Mesh Modification and Adaptation Within INMOST Programming Platform

Kirill Terekhov and Yuri Vassilevski

Abstract INMOST (Integrated Numerical Modelling Object-oriented Supercomputing Technologies) is a programming platform which facilitates development of parallel models. INMOST provides to the user a number of tools: mesh manipulation and mesh data operations, automatic differentiation, linear solvers, support for multiphysics modelling. In this paper, we present mesh modification and adaptation capabilities of INMOST.

1 Introduction

Parallel modelling of complex multiphysics phenomena is a challenge since the programmer has to implement numerical methods in parallel, manage the unstructured grid, data exchanges and assembly of large distributed linear systems with MPI, solve the resulting linear and nonlinear systems and finally postprocess the result. INMOST [8] is an open-source library that alleviates the most of the burden from the programmer providing a unified set of tools to address each of the aforementioned issues. We have used the INMOST platform to implement the fully implicit black-oil reservoir model and fully coupled blood coagulation

K. Terekhov (✉)
Marchuk Institute of Numerical Mathematics of the Russian Academy of Sciences, Moscow, Russia
e-mail: terekhov@dodo.inm.ras.ru

Yu. Vassilevski
Marchuk Institute of Numerical Mathematics of the Russian Academy of Sciences, Moscow, Russia

Lomonosov Moscow State University, Moscow, Russia

Moscow Institute of Physics and Technology, Dolgoprudny, Russia

Sechenov University, Moscow, Russia

V. A. Garanzha et al. (eds.), *Numerical Geometry, Grid Generation and Scientific Computing*, Lecture Notes in Computational Science and Engineering 131,
https://doi.org/10.1007/978-3-030-23436-2_18

model [12]. The black-oil reservoir model involves simultaneous solution of three Darcy equations that describe flows of the mixture of water, oil and gas. The blood coagulation model couples the Navier-Stokes equations with the Darcy term and nine additional advection-diffusion-reaction equations that participate in a reaction cascade responsible for blood coagulation [3]. Both multiphysics applications can greatly benefit from adaptation of the computational mesh to the solution during the simulation.

Mesh modifications are demanded in parallel generation of huge computational meshes. Static and dynamic mesh adaptations through refinement and coarsening reduces the computational work. These mesh operations require very flexible and efficient data structure for storage of mesh elements, adjacency information, allowing for fast removal and addition of elements. Mesh libraries allowing for mesh modification, such as Dune [5], project DuMuX [6] based on Dune, STK mesh from Trilinos package [14], have attracted ever-growing attention. There are other notable packages for parallel mesh management, such as MOAB [10] and MSTK [7], they offer basic mesh modification functionality: delete and add mesh elements. INMOST platform provides mesh modification tools which are applicable to meshes composed of arbitrary polyhedral star-shaped cells. The algorithms and functionality that form the basis of the INMOST mesh operations module were previously reported in [2, 4, 11, 12, 15]. In this work we address sequential modification of general meshes leaving parallel mesh modification to further works.

2 Mesh and Mesh Data

The primary mesh functionality of the INMOST platform is formed by the following operations:

- load a mesh and associated data;
- compute partitioning of the mesh;
- redistribute the mesh;
- build multiple layers of ghost cells;
- access mesh elements, status of elements, mesh data;
- save the result in parallel format.

The data structure supports the full zoo of elements: nodes, edges, faces, cells as well as bidirectional adjacency connections between them as depicted in Fig. 1a. The traversal of adjacencies should be ordered: edges of each face form a loop that defines the normal orientation, nodes of certain types of cells appear in a predefined order. The elements can be organized into sets of elements which in turn can be organized into a tree structure as depicted in Fig. 1b.

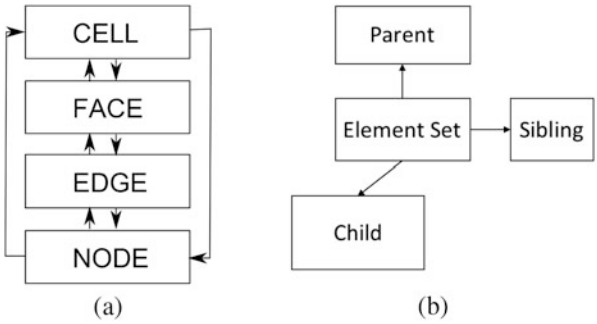


Fig. 1 (**a**) Element zoo and adjacency connections. (**b**) Organization of sets of elements into a tree structure

2.1 Data Structure and Algorithms for Mesh Modification

Various scenarios of data usage imply a large variety of mesh data representations shown in Fig. 2. Mesh data can be dense or sparse, i.e., given on all or some elements. The data can have fixed or variable size, various data types: bulk (single character), integer, double, a reference to an element, double with single or multiple variations (variation is represented by a sparse vector consisting of an index and a coefficient), a reference to an element of another mesh. The user can access the data

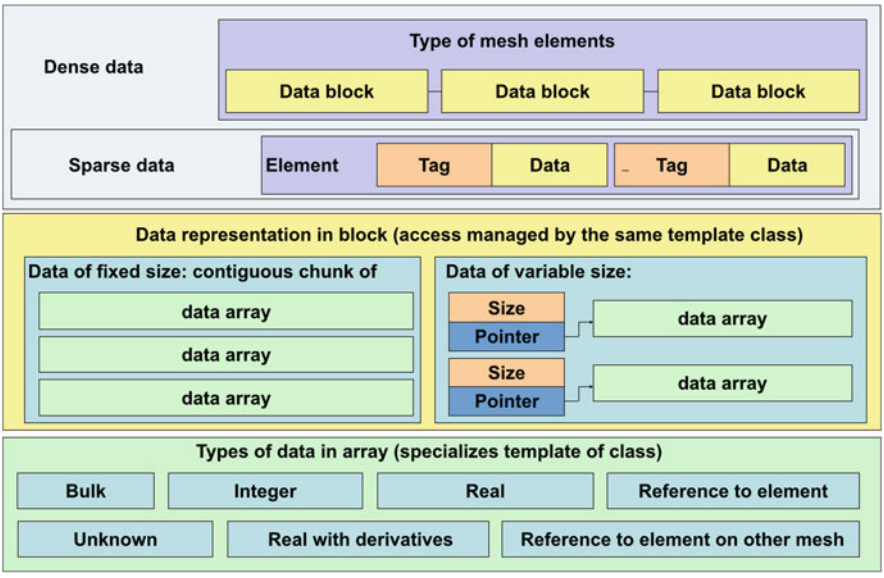


Fig. 2 Data representation in a mesh

directly in the memory through provided classes or can request to copy the data into provided arrays. Data on a mesh is associated with a tag.

All data of a mesh (including adjacency information) is stored using mesh data. Mesh operations imply possible increase of the number of elements and the size of the stored data. To cope with this, we separate dense data array into chunks, each chunk is capable of storing m-bit data for N elements. We keep in memory the contiguous blocks of size mN bits and links to these blocks. When we need to store more data, another block is added. The data fragmentation allows us to access the data in memory during reallocation.

We want to store references to elements in the mesh that are valid during modification and to minimize the memory required to store the data. To this end, we represent each element by a unique identificator: the first three bits store the element type, the rest of the bits store the position in a non-shrinkable array that holds position of the data. As a result, we can move the data and change its position keeping its identificator. When the element is deleted, the position of its data is set to -1. To handle removal of elements efficiently, we keep an array of positions of deleted elements. When an element is deleted, its position is added to the array and its data is deallocated or zeroed-out. When a new element is to be added, we first try to add it to the array of empty data positions. If no empty positions are available, we extend the data set by the new element. Compaction of data is achieved by repetitive motion of elements stored at the last positions to the empty positions if they occur.

New mesh elements can be added or deleted and the adjacency connections can be modified. An element is deleted by disconnection of the lower level adjacency connections and deletion of all the adjacency elements dependent on the deleted element. This is needed to keep the mesh consistent. The lower level adjacencies are two nodes for an edge, edges or nodes for a face, faces or nodes (in some cases) for a cell. Adjacencies of mesh elements can be disconnected and connected by functions *Element::Disconnect, Element::Connect*, element can be deleted or hidden depending on mesh state by *Element::Delete*, or completely destroyed disregarding the state of the mesh by *Element::Destroy*.

A new element is inserted into the mesh by its lower adjacency dependence. The elements are created with the functions *Mesh::CreateNode*, *Mesh::CreateEdge*, *Mesh::CreateFace*, *Mesh::CreateCell*.

A local mesh modification on an element is performed as follows. Upper adjacencies of the element are disconnected first and then the element is deleted to keep upper adjacencies intact. The mesh becomes inconsistent at this point. Then new elements are added to the mesh and the upper adjacencies are reconnected by function *Element::Connect* to make the mesh consistent again.

The user can calculate and store geometrical quantities, such as edge length, face area and normal, cell volume, barycentres for elements. The quantities are recomputed only for modified elements during mesh modification. The function computing geometrical quantities is *Mesh::PrepareGeometricData*, removing geometrical data *Mesh::RemoveGeometricData*, checking data availability *Mesh::HaveGeometricData*. The data can be accessed through functions *Element::Barycenter*, *Edge::Length*, *Face::Area*, *Cell::Volume*, *Face::Normal*, *Face::OrientedNormal*, *Face::FixNormalOrientation* or, in general, *Mesh::GetGeometricData*.

2.2 Topology Correctness Control

During generation and modification, it is easy to produce a topologically inconsistent configuration. A number of controls of mesh topological correctness are provided:

- Check for duplication of elements: when a new element is created, the adjacency is checked and if the element already exists, it is returned to the user.
- Check for element degeneracy: a face is prohibited to have less then 3 edges and a cell is prohibited to have less then 4 faces.
- Check and fix order of edges within a face: the edges should form a closed loop.
- Check for normal orientation of faces: the traversal of face nodes should match the face normal direction.
- Check for face planarity: an error is returned if the face is non-planar.
- Check for interleaved faces: an error is returned if multiple faces share the same nodes.
- Check for mesh conformity: each interior face is shared by exactly two neighbouring cells.
- Check for slivers: a cell face should not contain all the cell nodes.
- Check for adjacent elements on element creation: check for duplicated adjacencies, deleted adjacencies or their improper dimensionality.
- Prohibit existence of general polygons and polyhedra: only known types of elements are allowed in the mesh, such as triangles, quads, tetrahedrons, hexes and so on.
- Prohibit existence of multi-line or multi-polygon: detect and prohibit elements whose lower adjacencies do not form a closed loop.

The tests are performed during mesh modification. The function *Mesh::SetTopologyCheck* sets topology test, *RemTopologyCheck* removes it, *Mesh::GetTopologyCheck* returns the current set of tests, The erroneous elements are marked by a data whose tag can be accessed by *Mesh::TopologyErrorTag*. The topology correctness tests have to be enriched by checks for appearance of concave or non-star-shaped elements, self intersections of edges of a face or faces of a cell.

2.3 High-Level Modification Routines

A number of procedures facilitate the mesh modification managing all necessary reconnections in the mesh. Two types of procedures are provided: functions uniting a set of elements into a single element and functions splitting an element into subelements. To unite a set of edges, faces or cells, one uses functions *Edge::UniteEdges*, *Face::UniteFaces*, *Cell:UniteCells*. To split an edge by nodes, a face by edges and a cell by faces, one uses functions *Edge::SplitEdge*, *Face::SplitFace*, *Cell::SplitCell*.

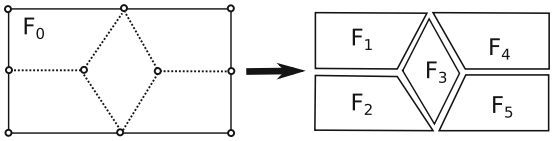


Fig. 3 Separation of a face F_0 by a set of edges into faces $F_1, \ldots, F_5$

Algorithm 1 Find all the loops $\mathscr{L}$ that form new faces

1: First we set a visit counter for all the edges. The original set of edges of the face should be visited just once and get the counter "1", the new edges that split the face should be visited twice and get the counter "2".
2: Start from an edge that has a visit counter "1". Add it to a loop ℓ_m. Set the recursion depth $k = 0$ and a counter of the loops $m = 0$.
3: Consider all the adjacent edges $\mathscr{E}_k$ with non-zero visit counter of the last added edge $l_k \in \ell_m$.
4: Add a next non-considered edge $e \in \mathscr{E}_k$ to the loop ℓ_m. If there are no more edges, then go to step 7.
5: Compute the number of visits of nodes by edges in ℓ_m, if all the nodes are visited twice, then ℓ_m is a closed loop and we go to step 6, if all the nodes are visited twice or once then continue to step 3, if some nodes are visited more then twice then we go to step 7.
6: The ℓ_m is a closed loop. We increase $m = m + 1$ and copy $\ell_m = \ell_{m-1}$.
7: We remove last added edge l_k from ℓ_m and reduce the recursion depth $k = k - 1$. If $k = -1$ then go to the step 8, otherwise return to the step 4.
8: Select such an m that the area covered by ℓ_m is the smallest and add ℓ_m to the set of all loops $\mathscr{L}$. For all the edges in ℓ_m we reduce the visit counter by one.
9: If there are still any edges with visit counter "1" return to step 2, otherwise exit the algorithm.

The unification scenario implies detection and elimination of all lower-level adjacencies internal to the union, connection of a new element to lower-level adjacencies external to the union, and reconnection of upper-level adjacencies. In certain cases merging of elements is not possible without topological issues and an error is returned. For instance, if a hexahedron is surrounded by other hexahedra, its faces can not be united. If a set of adjacencies external to a desirable union forms multiple disjoint loops, an error occurs as well.

The splitting scenario implies finding all closed loops in a graph formed by the adjacency connections such that the geometric measure (length, area or volume) is minimal. The problem of dividing a face by a set of edges is depicted in Fig. 3. To solve the problem of finding closed loops, we use recursive Algorithm 1 implemented in function *Face::SplitFaces*.

Non-flat face may complicate its splitting: face projection to a plane should be performed before application of Algorithm 1. Recursive Algorithm 1 is rather expensive, a more efficient algorithm stems from the divide and conquer strategy.

2.4 Mesh Modification and Mesh Data Transfer

Mesh modification is often accompanied by data transfer based on modification epochs. The user can switch the mesh into a modification state. In this state, the

Algorithm 2 Mesh modification epoch

1: Call *Mesh::BeginModification* to enter the modification state. From this point all deleted elements are only marked for deletion but remain in the mesh. Requests for adjacent elements skip elements marked for deletion.
2: Perform modification of the mesh. Delete old elements and create new elements.
3: Call *Mesh::ResolveModification* to setup the data necessary for exchange of data in parallel on the new mesh.
4: Call *Mesh::SwapModification* to recover the old mesh and transfer mesh data.
5: Call *Mesh::ApplyModification* to apply all changes. All the elements marked for deletion and their data are irreversibly destroyed. Links to deleted mesh elements are replaced to invalid links.
6: Call *Mesh::EndModification* to exit the modification state.

deleted elements are hidden from the mesh, but their data is still available for data transfer. The steps are presented in Algorithm 2.

The transfer of physical quantities is not provided during mesh modification as their interpolation is problem-dependent. The user can implement his interpolation procedures on the basis of the above functions.

3 Examples

The following examples involve mesh modification and use the aforementioned structure and algorithms. All examples are available in the INMOST repository [8].

3.1 Mesh Repair and Improvement

The following two examples represent only a small subset of possible mesh improvements that can be performed.

Example *GridTools/FixFaults* addresses the following mesh inconsistency. The conventional *Corner Point Grid* format stores a geological grid with a fault as a combination of two grids shifted vertically with respect to each other. The two grids are disjoint and the geological grid is not conformal. To fix this issue, we have to find intersection of faces that are in contact, introduce new edges and split the contacting faces as depicted in Fig. 4. The fault displayed in Fig. 5a has two disjoint mesh traces as shown in Fig. 5b. The corrected conformal grid on the fault is shown in Fig. 5c.

Example *GridTools/FixTiny* addresses meshes which have faces or edges of very small size. This may result in instability of discretization methods, deterioration of accuracy, very stiff matrices, etc. Our tool collapses such elements. An edge can be collapsed to a node, a face to either an edge or a node, a cell to either a face or an edge or a node. Moreover, the tool is able to repair an issue in the grid illustrated in Fig. 5a. The choice of the collapse operation for an element is based on the analysis

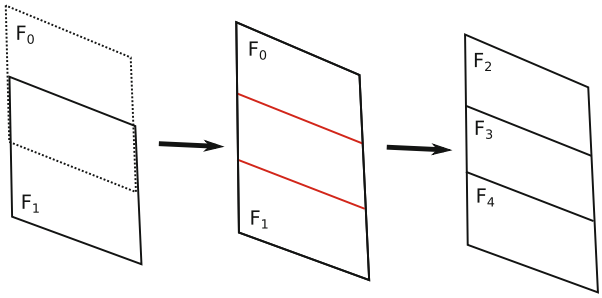
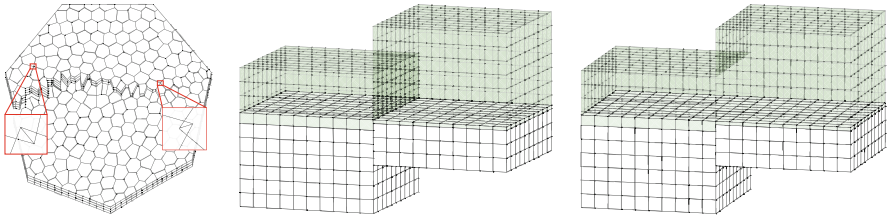


Fig. 4 Finding intersection of overlapping faces and separation into a new set of faces



(a) (b) (c)

Fig. 5 (**a**) Fixing the grid with self-intersections in edges of the mesh. (**b**) Traces of two disjoint faces on the fault. (**c**) Faces are split at the fault between two meshes

of the bounding ellipsoid with minimum volume [13]. For a 3D cell, comparing the ellipsoid semi-axes of the matrix corresponding to the ellipse, we:

- collapse the cell to an edge, if one semi-axis is significantly larger than the others;
- collapse the cell to a face, if two largest semi-axes are relatively close;
- collapse the cell to a node, if all three semi-axes are almost equal.

3.2 Dual Meshes

Vertex-centered finite volume (FV) methods exploit polyhedral dual grid which benefits from the reduction of the number of degrees of freedom keeping the same accuracy if one compares them with cell-centered FV methods on tetrahedral or triangular prismatic grids. The construction of the dual grid is straightforward, see Fig. 6. The example is *GridTools/Dual*.

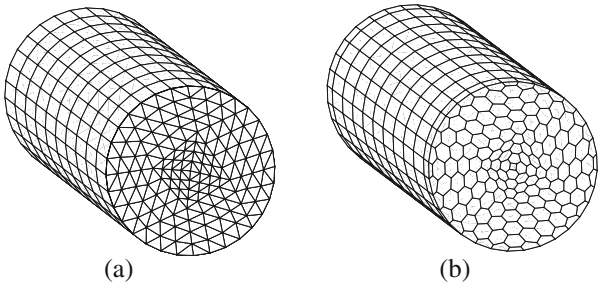


Fig. 6 Initial triangular prismatic grid (**a**) converted into dual grid (**b**)

3.3 Cutting Grids

This tool allows to cut cells by a plane or by the zero-level of a signed-distance function as demonstrated in Fig. 7a. The first example forms a geological layer into an uniform grid. The second example cuts an uniform grid to a shape of interest. In both cases tiny cell may be produced and we recommend to collapse these cells by the tools discussed in Sect. 3.1. The grid is cut by the zero-level of a given function according to Algorithm 3.

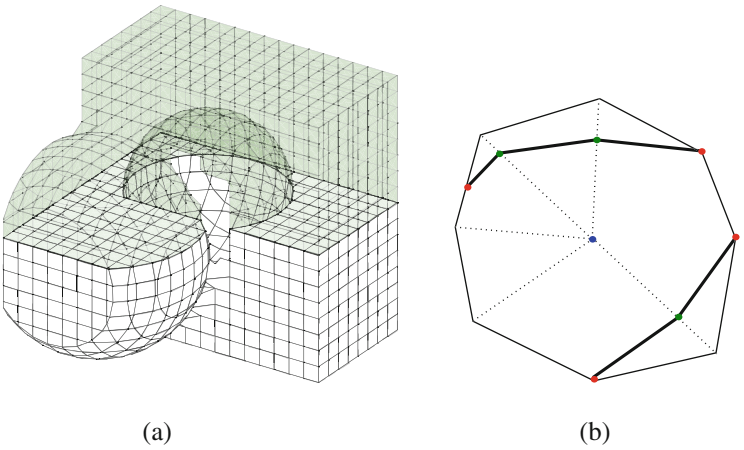


Fig. 7 (**a**) Results of slicing the grid with a function that defines the domain inside of one sphere in one half of the grid and outside of another sphere in another part of the grid. Transparent green surface defines the boundaries of the mesh, slice of the mesh is displayed with the white color. (**b**) The result of cutting the face with more than two cut nodes (red dots): the barycentre is introduced (blue dot), the segments to non-cut nodes are considered (dashed lines) and new cut nodes along these segments are introduced (green dots), connecting the dots results in a cut trajectory (bold lines)

Algorithm 3 Slicing grid by zero-level of a function

```
for all nodes of the gird do
    if the function is zero on the node then
        mark it as cut
    end if
end for
for all edges of the grid do
    if the function changes sign on the edge nodes then
        search for zero of the function along the edge and insert a zero-level node, split the edge by this node
    end if
end for
for all faces of the grid do
    if there are only two cut nodes adjacent to the face then
        split the face by an edge connecting these two nodes
    else if all nodes of the face are cut then
        mark entire face as cut
    else if there are more then two cut nodes then
        consider barycentre of the face and introduce nodes at zero of the function on segments connecting non-cut nodes of the face and the barycentre: these cut nodes form a trajectory that cuts the face, see Fig. 7b
    end if
end for
for all cells of the grid do
    if cut edges of the cell form a simple loop then
        insert a face defined by this loop of edges and cut a cell by this face
    else
        insert the barycentre of the cell
        consider segments connecting the barycentre and the uncut nodes of the cell and insert new cut nodes along the segments
        consider triangles formed by the uncut edges and the barycentre and insert new cut edges
        consider pyramids formed by the barycentre and the uncut faces of the cell and connect all cut edges into a face
        split the cell by resulting set of faces
    end if
end for
```

The examples are *GridTools/Slice* for slicing by a plane and *GridTools/SliceFunc* for slicing by a function.

3.4 Mesh Adaptation

Example *AdaptiveMesh* provides dynamic refinement of general polyhedral meshes shown in Fig. 8. For each cell to be refined, the algorithm adds nodes at barycentres of the cell, its faces and edges. For each cell edge, a quadrilateral is formed by the mid-edge, the cell barycentre and the barycentres of two cell faces adjacent to the edge. The cell is split into subcells by these quadrilaterals, and the conformity of the refined mesh is recovered.

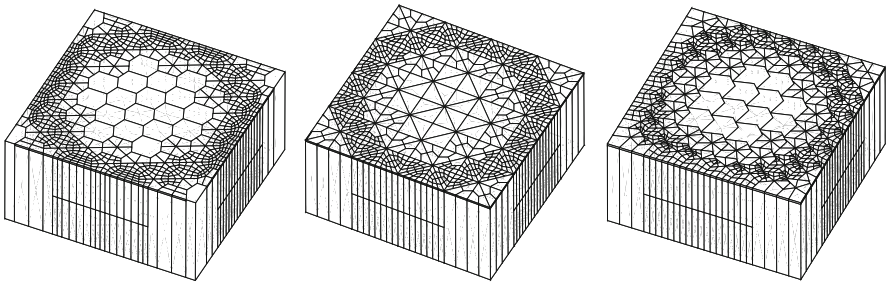


Fig. 8 Local adaptation of three kinds of prismatic meshes, from left to right: hexagonal, triangular, non-convex squama. The middle cutaway of the grids is displayed

The local coarsening is based on hierarchy of sets. The leaf sets store all mesh elements, the union of these elements restore a coarse element. When a cell is refined, a new leaf set is created and the subcells are added to this set. The level of refinement in adjacent elements can differ by no more than one, in order to avoid highly graded meshes. Sequential and parallel algorithms for refinement and coarsening will be reported in other works.

4 Real-Life Application

INMOST was used for the solution of poromechanics problem, formulated similar to the one in [1]. The problem was solved on the grid of Norne oil field [9] with one injection well and two production wells. The grid features multiple faults and pinch-outs. For filtration part of the problem the original porosity and permeability data was used, for coupling and mechanical part, the Biot coefficient, Biot modulus, and anisotropic 4-th rank compliance tensor were synthetically defined based on permeability and porosity fields. Figure 9 illustrates the magnitude of displacement field in logarithmic scale after 1000 days of injection.

5 Conclusion

We have presented briefly the open-source platform INMOST for the development of parallel mathematical models on general meshes. The platform allows the user to perform mesh generation and mesh modifications such as mesh repair and mesh adaptation.

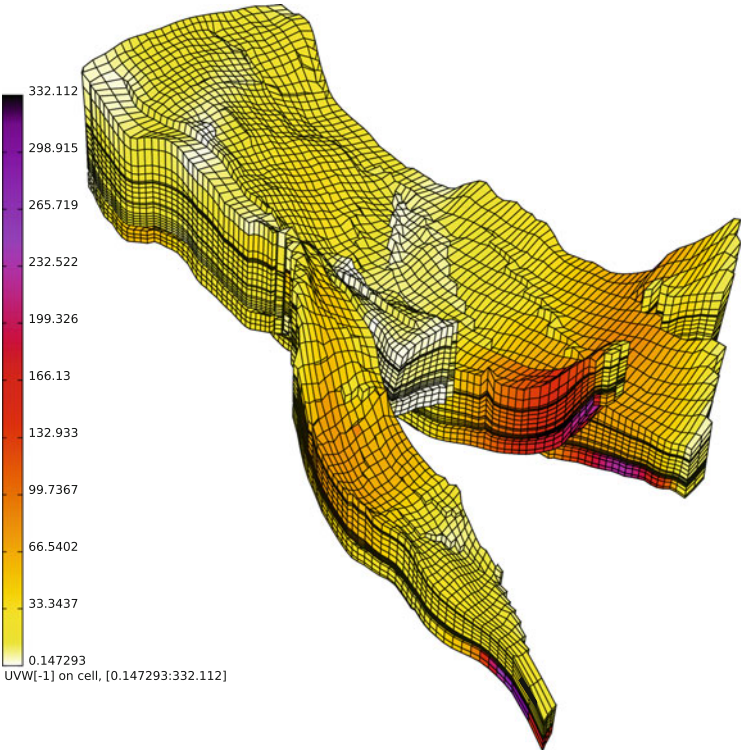


Fig. 9 Grid for Norne oil field colored in magnitude of displacement for poromechanics problem

Acknowledgements This work was supported by the Russian Foundation for Basic Research (RFBR) under grants 17-01-00886 and 18-31-20048.

References

1. Badia, S., Quaini, A., Quarteroni, A.: Coupling biot and Navier–Stokes equations for modelling fluid–poroelastic media interaction. J. Comput. Phys. **228**(21), 7986–8014 (2009)
2. Bagaev, D.V., Burachkovskii, A.I., Danilov, A.A., Konshin, I.N., Terekhov, K.M.: Development of INMOST programming platform: dynamic grids, linear solvers and automatic differentiation. In: Russian Supercomputing Days, pp. 543–555. http://2016.russianscdays.org/files/pdf16/543.pdf (2016, in Russian)
3. Bouchnita, A.: Mathematical modelling of blood coagulation and thrombus formation under flow in normal and pathological conditions. Ph.D. Thesis, Université Lyon 1 - Claude Bernard, Ecole Mohammadia d'Ingénieurs - Université Mohammed V de Rabat, Maroc (2017)
4. Danilov, A.A., Terekhov, K.M., Konshin, I.N., Vassilevski, Yu.V.: Parallel software platform INMOST: a framework for numerical modeling. Supercomput. Front. Innov. **2**(4), 55–66 (2015)

5. Distributed and Unified Numerics Environment. https://dune-project.org/. Accessed 30 May 2018
6. Flemisch, B., Darcis, M., Erbertseder, K., Faigle, B., Lauser, A., Mosthaf, K., Müthing, S., Nuske, P., Tatomir, A., Wolff, M., Helmig, R.: DuMux: DUNE for multi-phase, component, scale, physics,... flow and transport in porous media. Adv. Water Resour. **34**(9), 1102–1112 (2011)
7. Garimella, R.V.: MSTK - a flexible infrastructure library for developing mesh based applications. In: Proceedings, 13th International Meshing Roundtable, pp. 213–220 (2004)
8. INMOST – a toolkit for distributed mathematical modeling. http://www.inmost.org. Accessed 15 April 2018
9. Norne: the full Norne benchmark case, a real field black-oil model for an oil field in the Norwegian Sea. https://opm-project.org/?page_id=559. Accessed 26 February 2019
10. Tautges, T.J.: MOAB-SD: integrated structured and unstructured mesh representation. Eng. Comput. **20**(3), 286–293 (2004)
11. Terekhov, K.M.: Application of unstructured octree grid to the solution of filtration and hydrodynamics problems (in Russian). Ph.D. Thesis, INM RAS (2013)
12. Terekhov, K., Vassilevski, Y.: INMOST parallel platform for mathematical modeling and applications. In: Voevodin, V., Sobolev, S. (eds.) Supercomputing. RuSCDays 2018. Communications in Computer and Information Science, vol 965. Springer, Cham. https://doi.org/10.1007/978-3-030-05807-4_20 (2019)
13. Todd, M.J., Yıldırım, E.A.: On Khachiyan's algorithm for the computation of minimum-volume enclosing ellipsoids. Discrete Appl. Math. **155**(13), 1731–1744 (2007)
14. Trilinos – platform for the solution of large-scale, complex multi-physics engineering and scientific problems. http://trilinos.org/. Accessed 15 April 2018
15. Vassilevski, Yu.V., Konshin, I.N., Kopytov, G.V., Terekhov, K.M.: INMOST - Programming Platform and Graphical Environment for Development of Parallel Numerical Models on General Grids (in Russian). Moscow University Press, Moscow (2013)