted methods. Note that in this approach, there is no regular data structure similar to the uniform Cartesian grid, but there are semi-structured data that are preferable in contrast to the fully unstructured data. In [28] a robust and efficient method has been proposed to generate octree Cartesian grids over complex geometries. This method uses sophisticated computational geometry techniques which makes it very difficult to implement (furthermore, needs deep background in the field of computational geometry).

In the present study, a robust, efficient and easy to implement Cartesian grid generator is proposed. The presented method generates the uniform grid, octree grid and essential information required for embedded boundary finite volume solvers. It supports the variation of grid size along each spatial direction for either uniform or octree-refined grids. The generation of irregularly refined (non-graded) grids can be included easily in the presented method. In contrast to the alternative methods, the ease of implementation is the outstanding feature of the presented method. This feature greatly simplifies the procedure of further extension and tuning for particular applications.

The rest of this paper is organized as follows. The required input geometry that is used in the present study is discussed in Section 2. Section 3 presents the grid generation algorithm. Section 4 is devoted to numerical experiments on the robustness and efficiency of the presented method and Section 5 summarizes the current study.

DOI: 10.1002/fld

2. INTERFACE WITH CAD

Many CAD packages use boundary representation (BREP) geometries to define solids. With BREP, the boundary surface is composed of multiple boundary patches, which can be planar polygons or non-uniform rational B-spline patches. The BREP geometry is said to be 'water tight' if each boundary curve (of a patch) is shared, and shared only by two patches. To generate a computational grid with almost any current grid generators, a 'water-tight' (topologically closed) geometry has to be defined first. Sometimes, 'dirty geometries' are caused by cracks, overlaps or invalid manifolds in the geometry [29]. There are several methods that can be used to repair dirty geometries before grid generation (e.g. see [30–38]). In the present study it is assumed that the input geometry is water tight.

Most CAD software on the market can generate stereolithography (STL) files, which are generally used for prototyping, visualization and grid generation purposes. These files represent a triangulation boundary of the solid. Algorithms to generate STL triangulation are highly efficient and surface approximations are very precise [39]. The transfer of STL files from one system to another is exact [29]. An STL file is composed of a list of triangle facets. These facets are composed of the coordinates of three vertices of the triangle, in addition to the coordinates of the normal oriented to the exterior of the solid. This triangulation is built to minimize a geometric approximation criterion that is related to the real boundary of the solid. CAD softwares usually export the STL file in the binary or ASCII format. Figure 1 shows the content of a sample ASCII STL file.

The CARTGEN code reads input geometry in STL format (either ASCII or binary). Note that other surface triangulation formats can be used in the presented method. The input geometry

```
solid AutoCAD
   facet normal 0.0000000e+000 0.0000000e+000 1.0000000e+000
      outer loop
         vertex 1.0000000e+002 1.0000000e+002 1.0000000e+002
         vertex 0.0000000e+000 1.0000000e+002 1.0000000e+002
         vertex 1.0000000e+002 0.0000000e+000 1.0000000e+002
      endloop
   endfacet
   facet normal 0.0000000e+000 0.0000000e+000 1.0000000e+000
      outer loop
         vertex 0.0000000e+000 1.0000000e+002 1.0000000e+002
         vertex 0.0000000e+000 0.0000000e+000 1.0000000e+002
         vertex 1.0000000e+002 0.0000000e+000 1.0000000e+002
      endloop
   endfacet

...

   facet normal 1.0000000e+000 0.0000000e+000 0.0000000e+000
      outer loop
         vertex 1.0000000e+002 0.0000000e+000 0.0000000e+000
         vertex 1.0000000e+002 1.0000000e+002 0.0000000e+000
         vertex 1.0000000e+002 1.0000000e+002 1.0000000e+002
      endloop
   endfacet
endsolid AutoCAD
```

Figure 1. Example of an ASCII STL file.

DOI: 10.1002/fld

can be composed from multiple non-intersecting objects (STLs). With some additional efforts, handling multiple objects that intersect with each other is possible in the presented method too (see Section 3.3).

3. CARTESIAN GRID GENERATION

At the beginning of this section, we introduce some definitions that are used through this paper:

- Voxel: each element of a uniform Cartesian grid is called a voxel.
- COV: center of voxel, geometrical center of voxel.
- IV: interior (internal) voxel, a voxel that is located inside the solid object.
- EV: exterior (external) voxel, a voxel that is located outside the solid object.
- BV: boundary voxel, a voxel that is intersected with the solid object's boundary.
- IBV: interior (internal) BV, a BV whose center is located on the surface or inside the solid object.
- EBV: exterior (external) BV, a BV whose center is located outside the solid object.

3.1. 3D uniform Cartesian grid generation

Consider the STL representation of a 3D solid object (CAD representation of 3D geometry). To generate a 3D Cartesian grid, at the first step, the spatial domain (the environmental box of the solid object) is determined such that the solid object does not touch its boundaries. At the next step, the spatial domain is filled with a 3D array of voxels based on user-defined spatial step sizes. The spatial step sizes are defined as δx, δy and δz along the x, y and z directions, respectively.[‡] The size of the spatial domain should be large enough such that there is at least one row of voxels outside the solid object. In the Cartesian grid approach, each voxel is determined by its i, j and k indexes in the mentioned 3D array and we have

$$i = 1, 2, \ldots, i_{\max}$$

$$j = 1, 2, \ldots, j_{\max}$$

$$k = 1, 2, \ldots, k_{\max}$$

where $i_{\max}$, $j_{\max}$ and $k_{\max}$ are maximum number of voxels in the x, y and z directions, respectively. The x, y and z coordinates of each voxel (center of voxel) are determined by

$$x = (i - 0.5)\delta x$$

$$y = (j - 0.5)\delta y$$

$$z = (k - 0.5)\delta z$$

[‡]The variation of step size along each spatial direction is also possible.

DOI: 10.1002/fld

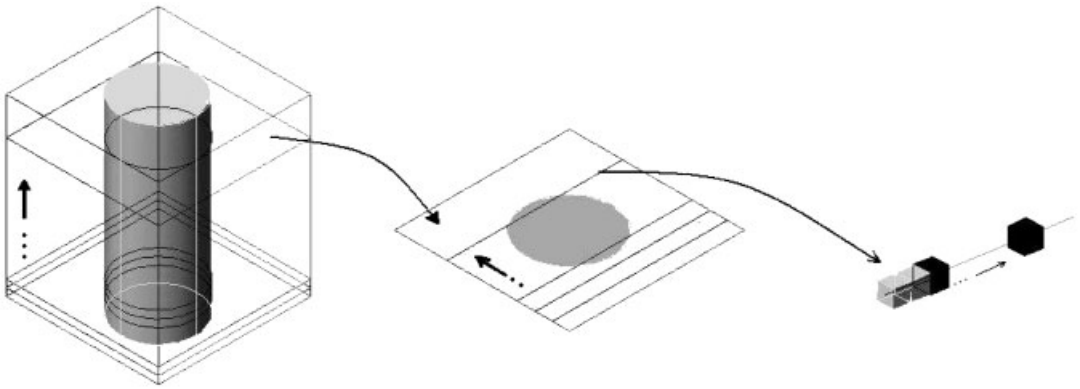

Figure 2. Schematic illustration of simplified 3D grid generation.

The task of Cartesian grid generation is voxel coloring, i.e. setting black color to IVs or IBVs and white color to EVs or EBVs. Since in the present study the type of voxel is determined based on the position of its center, we only work with the COVs (one point per voxel). This permits us to convert the 3D grid generation to a sequence of plane-by-plane 2D grid generation and similarly to convert a planar 2D grid generation to a sequence of line-by-line 1D grid generation as illustrated in Figure 2.

As illustrated in Figure 2, the voxel coloring procedure in the present study contains sweeping along the z direction and voxel coloring, plane-by-plane (the horizontal xy-plane passing from COVs). The z-coordinate of these planes are

$$z_{\min}+0.5\delta z, z_{\min}+1.5\delta z, \ldots, z_{\max}-1.5\delta z, z_{\max}-0.5\delta z$$

where $z_{\min}$ and $z_{\max}$ are the minimum and maximum bounds of z-coordinates, respectively. To color voxels on a horizontal plane, at the first step, the intersection of the solid object with the plane should be extracted. Since the solid object is a closed 3D surface (or surfaces), its intersection with a horizontal plane is a closed 2D curve (or multiple closed 2D curves), and as the solid object is represented with its surface triangulation (STL format) in this study, the result of intersection is a (or multiple) piecewise linear closed 2D curve(s). Note that for complex geometries (or multi-component geometries), we may have more than one closed curve but for convenience, without losing generality, we assume only one curve. Since a triangle intersects with a plane when it has one vertex above the plane and one vertex below the plane, to improve performance, we sort the triangle list based on the maximum z-coordinates and start the intersection procedure from the first triangle that has one vertex above the intersecting plane.

After extraction of the mentioned closed 2D curve on each horizontal plane, we have a 2D array of points (COVs) and a closed curve on the desired plane. The task of planar voxel coloring is to determine the surface and interior points in relation to the 2D closed curve and blacking their corresponding voxels. To simplify this stage, we convert the planar voxel coloring to a sequence of linear voxel coloring. For this purpose, we sweep along the y direction (on the desired plane) and intersect y-constant lines with the extracted 2D closed curve. The y-coordinates of these

DOI: 10.1002/fld

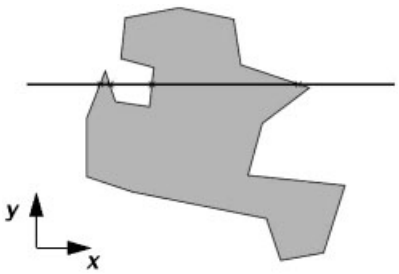

Figure 3. Intersection of a straight line with a closed 2D curve that results in an even number of points.

lines are

$$y_{\min}+0.5\delta y, y_{\min}+1.5\delta y, \ldots, y_{\max}-1.5\delta y, y_{\max}-0.5\delta y$$

where $y_{\min}$ and $y_{\max}$ are the minimum and maximum bounds of y-coordinates, respectively. Intersection of each y-constant line with a closed 2D curve results in some points having different x-coordinates (and the same y- and z-coordinates). To improve the performance of this stage, we sort the line segments of the closed 2D curve based on their y-coordinates and start the intersection procedure from the first line segment that has one vertex with a larger y-coordinate than the y-coordinate of the intersecting line.

For linear voxel coloring purpose, the resulting points (result of intersection) should be sorted based on their x-coordinates. Then the index of the corresponding voxel of each point should be determined based on the following relation (these voxels are BVs as defined previously):

$$i=\mathrm{INT}(x_p/\delta x)+1, \quad j=\mathrm{INT}(y_p/\delta y)+1, \quad k=\mathrm{INT}(z_p/\delta z)+1$$

where x_p, y_p and z_p are coordinates of each point. Note that z_p is equal to the z-coordinate of the intersecting horizontal plane, y_p is equal to the y-coordinate of the intersecting y-constant line and x_p is computed from intersection of the y-constant line with the closed 2D curve. Since these points are the result of intersecting a closed 2D curve with a straight line, the number of points is an even number (see Figure 3).

The resulting BVs are colored based on their type (black for IBVs and white for EBVs). To color the other voxels (along a prescribed straight line), we sweep along the x direction and set black color for those voxels that are located between each of the two consecutive BVs as illustrated in Figure 4.

Sometimes, we encounter overlapped points during linear voxel coloring. In these cases two or more than two points have the same position. This problem has two sources. The first source is related to a repeated line segment in the closed 2D curve. The repeated line segment occurs when a horizontal plane passes through a shared edge of two triangles (see Figure 5(a)). In this manner, the shared edge appears two times in the closed 2D curve. These repeated line segments lead to repeated points after intersection of a y-constant line with the closed 2D curve. The other source of repeated points is related to passing a y-constant line from the intersection point of two line segments (see Figure 5(b)). Removing repeated points from the sorted point list in each linear voxel coloring stage is a simple and efficient solution to this problem.

DOI: 10.1002/fld

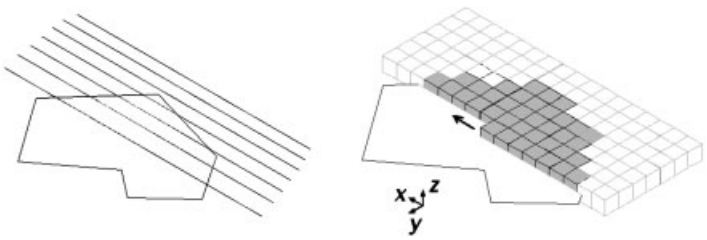

Figure 4. A schematic illustration of linear voxel coloring.

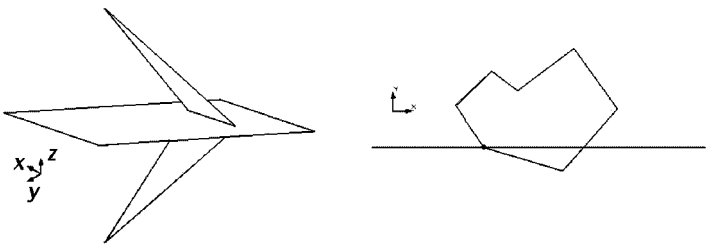

Figure 5. Production of a repeated line segment by passing a horizontal plane from the shared edge of two triangles (a) and production of a repeated point by passing an intersecting line from the intersection point of two line segments (b).

It is clear that the variation of spatial step sizes along each spatial direction can be easily included in the presented method. For this purpose, it is sufficient to replace δx, δy and δz with variable step sizes δx_i, δy_j and δz_k, respectively.

In the present study (CARTGEN code), 1 bit of memory is used to store each voxel (to save memory). In this manner, every 8 voxels are stored in an 8-bit integer (INTEGER*1 in FORTRAN). In this manner, each bit has value 1 when its corresponding voxel has black color and 0 in the other case. Therefore, every two xy-planes are stored in an array with dimension $i_{\max}/2 \times j_{\max}/2$ (for more details see the CARTGEN code).

3.2. Related computational geometry procedure

In this subsection, we describe all of the computational geometry procedures that are used in the current study.

The first computational operation is intersection of a horizontal plane with a triangle which makes a line segment. Since a horizontal triangle does not intersect with a horizontal plane, the horizontal triangles are removed from the triangles list before grid generation. A triangle is considered horizontal if the z component of its normal is smaller than *a priori* defined threshold ($1.e-10$ in the present study). The equation of the desired line segment results by putting the z-coordinate of the horizontal plane in the equation of the triangle's plane. To compute two endpoints of the line segment, we calculate the intersection of the line segment with three lines that pass from each of the two vertexes of the triangle. If one of these lines is parallel with the horizontal plane, the result of intersections would be our desired points (two endpoints). In the

DOI: 10.1002/fld

other case, the result of intersections would be three points. One of these points is not located on the triangle's edge. To determine this point, the distance of each resulting point is computed from two vertexes of the triangle (those vertexes that are used for the computation of the intersection point). If summation of these distances is equal to the triangle's edge length, then the point is located on the triangle edge.

The second computational procedure is intersection of a line segment with a y-constant line which makes a point. For this purpose, it is sufficient to put the y-coordinate of the y-constant line in the equation of the line segment. Note that before intersection the possibility of intersection should be studied.

3.3. Multi-component geometries

The algorithm described above works naturally for non-intersecting multi-component geometries. For this purpose, it is sufficient to load each component as a separate STL file and add its triangles to the end of the triangles list (the CARTGEN code supports this feature). When we have intersecting components, there are generally two solutions. A formal solution is to use the components intersection algorithm (as discussed in [28]) to generate the result of intersection as a new triangulation. The second solution, which is easy to implement in the context of the presented algorithm, is to generate grids separately for each component, and then combine the resulting grids by a Boolean rule to create the final grid. Note that all components should have the same spatial domain and step sizes. In this manner, each voxel of the final desired grid has the black color when at least one of its corresponding voxels in the separately generated grids (for each component) has the black color.

3.4. Octree grid generation

The above-described algorithm is sufficient to generate the uniform Cartesian grids. For the generation of octree-refined grids, some additional procedure should be included.

Unlike the traditional octree grid generators that use an up-to-down approach to generate octree grids, in the present study we use a down-to-up approach. In the up-to-down approach, the grid generation is started from the coarsest level and is continued by adaptive grid refinement based on the geometry curvature toward the finest level. In the down-to-up approach, the finest grid level is generated first and the coarser levels are generated directly from the finest level. The main benefits of the down-to-up approach are high efficiency, ease of implementation and flexibility to generate various kinds of refined grids (e.g. anisotropic and non-graded refinements). The main drawback of down-to-up approach is high memory usage due to storage of the all grid levels. In Section 3.6, the memory problem of the presented method is studied in detail and to tackle it, an effective solution is presented.

In the presented method, we construct coarser grid levels hierarchically from the generated finest level. For convenience, we consider the isotropic double coarsening method in which the number of voxels along each spatial direction of the next coarser level (started from the finest level) is one-half its corresponding finer level. Therefore, each of the 8 voxels of the finer level, which are called child voxels, fall within one voxel of the coarser level, which is called parent voxel.

For voxel coloring of a coarser level, we loop on its corresponding voxels and color each (parent) voxel when its corresponding 8 voxels (child voxels) in the finer level have the same color (black color when its child is black and *vice versa*). When a parent voxel is colored in the coarser level, its corresponding child voxels are removed from the finer level. The other coarser levels are

DOI: 10.1002/fld

generated hierarchically in the same manner. Finally, the desired octree data structure is extracted from these hierarchical grid levels.

As discussed previously, we use a 1-byte (8 bits) integer to store each of the 8 voxels. This issue helps to increase efficiency of the grid coarsening procedure. When all 8 bits of a 1-byte signed integer have value 1, the value of this integer should be -1 (note that for an unsigned integer this value is 255; in our implementation, i.e. CARTGEN, a signed integer is used) and in the other case when all 8 bits have a value of zero, the value of this integer should be zero. Therefore, for voxel coloring of a coarser level from its corresponding finer level, it is sufficient to check each 1-byte integer (corresponding to every 8 voxels of the finer level) for values 0 and -1 (for more detail see the CARTGEN code). The generated octree from this method is not essentially graded (in a graded octree, every two neighbor voxels could have at maximum one octree level difference). Since some of the finite volume octree Cartesian solvers only use the graded octrees, a post-processing procedure should be performed on the resulted octree to ensure this criterion.

Using the down-to-up grid coarsening approach in the presented method, all of the grid levels are directly accessible; hence, other coarsening strategies such as anisotropic coarsening or non-graded coarsening can easily be included.

3.5. Geometric requirements of embedded boundary Cartesian finite volume solvers

In this section we present a simple and easy to implement method to extract geometric information required for embedded boundary Cartesian grid finite volume solvers.

Implementation of embedded boundary cell-centered FVMs are usually performed based on the volume-of-fluid approach (in this study each voxel represents a control volume). In this approach, we need to know the occupied volume fraction and the wetted surface fraction of boundary voxels. The occupied volume fraction of each boundary voxel is defined as the ratio of the volume occupied by the solid object to the total volume of the voxel. The wetted surface fraction for each face of a boundary voxel is defined as the ratio of area of the face wetted by the solid object to the total area of face [20, 21, 26].

To extract this information, we use the efficiency feature of the presented method and after a primary grid generation, we generate a secondary grid with smaller grid sizes δx_2, δy_2 and δz_2,

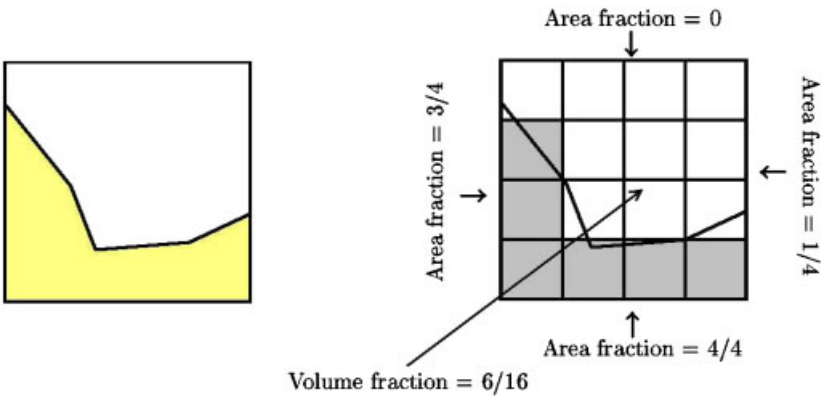

Figure 6. Division of an interfacial voxel into 16 baby voxels to extract the embedded boundary information required for 2D finite volume solvers.

DOI: 10.1002/fld

where $\delta x_2 = \delta x/2^n$, $\delta y_2 = \delta y/2^n$, $\delta z_2 = \delta z/2^n$ and n is a positive integer that controls the accuracy of method. In this manner, each primary voxel is divided into $2^n \times 2^n \times 2^n$ smaller voxels. We call these small voxels the baby voxels of the corresponding primary voxel. Then, the occupied volume fraction of each primary voxel is computed by counting the number of its black baby voxels and in the same manner, the wetted area fraction of each face is computed by counting the number of black baby voxels in touch with the face. Figure 6 schematically shows this procedure in two dimensions.

Since the presented method is a z-buffering algorithm (layer-by-layer grid generator), it is needed to generate only n layers of the baby voxels for each primary voxel layer. Hence, the additional memory requirement is $2^{3n-3} \times i_{\max} \times j_{\max}$ bytes (consider 1 bit of memory per baby voxel).

It is clear that the generated embedded boundary information by the presented method is not essentially exact. However, in the numerical simulation, it is not essential to have the exact embedded boundary information. As stated in [40, 41] for accessing a second-order embedded boundary finite volume solver it is sufficient to have the embedded boundary information with a second order accuracy with respect to the spatial step size. Since the presented method uses the central approach to determine in/out voxels, it is expected that any geometrical information generated by the presented method has a second-order accuracy. Therefore, the presented approach to generate embedded boundary information is consistent with second-order finite volume solvers when $n \geqslant 1$. It is worth noting that the absolute level of error in this approximation is not essentially same as the situation when the exact embedded boundary information is used; hence this error might be 4–8 times larger while we have still a second-order slope [42].

For efficient implementation of this method, we use $n = 2$ (64 baby-voxels per each primary voxel) and assign 8-byte data (64 bits) for each primary voxel (the size of additional working array is $8 \times i_{\max} \times j_{\max}$ bytes). It should be noted that the storage of the produced embedded boundary data does not need floating point data type (4 byte), but needs only one integer (1 byte) per embedded boundary information. As an example, the stored value of the volume fraction is the number of black baby voxels (varies from 0 to 64); it is obvious that the real value of the volume fraction can be computed by multiplication of this number with factor $\delta x \delta y \delta z/64$.

3.6. Memory reduction

The main drawback of the presented method is the high memory usage that could limit its application for large-scale problems. In this section, the memory requirement of the presented method is studied and an effective solution for the memory problem is presented.

The consumed memory of the presented method is composed from two types of the allocated memory. The first type is the working arrays, contains a list of triangles, line segments, etc. The presented method takes about 70 bytes memory per triangle for this purpose. As an example, for a complex input geometry with about 1 million triangles the required memory for this purpose is about 70 mega bytes (MB).

The consumed memory to store the generated grid (one bit per voxel) is the other type of the allocated memory. In the octree grid case, each octree level should also be stored. As an example, for an 11-level refined octree (2048^3 grids on the finest level), the total memory consumption is about 1.23 giga bytes (GB). Therefore, the presented method does not have the memory problem up to 11-level refinement on a personal computer with about 1.5–2 GB physical memory. It is worth noting that in practical applications a 12-level grid falls in the category of very large-scale problems. For example in [28], generated octree grids over complicated multi-component

DOI: 10.1002/fld

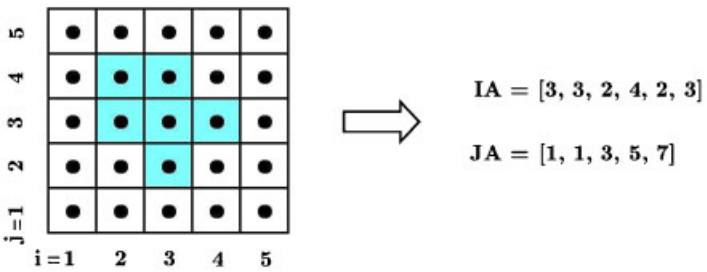

Figure 7. Schematic of voxelization compression in 2D (shaded voxels represent the black voxels).

geometries did not exceed 11 levels. For the heaviest case presented in [28], the grid generation is started from a $6\times6\times6$ coarse level and continued to 9-level refinement. The crude version of the presented method cannot handle this case on traditional personal computers.

To solve the memory problem of the presented method, a simple data compression algorithm is used. This method is similar to the compressed sparse row (CSR) format that is used in linear algebra to store sparse matrices (see [43] for more details about CSR). In this method, instead of saving the color of each voxel, we only store the index of interfacial voxels. Hence, the required memory is decreased about one order of magnitude. In this way, during each sweep along a y-constant line, i-index of the interfacial voxels (i.e. IBV voxel) is stored in an additional array. We call this array the IA array. Each item of this array is a 2-byte integer number (INTEGER*2 in FORTRAN). Another integer array, the JA array, is also essential to store the pointers to the beginning of each row in the IA array. The length of this array is $j_{\max}*k_{\max}+1$ and its items are 4-byte integers (INTEGER*4 in FORTRAN). In this manner, for accessing the i-index of voxels which are located on $j=j_c$ line, it is sufficient to loop on the IA array from item JA(j_c) to JA(j_c+1)-1. It should be noted that voxels which are located between each of the two consecutive black voxels are taken as black voxels (we do not store their indexes). In this method for a y-constant line ($j=j_c$) that has no black voxel JA(j_c) is taken to be equal to JA(j_c+1). Figure 7 schematically shows this data compression technique in two dimensions. Note that in this method, after storing each xy-plane, the next xy-plane is stored at the end of the previously stored data.

As discussed in Section 3.1, CARTGEN uses 1 bit of memory to store each voxel and hence each of the two consecutive xy-planes are mapped to an (INTEGER*1) array with one-eight size of the original voxelization. We call this storage format the primary compressed voxelization. To reduce memory usage and to decrease number of operations during data compression, we compress primary compressed voxelization instead of the direct voxelization in our data compression implementation. Since items of this array are not essentially 0 or 1, an additional integer (INTEGER*1) array, which is called AA, is used to store the values of boundary voxels of the primary compressed voxelization. To increase efficiency and ease of implementation, CARTGEN uses a working buffer array to store one plane of the primary compressed voxelization. After filling of this working buffer, CARTGEN compresses its content (for more details see the CARTGEN code).

3.7. Limitations

Although generation of Cartesian grids is automatically and efficiently possible over complex geometries, Cartesian grids have some intrinsic limitations that could limit their application. Since

DOI: 10.1002/fld

Cartesian grids are non-body fitted, they have fine features of geometry, e.g. sharp corners and ridges are rounded or missed during grid generation. The only solution to this problem is increasing grid resolution near sharp features (high curvature surfaces).

As stated in Section 2, the input geometry of the presented method should be a water-tight surface triangulation. Therefore, non-manifold and dirty geometries (e.g. have crack, hole, self-intersection and degenerated feature) should be fixed before grid generation.

3.8. Software availability

The FORTRAN 90 implementation of the presented method is freely available under the terms of the GNU Lesser GPL[§] from URL: http://mehr.sharif.ir/~tav/cartgen.htm and http://sourceforge.net/projects/cartgen/.

4. RESULTS AND DISCUSSION

In this section, demonstrative examples are presented to show the capability of the proposed method. All computations were performed on a personal computer with AMD 2.4 GHz central processing unit (CPU) and 2 GB RAM.

Figure 8 shows the 3D configuration of test cases 1–6 which are used in the present study. The number of triangles in STL files related to test cases 1–6 are 12 094, 8176, 2668, 41 850, 355 870 and 871 414, respectively. For each test case, the Cartesian grid was generated with various grid resolutions.

Figures 9–11 show results of grid generation (uniform and octree) for test cases 1–6. The number of black voxels related to results of test case 1 is 1134, 12 712, 126 365, 1242, 8116 and

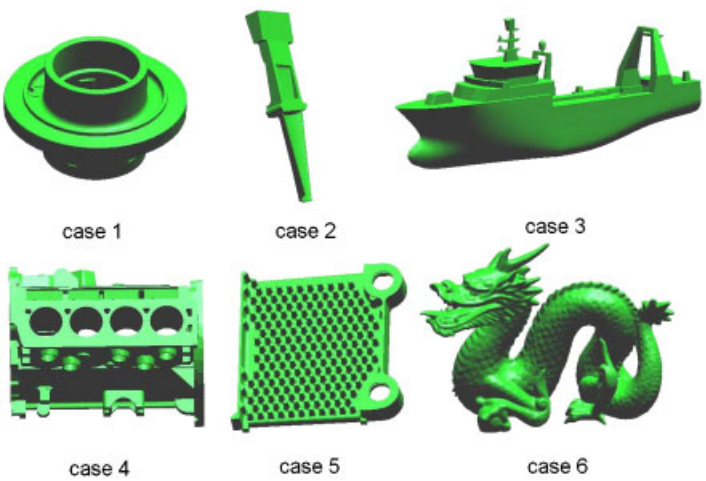

Figure 8. 3D configuration of test cases 1–6 used in the present study.

[§] http://www.gnu.org/copyleft/gpl.html.

DOI: 10.1002/fld

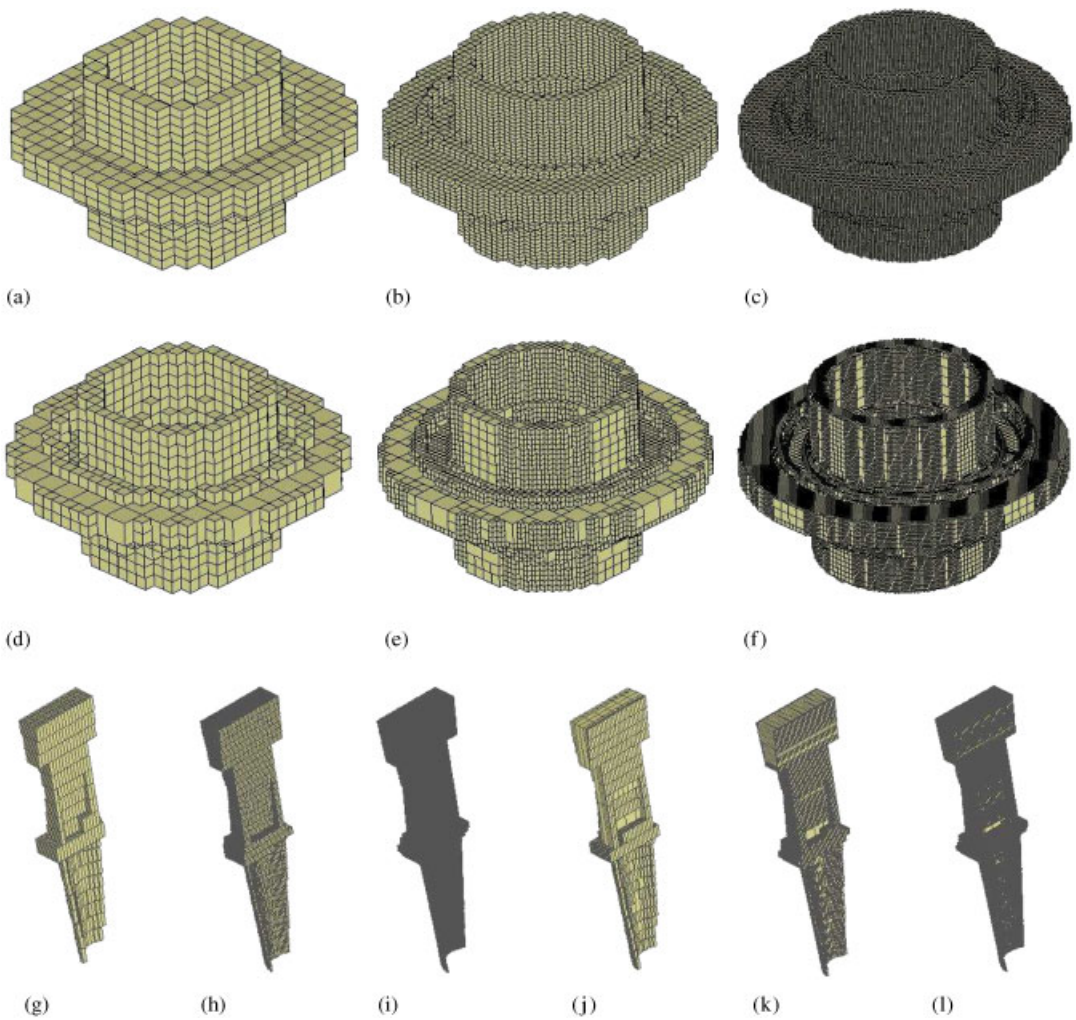

Figure 9. Cartesian grid generation results. Test case 1: (a) $25 \times 25 \times 25$ uniform grids; (b) $50 \times 50 \times 50$ uniform grids; (c) $100 \times 100 \times 100$ uniform grids; (d) 5-level octree; (e) 6-level octree; and (f) 7-level octree. Test case 2: (g) $25 \times 25 \times 25$ uniform grids; (h) $50 \times 50 \times 50$ uniform grids; (i) $100 \times 100 \times 100$ uniform grids; (j) 5-level octree; (k) 6-level octree; and (l) 7-level octree.

40 747 for Figure 9(a)–(f), respectively. The number of black voxels related to results of test case 2 is 1621, 18 931, 192 264, 1413, 9483 and 61 352 for Figure 9(g)–(l), respectively. The number of black voxels related to results of test case 3 is 2025, 23 302, 240 049, 1629, 8670 and 46 275 for Figure 10(a)–(f), respectively. The number of black voxels related to results of test case 4 is 1437, 13 091, 123 807, 2882, 20 986 and 126 208 for Figure 10(g)–(l), respectively. The number of black voxels related to results of test case 5 is 4301, 41 782, 5854 and 39 703 for Figure 11(a)–(d), respectively. The number of black voxels related to results of test case 6 is 12 826, 132 715, 21 435 and 99 537 for Figure 11(e)–(h), respectively.

DOI: 10.1002/fld

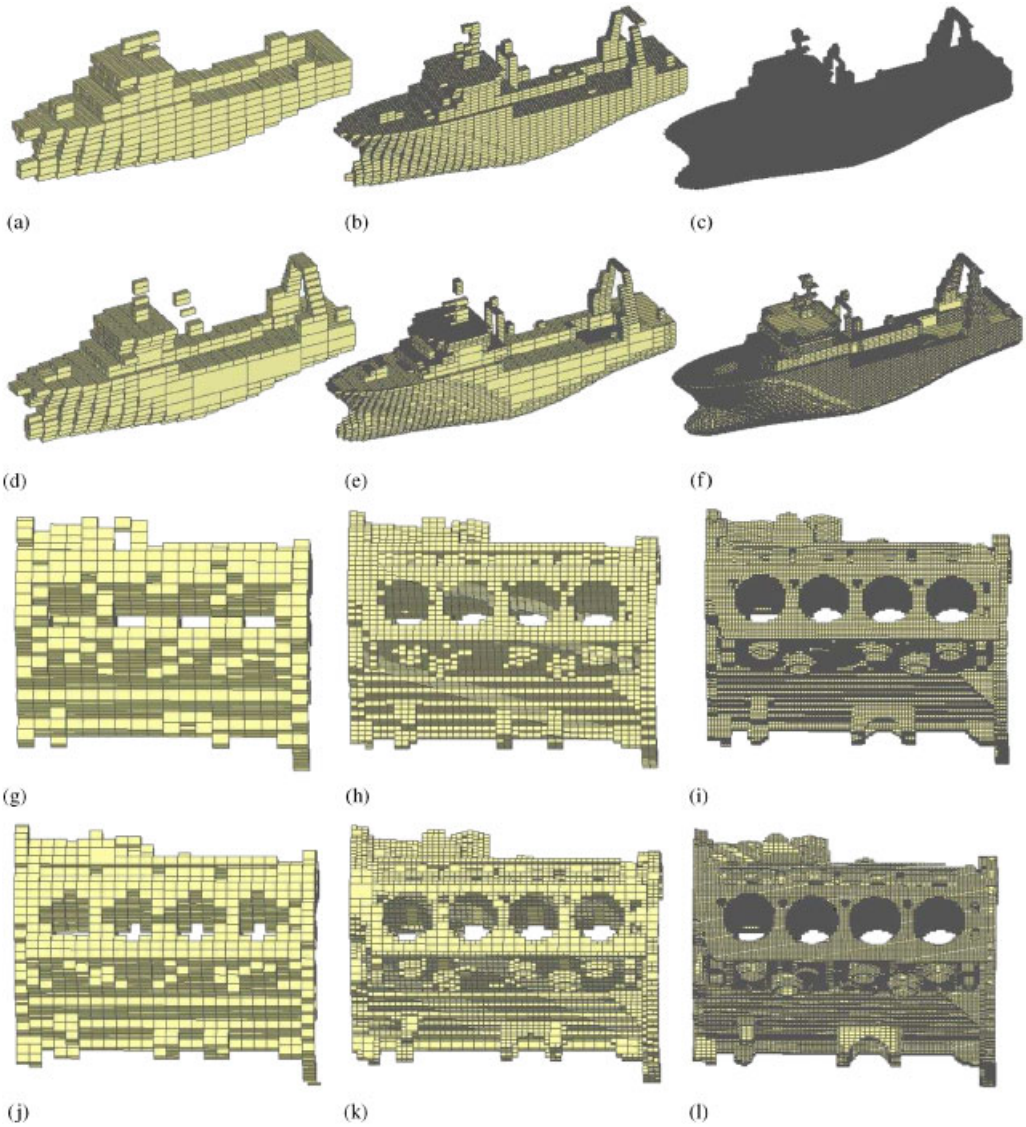

Figure 10. Cartesian grid generation results. Test case 3: (a) $25 \times 25 \times 25$ uniform grids; (b) $50 \times 50 \times 50$ uniform grids; (c) $100 \times 100 \times 100$ uniform grids; (d) 5-level octree; (e) 6-level octree; and (f) 7-level octree. Test case 4: (g) $25 \times 25 \times 25$ uniform grids; (h) $50 \times 50 \times 50$ uniform grids; (i) $100 \times 100 \times 100$ uniform grids; (j) 5-level octree; (k) 6-level octree; and (l) 7-level octree.

To study the accuracy of the embedded boundary information generated with the presented method, a sphere with a sufficiently fine surface triangulation and known dimensions was considered (diameter = 200 units and 28 560 triangles). In this case, we only investigate the accuracy and convergent of generated occupied volume fraction related to interfacial voxels (see Section 3.5).

DOI: 10.1002/fld

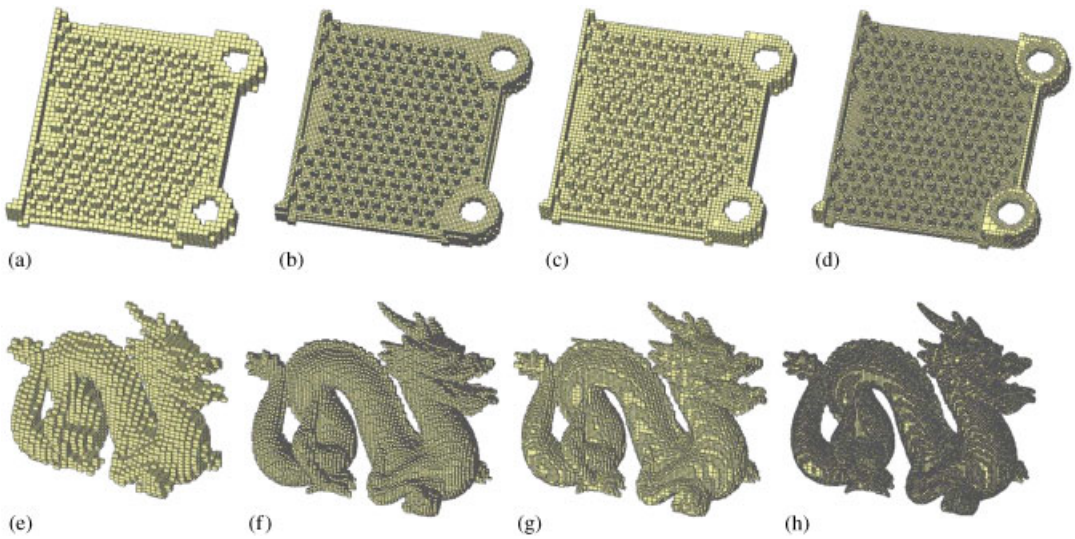

Figure 11. Cartesian grid generation results. Test case 5: (a) $50 \times 50 \times 15$ uniform grids; (b) $100 \times 100 \times 30$ uniform grids; (c) 7-level octree; and (d) 8-level octree. Test case 6: (e) $50 \times 50 \times 50$ uniform grids; (f) $100 \times 100 \times 100$ uniform grids; (g) 7-level octree; and (h) 8-level octree.

Table I. Accuracy and convergence of the generated occupied volume fraction field.

Grid spacing	n	L^1 error	Order
20	0	2584	—
10	1	414	1.64
5	2	143	1.53
2.5	3	30.1	2.45
1.25	4	10.17	1.59
0.625	5	0.38	4.718
0.3125	6	0.01	5.25

Based on the divergence theorem, the exact volume of the surface triangulated sphere was calculated and used as reference volume (it was equal to $4\,186\,400\,\text{units}^3$).

A primary uniform grid was generated with $\delta_x = \delta_y = \delta_z = 20\,\text{units}$. To extract the occupied volume fraction information, baby voxels (for $n = 1, 2, 3, 4, 5$ and 6) are generated and the occupied volume fraction field is calculated based on the method discussed in Section 3.5. Then the approximate volume of the sphere is computed based on the primary voxelization and generated occupied volume fraction field. Table I shows the L^1 error and convergence of the computed volume. It shows that the approximated volume faction information has about second-order convergence to the exact value.

To study performance of the presented method in terms of the CPU time and memory usage, test cases 5 and 6 are considered and their octree grid representation is produced up to 13-level

DOI: 10.1002/fld

Table II. Computational performance in terms of the CPU time (s) and memory usage (MB).

Octree levels	Finest level	No. black voxels	CPU time	Working memory	Grid memory
Case 5					
8-level	256^3	159 865	1.12	25.29	0.31
9-level	512^3	857 499	3.73	25.36	1.48
10-level	1024^3	2 760 119	13.87	25.62	4.79
11-level	2048^3	11 432 070	61.0	26.67	19.90
12-level	4096^3	52 503 027	300.56	30.86	88.69
13-level	8192^3	218 193 823	1706.61	47.64	366.66
Case 6					
8-level	256^3	99 537	1.92	61.89	0.20
9-level	512^3	426 596	5.31	61.96	0.83
10-level	1024^3	1 782 631	18.40	62.22	3.40
11-level	2048^3	7 448 039	76.27	63.27	14.01
12-level	4096^3	31 594 618	379.55	67.46	58.56
13-level	8192^3	135 938 537	2155.34	84.24	248.01

refinements (8192^3 grid on the finest level). Table II gives the result of this experiment. It is obvious that the presented method is efficient from both the CPU time and memory usage view points.

5. CONCLUSION

An efficient and easy to implement (uniform/octree) Cartesian grid generator is presented. It supports the variation of grid size along each spatial direction as well as anisotropic and non-graded refinements. The presented method could generate the required information for second-order embedded boundary finite volume solvers. It benefits from a simple and effective data compression method that allows one to store all the octree levels without considerable memory consumption. The efficiency and robustness of the presented method are supported by illustrative examples.

REFERENCES

1. Lee CK, Xu QX. A new automatic adaptive 3D solid mesh generation scheme for thin-walled structures. *International Journal for Numerical Methods in Engineering* 2005; **62**:1519–1558.
2. Allaire G, Bonnetier E, Francfort G, Jouve F. Shape optimization by the homogenization method. *Numerische Mathematik* 1997; **76**:27–68.
3. Chen BC, Silva ECN, Kikuchi N. Advances in computational design and optimization with application to MEMS. *International Journal for Numerical Methods in Engineering* 2001; **52**:23–62.
4. Borrvall T, Petersson J. Large-scale topology optimization in 3D using parallel computing. *Computer Methods in Applied Mechanics and Engineering* 2001; **190**(46):6201–6229.
5. Li Y, Saitou K, Kikuchi N. Topology optimization of thermally actuated compliant mechanisms considering time-transient effect. *Finite Elements in Analysis and Design* 2004; **40**:1317–1331.
6. Cho S, Choi JY. Efficient topology optimization of thermo-elasticity problems using coupled field adjoint sensitivity analysis method. *Finite Elements in Analysis and Design* 2005; **41**(15):1481–1495.

DOI: 10.1002/fld

7. Tavakoli R, Davami P. Automatic optimal feeder design in steel casting process. *Computer Methods in Applied Mechanics and Engineering* 2007; DOI: 10.1016/j.cma.2007.09.018.
8. Tavakoli R, Davami P. A fast method for numerical simulation of casting solidification. *Communications in Numerical Methods in Engineering* 2007; DOI: 10.1002/cnm.
9. Tavakoli R, Davami P. Optimal riser design in sand casting process with evolutionary topology optimization. *Structural and Multidisciplinary Optimization* 2007; submitted.
10. Tavakoli R, Davami P. Feeder growth: a new method for automatic optimal feeder design in gravity casting processes. *Structural and Multidisciplinary Optimization* 2007; submitted.
11. Si HM, Cho C, Kwahk SY. A hybrid method for casting process simulation by combining FDM and FEM with an efficient data conversion algorithm. *Journal of Materials Processing Technology* 2003; **133**(3):311–321.
12. Grill A, Sorimachi K, Brimacombe JK. Heat flow, gap formation and break-outs in the continuous casting of steel slabs. *Metallurgical Transactions B* 1976; **7**(2):177–189.
13. Maronnier V, Picasso M, Rappaz J. Numerical simulation of free surface flows. *Journal of Computational Physics* 1999; **155**(2):439–455.
14. Caboussat A, Picasso M, Rappaz J. Numerical simulation of free surface incompressible liquid flows surrounded by compressible gas. *Journal of Computational Physics* 2005; **203**(2):626–649.
15. Wang SY, Wang MY. A moving superimposed finite element method for structural topology optimization. *International Journal for Numerical Methods in Engineering* 2005; **65**(11):1892–1922.
16. Morinishi K. A finite difference solution of the Euler equations on non-body-fitted Cartesian grids. *Computers and Fluids* 1992; **21**(3):331–344.
17. Peskin CS, Printz BF. Improved volume conservation in the computation of flows with immersed elastic boundaries. *Journal of Computational Physics* 1993; **105**(1):33–46.
18. Leveque RJ, Li Z. The immersed interface method for elliptic equations with discontinuous coefficients and singular sources. *SIAM Journal on Numerical Analysis* 1994; **31**(4):1019–1044.
19. Almgren AS, Bell JB, Colella P, Marthaler T. A Cartesian mesh method for the incompressible Euler equations in complex geometries. *SIAM Journal on Scientific Computing* 1997; **18**:1289–1309.
20. Johansen H, Colella P. A Cartesian grid embedded boundary method for Poissons equation on irregular domains. *Journal of Computational Physics* 1998; **147**:60–85.
21. Day MS, Colella P, Lijewski M, Rendleman C, Marcus D. Embedded boundary algorithms for solving the poisson equation on complex domains. *LBNL-41811*, *Lawrence Berkeley National Laboratory*, May, 1998.
22. Fadlun E, Verzicco R, Orlandi P, Yusof JM. Combined immersed-boundary finite-difference methods for three-dimensional complex flow simulations. *Journal of Computational Physics* 2000; **161**(1):35–60.
23. Khokhlov AM. Fully threaded tree algorithms for adaptive refinement fluid dynamics simulations. *Journal of Computational Physics* 1998; **143**(2):519–543.
24. Aftosmis MJ, Berger MJ, Adomavicius G. A parallel multilevel method for adaptively refined Cartesian grids with embedded boundaries. *AIAA Paper 808*:*38*, 2000.
25. Aftosmis MJ, Berger MJ. Multilevel error estimation and adaptive h-refinement for Cartesian meshes with embedded boundaries. *AIAA Paper 863*, 2002.
26. Popinet S. Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. *Journal of Computational Physics* 2003; **190**(2):572–600.
27. Popinet S. Accurate adaptive ocean modelling using quadtrees. *Geophysical Research Abstracts* 2005; **7**:03789.
28. Aftosmis MJ, Berger MJ, Melton JE. Robust and efficient Cartesian mesh generation for component-based geometry. *AIAA Journal* 1998; **36**(6):952–960.
29. Wang ZJ, Srinivasan K. An adaptive Cartesian grid generation method for dirty geometry. *International Journal for Numerical Methods in Fluids* 2002; **39**(8):703–717.
30. Barequet G, Kumar S. Repairing CAD models. *Visualization'97*, *Proceedings*, Phoenix, AZ, U.S.A., 1997; 363–370.
31. Barequet G, Duncan CA, Kumar S. RSVP: a geometric toolkit for controlled repair of solid models. *IEEE Transactions on Visualization and Computer Graphics* 1998; **4**(2):162–177.
32. Davis J, Marschner SR, Garr M, Levoy M. Filling holes in complex surfaces using volumetric diffusion. *First International Symposium on 3D Data Processing Visualization and Transmission*, Padova, Italy, 2002; 428–861.
33. Zelinka S, Garland M. Permission grids: practical, error-bounded simplification. *ACM Transactions on Graphics* 2002; **21**(2):207–229.
34. Yngve G, Turk G. Robust creation of implicit surfaces from polygonal meshes. *IEEE Transactions on Visualization and Computer Graphics* 2002; **8**(4):346–359.

35. Liepa P. Filling holes in meshes. *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry Processing*, Aachen, Germany, 2003; 200–205.
36. Nooruddin FS, Turk G. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics* 2003; **9**(2):191–205.
37. Wood ZOE, Hoppe H, Desburn M, Schroder P. Removing excess topology from isosurfaces. *ACM Transactions on Graphics* 2004; **23**(2):190–208.
38. Ju T. Robust repair of polygonal models. *ACM Transactions on Graphics* (*TOG*), 2004; **23**(3):888–895.
39. Béchet E, Cuilliere JC, Trochu F. Generation of a finite element MESH from stereolithography (STL) files. *Computer-Aided Design* 2002; **34**(1):1–17.
40. Colella P, Graves DT, Keen BJ, Modiano D. A Cartesian grid embedded boundary method for hyperbolic conservation laws. *Journal of Computational Physics* 2006; **211**:347–366.
41. Schwartz P, Barad M, Colella P, Ligocki T. A Cartesian grid embedded boundary method for the heat equation and Poisson's equation in three dimensions. *Journal of Computational Physics* 2006; **211**:531–550.
42. Aftosmis MJ. http://www.nas.nasa.gov/∼aftosmis/home.html. *Private Communication*, 2007.
43. Saad Y. *Iterative Methods for Sparse Linear Systems* (2nd edn). SIAM: Philadelphia, PA, 2003.

DOI: 10.1002/fld