Cassiopee: a CFD pre-and post-processing tool

C. Benoit, S. Péron, S. Landier

▶ To cite this version:

C. Benoit, S. Péron, S. Landier. Cassiopee: a CFD pre-and post-processing tool. 2015. hal-01141585

HAL Id: hal-01141585
https://hal.science/hal-01141585

Preprint submitted on 13 Apr 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

Cassiopee: a CFD pre- and post-processing tool

Christophe Benoit*, Stéphanie Péron*, Sâm Landier*

ONERA - The French Aerospace Lab, F-92322 Châtillon, France

Abstract

This paper presents an overview of the capabilities of a new open-source pre- and post-processing tool for Computational Fluid Dynamics simulations, called Cassiopee. Its architecture, which is basically a set of Python modules, and the handled data, which is based on CGNS standard, are described. Some examples of workflows that can be built with Cassiopee functions are provided. Finally, some applications of Cassiopee functions to realistic CFD configurations are briefly presented.

Keywords: Computational Fluid Dynamics, Pre-processing, Post-processing, CGNS, Open source software

1. Introduction

In 2008, ONERA started a project to gather pre- and post-processing tools in a single software, called Cassiopee, following other initiatives as Salome [1] or gmsh [2]. At that time, pre- and post-processing tools were spread among scientists, each one generally having its file formats and data representations. Thus, it seemed clear that capitalizing all the knowledge in a single software environment and making it interoperable was worth the effort. However, this effort pays off only if the common data representation is well accepted and easily accessible to users.

For this reason, the Python language was chosen as a high level interface, since it is very fast to start with, easy to use and is spread in the scientific community (scipy, matplotlib, ...).

*Corresponding author

Email addresses: christophe.benoit@onera.fr (Christophe Benoit), stephanie.peron@onera.fr (Stéphanie Péron), sam.landier@onera.fr (Sâm Landier)

To simplify the use of Cassiopee, two interfaces are provided: the first one, that we call the "array" interface, is directly based on numpy arrays. In this way, Cassiopee functions can be easily mixed with functions using the numpy library.

The second interface is based on the Python representation of the CFD General Notation System (CGNS) of data [3]. The data handled by Cassiopee functions is then an imbricated set of lists describing a computation tree, compliant with the CGNS standard, as described by Poinot [4]. We call this interface the "pyTree" interface.

Functions are classified in thematic Python modules, according to their features. Each Python module is independent and can be compiled and installed separately from the others.

Currently, among all the available functionalities, one can for example perform minor mesh modifications, mesh improvements (e.g. mesh smoothing), preprocessing of a CFD computation (e.g. connectivity computation or mesh splitting), code coupling or solution post-processing. These modules are commonly used for CFD and CAA applications.

In 2013, Cassiopee modules were released under the GNU General Public License GPL3, in the hope that it will be useful to the CFD community [5].

2. Design choices

As previously presented, the pyTree is one of the two choices of data representations available in Cassiopee. From now on, we will focus on this interface only. A pyTree is a hierarchical data set where each node is a Python list of this type:

```
['name', v, [], 'Type']
```
(1)

The string `'name'` is simply the name of the node, `v` is a numpy array defining the value of the node (for instance, for the first component of the grid coordinates, say 'CoordinateX', the numpy array stores all the x-coordinate values for the corresponding zone), the string `'Type'` is a CGNS-compliant name and describes the node type (e.g. `'Zone_t'` for a zone node). The third element of the Python list is `[]` and define the list of nodes that are the current node's children.

This structure enables to store the mesh coordinates (in a node named 'GridCoordinates'), the flow solution (located at nodes and at centers), bound-

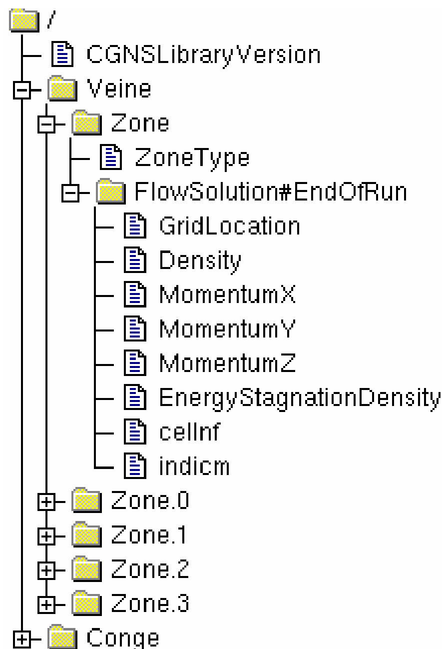


Figure 1: Example of a CGNS/Python (or pyTree) representation.

ary conditions (in a node of name 'ZoneBC' for each zone) or the grid connectivity. An example of a representation of a pyTree is displayed in figure 1.

This data structure is common to all the modules of Cassiopee. In addition, the philosophy of Cassiopee modules is purely functional, hence a Cassiopee function can be written as:

$$b = f(a) \text{ or } _f(a) \tag{2}$$

where a and b are pyTrees. Function f returns a copy b of the pyTree, whereas function prefixed with a "_" results in an in-place modification of the pyTree (the given pyTree is directly modified by the function $_f$). For the sake of generality, we try (as far as possible) to make all the Cassiopee functions deal with all mesh types: structured meshes, unstructured meshes with basic elements, and polyhedral meshes, as displayed in figure 2. Element names are also defined according to the CGNS standard: for instance a 'NGON/NFACES'-type mesh describes a general polyhedral mesh.

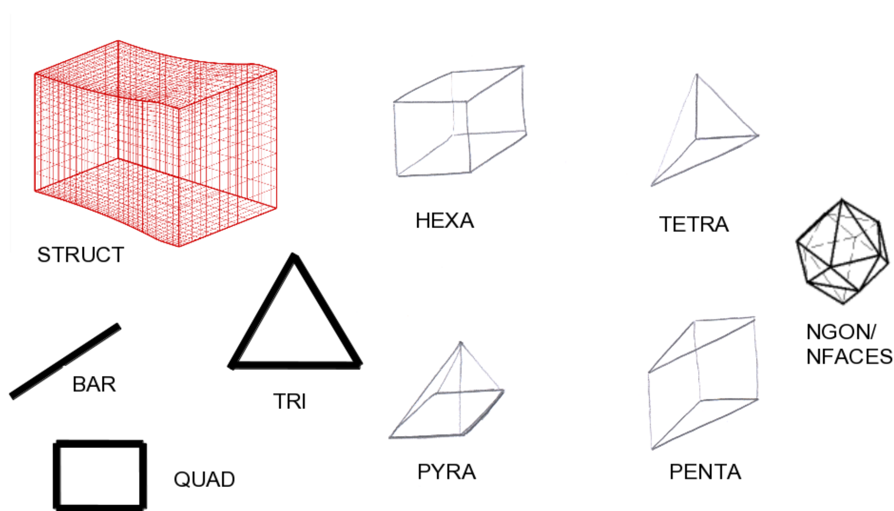


Figure 2: Mesh types compliant with Cassiopee functions.

Another original feature of Cassiopee functions is that boundary conditions and solutions (defined both at nodes and centers) are also modified by the functions wherever this is relevant.

For example, a simple "subzoning" function, that extracts a subzone from a zone, can be applied on structured grids (using min-max indices in the three directions of the zone) or unstructured grids (given a list of element indices). For instance, when a subzone is created, not only the grid coordinates are extracted, but also the solutions at nodes and cell centers and the boundary conditions as well.

3. Content of Cassiopee modules

The software is made of independent Python modules. Each module can be compiled and installed separately. The release R-3 of Cassiopee contains thirteen modules [5], including `Converter`, `Geom`, `Generator`, `Transform`, `Connector`, `Post`, described briefly in the following.

3.1. Converter module

`Converter` module enables input/output between the in-memory data representation and various file formats of discrete data, such as Tecplot,

plot3d, mesh, STL, OBJ... and of course CGNS/ADF and CGNS/HDF5 formats. `Converter` module also provides low-level functions to handle pyTrees, such as pyTree node creation/removal or specific nodes search. Basic functions are also implemented to add boundary conditions to a zone, to initialize a solution, to convert a solution from centers to nodes and vice versa.

3.2. Geom module

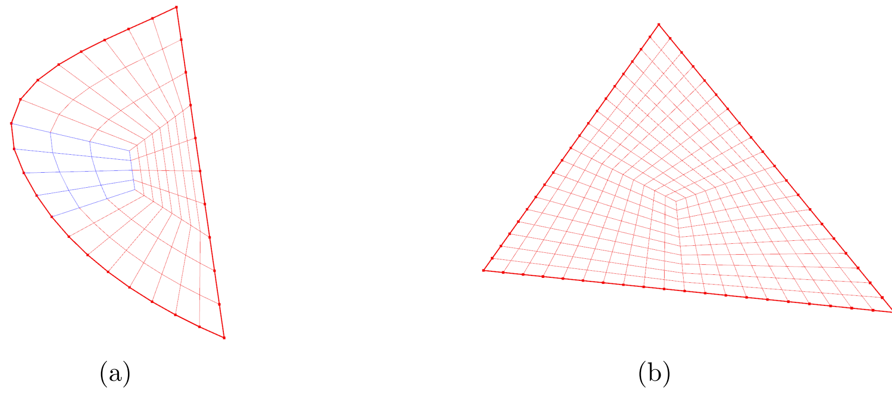
`Geom` module is used to generate specific elements or surfaces, such as points, lines, Bézier surfaces [6], spline surfaces, parametric surfaces...

3.3. Generator module

`Generator` module provides mesh generation functions. Classical transfinite interpolation (TFI) from quadrangular mesh boundaries [7] is available. A function that makes use of TFI and automatically performs a suitable decomposition of the boundaries is also available to build structured meshes starting from different boundary shapes. Some examples of 2D TFI meshes starting from a half-moon curve and a triangle describing the mesh boundaries are displayed in figure 3. A generalization of the TFI to generate unstructured meshes is also implemented, according to Perronnet algorithm [8]. Combined with a projection technique onto a surface, this method can be used to generate structured patches on a surface that is not too distorted, as displayed in figure 4.

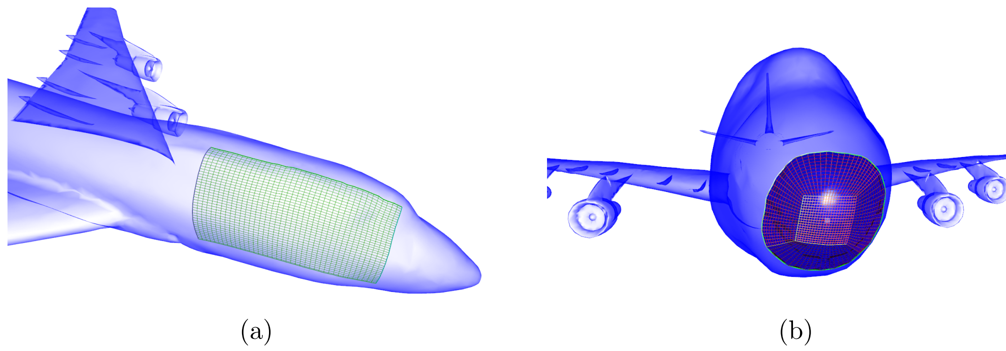
A basic algorithm of algebraic extrusion starting from a discretized surface mesh is also proposed. It is based on an iterative averaging of neighbouring normals at a given point. It works on any type of surface meshes, defined by a single zone or a set of abutting zones, as shown in figure 5. This algorithm enables to generate a surface mesh by an orthogonal walk onto a surface S, combined with a projection onto S at each step. In figure 6-(a), a O-type 2D structured mesh is generated starting from a closed curve onto a surface and the surface.

Finally, such basic mesh generation functions can be assembled to create more complex mesh functions, such as the generation of collar grids, which are structured grids at the junction of two surfaces [9]. Figure 6-(b) provides an example of a collar grid at the junction between a helicopter fuselage and one of its horizontal tail planes (HTP). The orthogonal walk function is used to create the surface mesh onto the fuselage and the HTP. Then the volume mesh is obtained here by transfinite interpolation.



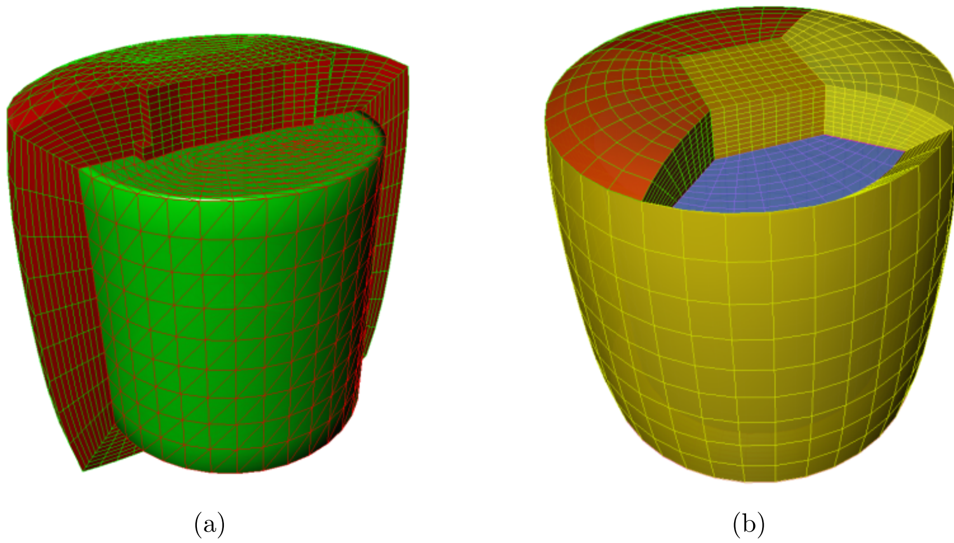
(a) (b)

Figure 3: Simple TFI meshes: (a) bounds defined by a half-moon shape, (b) bounds defined by a triangle.



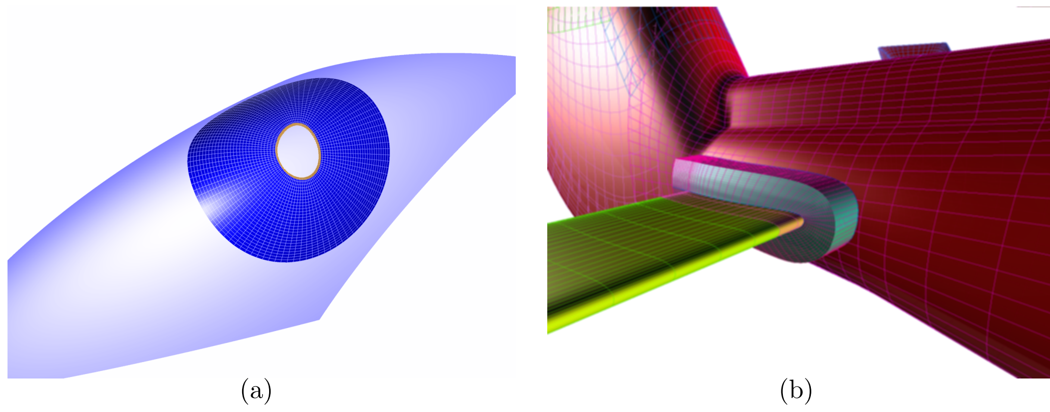
(a) (b)

Figure 4: Projected TFI meshes: (a) bounds defined by a quadrangle, (b) bounds defined by a circle.



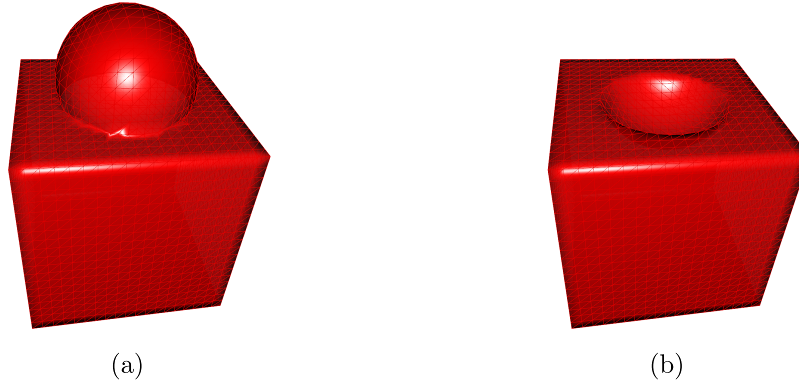
(a) (b)

Figure 5: Normal extrusion from (a) a triangular surface, (b) a set of structured 2D abutting grids.



(a) (b)

Figure 6: (a) Orthogonal walk onto a surface starting from a curve resulting in a surface mesh (in blue) (b) generation of a collar grid at a fuselage/HTP junction.



(a) (b)

Figure 7: Surfaces resulting from (a) a boolean union and (b) a boolean difference between a cube and a sphere.

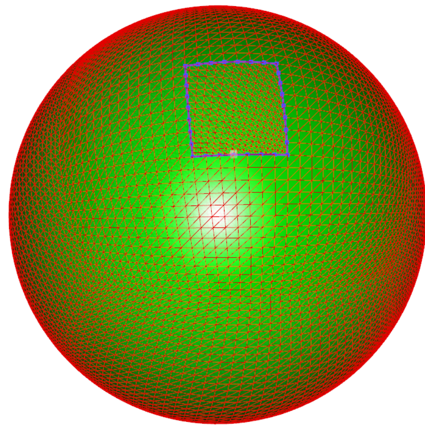
In the context of triangular surfaces, `Generator` module provides boolean operators (union, difference, intersection) [10] applied to a pair of triangular surface meshes. Figure 7 represents the resulting meshes after a boolean union and a boolean difference between a cube and a sphere, described by triangular zones.

A remeshing algorithm depending on a metrics [11] based on Delaunay triangulation is also available for triangular meshes. This algorithm can be combined with a spline surface fitting to fix holes in surfaces, as shown in figure 8.

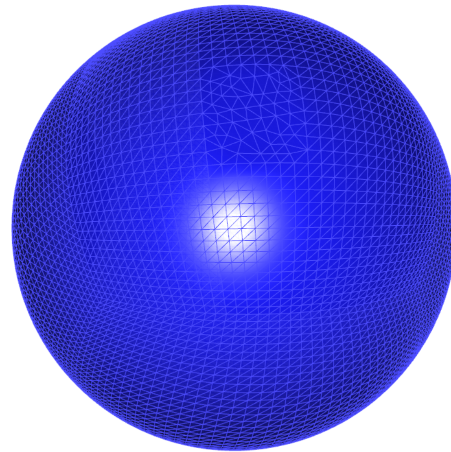
Octree mesh generation [12] starting from a set of surfaces is available as a `Generator` function. An octree mesh adaptation function enables to refine or coarsen elements according to an indicator. These functions can be used to generate adaptive set of Cartesian grids that are abutting or fully overlapping (AMR-type), as described in reference [13]. An example of an octree mesh generated around a helicopter fuselage and its strut is presented in figure 9-(a). The set of Cartesian grids that are derived from this octree are displayed in figure 9-(b).

3.4. Transform module

The `Transform` module gathers functions that operate on mesh coordinates such as translation, rotation, homothety... Some split functions are implemented for structured grids in this module. They can be used for parallel load balancing of a CFD computation or for post-processing purposes.

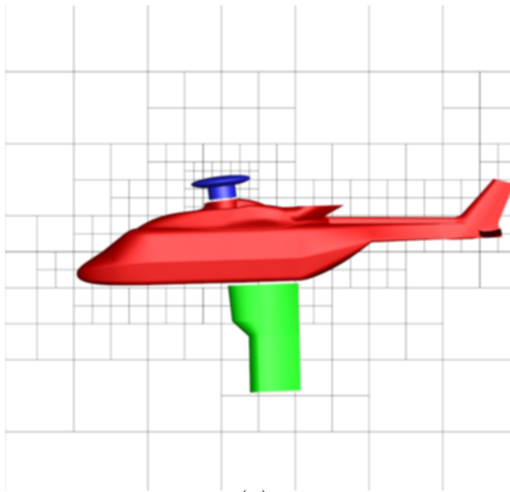


(a)

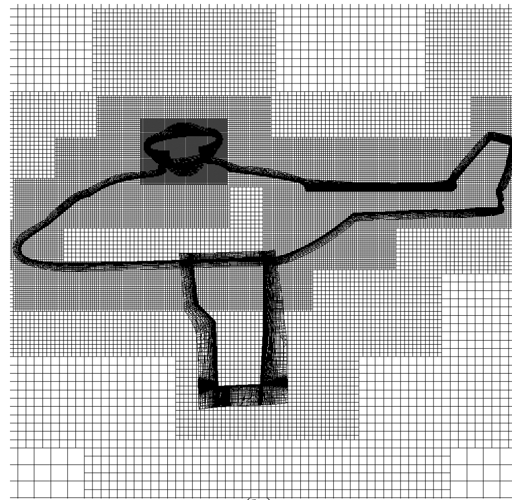


(b)

Figure 8: (a) Hole to be fixed in a sphere; (b) fixed surface.



(a)



(b)

Figure 9: (a) Octree mesh around a helicopter fuselage; (b) corresponding Cartesian grids.

Figure 10 displays a 2D mesh around a NACA0012 profile that is split according to the maximum number of points required in each zone. For this case, if the initial pyTree contains boundary conditions, flow solution or connectivities, they are also subzoned by the split function.

Reverse algorithms enable to join pairs of structured grids or to merge all grids automatically in a minimum number of blocks, according to the algorithm of Rigby [14] (figure 11).

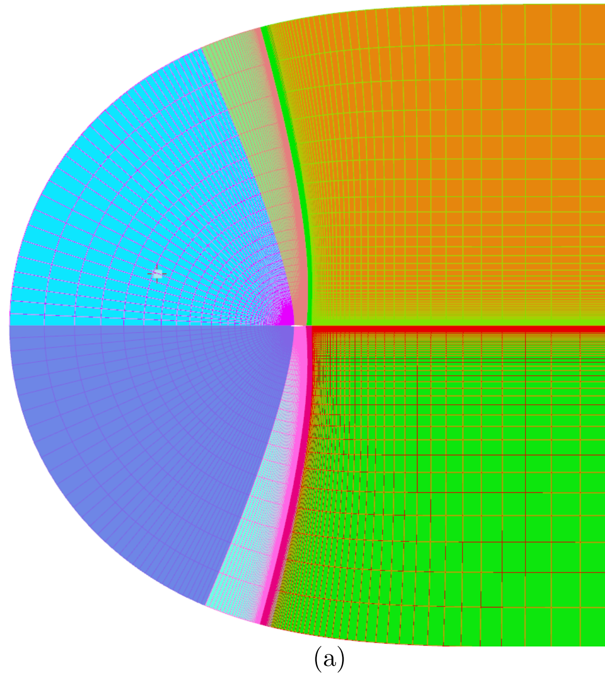
Finally, structured or unstructured meshes can be smoothed by functions using Laplace/umbrella operators or Taubin smoothing [15]. Note that the smoothing functions work for a set of abutting grids: when the function is applied to a set of abutting grids, the borders are still abutting after smoothing, as displayed in figure 12.

3.5. Connector module

`Connector` module contains three main capabilities: a hole-cutting function, the computation of 1-to-1 (and more generally n-to-m) abutting connectivity and the computation of overset grid connectivity.

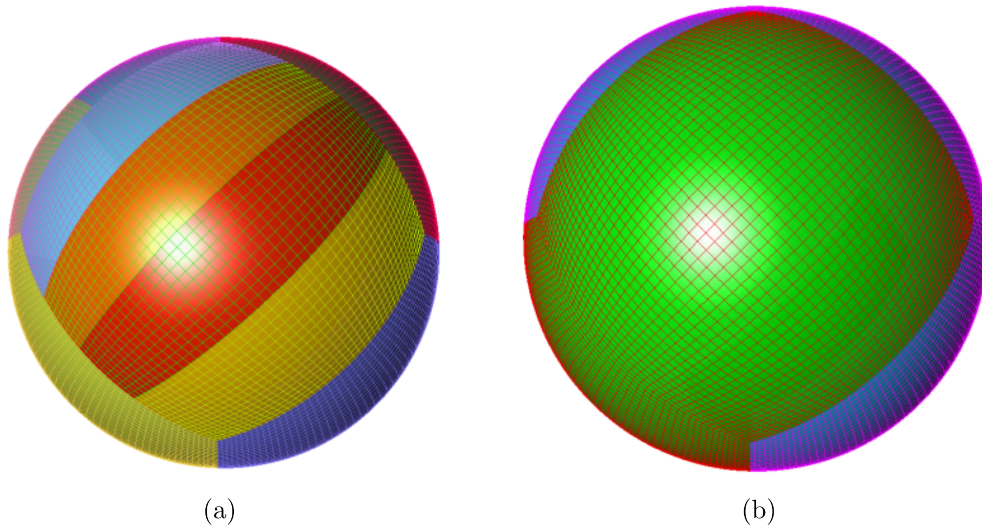
Hole-cutting or blanking is the action of determining mesh points that are inside a solid. Two blanking algorithms exist: the first one is based on the Object X-Rays technique [16] introduced by Meakin, for which the solid is defined by a closed surface mesh; the second one is based on in-tetra tests: points are blanked if they are inside a tetra cell. In that case, a 3D Tetra mesh of the volume inside the solid is required. This mesh can be generated by a 3D Delaunay mesh generation starting from a discrete representation of the solid surface. A view of a Cartesian mesh blanked by a triangular surface mesh defining a human body, using the X-Ray technique is displayed in figure 13-(a).

The computation of the 1-to-1 abutting connectivity on a set of structured grids is performed automatically. It is achieved in three steps: first, a k-d tree is built [17] in which are stored the centers of the borders of a pyTree that are not already defined by a boundary condition or a grid connectivity. Matching centers are sought in the k-d tree and are then tagged by pairs. Finally, points are gathered into structured patches that define the abutting connectivity between the corresponding zones, as displayed in figure 13-(b).



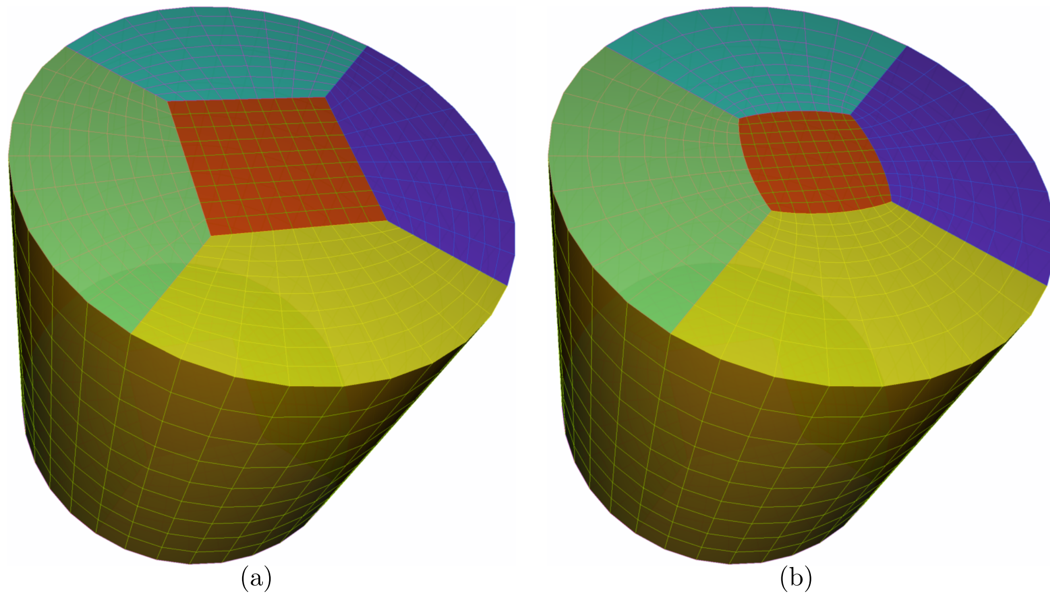
(a)

Figure 10: Split of a structured 2D mesh around a NACA0012 profile.



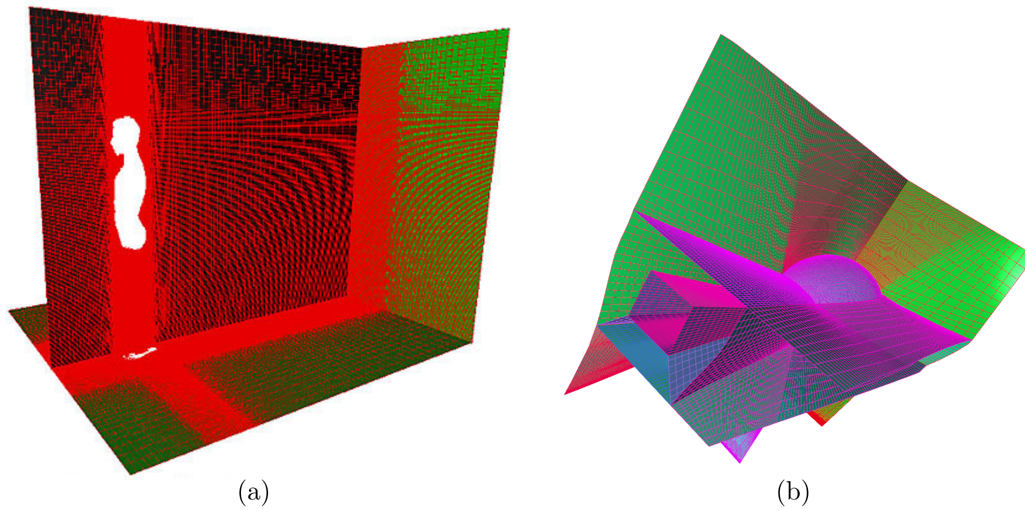
(a) (b)

Figure 11: Set of 2D structured grids (a) before and (b) after automatic block merging.



(a) (b)

Figure 12: Structured set of surface grids (a) before and (b) after smoothing.



(a) (b)

Figure 13: (a) Blanking of a Cartesian mesh inside a surface describing a human body; (b) view of abutting borders in a mesh.

Overset grid assembly is broken down into two major functions: overlap optimization and computation of the computation of overset grid connectivity.

The overlap optimization function aims at removing some points in overlapping regions in order to minimize the remaining overlap. The algorithm is based on the one developed in [18]. In `Connector` module, it is further split into two functions:

- first, `Connector.optimizeOverlap` results in marking points that can be removed in overlapping regions as interpolated, depending on a quality criterion between donor and receptor points.

- then, `Connector.maximizeBlankedCells` maintains d layers of interpolated points, the other points marked initially as interpolated are marked as blanked points.

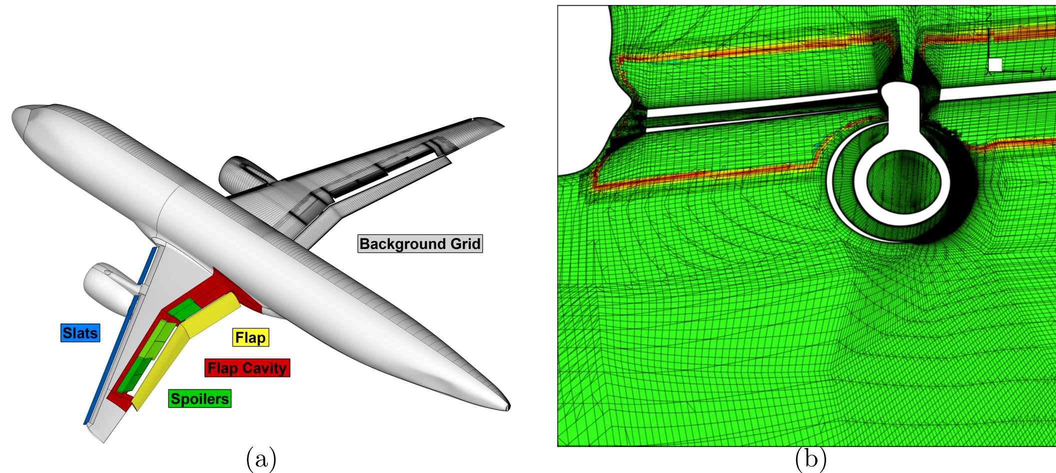
The computation of the overset grid connectivity is achieved by the function `Connector.setInterpData`: receptor and donor points, interpolation coefficients, interpolation method and the location of interpolated points are stored in the pyTree as a `ZoneSubRegion_t`-type node for each pair of donor-receptor zones. Different interpolation methods are available:

- the 2nd-order interpolation works for structured and Tetra donor grids. It is based on Tetra decomposition of cells [19]. The interpolation coefficients are computed first on the tetrahedron containing the interpolated point. If the donor mesh is structured, then the coefficients are passed on the corresponding structured cell.

- 3rd [20] and 5th order Lagrangian interpolations are available for structured grids.

- Moving Least Squares approximations [21, 22] are available for all types of grids, and are currently 3rd-order accurate.

Figure 14-(a) describes a structured mesh defining a high-lift configuration of an aircraft, where the high-lift devices (colored in the figure) are overlapping the initial configuration. In figure 14-(b), a slice in the mesh aft the wing is displayed, where blanked points are not visualized. These blanked points are determined by the blanking function followed by the overlap optimization function. As it can be seen, the overlapping region has been made



(a) (b)

Figure 14: Overset grid assembly of a high-lift configuration (blanked points are not visualized). *Courtesy of C. Francois, ONERA, DAAP/ACI.*

An additional function, called `Connector.setInterpTransfers`, transfers the solution from the donor zones to the receptor zones, using interpolation data stored in the donor and receptor pyTrees.

3.6. Post module

`Post` module contains some classical post-processing tools, such as the computation of classical variables (pressure, vorticity, gradients...), the extraction of isosurfaces or slices, or the computation of field integrations.
An example of an isosurface of the Q-criterion in a CFD solution using Marching Cubes [23] is displayed in figure 15-(a).

In the case where the pyTree is defined by overset grids, the nature of points (namely computed, interpolated, blanked) is automatically taken into account by all post-processing functions.

Similarly, the connectivity between abutting grids, stored in the pyTree, is used if relevant: for instance, when computing a gradient, ghost cells are added temporarily within the function `P.computeGrad` such that gradients are correctly estimated near the borders of the grids.

`Post` module can also perform interpolation from a set of volume grids to another surface or volume mesh, as displayed in figure 15-(b) and (c). In

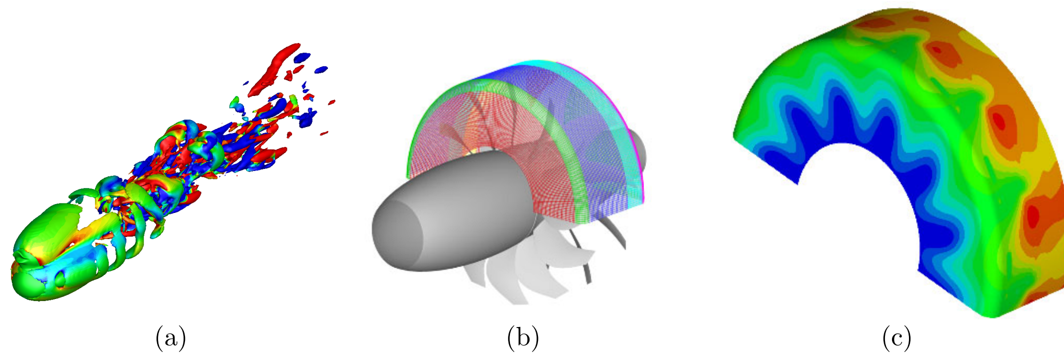


Figure 15: (a) Isosurface extraction; (b) and (c) interpolation of a CFD field to a CAA surface mesh.

this example, the CFD solution on a structured overset mesh is interpolated onto a given surface mesh used to propagate the acoustic field in a following process. If the surface mesh intersects overlapping CFD grids, then the cell with the smallest volume is chosen as donor for the interpolation onto the surface mesh, provided it is not a blanked cell.

3.7. Distance fields

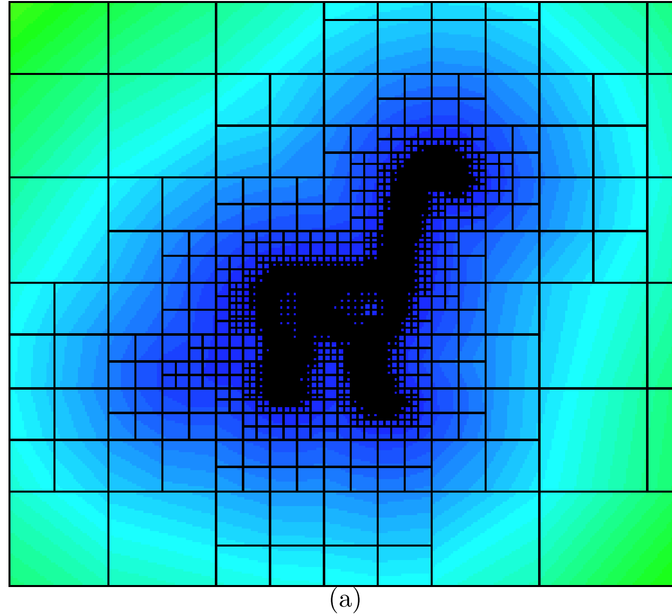
The `Dist2Walls` module provides functions to compute a distance field with respect to a set of bodies, as displayed in figure 16. This distance field can be signed, i.e. negative inside the bodies and positive outside. In that case, the Object X-Ray technique [16] is applied to determine if a point is inside or outside a body.

4. Some workflows using Cassiopee modules

In this section, two examples of workflows are presented, highlighting the ability of Cassiopee modules to build in an easy and extensible way many applications.
The first workflow consists in creating an offset surface starting from an initial surface. The second workflow illustrates how to perform a geometrical preprocessing for the Immersed Boundary Method.

4.1. Surface offset

The first example of workflow describes how to offset a surface at a given distance d from this surface. The resulting offset surface can be used for



(a)

Figure 16: Signed distance field defined on an octree mesh.

instance to define a refinement region before adaptation. It can be also useful for overset grid assembly, to inflate blanked regions to prevent them from lying within the boundary layer. The algorithm is summarized in figure 17.

- First, a file describing the bodies, defined discretely as surface meshes, is converted into a pyTree, named `surfaces`.

- An octree is generated around the surfaces, using `Generator.octree` function, resulting in a pyTree zone named `o`.

- A signed distance field is computed on the octree, using the function `Dist2Walls.distance2Walls`.
  The result of this function consists in storing the distance field of name `TurbulentDistance` as a `FlowSolution` node in the octree zone `o`.
  The offset region can cross octree elements of different levels. In order to ensure elements of finest level in that region, the octree `o` is refined at a distance d from the initial surfaces.

- For that purpose, an indicator field is defined for each element, such that an octree element is tagged for refinement if it is roughly at a distance d from the body and if its refinement level is not the finest one.

  This is achieved using `Converter.initVars` function, using a formula translating the previous condition. This function adds a tag field located at elements of the octree `o`.

- The octree is then refined around a distance d, according to this tag using `Generator.adaptOctree`.

- Once the octree mesh is refined enough in that region, an isosurface of distance d, defined by a pyTree `s`, is extracted from the octree mesh, using `Post.isosurfMC`.

- The resulting surface can then be smoothed for a better result, using `Transform.smooth`.

4.2. Immersed Boundary Method: geometrical preprocessing

The second example describes the geometrical preprocessing of the Immersed Boundary Method (IBM), using the Ghost Cell approach as described in [24]. In this approach, the wall is defined by a surface mesh that is not a boundary of the computational volume mesh. The wall boundary condition is then imposed in the first layer of cells lying inside the surface. Those cells are called ghost cells. To update the ghost cell field during computation, the following process is used (see figure 18): first, the symmetric point B of a ghost cell center A with respect to the wall boundaries is sought. Then, the flow field is interpolated for point B, using a donor cell containing it. Finally, using a no slip boundary condition onto the wall leads to $u_A = -u_B$ and $p_A = p_B$.

The full algorithm using Cassiopee functions is described in figure 19. The starting point in this case is the solid, defined by a cylinder, and the spatial resolution required in its neighbourhood. The first three steps are the setup of the case and then the geometrical preprocessing for the IBM is performed:

- First, the cylinder is defined in a file as a surface mesh. It is converted to a pyTree, named `bodies`, using `Converter.convertFile2PyTree` function.

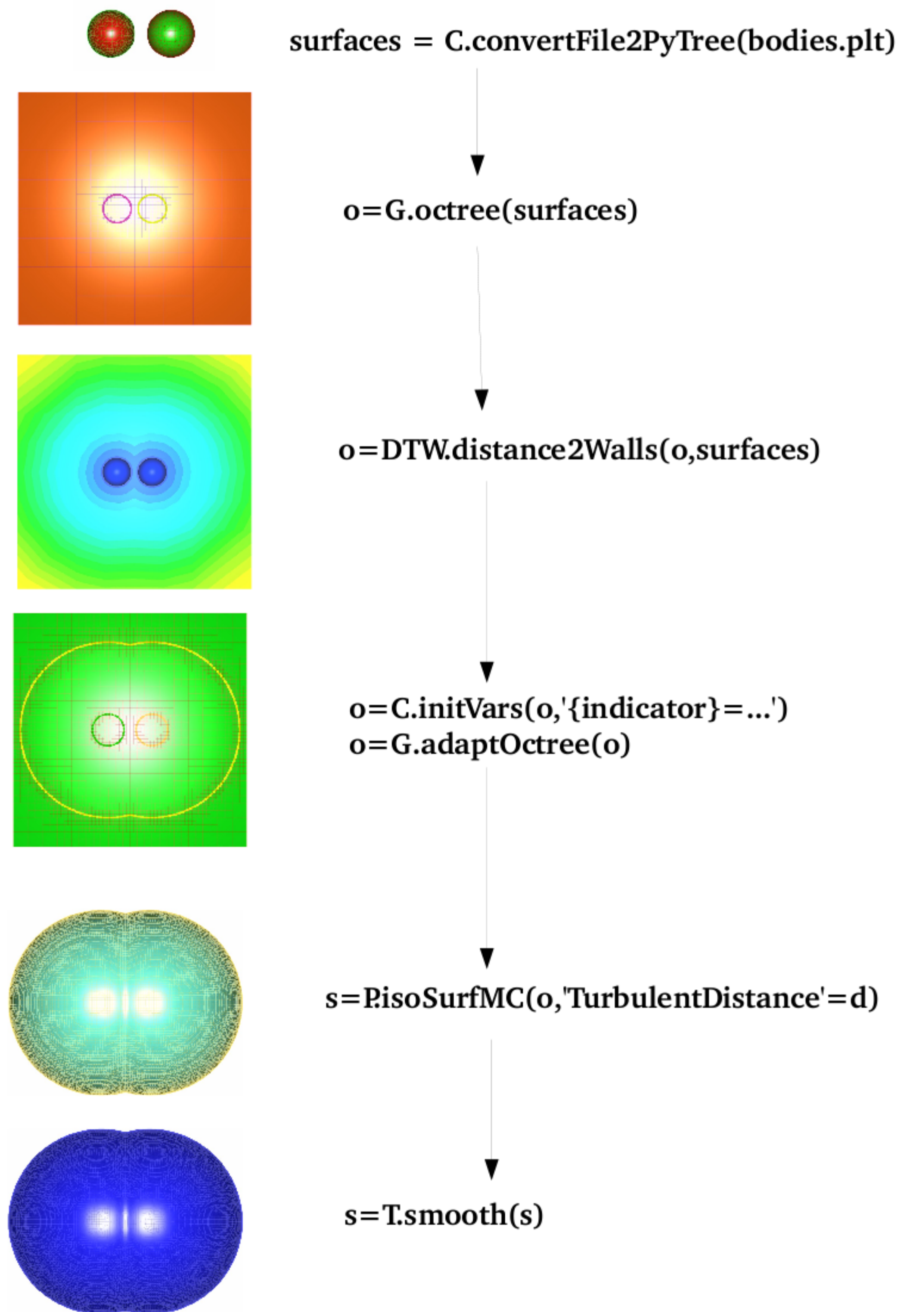


Figure 17: Example of workflow to build an offset surface.

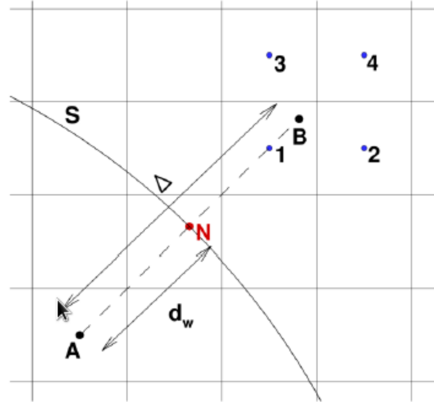


Figure 18: Immersed Boundary Method: A is a ghost point, B its symmetric, S the solid, and [1,2,3,4] the donor points for interpolation to get the flow field at B.

- An octree unstructured zone is built around the cylinder, with a given spacing near the cylinder.
- A set of abutting Cartesian grids are then built, using the same refinement pattern as the octree mesh, using `Generator.octree2Struct` function (the algorithm is detailed in [13]). The result is `t`, which is once again a pyTree, where are stored the mesh coordinates and the grid connectivity (one-to-one and n-to-m abutting connectivities).
- Cells lying inside the cylinder are marked as blanked thanks to the function `Connector.blankCells`.
  A field `cellN` is now stored in `t`, and is equal to 0 for blanked points, and 1 elsewhere.
- Then ghost cells are identified using `setHoleInterpolatedPts`, such that `depth` value is -2: this means that the first two layers of blanked points (inside the solid) are marked as ghost cells. The pyTree `t` is thus modified: the `cellN` field is equal to 2 for ghost cells.
- A signed distance is then computed using `Dist2Wall.distance2Walls` function.
- The gradient of the distance is then computed in order to get the normals at ghost cells, using `Post.computeGrad` function.

Note that since the pyTree `t` contains the abutting connectivity information, the gradient is estimated properly if the ghost cell region is defined by a set of abutting grids.

- The wall points and the symmetric point of the ghost cells can be easily determined then, since the distance to the cylinder and the normals are defined for the ghost points.

- The interpolation of the flow field outside the cylinder at symmetric points is achieved. The coordinates of the wall points and the symmetric points, but also the interpolation data (coefficients, indices of donor points, interpolation method,...) are stored in the pyTree `t` using `Connector.setIBCData`.

- Finally, the values of velocity and pressure are determined for ghost cells using `Connector.setIBCTransfers`, which can be sent to the CFD solver at each iteration.

5. Some CFD applications using Cassiopee

In this section are presented some CFD applications that use Cassiopee modules, mainly achieved at the Applied Aerodynamics Department of ONERA.

5.1. Adaptation of the wake past a helicopter fuselage

The first example concerns the application of the Cartesian mesh adaptation to a helicopter fuselage configuration. The configuration and results are discussed in [25]. The mesh is defined by near-body structured grids around the fuselage. An off-body Cartesian mesh is generated using `Generator` functions. The resulting mesh after blanking using `Connector` module is represented on the left-hand side of figure 20-(a). A first computation using *elsA* CFD solver [26] is achieved and the mesh is adapted according to the Q-criterion, using `Generator` and `Post` functions. The adapted Cartesian mesh is represented on the right-hand side of figure 20-(a). Figure 20-(b) highlights the better capture of the wake aft the helicopter fuselage due to the Cartesian mesh refinement in this region.

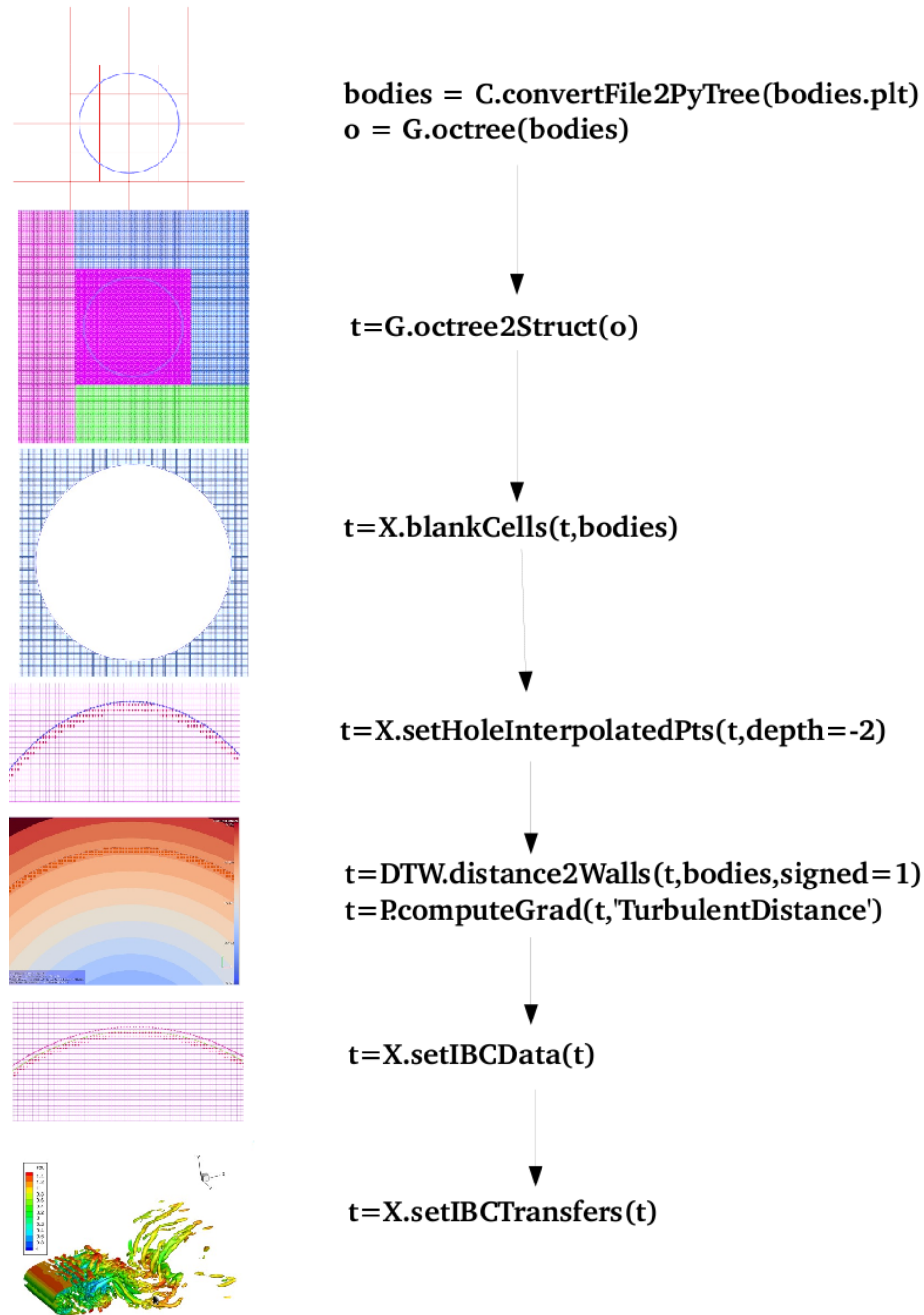
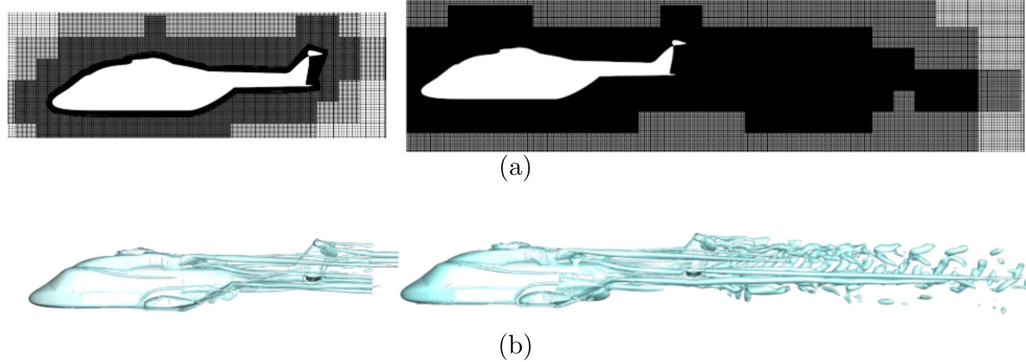


Figure 19: Example of workflow for geometrical preprocessing of the Immersed Boundary Method.



(a)

(b)

Figure 20: (a) Cartesian off-body mesh before and after adaptation and (b) corresponding isosurfaces of Q-criterion.

5.2. Embedded post-processing within the CFD simulation

This example demonstrates that it is possible to use Cassiopee functions during an unsteady CFD simulation to extract some flow features. The CFD simulation considered here is a ZDES simulation in poststall condition [27] using ONERA *elsA* solver. Here, the number of points is roughly equal to 50 millions, thus the storage of the whole volumic data resulting from the computation at each time step would be prohibitive and the storage of data on a specific region is up to now the only realistic way. Here, `Post.isoSurface` function is used to extract the isosurface of the Q-criterion every 500 iterations in parallel mode. The pyTree is the common data structure used by both *elsA* solver and Cassiopee modules during the following steps: first, the velocity gradients required for the computation of the Q criterion are computed in parallel by *elsA* solver for each block on its processor. The Q-criterion field is then computed by `Post.computeExtraVariable` function also locally. The isosurface of the Q criterion at a given value is then extracted onto the processor by using `Post.isoSurface`. On each processor is defined a part of the final surface mesh defining the isosurface, which part corresponds to the local blocks to the processors. Then the different surface grids are gathered on the same processor, and can be saved into a file.

Acknowledgements

The authors would like to thank Marc Poinot, for spreading the major ideas concerning the software architecture upon which Cassiopee is based

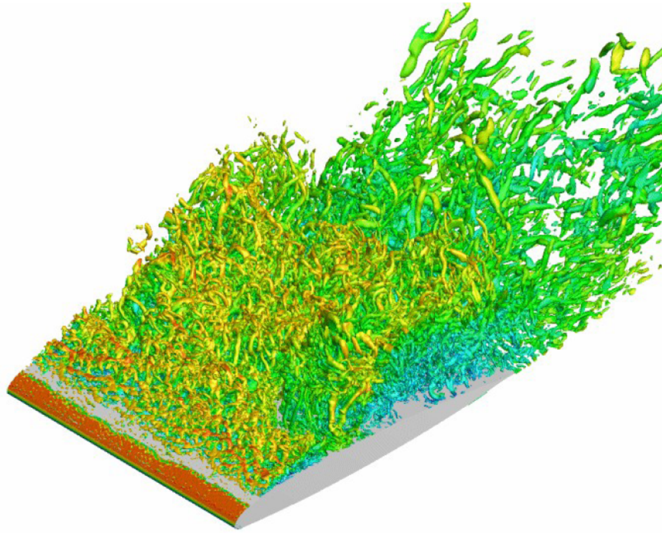


Figure 21: Extraction of an isosurface of the Q-criterion during a CFD simulation. *Courtesy of A. Le Pape, ONERA, DAAP/H2T.*

(CGNS/Python representation, independent modules...).
We are also extremely grateful to all the people that contribute to Cassiopee by making suggestions, bringing new ideas and reporting bugs.

[1] http:http://www.salome-platform.org.

[2] C. Gueuzaine, J. Remacle, Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities, International Journal for Numerical Methods in Engineering 79 (2009) 1309–1331.

[3] http://cgns.sourceforge.net.

[4] http://www.grc.nasa.gov/WWW/cgns/CGNS_docs_current/sids/index.html.

[5] http://elsa.onera.fr/Cassiopee/Userguide.html.

[6] P. Bézier, Courbes et surfaces, HERMES, 1986.

[7] W. Gordon, C. Hall, Construction of curvilinear coordinate systems and applications to mesh generation, Int. J. Num. Meth. Engr. 7 (1973) 461–477.

[8] A. Perronnet, Interpolation transfinie sur le triangle, le tétraèdre et le pentaèdre. Application la création de maillages et la condition de Dirichlet, C.R. Acad. Sci. Paris 326 (1) (1998) 117–122.

[9] S. J. Parks, P. G. Buning, W. M. Chan, J. L. Steger, Collar grids for intersecting geometric components within the Chimera overlapped grid scheme, 1991, pp. 672–682.

[10] A. A. G. Requicha, Representations for rigid solids: Theory, methods, and systems, ACM Computational Survey 12 (4) (1980) 437–464.

[11] P. L. George, H. Borouchaki, Delaunay triangulation and meshing : application to finite elements, HERMES, 1998.

[12] M. A. Yerry, M. S. Shephard, Three-dimensional mesh generation by modified octree technique, International Journal for Numerical Methods in Engineering 20 (1984) 1965–1990.

[13] S. Péron, C. Benoit, Automatic off-body overset adaptive Cartesian mesh method based on an octree approach, Journal of Computational Physics 232 (1) (2013) 153 – 173.

[14] D. L. Rigby, E. Steinthorsson, W. J. Coirier, Automatic Block Merging Methodology using the Method of Weakest Descent, AIAA paper 97-0197, 1997.

[15] G. Taubin, A signal processing approach to fair surface design, Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques, 1995, pp. 351–358.

[16] R. L. Meakin, Object X-Rays for Cutting Holes in Composite Overset Structured Grids, AIAA paper 2001-2537, 2001.

[17] J. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517.

[18] N. E. Suhs, S. E. Rogers, W. E. Dietz, PEGASUS5: an automated pre-processor for Overset-grid CFD, AIAA paper 2002-3186, 2002.

[19] C. Benoit, G. Jeanfaivre, E. Canonne, Synthesis of ONERA Chimera method developed in the frame of CHANCE program, 31st European Rotorcraft Forum, 2005.

[20] O. Saunier, C. Benoit, G. Jeanfaivre, A. Lerat, Third-order Cartesian Overset mesh adaptation method for solving steady compressible flows, Intern. J. Numer. Meth. Fluids 57 (7) (2007) 811–838.

[21] P. Lancaster, K. Salkauskas, Surfaces generated by moving least squares methods, Mathematics of computation 37 (155) (1981) 141–158.

[22] Z.-Q. Cheng, Y.-Z. Wang, B. Li, K. Xu, G. Dang, S.-Y. Jin, A survey of methods for moving least squares surfaces, in: Proceedings of the Fifth Eurographics/IEEE VGTC conference on Point-Based Graphics, 2008, pp. 9–23.

[23] W. Lorensen, H. Cline, Marching Cubes: A High Resolution 3D Surface Construction Algorithm, SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques, 1987.

[24] R. Mittal, G. Iaccarino, Immersed Boundary Methods, Annu. Rev. Fluid Mech. 37 (2005) 239–261.

[25] T. Renaud, A. Le Pape, S. Péron, Numerical analysis of hub anf fuselage drag breakdown of a helicopter configuration, CEAS Aeronautical Journal 4 (4) (2013) 409–419.

[26] L. Cambier, S. Heib, S. Plot, The Onera elsA CFD software: input from research and feedback from industry, Mechanics & Industry 14 (03) (2013) 159–174.

[27] A. Le Pape, F. Richez, S. Deck, Zonal Detached-Eddy Simulation of an Airfoil in Poststall Condition, AIAA Journal 51 (8) (2013) 1919–1931.