Seventh International Conference on Computational Fluid Dynamics (ICCFD7), Big Island, Hawaii, July 9-13, 2012

ICCFD7-1204

Wall Distance Search Algorithm Using Voxelized Marching Spheres

Beatrice Roget Jayanarayanan Sitaraman

University of Wyoming

Department of Mechanical Engineering

Laramie, WY 82071

Abstract: Minimum distance to a solid wall is a commonly used parameter in turbulence closure formulations associated with the Reynolds Averaged form of the Navier Stokes Equations (RANS). This paper presents a new approach to efficiently compute the minimum distance between a set of points and a surface. The method is based on sphere voxelization, and uses fast integer arithmetic algorithms from the field of computer graphics. An unstructured grid around an aircraft configuration (DLR-F6) is chosen as the test case for demonstration and validation. Multi-processor computations (up to 256 processors) are conducted to study efficiency and scalability. Encouraging results are obtained, with the sphere voxelization algorithm demonstrated to be more efficient than all of the alternate methods for computing minimum distances. However, a load imbalance does exist, which negatively impacts the scalability for large number of cores. A simple method for load re-balancing is formulated and tested, which results in significant improvements in both efficiency and scalability.

Keywords: Numerical Algorithms, Computational Fluid Dynamics, Turbulence Modeling, Computer Graphics.

1 Introduction

Minimum distance to a solid wall is a primary parameter that is utilized in several turbulence models. For example, in the commonly used Spalart-Allmaras turbulence model, the turbulence destruction source term is inversely proportional to the square of the wall distance. For a grid with N_p field points and N_b boundary faces, the cost of direct exhaustive computation of wall distances is of $O(N_p \times N_b)$, which is quite expensive for grids utilized in the state-of-the-art RANS based CFD calculations (where N_b is of order hundred thousand and N_p is of order several million). The applicability of wall-distance based damping functions for geometrically and topologically complex problems has been a topic of active discussion in the turbulence modelling community. The work presented here does not concern itself with this debate. In contrast, we tackle the algorithmic challenge and explore a new method of computing the minimum distance function efficiently and accurately. Several algorithms have been devised by various researchers in the past to compute the minimum wall distance. They can, in general, be classified into three groups:

1. $k-d$ tree based search approaches [1, 2, 3]
2. Differential equation based approaches [4, 5]
3. Advancing surface front methods [6]

In a parallel computing environment, all of the above approaches suffer from scalability and/or accuracy issues. The $k-d$ tree approach (octree, ADT etc are subsets of this group), which is an adaptation of the classic nearest-neighbor search, constructs a digital tree of the surface elements and follows a divide-and-conquer algorithm by eliminating regions that do not intersect with the sphere corresponding to the "current best" minimum distance. This approach is quite efficient for points that lie close to the boundary surface.

However, as points are further away from the surface, lesser number of regions (and hence tree branches) can be eliminated from comprehensive checking, leading to poor scalability–because grid partitions lying further from the surface will incur much more computations compared to those that lie closer to the surface. Differential equation based (Poisson, Eikonal) approaches are derived by posing the minimum distance search as a wave propagation problem, i.e. minimum distance is computed as the time taken by a wave front of unit speed emanating from the surface tesselation to reach a given query point. This is a purely hyperbolic problem that is observed to exhibit stability problems near sharp corners. Most often, dissipative terms are added to stabilize and smoothen the behavior of these equations. Once formulated, these equations can be discretized and solved using any of the standard approaches such as finite-difference, finite-volume of finite-element methods. Owing to discrete formulation, the differential equation based approaches are quite scalable and much easier to implement as parallel algorithms. However, the accuracy of the wall distances is driven by the order of accuracy of the discrete approximation (of the governing PDE) as well as the level of convergence obtained. True minimum distance can be achieved only in the limit of zero grid spacing and machine-zero convergence. Advancing front methods utilize a painting technique using the nodal connectivity of cells. Beginning at the cells with wall boundaries, cell vertices are incrementally tagged by using surface element indices stored at nodal neighbors as candidates for evaluating minimum distance. This approach can be perceived as a topological equivalent of the wave propagation approach. Therefore, in the presence of non-uniform grids and highly varying cell sizes, the computed minimum distances could accumulate errors. Furthermore, advancing front methods have inherent defects in scalability as processes controlling partitions further away have to wait idle until the front approaches them after passing through closer partitions. In summary, $k-d$ tree approaches can provide the true-estimate of the wall distance, but have limited scalability. Differential equation based approaches are scalable, but only provide approximate estimate of the wall distance and advancing surface front type methods are not scalable and not unconditionally accurate.

The objective of this work is to explore a different algorithm, still based on searching rather than differential equations, in an effort to mitigate the scalability and accuracy issues mentioned above. Similar to the tree approach, the proposed method attempts to eliminate as many surface elements as possible in order to compute exact distances for the smallest subset possible. It does so by considering discretized spheres on a structured auxiliary grid. These spheres are centered on each query point and their radius in increased until a subset of candidate faces is identified for exact minimum distance evaluation. This method does not however eliminate the load imbalance of the search process. For points that are a large distance away from the surface, the sphere radius will become large and hence the number of surface elements to consider will be larger than for points closer to the surface. However, it is expected that the imbalance will not be as large as for tree-based methods – since even at the limit of infinite distance only a subset of the faces need to be checked– and hence could be mitigated by re-balancing the load in anticipation of this problem. Furthermore, similar to the tree method, the present method has guaranteed accuracy since it is based on exact computation of minimum distances (albeit for a small subset of surface elements).

The primary motivation for this work stems from the need for efficiently recomputing minimum distance functions for unsteady simulations involving moving/deforming bodies. This is especially relevant in the case of massively parallel simulations involving overlapping grids. For example, the multi-mesh/multi-solver approach pioneered by the U.S Army HELIOS [7] framework uses a grid system that is composed of multiple body-conforming meshes (near-body meshes) that extend a short distance from the solid walls. Further, these near body meshes are embedded in a nested Cartesian mesh capable of adaptive mesh refinement (AMR). For a larger degree of automation in mesh generation and computational efficiency, the extent of the body-conforming meshes should be as limited as possible (such as in strand-type grids), with the bulk of the computations to be performed on the Cartesian meshes [8]. Clearly, the minimum distance function has to be evaluated accurately for all of the overlapping mesh systems to achieve consistent turbulence closure. In addition to scalability and efficiency, software modularity is central for successful implementation and demonstration in such multi-solver frameworks. In this context, differential equation based minimum distance approaches can be said to lack software modularity, since they require corresponding implementations in all participating solvers and grid assembly tools. Further, convergence of the distance function is not guaranteed in an overlapping mesh system, because of its strong non-linear dependence with complex geometry. Therefore, search-based methods that can exactly determine the distance function become attractive owing to their software modularity. However, neither the tree-based methods nor the advancing front methods are attractive because of their poor scalability. Furthermore, advancing front methods and tree based method to

some extent rely on cell-to-node connectivity maps to accelerate their search process. Requirement of cell-to-node connectivity maps adversely affects computing minimum distances in topologically disjoint overlapping mesh systems. Therefore, we explore a new method in an attempt to obtain all of the qualities desirable in a multi-solver framework, namely scalability, accuracy, efficiency and modularity. In particular, to maintain modularity, the problem is formulated as requiring the minimum possible input information–only the surface tesselation (surface nodes and their connectivity) and a point cloud (points to which minimum distance need to be estimated) are necessary. It is worth noting that a naive implementation of the proposed method of voxelized marching spheres leads to inadequate improvements in accuracy, efficiency, and scalability. Therefore, we perform several optimizations such as (1) modification of the baseline circle pixelization to prevent gaps (2) voxelization of partial spheres (3) generation of cache-efficient code (4) determination of optimal voxel size (5) reduction of the number of true minimum distance checks using approximate distances and (6) load re-balancing to achieve an accurate, efficient and scalable method, which is an attractive alternative to existing minimum distance evaluation approaches.

2 Problem Definition and Algorithm Overview

Given a set of query points and a surface (discretely represented as a collection of faces), the problem addressed here is to find the minimum distance between each query point and the tesselated surface. Considering a sphere centered on a query point and continuously increasing in radius, the minimum distance is, by definition, the radius of the sphere when it has the first intersection with the surface.

The central idea of the present algorithm is to superimpose a Cartesian, regular auxiliary grid on the surface geometry, and to approximate the surface of the sphere with initial contact using cells of this auxiliary grid. Borrowing a term from the field of computer graphics, these cells are called voxels (volume elements, by analogy with 2D pixels). The wall surface is also approximated using voxels in a pre-processing step, so that candidate surface elements for minimum distance can simply be found as those inside voxels of the sphere with initial surface contact.

Note that the voxelization of the wall surface and of the sphere surface are performed in fundamentally different ways. The wall surface voxels are identified as a pre-processing step, by considering the voxels overlapped by each surface element (one advantage of using a Cartesian auxiliary grid is that the containing voxels can be determined efficiently by very few floating point operations). On the other hand, the sphere surface has to be voxelized multiple times: for each query point, several spherical shells must be generated until initial contact with the wall occurs. For this reason, the sphere voxelization is generated using a very efficient circle-drawing algorithm from the field of computer graphics, modified and extended to three dimensions. Figure 1 shows the voxelized approximation of a spherical shell of radius $R = 12$, with a voxel highlighted.

The entire algorithm for minimum distance calculation using sphere voxelization consists of several tasks.

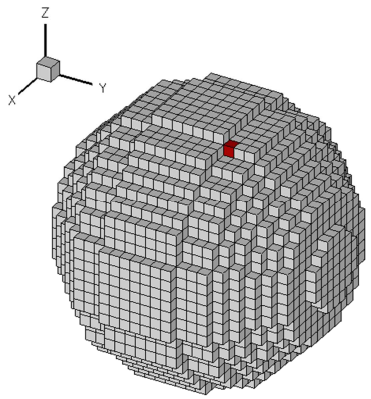


Figure 1: Voxelized sphere R=12 with highlighted voxel.

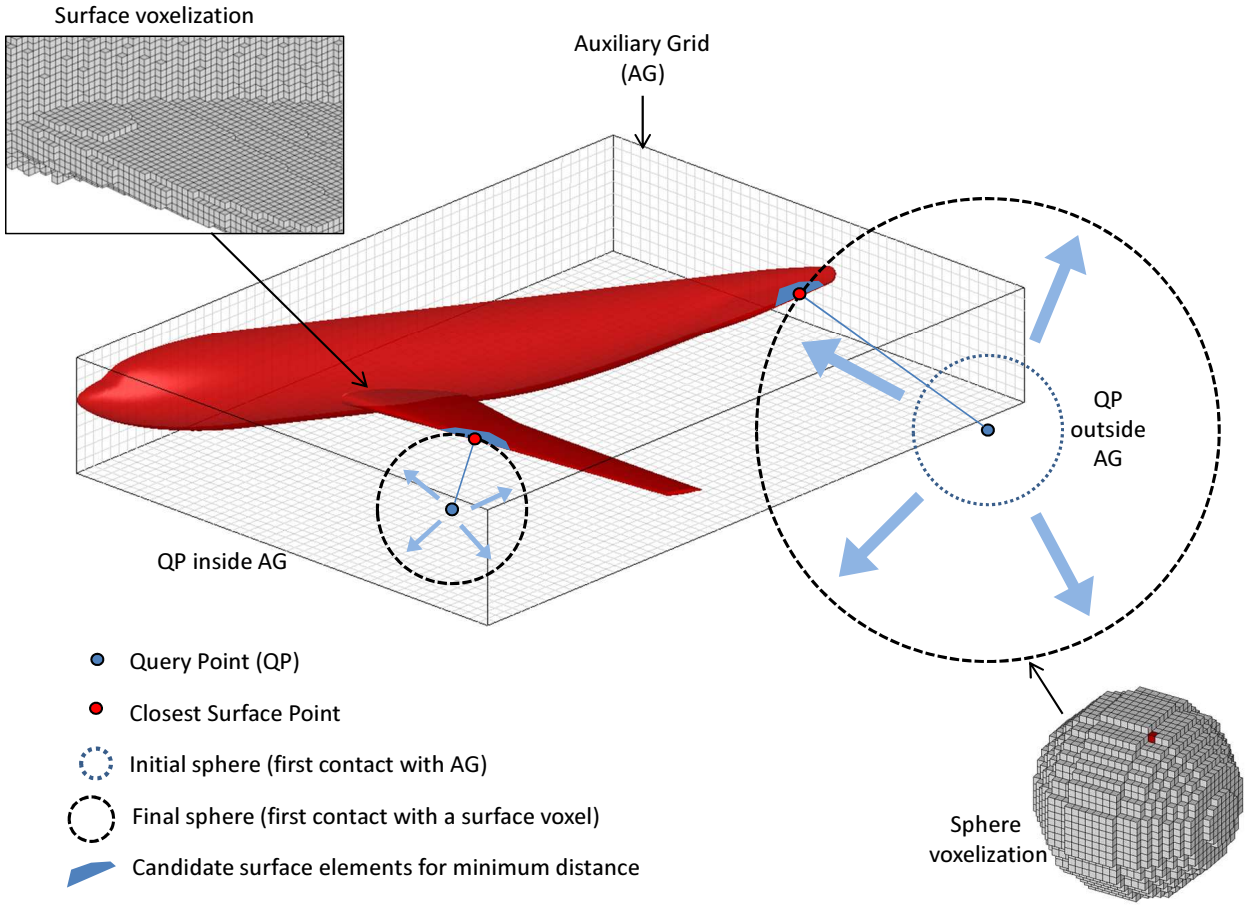


Figure 2: Overview of the minimum distance search method using voxelized marching spheres

These can, in general be grouped into two main categories, they are:

- Pre-processing: main sub-tasks are (a) calculation of floating point parameters (to accelerate actual distance calculation) for each surface element (b) computation of an oriented bounding box (OBB) for the surface elements (c) determination of the appropriate voxel size (d) generation of voxel-to-surface elements maps and (e) generation and storage of pixelized circle representations.
- Distance evaluation, consisting of two main sub-tasks:
  (a) sphere voxelization and identification of surface elements: approximated (voxelized) spheres of increasing radius are generated from each query point until initial intersection is detected. Voxelized spheres are generated using an extension of the mid-point circle algorithm to three dimensions. Once the first intersection is achieved, candidate surface elements are collected based on the bounds from a robustness/accuracy criteria.
  (b) distance computations: distances between query points and each candidate surface element are computed. Actual exact distances are computed only for a small subset (see section 6). The minimum distance for each query point is continuously updated by comparing the current minimum distance with the computed minimum distance.

Figure 2 shows the salient features and logic of the algorithm outlined above, where concepts of surface voxelization and marching voxelized spheres are illustrated. In Figure 3, the various tasks of proposed minimum distance search algorithm are summarized in block diagram format.

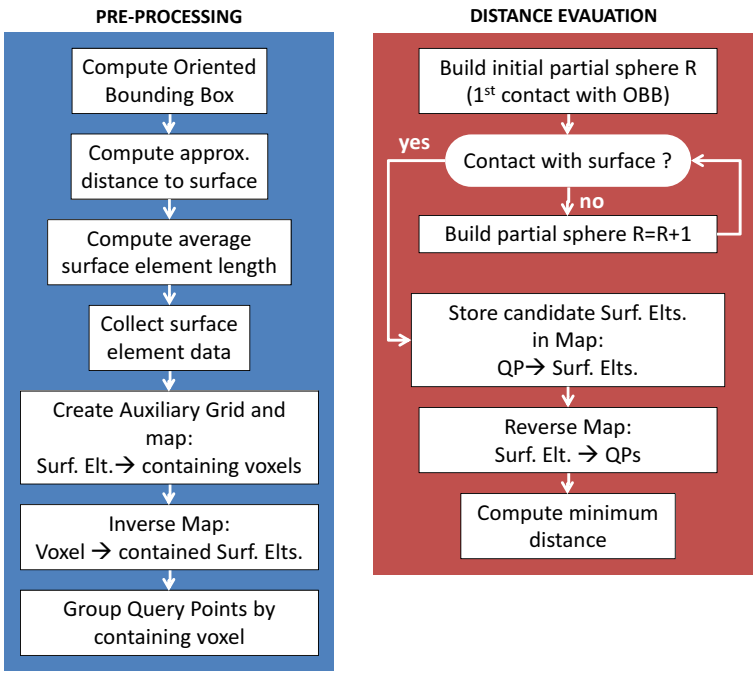


Figure 3: Summary of main algorithm tasks

The paper is organized as follows. In section 3, the pre-processing tasks are described. Section 4 describes the algorithm used to build the voxelized spheres. Section 5 explains the process by which candidate surface elements are selected (for which distance computations are required). This concludes the presentation of the basic algorithm. In section 6, various additions and modifications to the algorithm are detailed, that maximize code efficiency (including a method for load re-balancing). The final section presents performance results obtained using a partitioned unstructured grid around a body (DLR-F6 aircraft configuration), and compares these results with alternate methods.

3 Surface pre-processing using Cartesian auxiliary grids

Auxiliary grids are used to create an approximate representation of the surface. Such auxiliary grids have been used by several researchers [9, 10, 11, 12] to help efficiently solve various types of geometric problems.

These are typically Cartesian grids, consisting of equal-size box-shaped cells (voxels). The advantage of such grids is that they are entirely defined by their orientation, their outer bounds ($C_{min} = (x_{min}\ y_{min}\ z_{min})^T$ and $C_{max} = (x_{max}\ y_{max}\ z_{max})^T$) and the number of voxels in each direction ($N_x\ N_y\ N_z$), so that the actual coordinates of each cell are seldom required to be computed (for this reason, they have also been called "Virtual Grids" [9]).

In addition, for any point $P = (x\ y\ z)^T$ inside the auxiliary grid, the indices of the containing voxel ($i = 1$ to N_x, $j = 1$ to N_y, $k = 1$ to N_z) can be rapidly identified:

$$i = \left\lfloor \frac{x - x_{min}}{\delta} \right\rfloor + 1,\ j = \left\lfloor \frac{y - y_{min}}{\delta} \right\rfloor + 1,\ k = \left\lfloor \frac{z - z_{min}}{\delta} \right\rfloor + 1 \tag{1}$$

where δ is the size of a voxel, defined as:

$$\delta = \frac{x_{max} - x_{min}}{N_x} = \frac{y_{max} - y_{min}}{N_y} = \frac{z_{max} - z_{min}}{N_z} \tag{2}$$

The surface auxiliary grid is established by building an oriented bounding box (OBB) around the surface

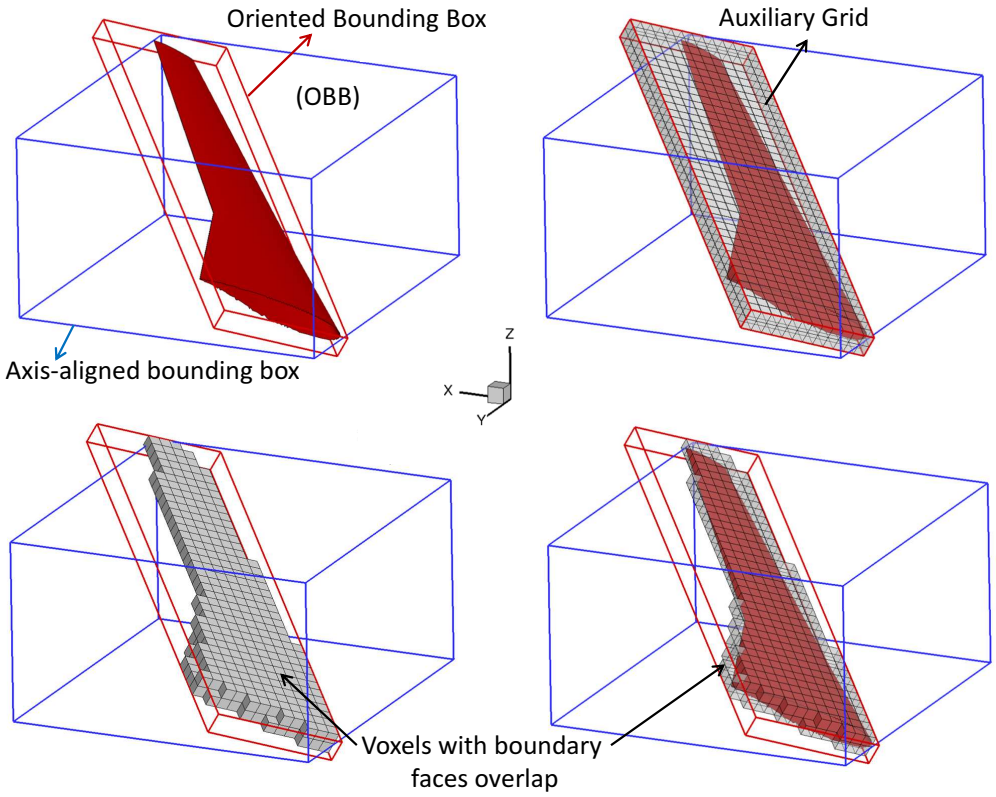


Figure 4: Creating the auxiliary grid using an Oriented Bounding Box.

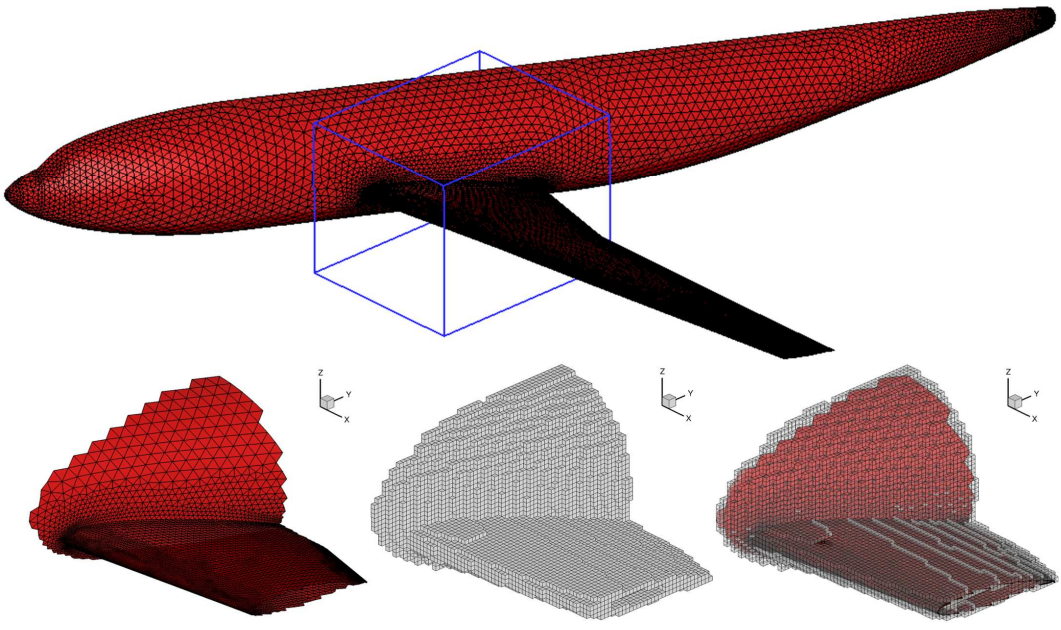


Figure 5: Identification of the voxels with possible surface overlap.

elements. The OBB axis are chosen to be the eigenvectors of the covariance matrix of the surface element centroids. Although this method does not always result in an optimal (minimum volume) bounding box, it achieves a good compromise between computational efficiency and bounding box quality. The size of voxels chosen is critical to the performance of the algorithm. With very small voxels, it is possible to reduce the number of candidate surface elements for which distance must be computed, but the number and the size of the spheres which must be built to identify them increases. The optimal voxel size is primarily a function of the minimum distance from the query points to the surface. Parametric studies and subsequent analysis of the performance statistics led to the following empirical rule for the voxel size δ :

$$\delta = \bar{l}\left[min\left(1 + \frac{\tilde{d}^{avg} + \tilde{d}^{std}}{12\,\bar{l}}\,,\ 4\right)\right] \tag{3}$$

where $\bar{l}$ is the average edge length of the surface elements, $\tilde{d}$ is the approximated minimum distance from query points to surface, and $\tilde{d}^{avg}$ and $\tilde{d}^{std}$ are the mean value and standard deviation of $\tilde{d}$, respectively. Approximated distance values are computed by considering the minimum distance from each query point to the six surface points that define the OBB. This heuristic method yields a good estimate of the optimal voxel size, which may then be further refined at each step of an unsteady simulation using on-line optimization techniques that can utilize an improved estimate of minimum distances.

Once the optimal voxel size is determined, the auxiliary grid is built and a list mapping each surface element to the voxels with potential overlap is established (which we abbreviate as "surfel-to-voxels" map). This task is performed efficiently by considering the bounding box of each surface element in the axis system of the auxiliary grid and marking all the voxels that overlap with this bounding box. This map is then inverted to obtain the voxel-to-surfels map. For each voxel, the voxel-to-surfels map provides a list of all the surface elements that potentially overlap with it.

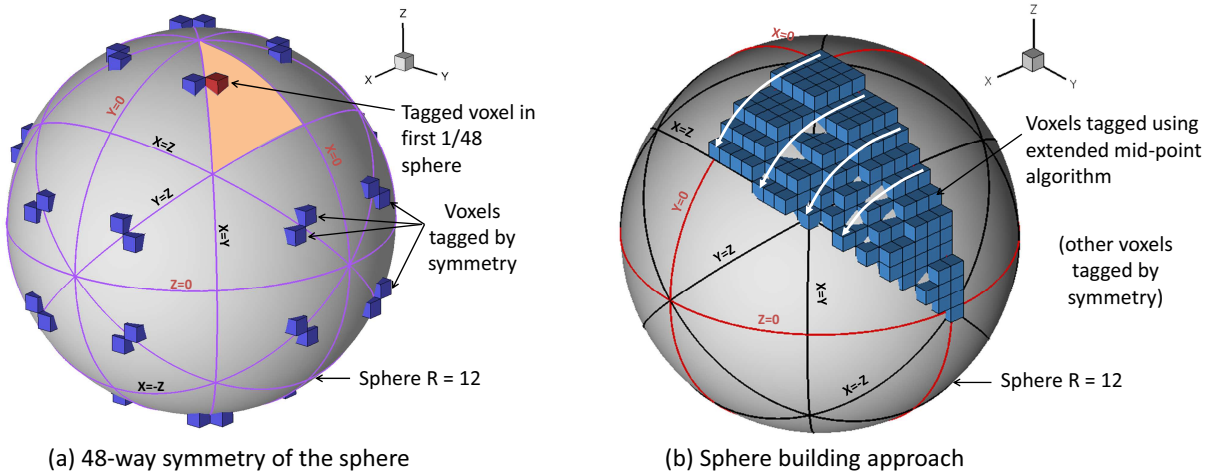
Figure 4 illustrates the surface pre-processing step for an aircraft wing. In Figure 4(a), the oriented bounding box is compared with the axis-aligned bounding box: in this case, the identified OBB is a good approximation of the minimum volume bounding box. Figure 4(b) shows the Cartesian auxiliary grid, using a relatively large voxel size for illustration clarity. Figures 4(c) and 4(d) show only the voxels with potential surface element overlap.

Finally, Figure 5 shows voxels with potential surface element overlap for the aircraft configuration tested in the results section. In this case, the voxel size is smaller (obtained using equation 3), so only a limited region (blue box) is shown for illustration clarity.

4 Generation of a voxelized sphere

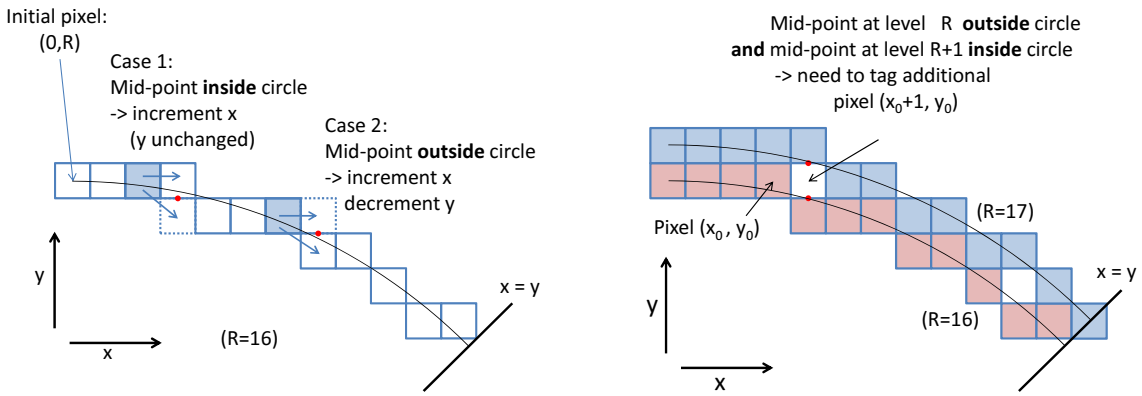
4.1 3D sphere voxelization algorithm

To generate the voxelized sphere, the 2D mid-point circle algorithm is extended to three dimensions in a manner similar to [13]. This algorithm determines the pixels required to draw a circle. It is very efficient because it only uses integer arithmetic and makes use of the circle 8-way symmetry: only one octant needs to be computed, the rest is drawn by symmetry [14, 15, 16]. For the sphere, there exists a 48-way symmetry, so that only 1/48th of the sphere needs to be voxelized. This is illustrated in Figure 6(a). Although attractive for construction of full spheres, the 48-way symmetry requires complex coding logic for adaptation to building of partial spheres. Therefore, we adopt a simpler sphere-building algorithm, that is be more easily adaptable for partial sphere building. The approach is based directly on the mid-point algorithm for circle pixelization, which is one of the elementary algorithms utilized in computer graphics. The sphere is built as a collection of circles of varying radius in planes parallel to the center plane (X-Z). The offset from the center and the radius are determined by the pixelization of the great circle in the plane (Y-Z). Figure 6(b) illustrates this sphere building approach. Owing to the circle's 8-way symmetry, only the first octants of each (X-Z) circle are built. Additionally, the circles located at negative offsets (along the Y axis) are also obtained by symmetry, so that only 1/16th of the sphere voxels need to be identified using the mid-point algorithm. To reiterate, this algorithm does not take full advantage of the 48-way sphere symmetry and can hence be optimized further.



(a) 48-way symmetry of the sphere (b) Sphere building approach

Figure 6: 48-way symmetry of the sphere illustrated for R=12, and sphere building approach using 1/16 sphere.



(a) Mid-point circle algorithm (b) Extended mid-point circle algorithm

Figure 7: Mid-point circle algorithm and modification for ensuring no gap between levels.

4.2 Modified 2D circle pixelization algorithm

To compute the sphere voxelization, several 2D circles need to be pixelized. This task is performed using the mid-point circle algorithm. The baseline algorithm, based on seminal work by Bresenham [14], identifies the pixels required to draw a circle (of integer radius R) in the plane (X-Y) in the following way: starting at the pixel $(0, R)$, the first octant is drawn ($x \geq 0$, $y \geq x$). For this octant, at each iteration, there are two choices: x is incremented and y is unchanged (case 1) or x is incremented and y is decremented (case 2). The decision is made by considering the mid-point between the two candidate pixels: if the mid-point is inside the circle, the pixel above is chosen (y unchanged - case 1), while if it is outside the circle, the pixel below is chosen (y decremented - case 2). For pixel (x,y), the decision variable is then defined as

$$d_x = F(x+1, y-\frac{1}{2}), \text{ where} \tag{4}$$

$$F(x,y) = x^2 + y^2 - R^2 - \frac{1}{4} \tag{5}$$

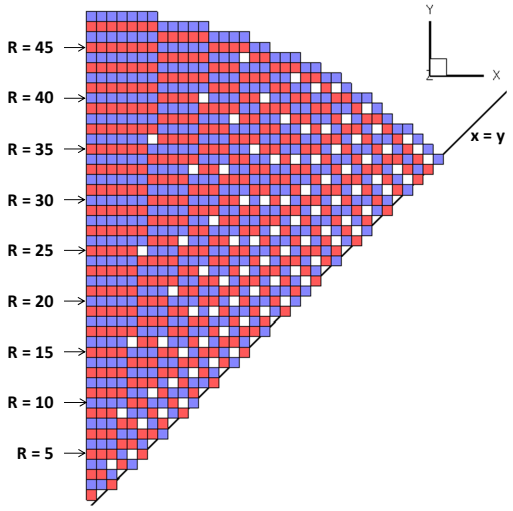


Figure 8: Gaps generated between radial levels when using the mid-point algorithm.

The quarter offset is introduced to avoid using non-integer arithmetic (since all variables are integers, $x^2 + y^2 > R^2$ is equivalent to $x^2 + y^2 > R^2 + \frac{1}{4}$). In addition, by updating the variable d_x at each iteration, computing square values can be avoided:

$$d_{x=0} = 1 - R \tag{6}$$

$$\text{case 1: } d_{x+1} = d_x + 2x + 1 \tag{7}$$

$$\text{case 2: } d_{x+1} = d_x + 2x - 2y + 5 \tag{8}$$

Figure 7(a) illustrates cases 1 and 2 for $R = 16$. One problem with the baseline mid-point algorithm is that it creates gaps between circles of successive radius. This situation is illustrated for $R = 16$ and $R = 17$ in Figure 7(b), where a gap is found to be generated at pixel locations $(4, 16)$ and $(10, 13)$. Figure 8, which shows the circle octants generated from R from 1 to 50 further illustrates this issue by representing the circle octants generated for R from 1 to 50. Radii which are odd integers are shown in red, while radii which are even integers are shown in blue. The white squares that are visible between radius levels are gaps corresponding to pixels which are never part of any pixelized circle (with the same center). Gaps will cause robustness issues and are clearly unacceptable for the minimum distance search problem. The algorithm must therefore be modified such that no pixel is missed. Analyzing figures 7(b) and 8 further, it can be noted that a gap is created in the following circumstance: consider the pixelized circles with radius R and $R + 1$; if circle R tags pixel (x, y) followed by $(x + 1, y - 1)$ and circle $R + 1$ tags $(x, y + 1)$ followed by $(x + 1, y + 1)$ a gap is created. The corresponding equations for circles R and $R + 1$ are:

$$\text{circle R (case 2)} : (x+1)^2 + (y - \frac{1}{2})^2 - R^2 - \frac{1}{4} > 0, \text{ and} \tag{9}$$

$$\text{circle R+1 (case 1)} : (x+1)^2 + (y + \frac{1}{2})^2 - (R+1)^2 - \frac{1}{4} < 0 \tag{10}$$

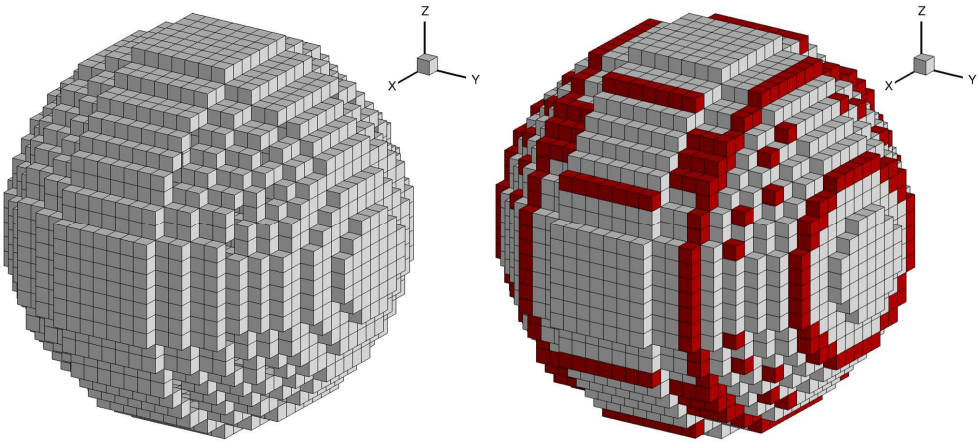


Figure 9: Comparison of voxelized spheres (R=12) with and without additional voxels.

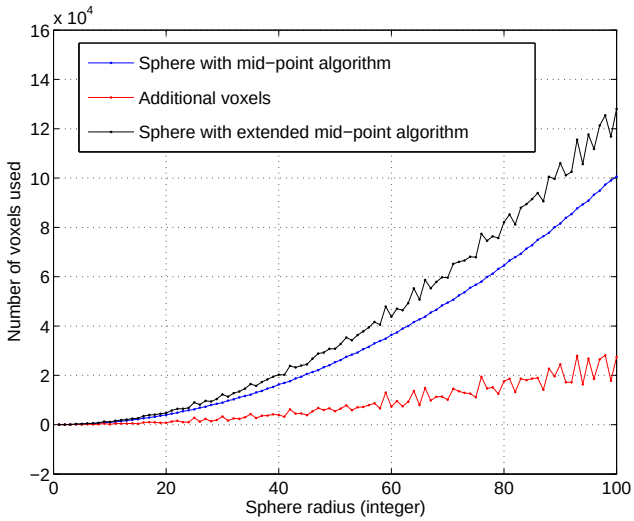


Figure 10: Number of voxels required for spheres of increasing radius

From Figure 7(b), it is clear that the pixel $(x+1, y)$ should also be tagged by circle R to prevent this problem. Therefore the following modification is applied to baseline mid-point algorithm:

$$\begin{aligned} &\text{if } d_x > 0 \\ &\quad \text{tag voxel } (x+1, y-1) \\ &\quad \text{compute } d'_x = d_x + 2y - 2R - 1 \\ &\quad \text{if } d'_x > 0 \\ &\qquad \text{tag additional voxel } (x+1, y) \end{aligned} \tag{11}$$

Figure 9 compares the voxelized spheres at R=12 with and without the present modification to avoid gaps. As expected, when using the baseline mid-point circle algorithm, the sphere created is not water-tight. However, with the modification, there is neither gap nor overlap between spherical shells of successive radial levels. Figure 10 shows the number of voxels required to generate spheres of increasing radius, using the

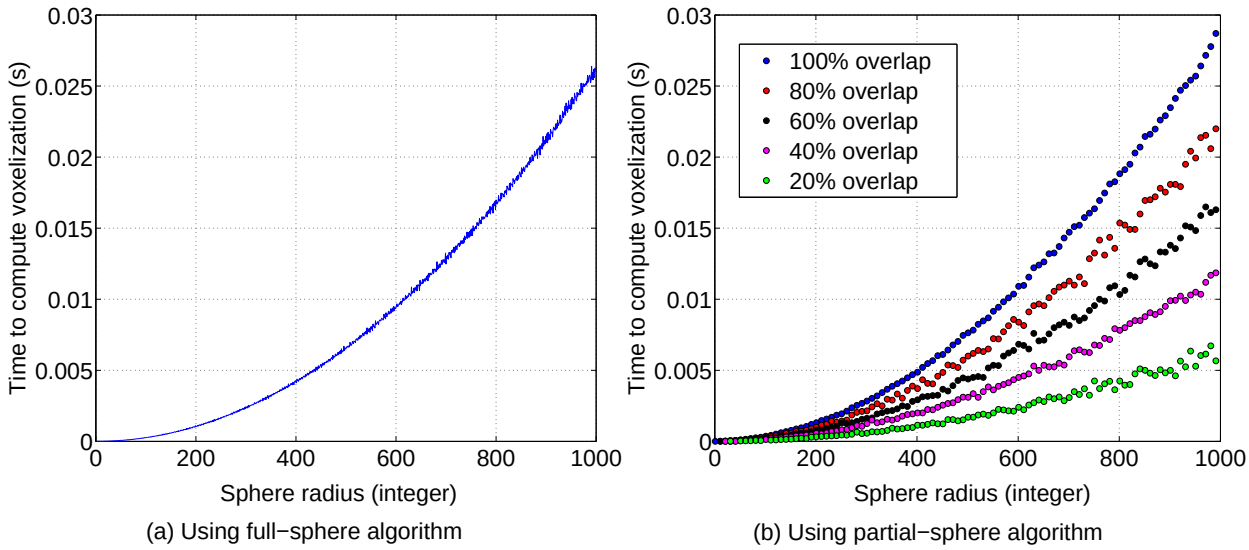


Figure 11: Time required to compute complete sphere voxelization vs. radius.

baseline mid-point circle algorithm, and using the modified algorithm with additional voxels. The number of regular voxels required varies linearly with the square of the radius (around ten times), and the number of additional voxels required is around 25% more, so that the total number of voxels required is around $12.5R^2$ (close to the surface area of the sphere, $4\pi R^2$).

4.3 Generation of a partially voxelized sphere

When query points are located far away from the surface (relative to the surface dimensions), the spheres required to identify the closest surface point can have a very large radius and the computational cost to build the entire sphere becomes unpractical. Figure 11(a) shows the time required to compute the full voxelization for spheres of increasing radius. The time required increases linearly with the square of the radius, and for a sphere of radius $R = 1000$, the time cost is almost 30 milli-seconds. A modified algorithm is therefore required that builds only the portion of the sphere that lies inside the oriented bounding box of the surface. This is illustrated in Figure 12, which shows the partial sphere containing the closest surface point to a query point located outside the surface bounding box (i.e., outside the auxiliary grid). In this case, that sphere has a radius R=51. Note that the spheres are generated starting from the sphere possessing the voxel closest to the query point voxel. In the illustrated case, the initial sphere has radius R=44 (dotted line), so that eight (partial) spheres need to be generated in total for this query point.

The partial sphere voxelization algorithm first identifies which circle octants overlap the surface bounding box, both for the great circle (X-Y), and for the (varying-radius) circles (X-Z). For these octants, only the circle arc contained in the bounding box is built by using direct look-up of the voxels required. In order to enable direct look-up, the 2-D pixelization of circles with radius varying from 1 to a large value (10000) is computed and stored as a pre-processing step (only the first 1/8 needs to be stored).

Figure 11(a) and (b) show the time required to compute the full and partial voxelization of spheres of increasing radius, respectively. For partial spheres, time is shown for different sphere/bounding box overlap ratios (overlap is defined as the ratio of the number of voxels for the partial sphere to the number of voxels for the full sphere). The computational cost varies almost linearly with the amount of overlap. Hence, for query points that are located far away from the surface (for which the sphere/bounding box overlap is small), the minimum distance search can be performed much more efficiently using the partial sphere algorithm.

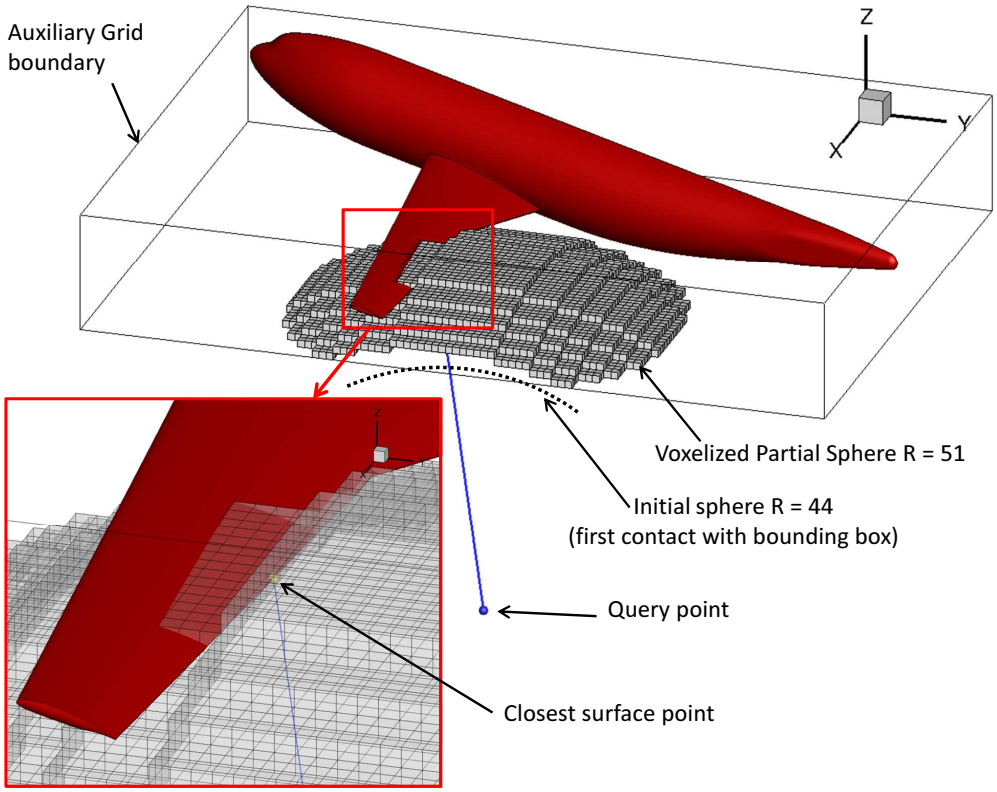


Figure 12: Using a partially voxelized sphere to identify the closest surface point.

5 Determining candidate surface elements

The voxelized partial spheres are used to identify candidate surface elements, which are then checked exhaustively to identify the exact closest surface point. To identify the candidate surface elements, the (partial) sphere containing the auxiliary grid voxel closest to the query point is first built. If (i_0, j_0, k_0) are the coordinates of the voxel containing the query point (note that these coordinates can be outside the range of the auxiliary grid), the closest voxel inside the bounding box has coordinates (i_c, j_c, k_c), with

$$i_c = \min\left(N_x,\ \max(1, i_0)\right),\ j_c = \min\left(N_y,\ \max(1, j_0)\right),\ \text{and}\ k_c = \min\left(N_z,\ \max(1, k_0)\right) \tag{12}$$

The initial sphere radius, R_{c0}, can then simply be computed as:

$$R_{c0} = \left\lfloor \sqrt{(i_c - i_0)^2 + (j_c - j_0)^2 + (k_c - k_0)^2} \right\rfloor \tag{13}$$

The marching sphere loop is then started: at each step, the sphere radius is incremented by one (beginning at R_{c0}), and the voxels of the corresponding partial sphere are checked to determine whether any surface elements are potentially contained in them (using the voxel-to-surfels map created as a pre-processing step). When the first non-empty voxel is reached, all boundary faces at that radius level (R_0) are collected. In most cases, the closest surface point can be found in one of these boundary faces. However, because the voxelized sphere is only an approximation of the true sphere, in some cases the closest surface point is located in a voxelized sphere of larger radius.

To determine the number of additional spheres that must also be checked, the distance between a point in a center voxel and a point in a sphere voxel must be examined. Considering a point randomly located in the center voxel, and a point randomly located inside one of the voxels of the sphere (radius R), the distance between them is observed to be always bounded between $R - 2.5$ and $R + 2.8$ (the additional voxels tagged

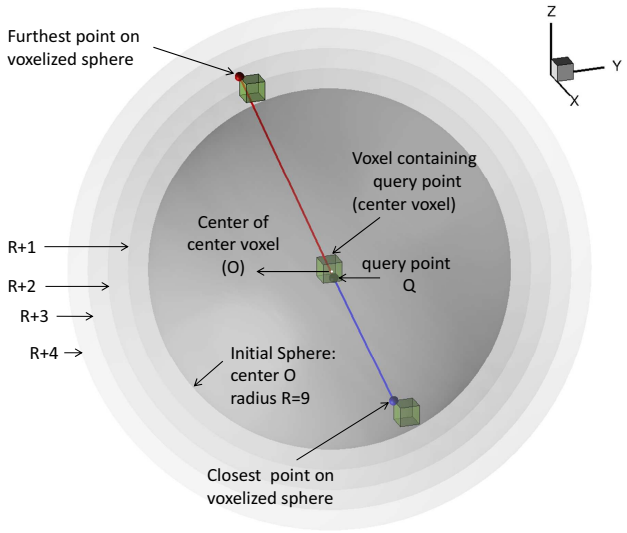


Figure 13: Closest and furthest points on sphere voxelization for R=9.

during sphere building are the reason for the asymmetry). From this observation, an upper bound for the number of additional spheres to check can be established as 5, since a point located in a (voxelized) sphere of radius $R_0 + 6$ will always have an actual distance above $R_0 + 3.5$, which is more than the maximum distance based on the initial sphere ($R_0 + 2.8$). In other words, if the initial sphere containing surface elements has radius R_0, the true closest point must be located in one of the (voxelized) spheres of radius R_0 to $R_0 + 5$.

To illustrate the point-to-center distance variation within a voxelized sphere, Figure 13 shows the closest and furthest points on a voxelized sphere (R=9), to a query point contained in the center voxel. Spheres of larger integer radius (R+1) to (R+4) are also shown.

Building and checking 5 additional spheres for all query points would increase the computational cost considerably, however it is possible to minimize this number for each query point, by taking into account (a) the distance from query point to the center of its containing voxel, d_c, and (b) the minimum distance computed based on the initial sphere with surface contact, d_{min}.

For a given query point Q and a voxelized sphere of radius R (centered on the voxel containing Q), let us define the maximum lower offset (ΔR_L) as the difference between R and the minimum distance between Q and any sphere voxel point (point located inside a sphere voxel):

$$\Delta R_L = R - \min\left(\|QP\|\right) \text{ , for P in voxelized sphere R} \tag{14}$$

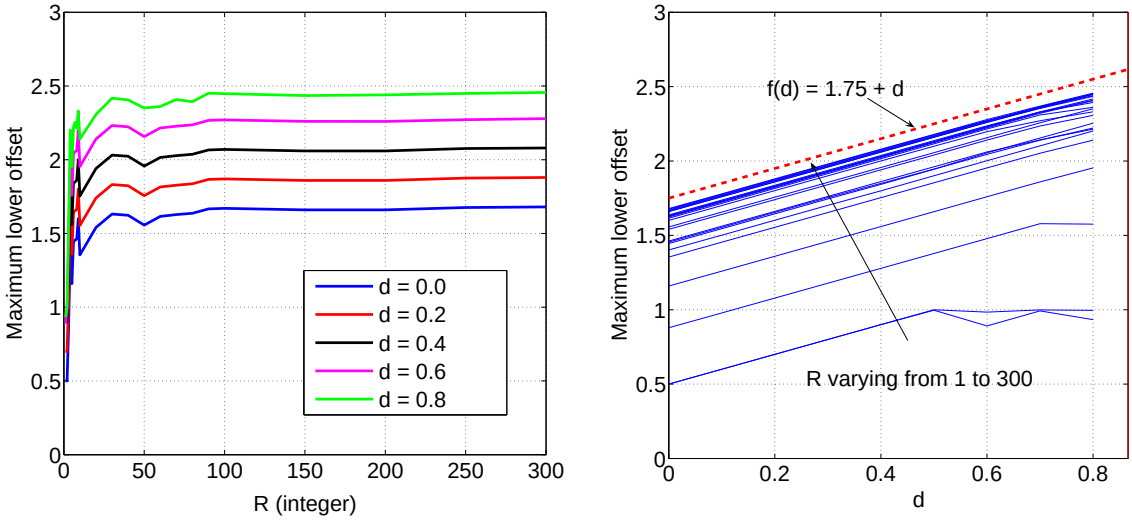
A conservative upper bound for ΔR_L is 2.5, but this can be refined further by examining the variation of ΔR_L with R and d_c, which is shown in Figure 14. Above $R \approx 100$, ΔR_L is almost constant with R. By considering the variation with $d = d_c/\delta$ (which ranges from 0 to $\frac{\sqrt{3}}{2}$), a conservative upper bound for the value of ΔR_L can be found as:

$$\Delta R_L < 1.75 + d \tag{15}$$

The number of additional sphere levels to check can then be conservatively estimated as,

$$n_{extra} = \lfloor d_{min} - R + 1.75 + d \rfloor \tag{16}$$

Statistically, it is observed that most query points (around 80%) require two additional levels to be checked. Approximately 10% require only one level, and about another 10% require three levels. The number of query points requiring no additional level, or more than three, is very small. By tailoring the number of additional spheres to consider for each query point using equation 16, the exact closest surface point can be determined more efficiently.



(a) Variation of maximum lower offset with radius R, for different values of d (normalized distance from query point to center of containing voxel)

(b) Variation of maximum lower offset with d, normalized distance from Query Point to center of containing voxel

Figure 14: Variation of the maximum lower offset with R (sphere radius) and d (distance from the query point to the center of its containing voxel, normalized by the voxel size).

6 Algorithm optimization

This section describes several additions and modifications to the basic algorithm for efficiency optimization.

Once the list of candidate surface elements is established, the distances from the query point to each of them need to be computed to find the true minimum distance. In order to speed up computations, only surface elements whose minimal bounding sphere is closer to the query point than the current minimum distance are considered (the minimal bounding sphere center and radius are pre-computed and stored as a pre-processing step). This is illustrated on an example query point in Figure 15. For this query point, the number of additional levels to be checked is 2. The surface elements for which distances are computed are shown in different color based on the radius of the sphere which tagged them: red for level 0 (first sphere with surface contact), green for level 1, and blue for level 2. In this case, surface elements requiring checking are located in two disjoint areas on the aircraft, one on the wing trailing edge, and another on the side of the fuselage. The actual closest point is located near the wing trailing edge, by the sphere at level 1 (the sphere at level 0 only tags a very small number of surface elements, see detail in Figure 15(a)). The zoomed-in views on the right of the figure identify which of these surface elements require only approximate distance computation (their minimal bounding sphere is further than the current minimum distance). Comparing Figures 15(b) and (c) shows that exact distance computations are required only for a small subset of the surface elements.

To further speed up the computation of the distance between a point and a surface element, a list of characteristics is pre-computed and stored for each surface element, consisting of 9 real numbers (this method is described in [17]). As a result, the cache efficiency of the algorithm can be improved if, instead of computing distances from each query point to its candidate elements, distances are computed from each surface element to its candidate query points. Although this requires the additional step of reversing the list mapping each query point to its candidate surface elements, computing distances between them then requires significantly less amount of memory access, resulting in overall improved efficiency. Furthermore, query points are processed in blocks to mitigate storage requirements for the surface element to query point map.

Another code optimization performed is to re-order the query points by containing voxel: since the spheres to generate are identical for all query points located inside a common voxel (center of the spheres), this pre-processing step is added to avoid repeating computations. Note that further improvements may be

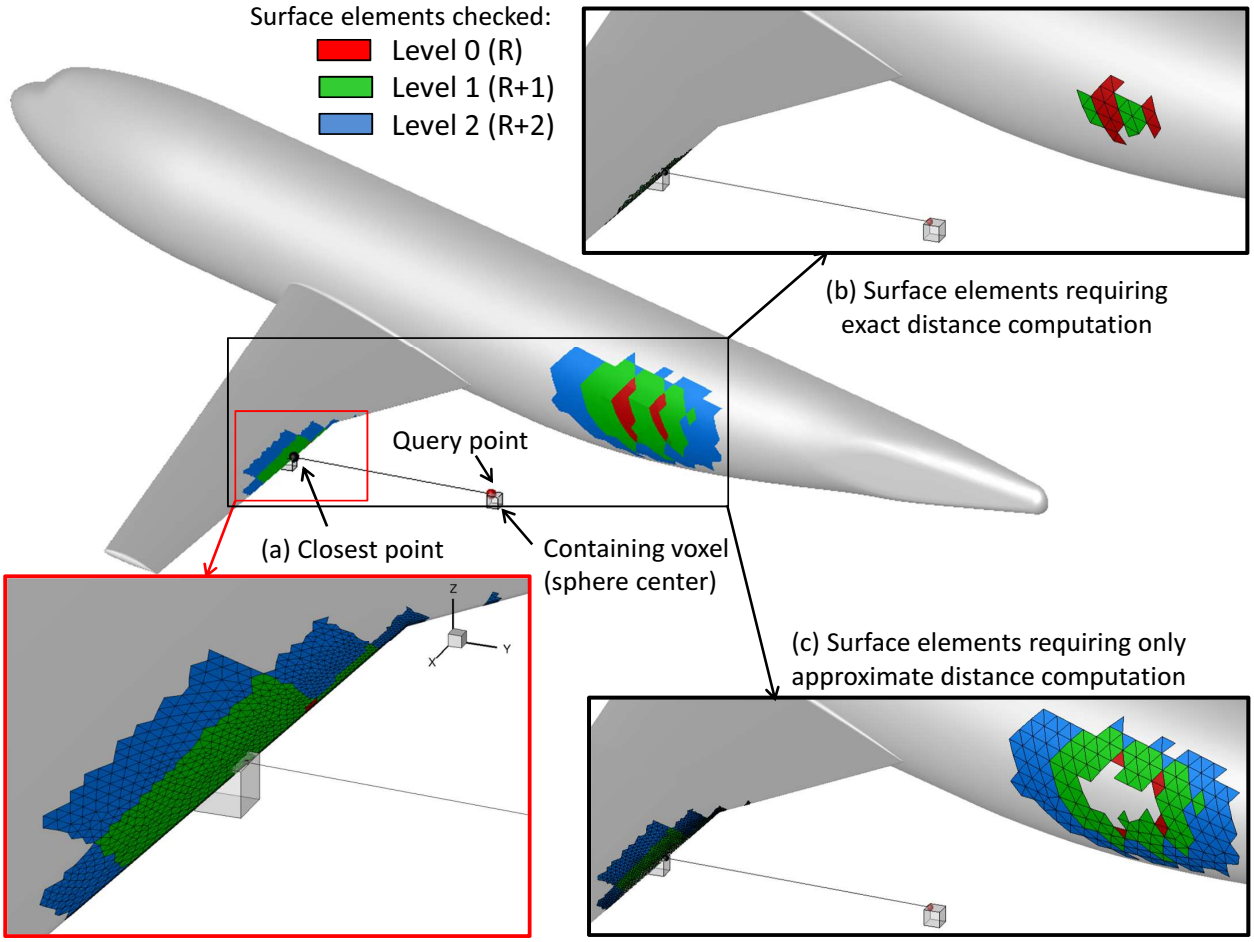


Figure 15: Example of surface elements requiring distance computations.

obtained by using minimum distance results to improve the initial radius guess for neighboring query points (and reduce the number of spheres to build), however this was not attempted here.

The last modification to the basic algorithm is to add a load re-balancing step. In general, query points are distributed among processors so that each receive a similar number of points. However, the resulting average wall distance can vary significantly between processors. For query points very far from the surface, the sphere with initial surface contact will tag a large number of surface elements, especially if the surface has low curvature near the point of contact. Furthermore, building spheres of larger radius is more computationally expensive. This imbalance can be mitigated by redistributing query points among processors. In this work, we adopt a simple approach to load re-balancing: the algorithm is first applied once without any point re-distribution, and the total task duration (processor load) is recorded for each processor. The following method is then used (by all processors) to determine the load fraction each processor should give (or receive) to (or from) which processor(s):

- The load average is first computed, which is the target load for each processor.
- The most loaded processor then donates to the least loaded processor (up to the target load).
- This is repeated until all processors are assigned a load within 10% of the target load.

In order to decide the number of query points to be exchanged, all query points within a processor are assumed to incur the same load. As a result, the number of query points to exchange can be directly computed as the product of the load fraction to be transferred and the total number of query points in the

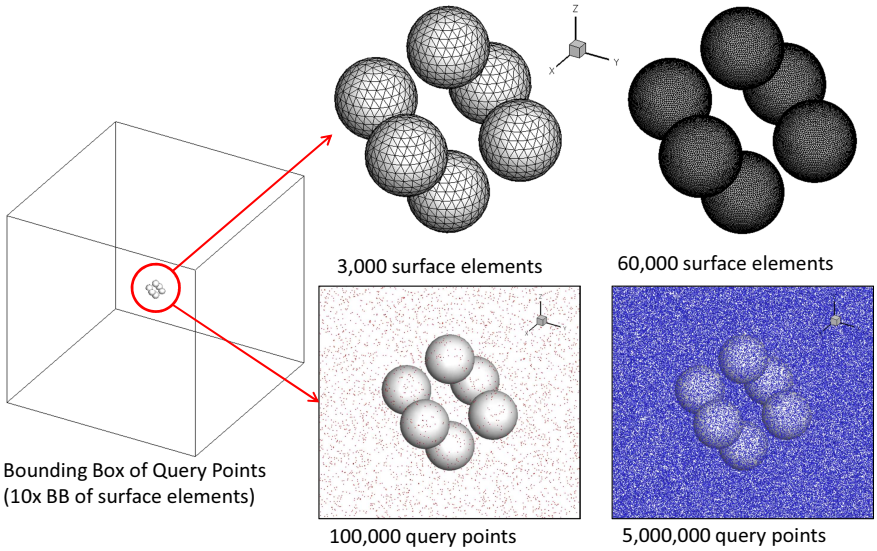


Figure 16: Test case for algorithm analysis: 6 spheres, varying number of surface elements and field points.

donor processor. This assumption can be quite inaccurate for small number of processors. However, this simple load re-balancing algorithm does become more efficient for a larger number of processors. It is worth noting that load re-balancing is critical and most useful for unsteady flow computations where repeated minimum distance searches need to be performed. In this case, processor loads can be estimated on the fly and load balance can be achieved within a few time steps.

7 Results

7.1 Computational complexity

In order to gain insight into the computational cost of the present method, a simple test case is first used where both the number of surface elements and the number of field points (query points) can be independently specified. This test case is illustrated in Figure 16. The surface consists of six unit spheres, uniformly triangulated. The spheres are equally spaced from each other, with their centers located on each axis at ($x = 2$, $x = -2$, $y = 2$, $y = -2$, $z = 2$ and $z = -2$). The field points are randomly distributed in a domain which is ten times larger than the bounding box of the spheres. The empirical order of growth is estimated by studying the variation of the total time required to compute minimum distances with number of surface elements, and number of query points, as shown in Figures 17(a) and (b). The number of surface elements (or boundary faces, N_b) is varied from 3,000 to 60,000, while the number of query points (or field points, N_p) is varied from 100,000 to 5,000,000. From these results, the order of growth follows the power rule for both parameters, i.e. the computational cost is $\mathcal{O}({N_p}^{\alpha}\ {N_b}^{\beta})$. The coefficients α and β are estimated by considering the variation of $log(time)$ with $log(N_p)$ and $log(N_b)$, respectively, and fitting a linear model to the data in a least-square sense. In addition, the value of the coefficients α and β is itself a function of N_b and N_p, respectively. This variation is shown in Figures 17(c) and (d). For large values of N_p and N_b, the complexity of the present method can be said to be $\mathcal{O}({N_p}^{0.8}\ {N_b}^{0.5})$.

7.2 Algorithm performance using DLR-F6 aircraft mesh

The proposed sphere-marching algorithm is then applied to the DLR-F6 aircraft configuration. This aircraft configuration is similar to an Airbus type aircraft [18] and has been used extensively as a test case for the international AIAA CFD Drag Prediction Workshop (DPW) series held since 2002 [19]. The unstructured mesh used in the present work was provided for the second DPW and consists of around 50,000 triangular

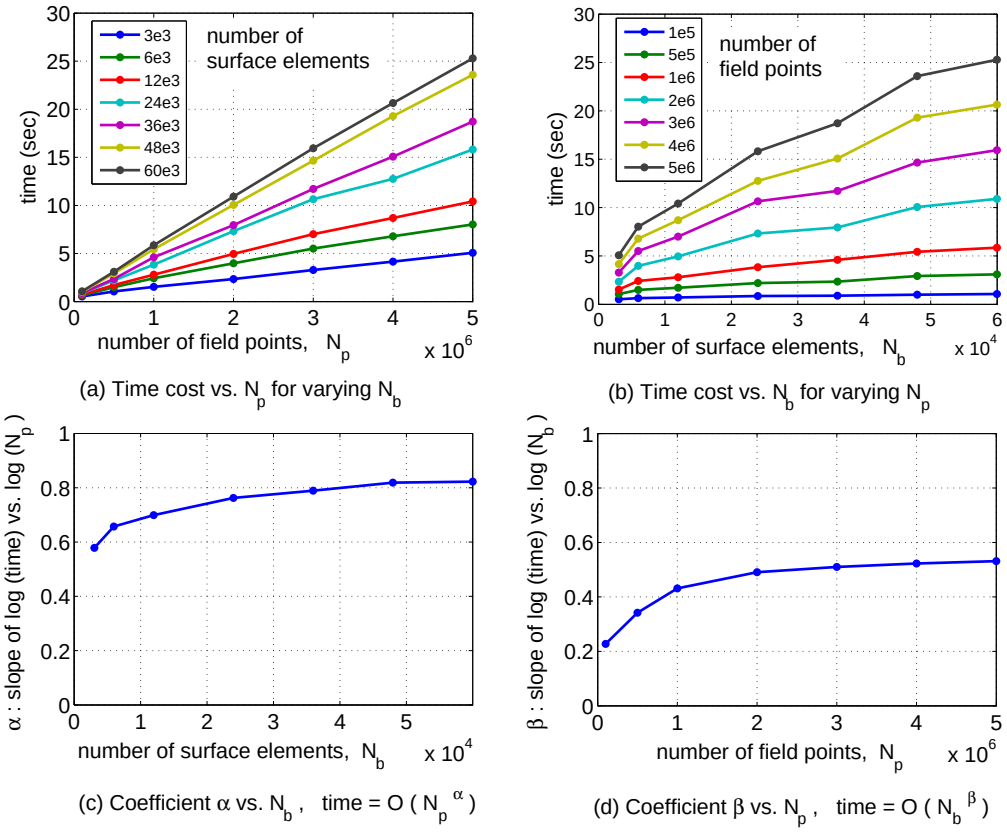


Figure 17: Computational cost variation with number of surface elements and field points.

boundary faces and around 1.2 million field points (query points for the wall distance algorithm). The mesh points are partitioned on multiple processors, while the complete list of surface elements is maintained on each processor. The maximum distance between a field point and the aircraft surface is around 8,500 times the average length of a surface element ($\bar{l}$), and the average distance is around $24\,\bar{l}$. More than half the field points are located less then $\bar{l}$ away from the surface, while only 1% are located further than $200\,\bar{l}$ away.

Figure 18(a) shows the time required to perform wall distance computations, for each processor, for varying number of processors used (8 to 256), before load re-balancing. It is clear that the load is initially quite imbalanced among processors. This is again because while each processor is assigned a similar number of query points, the average wall distance varies considerably between processors. In Figure 18, algorithm tasks have been divided into four main sub-tasks:

1. pre-processing,
2. sphere building and candidate surface element identification,
3. reversing the lists (to obtain the map of query points for each surface element), and
4. distance computations.

The cost of pre-processing operations (in black) is very small compared to that of other tasks. In general, it is observed that for the bottleneck processor, the time required to build spheres and identify candidate surface elements is similar to the time required for actual distance computations. Figure 18(b) shows the time cost per processor after load re-balancing is performed. The total time required is significantly reduced, owing to a better balance between processors (less then 2 seconds for 256 processors compared to 7.5 seconds before re-balancing). As expected, the performance of the re-balancing algorithm is not satisfactory for a small number of processors, but improves as the number of processors increases.

Table 1: Computational time (in seconds) to find minimum distances for the DLR-F6 mesh using a varying number of processors: comparison of different methods.

Number of processors	64	128	256
Exhaustive search	95.83	52.08	27.74
Octree method	51.77	35.12	22.89
Eikonal equation method	44.17	25.03	12.32
Marching spheres (no load balance)	16.68	11.89	7.27
Marching spheres (with load balance)	6.28	3.46	1.87

Figure 19 and Table 1 compare the performance of the present algorithm with alternate methods for a varying number of processors. The alternate methods compared are (a) octree based search (b) PDE based method that solves the discrete Eikonal/Poisson equations and (c) exhaustive search. All of these methods are available within the pre-processor suite of HELIOS [20] framework and developed by Mavriplis [21]. Even before load re-balance, the marching-spheres algorithm yields the shortest time for all number of processors tested (4 to 10 times faster than exhaustive search). The advantage becomes less apparent for large number of processors, partly because of the load imbalance. Figure 20(a) and (b) compare the marching spheres algorithm before and after load re-balancing, in terms of total time cost and speed-up, respectively. It is clear that load re-balancing must be performed in order to achieve satisfactory scalability. The re-balanced marching-sphere algorithm is 12 to 17 times faster than exhaustive search.

7.3 Wall distance contours

Finally, Figure 21 shows minimum distance contour lines obtained using the present algorithm applied on the DLR-F6 aircraft configuration as well as the MDART rotor-hub configuration. True minimum distance is a continuous single valued function throughout the domain as illustrated by the smooth contour lines. The iso-surface with a value of zero minimum distance recovers the original tesselation, whereas an iso-surface with a fixed non-zero value follows a smoothened shape of the original tesselation. The MDART rotor-hub configuration is composed of two overset unstructured grids with about 3.5 million grid nodes. This case is presented to highlight the modular nature of the present method which facilitates direct application to an overset grid system–i.e, only minimal amount of information, namely surface tesselation and a cloud of points are required for the minimum distance search. Figure 21(d) and (e) show details of the wall distance contour lines around the hub. Note that the contour lines in the regions of grid overlap are consistent between both grid systems. Furthermore, owing to the relatively short extent of the unstructured meshes in this case, the sphere voxelization method is found to be almost two orders of magnitude faster than the exhaustive search.

8 Conclusion and Future Work

This paper presented a new approach to compute the minimum distances between a set of query points and a tesselated surface using a method of voxelized marching spheres. Sphere voxelization is performed by extending the two-dimensional mid-point circle voxelization algorithm to three dimensions. The method is high performance computing compatible and very modular, i.e. it requires only the surface tesselation and a cloud of points as input. While the primary application for this method is minimum wall distance computations (in the context of turbulence modeling), several other related geometric problems such as nearest neighbor search, mesh generation, overset grid assembly, contact detection, etc. constitute potential areas of application.

Using a simple test case where the number of field points (N_p) and surface elements (N_b) can be independently specified, the present method is found to be $\mathcal{O}({N_p}^{0.8}\ {N_b}^{0.5})$.

When applied to a 1.2 million-node (50,000 surface elements) mesh around an aircraft configuration, the new approach shows improved efficiency compared to all of the alternate methods for minimum distance computations, namely: the octree method, the Eikonal equation method and the exhaustive search method.

Scalability is not satisfactory for the basic algorithm. However, including a load re-balancing step substantially mitigates this problem with almost 229 times speed-up obtained when using 256 processors. Further, the rebalanced algorithm is observed to be 12-17 times faster (on all processor counts) than the exhaustive search for the test case considered. Consistent and accurate wall distance prediction is also demonstrated for a more complex rotor-hub test case (3.5 million nodes) that is composed of multiple overlapping unstructured grids.

Several modifications such as (a) accuracy improvement to circle pixelization algorithm (b) partial sphere voxelization (c) improvement in cache efficiency (d) determination of a rule for optimal voxel size (d) efficient true minimum distance checks and (e) load re-balancing were formulated and implemented to achieve the observed efficiency improvement. Work is currently under way to explore the following additional areas for efficiency improvement:

1. Improving efficiency of the sphere building algorithm, using the full 48-way symmetry of the sphere.
2. Removing circle storage requirement by computing partial sphere voxels on-line.
3. Sorting query points by proximity to improve guess quality for initial sphere radius,
4. Improving load re-balancing algorithm using an estimated time cost for each query point.
5. Using on-line sphere voxel size optimization.

References

[1] E. van der Weide, G. Kalitzin, J. Schluter, and J.J. Alonso. Unsteady Turbomachinery Computations Using Massively Parallel Platforms. 44th AIAA Aerospace Sciences Meeting and Exhibit, AIAA Paper 2006-0421, Reno, NV, AIAA, 2006.

[2] A.D. Boger. Efficient Method for calculating Wall Proximity. *AIAA journal*, 39(12):2404–2406, 2001.

[3] J.A. Sethian. Fast Marching Methods. *SIAM review*, 41(2):199–235, 1999.

[4] P.G. Tucker. Differential equation-based wall distance computation for DES and RANS. *Journal of Computational Physics*, 190(1):229–248, 2003.

[5] J. Xu, C. Yan, and J. Fan. Computations of Wall Distances by Solving a Transport Equation. *Applied Mathematics and Mechanics*, 32(2):141–150, 2011.

[6] R. Lohner, D. Sharov, H. Luo, and R. Ramamurthi. Overlapping Unstructured Grids. AIAA 2001-0439, Reno, NV, AIAA, 2001.

[7] J. Sitaraman, M. Potsdam, B. Jayaraman, A. Datta, A. Wissink, D. Mavriplis, and H. Saberi. Rotor Loads Prediction Using Helios: A Multi-solver Framework For Rotorcraft CFD/CSD Analysis. In *AIAA Aerospace Sciences Meeting and Exhibit*, pages AIAA Paper 2011–1123, Orlando, FL, January 2011.

[8] A. Wissink, A. Katz, and J. Sitaraman. PICASSO : A Meshing Infrastructure For Strand-Cartesian CFD Solvers. In *30th AIAA Applied Aerodynamics Conference*, pages AIAA–2012–2916, New Orleans, Louisiana, June 2012.

[9] G. Zagaris, M.T. Campbell, D.J. Bodony, E. Shaffer, and M.D. Brandyberry. A Toolkit for Parallel Overset Grid Assembly Targeting Large-Scale Moving Body Aerodynamic Simulation. Proceedings of the 19th International Meshing Roundtable, pp.385-401, Sandia National Laboratories, 2010.

[10] M. Khoshniat, G.R. Stuhne, , and D.A. Steinman. Relative Performance of Geometric Search Algorithms for Interpolating Unstructured Mesh Data. 6th International Conference on Medical Image Computing and Computer Assisted Intervention, Montreal, QC, MICCAI, 2003.

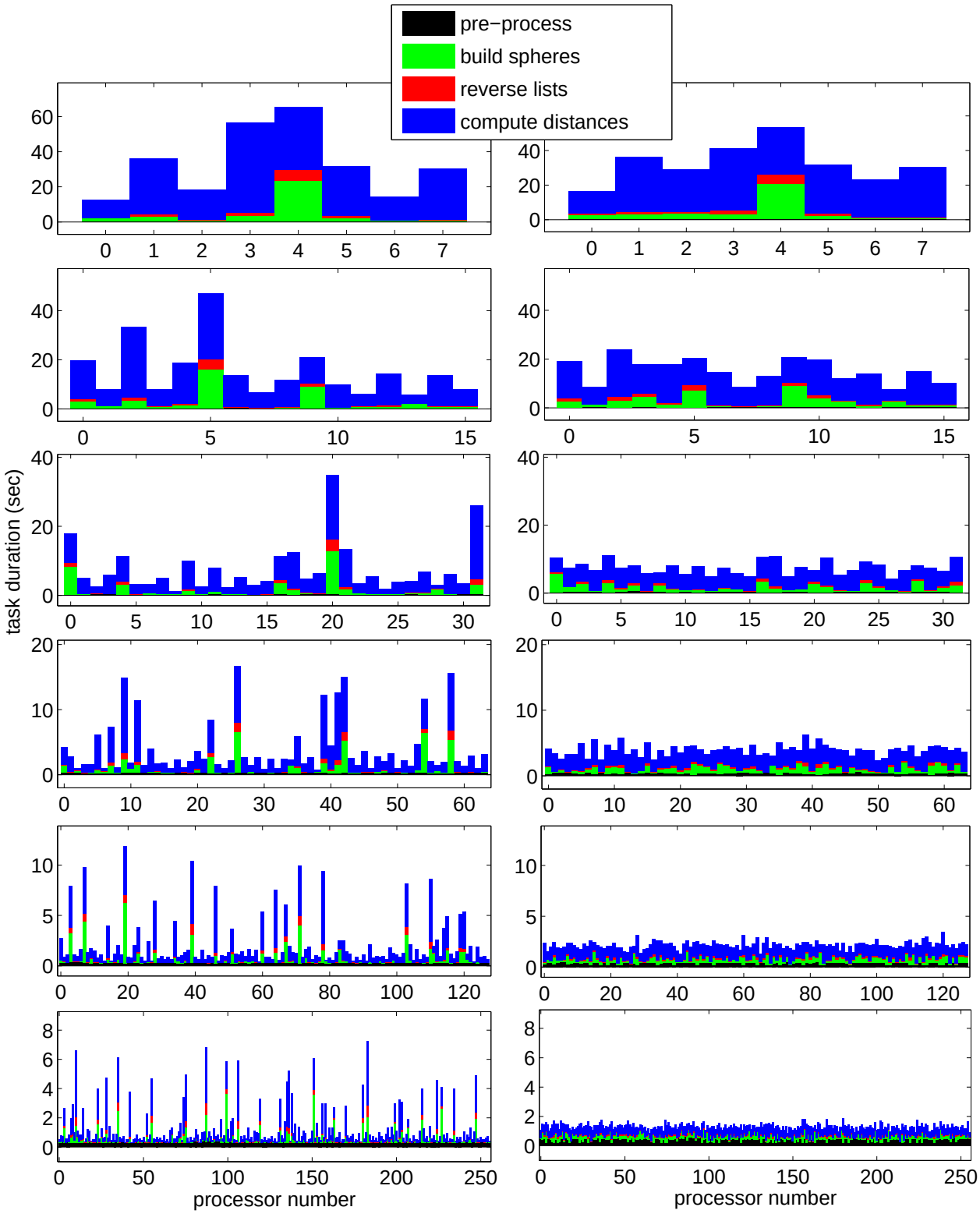
[11] S. Pissanetzky and F.G. Basombrio. Efficient Calculation of Numerical Values of a Polyhedral Function. *International Journal of Numerical Methods in Engineering*, 21:1–17, 1991.

[12] J. Sitaraman, M. Floros, A.M. Wissink, and M. Potsdam. Parallel Unsteady Overset Mesh Methodology for a Multi-Solver Paradigm with Adaptive Cartesian Grids. 26th AIAA Applied Aerodynamics Conference, Honolulu, Hawaii, AIAA, 2008.

[13] C. Montani and R. Scopigno. Spheres-to-Voxels Conversion. In Andrew S. Glassner, editor, *Graphics Gems*, pages 327–334. Academic Press, 1990.

[14] J.E. Bresenham. Algorithm for Computer Control of a Digital Plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[15] M.L.V. Pitteway. Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter. *Computer Journal*, 10(3):282–289, 1967.
[16] J.R. Van Aken. An Efficient Ellipse Drawing Algorithm. *CG & A*, 4(9):24–35, 1984.
[17] P.J. Schneider, , and D. Eberly. *Geometric Tools for Computer Graphics*. Morgan Kaufmann, 2002.
[18] O. Brodersen and A. Sturmer. Drag Prediction of Engine-Airframe Interference Effects using Unstructured Navier-Stokes Calculations. 19th AIAA Applied Aerodynamic Conference, Anaheim, CA, AIAA, 2001.
[19] AIAA CFD Drag Prediction Workshop. DPW Website. [http://aaac.larc.nasa.gov/tsab/cfdlarc/aiaa-dpw/], 2012.
[20] A. Wissink, J. Sitaraman, M. Potsdam, V. Sankaran, Z. Yang, and D. Mavriplis. A Coupled Unstructured Adaptive Cartesian CFD Approach For Hover Prediction. In *66th Forum of the American Helicopter Society*, Phoenix, AZ, May 2010.
[21] D. J. Mavriplis. Viscous Flow Analysis using a Parallel Unstructured Multigrid Solver. *AIAA Journal*, 38:2067–2076, 2000.



(a) Before load re−balance

(b) After load re−balance

Figure 18: Load distribution for each task for 8, 16, 32, 64, 128, and 256 processors

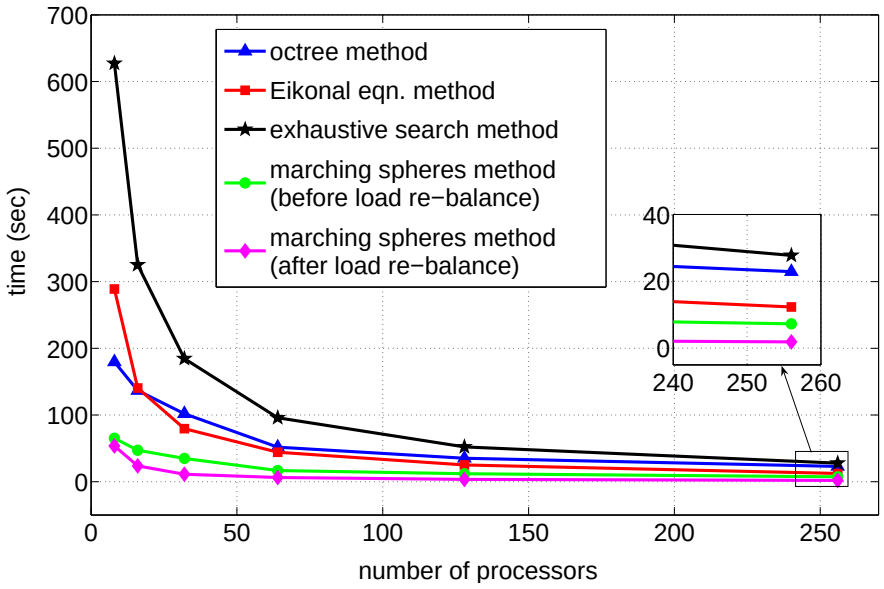
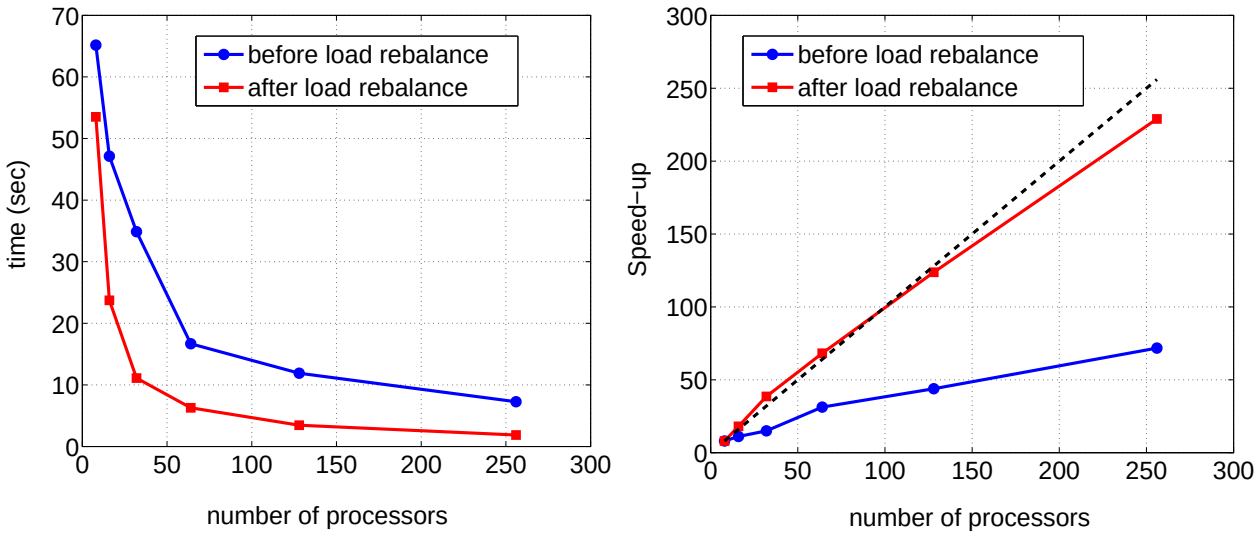


Figure 19: Performance comparison between different methods to compute minimum distances.



(a) Time vs. number of processors

(b) Speed−up vs. number of processors

Figure 20: Performance comparison for the present algorithm before and after load re-balancing.

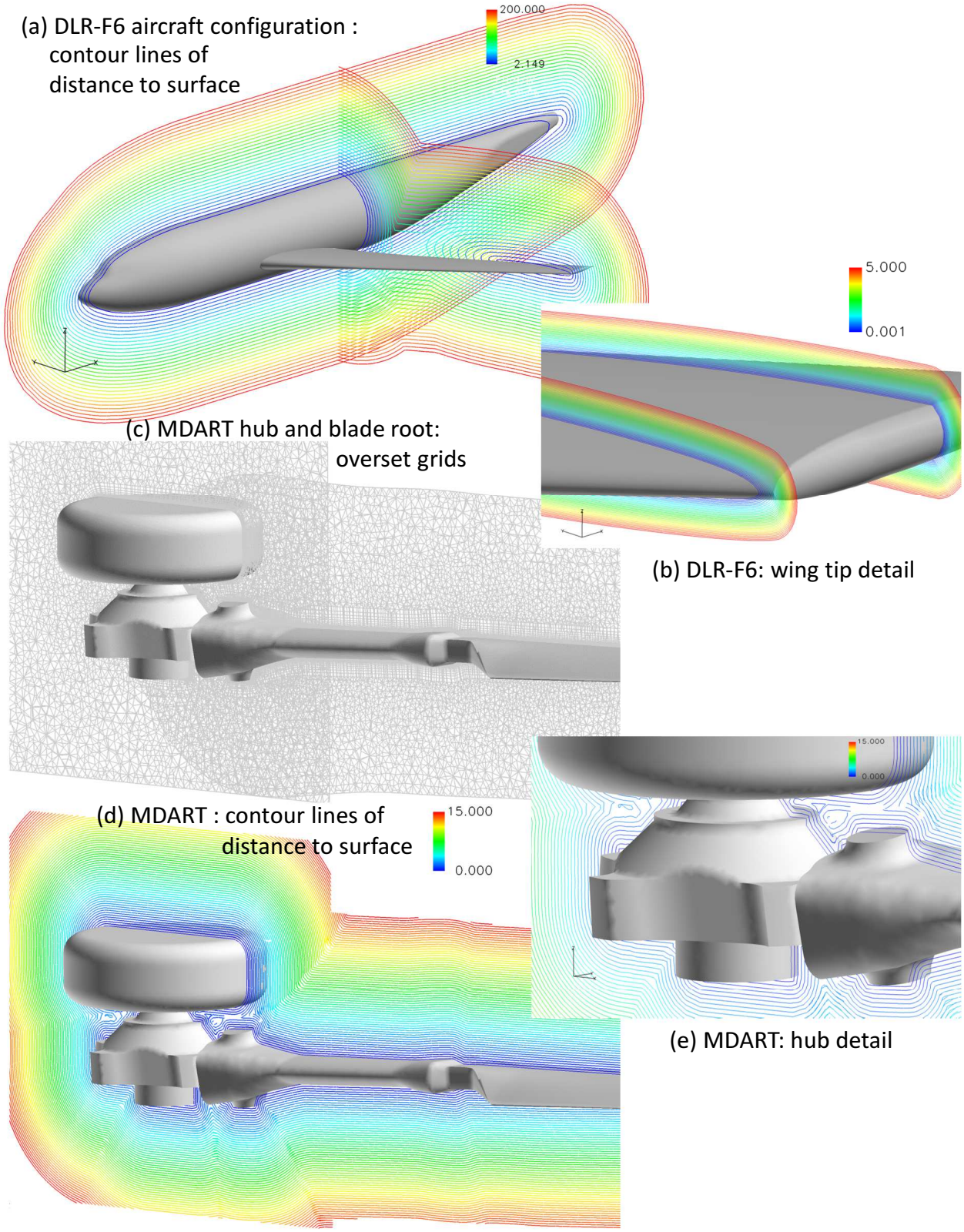


Figure 21: Minimum distance contours obtained for the DLR-F6 configuration and the MDART rotor hub.