Ninth International Conference on Computational Fluid Dynamics (ICCFD9), Istanbul, Turkey, July 11-15, 2016 **ICCFD9-xxxx**

girdap: Open source object oriented autonomous grid management library for multi-physics simulations

Eray Uzgoren[1]

[1]Mechanical Engineering Program, Middle East Technical University Northern Cyprus Campus, Guzelyurt TRNC, Mersin 10, Turkey

Corresponding author: uzgoren@metu.edu.tr

Abstract: This paper presents an object-oriented grid generation and management library that use finite volume operator objects for multi-physics problems. Grid objects, formed by quadrilateral cells, are capable of performing local anisotropic grid refinement (*h*-adaptation) as well as relocating their vertices (*r*-adaptation) based on solution field using finite volume based differential operator objects for conservation laws of motion. In addition, grid objects can interact with each other (including triangulated surfaces and one-dimensional strings) through interpolators which allows extension for tackling moving boundary problems. This study illustrates *h*- and *r*-refinement capabilities based on an error estimate that compares two different interpolation techniques to capture undesired levels of deviation from linear approximation. Code's capabilities are illustrated through example problems.

Keywords: Adaptive mesh refinement, multi-physics, finite volume method.

1 Introduction

Mesh-based simulation tools are increasingly being used to reduce the cost and the time of a typical engineering design cycle thanks to continuous developments in numerical methods and grid generation techniques. Open source development is central to nourish this development process by allowing researchers to focus more on numerical methods and by avoiding time consuming code repetition. Some examples of open source simulation tools include finite volume based OpenFOAM [1], finite element based FEniCS project [2], finite difference/finite volume based Overture [3], which provide mostly object oriented differential operators working on a grid tools that allows researchers to investigate new algorithms. Even though both tools come with simple grid generators, these tools lack adaptive solution strategies.

Managing grid quality and size in grid-based numerical simulations is essential to obtain accurate solutions in a reasonable computation time. According to Fidkowski and Darmofal [4], the present challenge in mesh-based simulation tools remains to be the need of robust and efficient tools for generating and maintaining quality grids around complex geometries for transient multi-scale problems. This is possible through adaptive solution techniques, which include (i) *h*-adaptation or local mesh refinement through cell splitting/merging, (ii) *r*-adaptation or relocating cell vertices, and (iii) *p*-adaptation or varying order of numerical approximation. All adaptation methods rely on the choice of error indicators. Adjoint methods have provided satisfactory results for steady flow problems while there are limited studies on its use for transient flows. Other examples include residual histograms, flow geometry and predictable flow features. Active research areas include (i) robust anisotropic grid adaptation techniques around complex geometries, (ii) identifying error bounds, and (iii) error estimation for unsteady problems.

This paper introduces an open source object oriented autonomous grid management library, named as *girdap* [5], to nourish research on grid-adaptation strategies for multi-physics problems. Primary focus of *girdap* is to provide tools for h-adaptation on auto generated grids with quadrilateral cells.

2 Overview of objects

Main objects are defined as below:

- *VecX*: Container for 1-D field data stored in vector class of STL library. Vector operators such as dot product, and cross product; element-wise addition, subtraction, division are overloaded on operators.
  - Vec3: a special sub-class of VecX for 3-D data
- *MatX*: Container for a square matrix stored in 1-D vectors in AIJ type sparse matrix storage scheme. Triplets, i.e. *row*, *column*, *value*, are used to construct the matrix.
- *LinSys*: Container for a linear equation system in the form of $Ax = b$. Members are one MatX (as A) and two VecX (as x and b). It also contains following solvers: Gauss-Seidel, conjugate gradient, biconjugate gradient stabilized.
- *Vertex*: Container for position vector stored as Vec3 to allow vector operations. It also contains information of surrounding cells.
- *Cell*: Container for an ordered set of vertices. Using the order; geometric information such as volume can be computed. Connectivity is assured through vertex-to-cell and cell-to-vertex lists.
  - Line: a sub-class of Cell that contains exactly two vertices
  - Quad: a sub-class of Cell that contains exactly four vertices
- *Boundary*: Container for a type (Dirichlet or Neumann) in the form of $a\phi + \mathrm{b}$. Coefficients a and b are stored in its definition.
- *Var*: Container for 1-D field data in the form of VecX along with an array of Boundary conditions. Typical array can contain 4 Boundary elements for east, west, north and south boundaries; but can be expanded as many as needed; i.e. east1 … eastN.
- *Grid*: Container for an array of Vertex; an array of Cell; and an array of faces; and an array of Var. Cell array contain different types as they are defined using polymorphism. For 2-D cases, only Quads are formed. Faces for Quad class are also formed by Line class. This allows 3-D extension by defining Hexa cells formed by faces made of Quad faces. It also contains methods for adaptation, grid-to-grid communication, as well as numerical operators such as interpolation, gradient, divergence, and Laplacian.

3 Grid

Grid objects are formed through lists of vertex objects and cell objects. Primary purpose of a single vertex object is to hold the location of a point in vector form. Newly created vertex object's pointer is stored in grid objects vertex list. Vertices also store their identity index, which corresponds the order in this list. Vertices can reach to its owner grid through a regularly defined pointer. Cells are formed through an ordered vertex list following the standards set by VTK tools, and vertices maintain an ordered list of cells IDs surrounding them, as illustrated in Fig. (1).

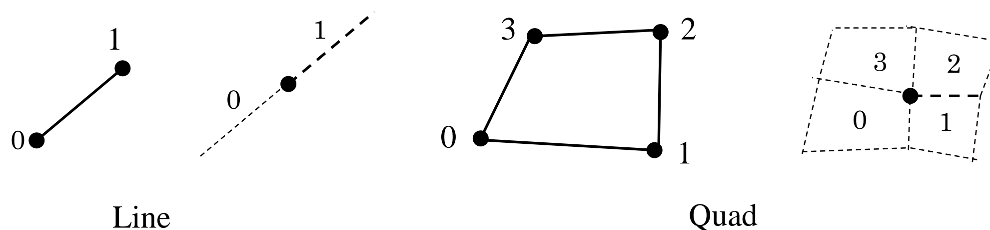

Figure 1: Vertex order for a cell and cell order for a vertex (a) for a cell type of Line, (b) for a cell type of Quad.

If the vertex is placed at a boundary, the value is replaced by a negative number. This number is used for imposing boundary conditions. In addition, a face list is introduced as faces are important in finite volume methods as they are used to account fluxes going in and out of a cell. Faces ease the handling of hanging vertices as they are not separately contained in a neighboring cell. Faces store an additional connectivity information of surrounding cells. Vertex and cell objects can be added or removed by the user while face list is only controlled by the grid object and is updated whenever the grid connectivity information is changed.

This data structure is selected considering a trade-off between storage (memory) and access (CPU) time. Explicit storage of faces with links to two cells (marked with (+) and (-)) gives an immediate access when compared to tree structures, in which traversing the tree becomes necessary to find the neighboring cells.

3.1 Generation

The simplest way is to define vertices and connectivity information explicitly; through *addVertex* and *addCell* operators on the grid. The vertices will be added to the list in the same order. Alternatively, a Cartesian 2-D block can be formed by directly creating a *Block2* object. Following presents an example for creating a single cell using existing vertices and directly creating another 5x5 grid in a square domain using a single command.

```
Grid* grid = new Grid();
grid->addVertex({ {0, 0, 0}, {1, 0, 0}, {1, 1, 0},
                  {0, 1, 0}, {0, 2, 0}, {1, 2, 0} });
grid->addCell({ { 3, 2, 5, 4} });
Block2* block = new Block2({0, 0, 0}, {1, 1, 0}, 5, 5);
```

3.2 *h*-adaptation

Each cell contains a current level information as well as a recommended adaptation level. This is identified during an error estimate procedure using either any primary/derived variable, or geometric information. The levels can be set to 0 at any time during computation, indicating that cells at level 0 cannot be further coarsened. Maximum possible level in each direction can also be set by the user. The default maximum number of levels for a grid is defined as 4.

When the desired level in a particular direction is higher than the current level, the cell is refined; split in to two in that particular direction. One of these cells becomes the master for that particular grid level. When the desired level in a particular direction is lower than the current level for two adjacent cells, they are merged into a single one. In order to facilitate an order, the coarsening procedure can only be initiated by the master of that particular level.

Before the refinement/coarsening procedure starts, the smooth level variation is enforced by checking adjacent cells. Even if they are not identified as either for refinement or coarsening, they can be refined to ensure smooth level change in a particular direction.

The procedure handles each direction separately. Refinement in a given direction is completed first starting from the highest cell level to the coarsest and the refinement continues in the other direction. Once refinement procedure is completed, the algorithm seeks for those remaining cells those are marked for coarsening. Similar to the refinement procedure, each direction is completed for refinement before going into the next. The coarsening procedure does not remove cells from the memory but it does mark both cells and vertices as unused. These unused cells are cleaned from the memory through a *clean*() command at certain frequencies. Faces are reconstructed using the new vertex and cell information.

Note that *h*-adaptation results in hanging nodes as shown in the figure. Hanging vertices hold the same cell index for two consecutive cell indices. No special treatment is needed for the hanging vertex as faces are formed using each vertex and possible connected neighboring vertices. For cells *E*, *P1*, and *P2*, four vertices are stored. This may cause cell *E* not to see the hanging vertex but through west faces of *P1* and *P2*, *E* becomes aware of the hanging vertex. A simple example to illustrate its use is given as follows,

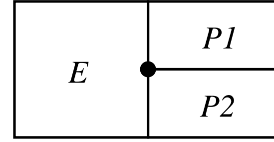

Figure 2: Hanging vertex

```
cell1->adapt[0] = 1;     // mark cell1 to be refined in x-direction
cell1->adapt[1] = 1;     // mark cell1 to be refined in y-direction
cell2->adapt[0] = -1;    // mark cell2 to be coarsened in x-direction
grid->adapt();           // start adaptation procedure
```

3.3 *r*-refinement

Complex geometries in Cartesian meshes can be handled various ways. One of them is based on the immersed boundary method, which uses forcing functions to impose no-slip surface on an interface. Another alternative is to use cut-cell approach. Both methods can produce quality results while there may face algorithmic problems such as avoiding formation of small cells. Body fitted grids do not face such algorithmic problems. As a result, one can choose to use either method when the flow happens around complex geometries. When combined with *h*-adaptation, *r*-adaptation can be quite powerful.

It is possible to shift vertices around a complex geometry using the solution of a Poisson's equation. Coordinates defining complex geometry can be supplied as the boundary condition for the solver.

3.4. Adaptation criteria

Error estimate can be based on any primary/derived variable. In isotropic adaptation, one decides to refine and coarsen a cell based on a scalar error estimate, whereas for anisotropic adaptation, scalars may not be useful as they cannot provide information on a direction biased refinement level. Gradient based error estimates are reported to produce ineffective grids resolutions.

This paper considers a measure linearity after comparing two different interpolation methods producing ϕ_f^a and ϕ_f^b at a face center. If the values of ϕ at the vertices of this face are ϕ_+ and ϕ_-, non-dimensional error is defined as follows:

$$\epsilon = \begin{cases} |\phi_f^a - \phi_f^b| / |\phi_+ - \phi_-| & \phi_+ \neq \phi_- \\ 0 & \phi_+ = \phi_- \end{cases} \tag{1}$$

If the function is linear or close to linear, the difference between ϕ_f^a and ϕ_f^b is small yielding an error close to zero. This can only happen when the region is adequately resolved. On the other hand, a large error hints for a split of this face, causing adjacent cells to be marked for refinement. We define lower and upper bound of acceptable non-dimensional thresholds for the error. If the error is larger than the upper threshold, the cell is marked to be refined so that the face is split into two. If the error is smaller than the lower threshold, the cell is marked to be coarsened in that particular direction.

```
grid->solBasedAdapt2(grid->getError(T)); // mark cells for adaptation
                                         // using the error in field var T
grid->adapt();
```

4 Solver

4.1 Field variables

Field variables are special 1D arrays that also contain boundary condition information. Variables can be defined as an unknown of a linear system or obtained through a solution of a partial differential

equations. New variables can be declared, solved and manipulated within the program. Boundary conditions can be defined as block or can also be defined for each individual cell through Dirichlet or Neumann conditions in the following form:

$$\phi_{BC} = a\phi_p + b \tag{2}$$

$$\frac{\partial \phi}{\partial n} = a\phi_p + b \tag{3}$$

where one needs to specify the type, and coefficients of a and b for a given cell. ϕ_p is the value stored at the cell of a boundary face. Variables also come with linear system (and their solvers). Customization to the behavior of the linear solver can be performed over the variable handle. An example for the use of variables is as follows:

```
grid->addVar({"T", "p"});
auto T = grid->getVar("T");
T->set(250);                              // Initialize all cells with 250
T->set(-2*sin(pi*y)*cos(pi*x));           // Initialize using formula (x
b = 0; a = 0;                             // and y can be 1D arrays)
T->setBC("west", "val", b, a);            // Dirichlet conditions on west
T->setBC("south", "grad", -250*20, 250);// Neumann conditions on south
T->itmax = 1000;                          // Maximum number of iterations
T->tol = 1e-6;                            // residual level
T->solver = "BiCGStab";                   // Solver to be used
```

4.2 Scheme

Scheme is a special type of construct that stores neighboring cells and their weights in a calculation, as well as a constant. This can be seen as a single row of a linear equation system. Schemes are generated by interpolation and differential operators and they can produce numerical values using their *eval* method.

4.3 Gradient operator

Gradient at cell centers are computed using Gauss's theorem. Cell-centered stored variables require interpolation operator on faces; integrated over the cell surface.

$$\vec{\nabla} T_p = \frac{1}{\Delta V_p} \sum_{faces} \phi_f \hat{n}_f A_f \tag{4}$$

where ϕ_f is computed at the faces using an interpolation scheme.

$$\phi_f = \frac{\alpha_N \phi_N + \alpha_P \phi_P}{\alpha_N + \alpha_P} \tag{5}$$

where α are the weights based on a distance projected on face normal. Normal gradient at the face centers are computed by central difference operator;

$$\left.\frac{\partial \phi}{\partial n}\right|_f = \frac{\phi_N - \phi_p}{\Delta \vec{x} \cdot \hat{n}} \tag{6}$$

Gradient operator comes in two types: (1) one that yields a row of stiffness matrix in terms of triplets, and (2) one that yields a value using current variables.

4.4 Laplacian operator

Laplacian term often appear in many conservation laws and represented as follows:

$$\oint_{CS} \Gamma \frac{\partial \phi}{\partial n} dA = \sum_{faces} \Gamma \left.\frac{\partial \phi}{\partial n}\right|_f A_f \tag{7}$$

where normal gradient is computed using Eqn. (6). Laplacian operator is only defined at the cell-centers and it is set to produce a row of the linear system. To illustrate its use, following lines can be used for solving heat equation:

```
grid->lockBC(T);                            // use T's boundary conditions
auto triplets = grid->laplace(k);           // => k[∂j∂j(T)]*vol
grid->unlockBC(T);                          // unsets current variable
```

4.5 Divergence operator

Divergence operator is used for advection term in conservation equations. For simplicity, first order upwind (FOU) scheme is considered at first. For FOU, fluxes are computed at face centers based on flow direction.

$$\vec{\phi}_f = \begin{cases} \vec{\phi}_U & u_f > 0 \\ \vec{\phi}_D & u_f < 0 \end{cases} \tag{8}$$

For a given cell, the scheme is obtained as follows:

$$\vec{\nabla} \cdot (\rho \vec{u} \phi) = \frac{1}{\Delta V_p} \sum_{faces} \rho \phi_f \vec{u}_f \cdot \hat{n}_f A_f \tag{9}$$

where ρ and $\vec{u}$ are provided as input parameters. The form of the operator is similar to the Laplacian operator:

```
grid->lock(T);                        // Sets current variable to T
auto triplets = grid->div(u, rho)     // =div(rho*u*T)
grid->unlock(T);                      // unsets current variable
```

4.6 Time integration

Generalized one step time integration is considered as follows:

$$\frac{\phi_P^{n+1} - \phi_p^n}{\Delta t} = w_1 F(\phi^{n+1}) + w_2 F(\phi^n) + w_3 F(\phi^{n-1}) \tag{10}$$

Time scheme stores the previous time step's linear system in Var container and uses when optionally specified weights on any available operators, i.e. laplace or div, separately. Following example, specifies Adam-Bashforth for advection and Crank-Nicolson for diffusion term in the following y-momentum equation:

```
grid->div(vel, rho), {0, 1.5, -0.5});       // w1=0, w2=1.5, w3=-0.5 → AB
grid->laplace(mu, {0.5, 0.5})               // w1=0.5, w2=0.5, w3=0 → CN
```

5 Examples of *girdap* code

5.1 Steady heat equation

Heat diffusion equation is solved considering various boundary conditions, including specified temperature, adiabatic wall temperature, Neumann condition and convection condition at various locations. Consider the following steady state example with volumetric heat generation, $\dot{q}$, in a unit square domain $x = [0,1]$ and $y = [0,1]$:

$$\frac{\partial}{\partial x}\left(k \frac{\partial T}{\partial x}\right) + \frac{\partial}{\partial y}\left(k \frac{\partial T}{\partial y}\right) + \dot{q} = 0 \tag{11}$$

$$\dot{q} = 5000 \text{ Wm}^{-3}, k = 2 \text{ Wm}^{-1}\text{K}^{-1}$$

Boundary conditions for Eqn. (11) are given as follows: at *x*=0 and *x*=1; temperature is specified as 100 ^{o}C, and 200 ^{o}C respectively, the surface at *y*=0 is insulated and the surface at *y*=1 is cooled by air flow at 20 ^{o}C with a convective heat transfer coefficient of 50 Wm^{-2}K^{-1}.

```
auto k = 2.0; auto qdot = 5000; auto h = 50; auto Tinf = 20;
Block2* grid = new Block2({0, 0, 0}, {1, 1, 0}, 40, 40);
grid->addVar("T");
auto T = grid->getVar("T");
```

```
T->set(100);

T->setBC("south", "grad", 0);
T->setBC("north", "grad", h/k, -h/k*Tinf); // h/k(T - Tinf)
T->setBC("east", "val", 200);
T->setBC("west", "val", 100);

grid->lockBC(T);                            // for boundary conditions
T->solve(   grid->laplace(k)                // thermal conductivity as Γ
          + grid->source(0, qdot) );
grid->unlockBC(T);

delete(grid);
```

Figure (3) shows the results on a uniform grid. Above algorithm is also tested after automatically refined several times based on the introduced error estimate and results are presented in Table (1).

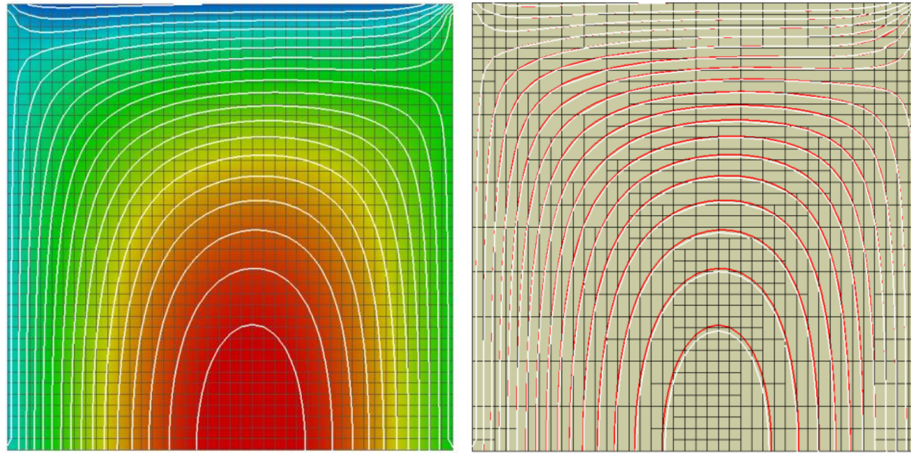

Figure 3: Steady state solution of heat equation represented by contours of temperature. Left plot is obtained using 40x40 grid; and right plot is obtained after h-adaptation of 10x10 by two levels. Contours are compared on the right plot.

Table 1. Comparison of *h*-adaptation for heat equation

	Ref (40x40)	*h*-adaptation on 10x10 with 2 level refinement in each dir					
N_{cell}	1600	118	299	638	753	795	804
T_{min} (ºC)	54.9	61.5	56.7	55.1	54.7	54.8	54.8
T_{max} (ºC)	426.0	422.3	424.7	424.7	424.4	424.4	424.4
*L*2 error	-	1.8357	0.47307	0.18305	0.12492	0.11204	0.11108

5.2 Fractional step method

Pressure based methods are suitable for incompressible flows. Fractional step method is considered to solve incompressible Navier-Stokes equations and consists of following steps:

[1] Solve for velocity components, without imposing continuity and pressure gradient

$$\frac{u_P^* - u_P^n}{\Delta t} + \vec{\nabla}\cdot(\rho\vec{u}u) = \frac{\partial}{\partial x_i}\left(\mu\frac{\partial u}{\partial x_i}\right) + \rho g_x$$

$$\frac{v_P^* - v_P^n}{\Delta t} + \vec{\nabla}\cdot(\rho\vec{u}v) = \frac{\partial}{\partial x_i}\left(\mu\frac{\partial v}{\partial x_i}\right) + \rho g_y$$

[2] Get face velocities and solve pressure Poisson equation

$$\frac{\partial}{\partial x_i}\left(\frac{1}{\rho}\frac{\partial p}{\partial x_i}\right) = \frac{\vec{\nabla}\cdot\vec{u}^*}{dt}$$

[3] Correct velocity to obtain divergence free flow field

$$u_f^{n+1} = u_f^* - \frac{dt}{\rho}\frac{\partial p}{\partial x}$$

$$v_f^{n+1} = v_f^* - \frac{dt}{\rho}\frac{\partial p}{\partial y}$$

[4] Interpolate velocities back to cell centers

This procedure can be defined as follows:

```
// timeloop begin:

//compute vorticity using mass consistent velocity field
vor->set(grid->valGrad(v).comp(0) - grid->valGrad(u).comp(1));

auto vel = grid->getVel();                  // collect current u,v,w

grid->lock(u);
u->solve(  grid->ddt(rho)                   // 1a – solve for u*
         + grid->div(vel, rho)              // backward in time
         - grid->laplace(mu, {0.5})         // Crank-Nicholson
           );
grid->unlock();

grid->lock(v);
v->solve(  grid->ddt(rho)                   // 1b – solve for v*
         + grid->div(vel, rho)              // backward in time
         - grid->laplace(mu, {0.5})         // Crank-Nicholson
         );
grid->unlock();

grid->solBasedAdapt(vor->data, 0.9);        // Sol. based adaptation
grid->adapt();                              // based on stat. variance

auto velstar = grid->getVel();              // collect uncorrected u* v*

grid->lockBC(p);
p->solve(   grid->laplace(dt/rho)           // solve for pressure
         - grid->source(0, grid->valDiv(velstar))
       );
grid->unlockBC();

gp = grid->valGrad(p);                      // get pressure gradient

u->set(velstar.comp(0)-dt/rho*gp.comp(0));  // correct velocities
v->set(velstar.comp(1)-dt/rho*gp.comp(1));

// timeloop end
```

Note that Statistical measures are considered for refinement and coarsening on vorticity. The adaptation is placed just before solving the pressure Poisson equation and criteria are based on 0.9σ up and down for refinement and coarsening, respectively. Figure (4) shows the results on an adaptively refined grid. .

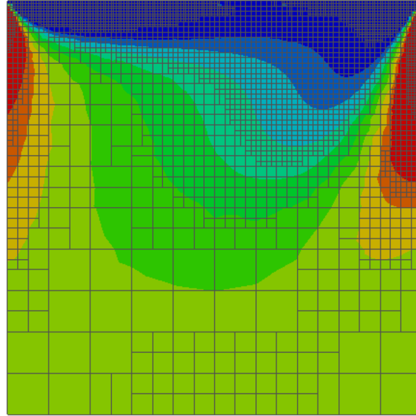

Figure 4: *h*-adaptation for lid-driven cavity flow at *Re*=100.

5.3 Other examples

Geometric based *r*-adaptation is demonstrated through solution of heat equation in a circular domain obtained by solving Poisson's equation. The results are shown for outer domain morphing as given in Fig 5(a) and for subtraction of a hole from a square domain as illustrated in Figure 5(b).

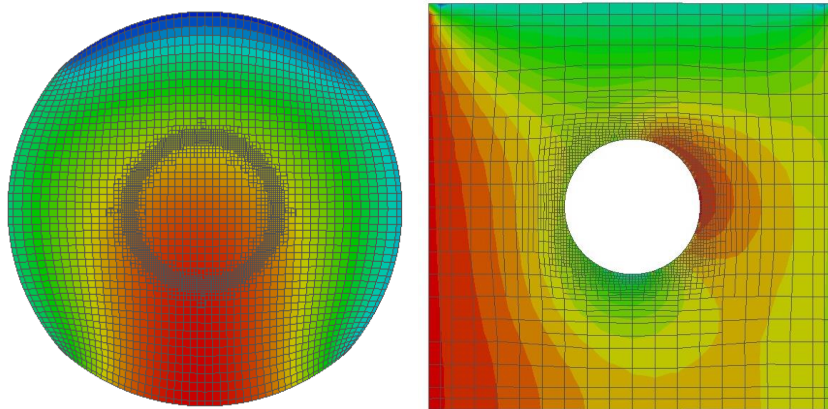

Figure 5: (a) r-adaptation on the outer domain (b) h*r*-adaptation around a cylindrical hole.

3 Conclusion and Future Work

This paper presents a new object oriented library for automated grid management that provides tools for researchers and educators to nourish new numerical methods using adaptive strategies.

Acknowledgement

This research was supported by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme (Project 268426).

References

[1] B. Nebenführ. OpenFOAM: A tool for predicting automotive relevant flow fields. 2010.

[2] A. Logg, K.-A. Mardal, and G. Wells, Eds. *Automated Solution of Differential Equations by the Finite Element Method.* 84, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[3] F. Bassetti, D. Brown, K. Davis, W. Henshaw, and D. Quinlan, "Overture: An Object-oriented Framework for High Performance Scientific Computing," in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, Washington, DC, USA, 1998, pp. 1–9.

[4] K. J. Fidkowski and D. L. Darmofal. Review of Output-Based Error Estimation and Mesh Adaptation in Computational Fluid Dynamics. *AIAA J.*, 49(4):673–694, 2011.

[5] "uzgoren/girdap," *GitHub*. [Online]. Available: https://github.com/uzgoren/girdap. [Accessed: 27-Jan-2016].