Serdica J. Computing **3** (2009), 249–268

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

AN ADAPTIVE QUADRILATERAL MESH IN CURVED DOMAINS

Sanjay Kumar Khattri

Abstract. An nonlinear elliptic system for generating adaptive quadrilateral meshes in curved domains is presented. The presented technique has been implemented in the C++ language with the help of the standard template library. The software package writes the converged meshes in the GMV and the Matlab formats. Grid generation is the first very important step for numerically solving partial differential equations. Thus, the presented C++ grid generator is extremely important to the computational science community.

1. Introduction. Quadrilateral meshes are used for visualization, interpolation and numerically solving partial differential equations. The accuracy of a numerical solution of partial differential equation strongly depends on the quality of the underlying mesh [22, 2, 3, 4, 7, 5]. Here, quality means orthogonality at the boundaries and quasi-orthogonality within the critical regions, smoothness, bounded aspect ratios and solution adaptive behavior.

ACM Computing Classification System (1998): G.1.0, G.4, G.1.7.
Key words: Adaptation, C++, STL, coupled elliptic system, grid generation.

Grid adaptation is used for increasing the efficiency of numerical schemes by focusing the computational effort where it is needed [19, 18]. In this work, we present the elliptic grid generation system for generating adaptive quadrilateral meshes. The presented method has been implemented in the C++ language. The presented method generates adaptive meshes without destroying the structured nature of the mesh. It is easier to develop solvers based on a structured mesh than on an unstructured mesh [20]. There are various software packages available for generating adaptive triangular meshes, but we do not know of any software package that can be used for generating adapting quadrilateral meshes. Thus, the presented method and its C++ implementation are very useful for solving partial differential equations.

For meshing a domain into non-simplex elements (quadrilaterals in 2D and hexahedra in 3D), we seek a mapping from a reference square or cube to the physical domain. This mapping can be algebraic in nature such as Transfinite Interpolation, or it can be expressed by a system of nonlinear partial differential equations such as an elliptic system [6, 8, and references therein]. We are looking for a vector mapping, $\mathcal{F}_k(\hat{k}) = (x, y)^t$, from a unit square in the reference space ($\hat{k} = [0, 1] \times [0, 1]$) to a physical space ($k$), that is $\mathcal{F}_k\colon \hat{k} \longmapsto k$. See Figure 1. Mapping $\mathcal{F}_k$ gives the position of a point in the physical space corresponding to a point in the computational or reference space. Let the physical space be given by the x and y coordinates, and the computational space be given by the ξ and η coordinates. Here, $\xi \in [0, 1]$ and $\eta \in [0, 1]$. We are using the following elliptic system for defining the mapping $\mathcal{F}_k = (x, y)^t$

$$g_{22}\,\frac{\partial^2 x}{\partial \xi^2} - 2\,g_{12}\,\frac{\partial^2 x}{\partial \xi \partial \eta} + g_{11}\,\frac{\partial^2 x}{\partial \eta^2} + P\,x_\xi + Q\,x_\eta = 0, \tag{1}$$

$$g_{22}\,\frac{\partial^2 y}{\partial \xi^2} - 2\,g_{12}\,\frac{\partial^2 y}{\partial \xi \partial \eta} + g_{11}\,\frac{\partial^2 y}{\partial \eta^2} + P\,y_\xi + Q\,y_\eta = 0. \tag{2}$$

Here, terms P and Q are used for grid adaptation and are given as

$$P = g_{22}\,P^1_{11} - 2\,g_{12}\,P^1_{12} + g_{11}\,P^1_{22}, \tag{3}$$

$$Q = g_{22}\,P^2_{11} - 2\,g_{12}\,P^2_{12} + g_{11}\,P^2_{22}. \tag{4}$$

Equations (1–2) are non-linear and are coupled through metric coefficients g_{ij} (coefficients of the metric tensor). Metric coefficients are given as

$$g_{11} = x_\xi^2 + y_\xi^2, \quad g_{22} = x_\eta^2 + y_\eta^2 \quad \text{and} \quad g_{12} = x_\xi\,x_\eta + y_\xi\,y_\eta. \tag{5}$$

For generating grids in the physical space, elliptic system (1–2) is solved for coordinates (x, y) on a unit square in the computational space by the method of Finite Differences. The boundary of the physical domain is specified as the Dirichlet boundary condition on the unit square in the computational space. In Figure 1, $\mathbf{g}_1$ $(= \mathbf{r}_\xi)$ and $\mathbf{g}_2$ $(= \mathbf{r}_\eta)$ are the covariant base vectors at point (x_i, y_j). Figure 2 shows a finite difference stencil around point (ξ_i, η_j) in the computational space. A finite difference approximation of x_ξ and x_η at point (i, j) (see Figure 2) is

$$x_\xi = \frac{[x(i+1,j) - x(i-1,j)]}{2\,\Delta\xi} \quad \text{and} \quad x_\eta = \frac{[x(i,j+1) - x(i,j-1)]}{2\,\Delta\eta}.$$

Similarly, y_ξ and y_η can be defined. Here, we are assuming that the grid in the computational space is uniform. However, the grid in the physical space can be compressed or stretched.

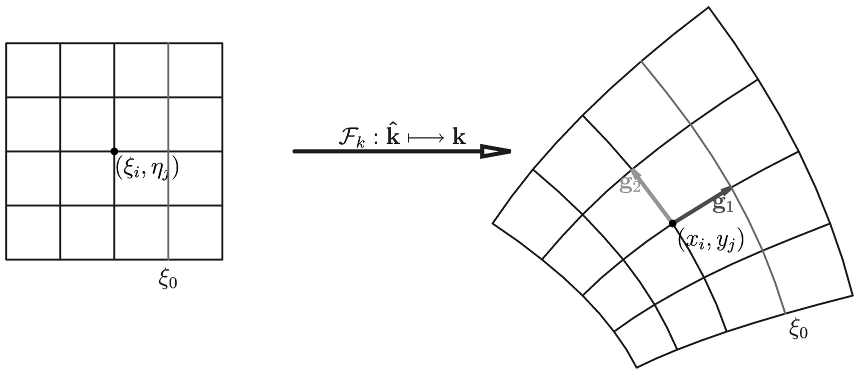

Fig. 1. Mapping $\mathcal{F}_k$ from a reference unit square $(\hat{k})$ on the left to a physical domain (k)

Terms P^k_{ij} in equations (1–2) are determined through another mapping $\mathcal{F}_1$. Here, $i = 1, 2$; $j = 1, 2$; $k = 1, 2$ and $P^k_{12} = P^k_{21}$. Mapping $\mathcal{F}_1$ is shown in Figure 3. This mapping maps a unit square in the computational space to a unit square in the parameter space. For defining mapping $\mathcal{F}_1 \colon \hat{k} \longrightarrow k_1$, the boundary and internal grid points of the parameter space are mapped to the reference space. The Jacobian matrix $\mathbf{T}$ of mapping $\mathcal{F}_1$ and vectors $\mathbf{P}_{11}$, $\mathbf{P}_{12}$ and $\mathbf{P}_{22}$ are given as follows

$$\mathbf{T} = \begin{pmatrix} s_\xi & s_\eta \\ t_\xi & t_\eta \end{pmatrix}, \qquad \mathbf{P}_{11} = -\mathbf{T}^{-1} \begin{pmatrix} s_{\xi\xi} \\ t_{\xi\xi} \end{pmatrix}, \tag{6}$$

$$\mathbf{P}_{22} = -\mathbf{T}^{-1} \begin{pmatrix} s_{\eta\eta} \\ t_{\xi\eta} \end{pmatrix}, \qquad \mathbf{P}_{12} = -\mathbf{T}^{-1} \begin{pmatrix} s_{\xi\eta} \\ t_{\xi\eta} \end{pmatrix}. \tag{7}$$

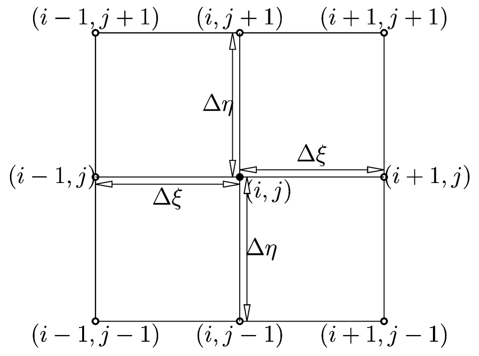

Fig. 2. Finite difference stencil in the ξ-η computational space

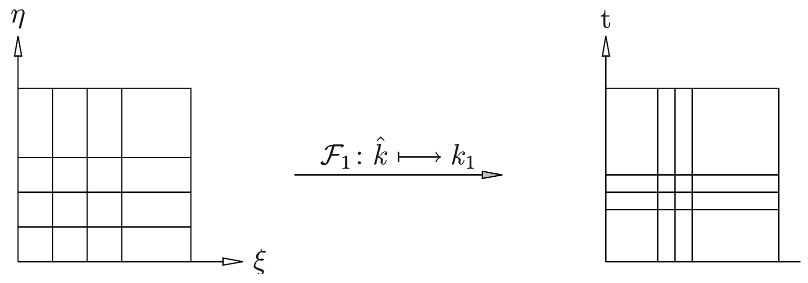

Fig. 3. Mapping $\mathcal{F}_1$ from a unit square $(\hat{k})$ in the reference space to a unit square in the parameter space (k_1)

Terms P_{ij}^1 $(i, j = 1, 2)$ are the first component of vector $\mathbf{P}_{ij}$, and terms P_{ij}^2 are the second component of the vector $\mathbf{P}_{ij}$. It should be noted that vectors $\mathbf{P}_{11}$, $\mathbf{P}_{12}$ and $\mathbf{P}_{22}$ can be computed a priori for clustering the grid points in the physical space. A second-order finite difference approximation of different operators required for computing vectors $\mathbf{P}_{11}$, $\mathbf{P}_{22}$, $\mathbf{P}_{12}$ and the Jacobian $\mathbf{T}$ are given in Table 1. We are using the stencil shown in Figure 2.

2. C++ Implementation. We have implemented the presented technique in the C++ language for generating adaptive grids. The software package can write meshes in the Matlab and GMV [21] formats. It consists of one Domain class. See subsections 2.2 and 2.3. Domain class is used for representing the unit square in the computational space, parameter space and the physical domain. See line numbers **44**, **22** and **37** in subsection 2.1.

The physical domain is defined in the file functions.h in subsection 2.6. For clustering grids in the parameter space different functions are defined in the domain class. See line numbers **25, 26, 28, 29, 31, 32** in subsection 2.2.

Table 1. Finite difference approximation of continuous operators

$$s_\xi = \frac{s(i+1,j) - s(i-1,j)}{2\,\Delta\xi}\,, \qquad s_{\xi\xi} = \frac{s(i+1,j) - 2\,s(i,j) + s(i-1,j)}{\Delta\xi^2}$$

$$t_\xi = \frac{t(i,j+1) - t(i,j-1)}{2\,\Delta\eta}\,, \qquad t_{\eta\eta} = \frac{t(i,j+1) - 2\,t(i,j) + t(i,j-1)}{\Delta\eta^2}$$

$$s_{\eta\eta} = \frac{s(i,j+1) - 2\,s(i,j) + s(i,j-1)}{\Delta\eta^2}\,, \qquad t_{\xi\xi} = \frac{t(i+1,j) - 2\,t(i,j) + t(i-1,j)}{\Delta\xi^2}$$

$$s_{\xi\eta} = \frac{s(i+1,j+1) + s(i-1,j-1) - s(i-1,j+1) - s(i+1,j-1)}{4\,\Delta\xi\Delta\eta}$$

$$t_{\xi\eta} = \frac{t(i+1,j+1) + t(i-1,j-1) - t(i-1,j+1) - t(i+1,j-1)}{4\,\Delta\xi\Delta\eta}$$

The coupled elliptic system are linearised by the method of Finite Differences, and the resulting system is solved by the SOR relaxation (see subsection 2.5). The SOR algorithm consists of three loops: the outer loop (see line number **42** in subsection 2.5) and two inner for loops (see line numbers **45** and **46** in subsection 2.5). Each iteration of an inner loop provides a new mesh by the SOR relaxation. The outer loop is controlled by the maximum number of SOR iterations (see line number **28** in subsection 2.5) and a given tolerance (see line number **26** in subsection 2.5).

The overall algorithm proceeds as follows. Generate grids in the computational and parameter spaces. Compute matrix $\mathbf{T}$ and vectors $\mathbf{P}_{ij}$ for defining mapping $\mathcal{F}_1$ (from computational space to parameter space). An initial grid, $\mathbf{r}_{\text{old}}$, in the physical region is generated by Transfinite Interpolation. This information is then passed to the SOR solver (see line number **42** in subsection 2.1).

2.1. main.cpp

```
 1 //++++++++++++++++
   #include <iostream>
 3 #include <iomanip>
   #include <vector>
 5 #include <iterator>
   #include <fstream>
 7 #include <sstream>
   #include <map>
 9 #include "domain.h"
   #include "matrix.h"
11 #include "sor_solver.cpp"
   //++++++++++++++++
13 int main(){
      bool grid_dist = true;
      bool run_ellip = true;
      unsigned xdim,ydim ;
      xdim = 31,ydim = 31;
      double del_xi = 1.0/double(xdim -1.0);
      double del_eta = 1.0/double(ydim -1.0);
      //Parameter Space ref(xdim,ydim);
      Domain parm(xdim,ydim);
      //Meshing the Parameter
      parm.Grid_Gen();
      //Clustering the Mesh
      //Example 1
      //parm.Cluster_X_Near(0.5);
      //parm.Cluster_Y_Near(0.5);
      //Example 2
      //parm.Cluster_Two_Lines_X(0.25,0.750);
      //parm.Cluster_Two_Lines_Y(0.25,0.750);
      //Example 3
      parm.Bound_Clust_X(0.5);
      parm.Bound_Clust_Y(0.5);

      parm.Fill_del_xi_eta(del_xi, del_eta);

      Domain physical(xdim,ydim);
      physical.Read_Bd();
      physical.Fill_del_xi_eta(del_xi, del_eta);
      unsigned max_iter = 100;
      double w = 1.90;
      SORSOLVER(physical, parm, xdim, ydim);
      //Reference or computational space
      Domain ref(xdim,ydim);

      //Writing the mesh in the physical space (GMV)
      std::ofstream outPhy("gmv_Physical.dat",std::ios::out);
      if (!(outPhy)) std::cerr << "ERROR␣:␣UNABLE␣TO␣OPEN␣\"outPhy\"\n";
      physical.GMV_Writer(outPhy);
      if (outPhy.is_open())  outPhy.close();

      //writing mesh in the parameter space (GMV)
      std::ofstream outParm("gmv_Para.dat",std::ios::out);
      if (!(outParm)) std::cerr << "ERROR␣:␣UNABLE␣TO␣OPEN␣\"outParm\"\n";
      parm.GMV_Writer(outParm);
      if (outParm.is_open())  outParm.close();
      parm.Matlab_Writer();

      return EXIT_SUCCESS;
}
```

2.2. domain.h

```
#ifndef PARAMETER_SPACE
#define PARAMETER_SPACE
//++++++++++++++++
#include<iostream>
#include<iomanip>
#include<vector>
#include<iterator>
#include<fstream>
#include<sstream>
#include<map>
//+++++++++++++++
#include "matrix.h"
//++++++++++++++
class Domain{
public:
      Domain();
      Domain(unsigned int xdim1, unsigned ydim1);
      Domain(const Domain & org);
      unsigned int XDIM() const ;
      unsigned int YDIM() const ;
      void Grid_Gen();
```

```
    std::vector<double> XCOORDS();
    std::vector<double> YCOORDS();
    double Eriksson_1(double eta);

    void Cluster_X_Near(double eta0);
    void Cluster_Y_Near(double eta0);

    void Cluster_Two_Lines_X(double eta1,double eta2);
    void Cluster_Two_Lines_Y(double eta1,double eta2);

    void Bound_Clust_X(double eta1);
    void Bound_Clust_Y(double eta1);

    double& XCOORD(unsigned int i , unsigned j);
    double& YCOORD(unsigned int i , unsigned j);
    void Read_Bd();
    void Matlab_Writer();
    void GMV_Writer(std::ofstream & outFile);
    void Fill_del_xi_eta(double xi,double eta);
    //void Call_Grid_Adapter();

    //+++++++++++++
    std::vector<double> P11(unsigned int i , unsigned int j);
    std::vector<double> P22(unsigned int i , unsigned int j);
    std::vector<double> P12(unsigned int i , unsigned int j);
    //++++++++++
    Matrix MeshX();
    Matrix MeshY();
    //+++++++++++++
    double G22(unsigned int i , unsigned int j);
    double G11(unsigned int i , unsigned int j);
    double X_xi(unsigned int i , unsigned int j);
    double Y_xi(unsigned int i , unsigned int j);
    double X_eta(unsigned int i , unsigned int j);
    double Y_eta(unsigned int i , unsigned int j);
    double X_xieta(unsigned int i , unsigned int j);
    double Y_xieta(unsigned int i , unsigned int j);

private:
    double del_eta,del_xi;
    unsigned xdim,ydim;
    Matrix x,y;
    std::vector<double> xcoords,ycoords;
};
#endif
```

2.3. domain.cpp

```
#include "domain.h"

#ifndef _FUNCTIONS_
#include "functions.h"
#endif

#include <cassert>

Domain::Domain(){
    xdim = 0 ; ydim = 0;
}
Domain::Domain(unsigned int xdim1,unsigned int ydim1){
    xdim = xdim1 ; ydim = ydim1;
}
Domain::Domain(const Domain & org){
    xdim = org.XDIM();
    ydim = org.YDIM();
    Grid_Gen();
}
unsigned int Domain::XDIM() const{
    return xdim;
}
unsigned int Domain::YDIM() const{
    return ydim;
}
void Domain::Grid_Gen(){
    Matrix xt(xdim,ydim),yt(xdim,ydim);
    assert(0 != xdim && 0 != ydim);
    for (unsigned int j = 0 ; j < ydim ; ++j){
        for (unsigned int i = 0 ; i < xdim ; ++i){
            double t_x = double(i)/double(xdim-1.0);
            double t_y  = double(j)/double(ydim-1.0);
            xt(i,j) = t_x ; yt(i,j) = t_y;
        }
    }
    x = xt ; y = yt;
}
std::vector<double> Domain::XCOORDS(){
    xcoords.resize(xdim*ydim); ycoords.resize(xdim*ydim);
    for (int j = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){
            int no = i+j*xdim;
            xcoords[no] = x(i,j);
            ycoords[no] = y(i,j);
        }
    }
    return xcoords;
}
std::vector<double> Domain::YCOORDS(){
    xcoords.resize(xdim*ydim); ycoords.resize(xdim*ydim);
    for (int j = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){
            int no = i+j*xdim;
            xcoords[no] = x(i,j);
            ycoords[no] = y(i,j);
        }
    }
    return ycoords;
}

void Domain::Bound_Clust_X(double eta1){

    double alpha = 4.0;
    double h = 1.0;
    double h2 = 1.0;
    double h1 = 0.0;

    for (int j  = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){

            if (x(i,j) <= eta1 && 0 <= x(i,j)){
                double eta = x(i,j);
                x(i,j) =(h2-h1)* eta1*(std::exp(alpha*eta/eta1)-1.0)/(std::exp(alpha)-1.0)+h1;
            }

            if (x(i,j) >= eta1 && x(i,j) <= 1.0){
                double eta = x(i,j);
                x(i,j) = (h2-h1)*(1.0-(1.0-eta1)*(((std::exp(alpha*(1.0-eta)/
                (1.0-eta1)))-1.0)/(std::exp(alpha)-1.0)));   }

        }
    }

}

void Domain::Bound_Clust_Y(double eta1){

    double alpha = 4.0;
    double h = 1.0;
    double h2 = 1.0;
    double h1 = 0.0;

    for (int j  = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){

            if (y(i,j) <= eta1 && 0 <= y(i,j)){
                double eta = y(i,j);
                y(i,j) =(h2-h1)* eta1*(std::exp(alpha*eta/eta1)-1.0)/(std::exp(alpha)-1.0)+h1;
            }

            if (y(i,j) >= eta1 && y(i,j) <= 1.0){
                double eta = y(i,j);
                y(i,j) = (h2-h1)*(1.0-(1.0-eta1)*(((std::exp(alpha*(1.0-eta)/
                (1.0-eta1)))-1.0)/(std::exp(alpha)-1.0)));}

        }
    }

}

//==========
double Domain::Eriksson_1(double eta){
    double h = 1.0;
    double alpha = 3.0;

    return h*((std::exp(alpha*eta)-1.0)/(std::exp(alpha)-1.0));


}

void Domain::Cluster_Two_Lines_X(double eta1,double eta2){

    double alpha = 5.0;
    double h = 1.0;
    double eta0 = (eta1+eta2)*0.5;

    for (int j  = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){

            if (x(i,j) <= eta1 && 0 <= x(i,j)){
                double eta = x(i,j);
                x(i,j) = eta1*(h-Eriksson_1(1-eta/eta1));
            }

            if (x(i,j) >= eta1 && x(i,j) <= eta0 ){
                double eta = x(i,j);
                x(i,j) = h*eta1+ (eta0-eta1)*Eriksson_1((eta-eta1)/(eta0-eta1));

            }

            if (x(i,j) >= eta0 && x(i,j) <= eta2){
                double eta = x(i,j);
                x(i,j) = h*eta0 + (eta2-eta0)*(h-Eriksson_1((eta2-eta)/(eta2-eta0)));
            }

            if (x(i,j) >= eta2 && x(i,j) <= 1.0){
                double eta = x(i,j);
                x(i,j) = h*eta2 + (1.0-eta2)*Eriksson_1((eta-eta2)/(1.0-eta2));
            }

        }
    }

}

void Domain::Cluster_Two_Lines_Y(double eta1,double eta2){

    double alpha = 5.0;
    double h = 1.0;
    double eta0 = (eta1+eta2)*0.5;

    for (int j  = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){

            if (y(i,j) <= eta1 && 0 <= y(i,j)){
                double eta = y(i,j);
                y(i,j) = eta1*(h-Eriksson_1(1-eta/eta1));
            }

            if (y(i,j) >= eta1 && y(i,j) <= eta0 ){
                double eta = y(i,j);
                y(i,j) = h*eta1+ (eta0-eta1)*Eriksson_1((eta-eta1)/(eta0-eta1));

            }

            if (y(i,j) >= eta0 && y(i,j) <= eta2){
                double eta = y(i,j);
                y(i,j) = h*eta0 + (eta2-eta0)*(h-Eriksson_1((eta2-eta)/(eta2-eta0)));
            }

            if (y(i,j) >= eta2 && y(i,j) <= 1.0){
                double eta = y(i,j);
                y(i,j) = h*eta2 + (1.0-eta2)*Eriksson_1((eta-eta2)/(1.0-eta2));
            }

        }
    }

}
//===========
void Domain::Cluster_X_Near(double eta0){
    double alpha = 3.0;
    for (int j  = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){
            if (x(i,j) < eta0){
                double eta = x(i,j);
                x(i,j) = (double) eta0*(exp(alpha)-exp(alpha*(double)(1-eta/eta0)))/
                (exp(alpha)-0.1e1);           }
            if (x(i,j) > eta0){
                double eta = x(i,j);
                x(i,j) = (double) eta0+(double)(1-eta0)*(exp((double)(alpha*(eta-eta0)/
                         (1-eta0)))-0.1e1)/(exp((double) alpha)-0.1e1);
            }
        }
    }

}

void Domain::Cluster_Y_Near(double eta0){
    double alpha = 3.0;
    for (int j  = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){
            if (y(i,j) < eta0){
                double eta = y(i,j);
                y(i,j) = (double)eta0*(exp(alpha)-exp(alpha*(double)(1-eta/eta0)))/
                (exp(alpha)-0.1e1);           }
            if (y(i,j) > eta0){
                double eta = y(i,j);
                y(i,j) = (double)eta0+(double)(1-eta0)*(exp((double)(alpha*(eta-eta0)/
                         (1-eta0)))-0.1e1)/(exp((double)alpha)-0.1e1);
            }
        }
    }
}
double& Domain::XCOORD(unsigned int i , unsigned int j) {
    if (i >= xdim || j >= ydim || i < 0 || j < 0){
        std::cerr << "In XCOORDS(..,..) dim mismatch\n";
    }
    return x(i,j);
}
double& Domain::YCOORD(unsigned int i , unsigned int j){
    if (i >= xdim || j >= ydim || i < 0 || j < 0){
        std::cerr << "In YCOORDS(..,..) dim mismatch\n";
    }
    return y(i,j);
}
void Domain::Read_Bd(){
//Read the boundary of the physical domain
    Matrix  xt(xdim,ydim),yt(xdim,ydim);
    for (int j = 0 ; j < ydim ; ++j){
        for (int i =0 ; i < xdim ; ++i){
            if (0 == i || xdim-1 == i || 0 == j || ydim-1 == j){
                double zeta1 = double(i)/double(xdim -1.0);
                double eta1  = double(j)/double(ydim -1.0);
                xt(i,j) = zeta1; yt(i,j) = eta1;
                xt(i,j) = XYcircle(zeta1,eta1,1);
                yt(i,j) = XYcircle(zeta1,eta1,2);

            }
        }
    }
//Create grid by the TFI
    for (int j = 1 ; j < ydim-1 ; ++j){
        for (int i = 1 ; i < xdim-1 ; ++i){
            double zeta1 = double(i)/double(xdim - 1.0);
            double eta1  = double(j)/double(ydim - 1.0);
            xt(i,j)=(1.0-zeta1)*xt(0,j)+zeta1*xt(xdim-1,j)+(1.0-eta1)*xt(i,0)+eta1*xt(i,ydim-1)-
                   ((1.0-zeta1)*(1.0-eta1)*xt(0,0)          +(zeta1)*(1.0-eta1)*xt(xdim-1,0)
                   + (zeta1)*(eta1)*xt(xdim-1,ydim-1)       +(1.0-zeta1)*eta1*xt(0,ydim-1));
            yt(i,j)=(1-zeta1)*yt(0,j)+zeta1*yt(xdim-1,j) +(1.0-eta1)*yt(i,0)+eta1*yt(i,ydim-1)-
                      ((1-zeta1)*(1-eta1)*yt(0,0)          + (zeta1)*(1-eta1)*yt(xdim-1,0)
                       + (zeta1)*(eta1)*yt(xdim-1,ydim-1)+ (1-zeta1)*eta1*yt(0,ydim-1));
        }
    }
    x = xt ; y = yt ;
}
//==============
void Domain::Matlab_Writer(){

    std::vector<double> x1 = XCOORDS();
    std::vector<double> y1 = YCOORDS();
    std::vector<double>::const_iterator viter;
    std::ofstream outfile("matlab_out.m",std::ios::out);
    if (!outfile) std::cerr << "Unable␣to␣open␣the␣matlab␣outfile\n";
    outfile << "clear;\n";
    outfile << "holdon=ishold;\n";

    for (int j = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){
            int no = i + j*xdim;
            outfile << "x1(" << i+1 << "," << j+1 << ")=" << x1[no] << ";␣␣␣"
           << "y1(" << i+1 << "," << j+1 << ")=" << y1[no] << ";" << std::endl;
        }
    }

    outfile << "m␣=␣␣" << xdim << std::endl;
    outfile << "n␣=␣␣" << ydim << std::endl;

    outfile << "plot(x1(1,:),y1(1,:),'r');␣hold␣on" << std::endl;
    outfile << "plot(x1(m,:),y1(m,:),'r');" << std::endl;
    outfile << "plot(x1(:,1),y1(:,1),'r');" << std::endl;
    outfile << "plot(x1(:,n),y1(:,n),'r');" << std::endl;

    outfile << "%␣Plot␣internal␣grid␣lines\n";
    outfile << "for␣i=2:m-1,␣plot(x1(i,:),y1(i,:),'b');␣end\n";
    outfile << "for␣j=2:n-1,␣plot(x1(:,j),y1(:,j),'b');␣end\n";

    outfile << "if␣(~holdon),␣hold␣off,␣end" << std::endl;

    outfile << "axis␣off;\n";

    outfile.close();
}

void Domain::GMV_Writer(std::ofstream & outFile){

    std::vector<double> xcoords = XCOORDS();
    std::vector<double> ycoords = YCOORDS();

    outFile << "gmvinput␣ascii\n";
    outFile << "nodes␣␣" << xdim*ydim << std::endl;

    for (int j = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){
            int no = i + j*xdim;
            outFile << xcoords[no] << "␣␣␣␣␣␣␣␣␣";

        }
    }
    outFile << std::endl << std::endl;
    //writing y coord
    for (int j = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){
            int no = i + j*xdim;
            outFile << ycoords[no] << "␣␣␣␣␣␣␣␣" ;

        }
    }
    outFile << std::endl << std::endl;
    //forming cells
    outFile << "cells␣␣" << (xdim-1)*(ydim-1) << std::endl;
    for (int j = 0 ; j < (ydim-1) ; ++j){
        for (int i = 0 ; i < (xdim-1) ; ++i){
            int no =  ( i + j*(xdim)  )+1;
            int no1 = ( i + (j+1)*(xdim)+1);
            outFile << "quad␣␣4␣␣" << std::endl;
            outFile << no << "␣␣␣" << no+1 << "␣␣␣"
           << no1+1 << "␣␣"  << no1 << std::endl;
        }
    }
    outFile << std::endl;

    outFile << std::endl << "endgmv\n";
    outFile.close();
}
//++++++++++++
Matrix Domain::MeshX(){
    return x;
}
Matrix Domain::MeshY(){

    return y;
}
//++++++++++++++++++
double Domain::G22(unsigned int i , unsigned int j){
    double x1;double x2;double y1;double y2;
    x1 = x(i,j-1); x2 = x(i,j+1) ;
    y1 = y(i,j-1); y2 = y(i,j+1);
    double g22 = std::pow((x2-x1)/(2.0*del_eta),2) +
             std::pow((y2-y1)/(2.0*del_eta),2);
    return g22;
}
double Domain::G11(unsigned int i , unsigned int j){
    double x1 = x(i-1,j); double x2 = x(i+1,j);
    double y1 = y(i-1,j); double y2 = y(i+1,j);
    double g11 = std::pow((x2-x1)/(2.0*del_xi),2) +
             std::pow((y2-y1)/(2.0*del_xi),2);
    return g11;
}
double Domain::X_xi(unsigned int i , unsigned int j){
    double x_xi;
    x_xi = (x(i+1,j)-x(i-1,j))/(2.0*del_xi);
    return x_xi;
}
double Domain::X_eta(unsigned int i , unsigned int j){
    double x_eta;
    x_eta = (x(i,j+1)-x(i,j-1))/(2.0*del_eta);
    return x_eta;
}
double Domain::Y_xi(unsigned int i , unsigned int j){
    double y_xi;
    y_xi = (y(i+1,j)-y(i-1,j))/(2.0*del_xi);
    return y_xi;
}
double Domain::Y_eta(unsigned int i , unsigned int j){
    double y_eta;
    y_eta = (y(i,j+1)-y(i,j-1))/(2.0*del_eta);
    return y_eta;
}
double Domain::X_xieta(unsigned int i , unsigned int j){
    return (x(i+1,j+1)+x(i-1,j-1)-x(i-1,j+1)-x(i+1,j-1))/(4.0*del_xi*del_eta);
}
```

```
double Domain::Y_xieta(unsigned int i , unsigned int j){
     return (y(i+1,j+1)+y(i-1,j-1)-y(i-1,j+1)-y(i+1,j-1))/(4.0*del_xi*del_eta);
}
void Domain::Fill_del_xi_eta(double xi,double eta){
     del_xi = xi; del_eta = eta;
}
std::vector<double> Domain::P11(unsigned int i , unsigned int j){
     //x-t coordinate
     //compute the jacobian at the point
     double s_xi   = (x(i+1,j)-x(i-1,j))/(2.0*del_xi);
     double s_eta  = (x(i,j+1)-x(i,j-1))/(2.0*del_eta);
     double t_xi   = (y(i+1,j)-y(i-1,j))/(2.0*del_xi);
     double t_eta  = (y(i,j+1)-y(i,j-1))/(2.0*del_eta);
     double det = s_xi*t_eta-t_xi*s_eta;
     double TI_11 =  t_eta/det; double  TI_12 = -t_xi/det;
     double TI_21 = -s_eta/det; double  TI_22 = s_xi/det;
     double s_xixi = (x(i+1,j)-2.0*x(i,j)+x(i-1,j))/(del_xi*del_xi);
     double s_etaeta = (x(i,j+1)-2.0*x(i,j)+x(i,j-1))/(del_eta*del_eta);
     double s_xieta = (x(i+1,j+1)+x(i-1,j-1)-x(i-1,j+1)-x(i+1,j-1))/(4.0*del_xi*del_eta);
     double t_xieta = (y(i+1,j+1)+y(i-1,j-1)-y(i-1,j+1)-y(i+1,j-1))/(4.0*del_xi*del_eta);
     double t_xixi   = (y(i+1,j)-2.0*y(i,j)+y(i-1,j))/(del_xi*del_xi);
     double t_etaeta = (y(i,j+1)-2.0*y(i,j)+y(i,j-1))/(del_eta*del_eta);
     std::vector<double> P11(2);

     P11[0] = -(s_xixi*TI_11+t_xixi*TI_12);
     P11[1] = -(s_xixi*TI_21+t_xixi*TI_22);

     return P11;
}
std::vector<double> Domain::P22(unsigned int i , unsigned int j){
     //x-t coordinate
     //compute the jacobian at the point
     double s_xi   = (x(i+1,j)-x(i-1,j))/(2.0*del_xi);
     double s_eta  = (x(i,j+1)-x(i,j-1))/(2.0*del_eta);
     double t_xi   = (y(i+1,j)-y(i-1,j))/(2.0*del_xi);
     double t_eta  = (y(i,j+1)-y(i,j-1))/(2.0*del_eta);
     double det = s_xi*t_eta-t_xi*s_eta;
     double TI_11 =  t_eta/det; double  TI_12 = -t_xi/det;
     double TI_21 = -s_eta/det; double  TI_22 = s_xi/det;
     double s_xixi = (x(i+1,j)-2.0*x(i,j)+x(i-1,j))/(del_xi*del_xi);
     double s_etaeta = (x(i,j+1)-2.0*x(i,j)+x(i,j-1))/(del_eta*del_eta);
     double s_xieta = (x(i+1,j+1)+x(i-1,j-1)-x(i-1,j+1)-x(i+1,j-1))/(4.0*del_xi*del_eta);
     double t_xieta = (y(i+1,j+1)+y(i-1,j-1)-y(i-1,j+1)-y(i+1,j-1))/(4.0*del_xi*del_eta);
     double t_xixi   = (y(i+1,j)-2.0*y(i,j)+y(i-1,j))/(del_xi*del_xi);
     double t_etaeta = (y(i,j+1)-2.0*y(i,j)+y(i,j-1))/(del_eta*del_eta);
     std::vector<double> P22(2);
     P22[0] = -(s_etaeta*TI_11+t_etaeta*TI_12);
     P22[1] = -(s_etaeta*TI_21+t_etaeta*TI_22);
     return P22;
}
std::vector<double> Domain::P12(unsigned int i , unsigned int j){
     //x-t coordinate
     //compute the jacobian at the point
     double s_xi   = (x(i+1,j)-x(i-1,j))/(2.0*del_xi);
     double s_eta  = (x(i,j+1)-x(i,j-1))/(2.0*del_eta);
     double t_xi   = (y(i+1,j)-y(i-1,j))/(2.0*del_xi);
     double t_eta  = (y(i,j+1)-y(i,j-1))/(2.0*del_eta);
     double det = s_xi*t_eta-t_xi*s_eta;
     double TI_11 =  t_eta/det; double  TI_12 = -t_xi/det;
     double TI_21 = -s_eta/det; double  TI_22 = s_xi/det;
     double s_xixi = (x(i+1,j)-2.0*x(i,j)+x(i-1,j))/(del_xi*del_xi);
     double s_etaeta = (x(i,j+1)-2.0*x(i,j)+x(i,j-1))/(del_eta*del_eta);
     double s_xieta = (x(i+1,j+1)+x(i-1,j-1)-x(i-1,j+1)-x(i+1,j-1))/(4.0*del_xi*del_eta);
     double t_xieta = (y(i+1,j+1)+y(i-1,j-1)-y(i-1,j+1)-y(i+1,j-1))/(4.0*del_xi*del_eta);
     double t_xixi   = (y(i+1,j)-2.0*y(i,j)+y(i-1,j))/(del_xi*del_xi);
     double t_etaeta = (y(i,j+1)-2.0*y(i,j)+y(i,j-1))/(del_eta*del_eta);
     std::vector<double> P12(2);
     P12[0] = -(s_xieta*TI_11+t_xieta*TI_12);
     P12[1] = -(s_xieta*TI_21+t_xieta*TI_22);
     return P12;
}
```

2.4. matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H
class Matrix{
public:
    unsigned int nx,ny;
    std::vector<std::vector<double> > Elements;
    Matrix(){
    }

    Matrix(unsigned int nx1,unsigned int ny1){
        nx = nx1 ; ny = ny1;
        //number of rows
        Elements.resize(nx);
        //fill each rows
        for (unsigned int i = 0 ; i < nx ; ++i)
            Elements[i].resize(ny,99.0);
    }

    void Clear(){
        for (unsigned int i = 0 ; i < nx ; ++i)
            Elements[i].clear();
    }

    double& operator()(unsigned int ix, unsigned int iy){
        return Elements[ix][iy];
    }
};
#endif
```

2.5. sor_solver.cpp

```
#ifndef SOR_SOLVER
#define SOR_SOLVER

#include <vector>

#include "domain.h"
#include "matrix.h"

double Mesh_Residual(Matrix x,   Matrix y ,
                     Matrix x_old , Matrix y_old ,
                     unsigned int xdim, unsigned int ydim){
    double resid = 0 ;
    for (int j = 0 ; j < ydim ; ++j){
        for (int i = 0 ; i < xdim ; ++i){
            double x_resd = (x(i,j)-x_old(i,j));
            double y_resd = (y(i,j)-y_old(i,j));
            resid += (x_resd*x_resd+y_resd*y_resd);
        }
    }
    return std::sqrt(resid);
}

bool SORSOLVER( Domain& physical , Domain & parm ,   unsigned int xdim1,unsigned int ydim1){
    bool grid_dist = true;
    bool run_ellip = true;

    double tolerance  = 1.0e-4;
    unsigned max_iter = 100;
    double w = 1.90;
    double residual = 10.0;
    unsigned int iter = 0;

    unsigned int xdim =xdim1; unsigned int ydim = ydim1;
    Matrix x_old(xdim,ydim), y_old(xdim,ydim);

    double del_xi = 1.0/double(xdim -1.0);
    double del_eta = 1.0/double(ydim -1.0);

    std::ofstream fout("gauss.dat",std::ios::out);
    if (!(fout.is_open())) std::cerr << "ERROR␣:␣UNABLE␣TO␣OPEN␣THE␣FILE␣\"gauss.dat\"\n";

    if (run_ellip){
        while (iter < max_iter & residual > tolerance){
            iter++;
            x_old = physical.MeshX() ; y_old = physical.MeshY();
            for (unsigned int j = 1 ; j < ydim-1 ; ++j){
                for (unsigned int i = 1 ; i < xdim-1 ; ++i){

                    double g22     = physical.G22(i,j);
                    double g11     = physical.G11(i,j);
                    double x_xi    = physical.X_xi(i,j);
                    double x_eta   = physical.X_eta(i,j);
                    double y_xi    = physical.Y_xi(i,j);
                    double y_eta   = physical.Y_eta(i,j);
                    double x_xieta = physical.X_xieta(i,j);
                    double y_xieta = physical.Y_xieta(i,j);
                    double g12 = x_xi*x_eta+y_xi*y_eta;
                    double g = std::pow(x_xi*y_eta-y_xi*x_eta,2);

                    std::vector<double> P11 , P22,P12;

                    if (grid_dist){
                        P11 = parm.P11(i,j);
                        P22 = parm.P22(i,j);
                        P12 = parm.P12(i,j);
                    }else{
                        P11.push_back(0);P11.push_back(0);
                        P22.push_back(0);P22.push_back(0);
                        P12.push_back(0);P12.push_back(0);
                    }

                    double tmpx, tmpy;

                    tmpx = (g22*P11[0]-2.0*g12*P12[0]+g11*P22[0])*x_xi+
                           (g22*P11[1]-2.0*g12*P12[1]+g11*P22[1])*x_eta;

                    tmpy = (g22*P11[0]-2.0*g12*P12[0]+g11*P22[0])*y_xi+
                           (g22*P11[1]-2.0*g12*P12[1]+g11*P22[1])*y_eta;

                    double lhsx = 2.0*(g22/(del_xi*del_xi)+g11/(del_eta*del_eta));
                    double rhsx = g22*(physical.XCOORD(i+1,j)+physical.XCOORD(i-1,j))/
                       (del_xi*del_xi)+g11*(physical.XCOORD(i,j+1)+physical.XCOORD(i,j-1))/
                       (del_eta*del_eta)-2.0*g12*x_xieta+tmpx;

                    physical.XCOORD(i,j)=physical.XCOORD(i,j)+w*(rhsx/lhsx-physical.XCOORD(i,j))

                    double lhsy = lhsx;
                    double rhsy = g22*(physical.YCOORD(i+1,j)+physical.YCOORD(i-1,j))/
                       (del_xi*del_xi)+g11*(physical.YCOORD(i,j+1)+physical.YCOORD(i,j-1))/
                       (del_eta*del_eta)-2.0*g12*y_xieta+ tmpy;

                    physical.YCOORD(i,j)=physical.YCOORD(i,j)+w*(rhsy/lhsy-physical.YCOORD(i,j))
                }
            }

            double mesh_resid = Mesh_Residual(physical.MeshX(),physical.MeshY(),x_old,
                                y_old,xdim,ydim)/((xdim-2)*(ydim-2));
            std::cout << "Iteration␣=␣␣␣" << iter << ",␣␣Residual␣=␣␣"
                      << mesh_resid << std::endl;

            fout << iter << "␣␣␣␣␣␣␣␣␣" << mesh_resid << std::endl;
            residual = mesh_resid;
        }
    }

    return true;
}
#endif
```

2.6. functions.h

```
#ifndef _FUNCTIONS_
#define _FUNCTIONS_

#include <vector>
#include <cmath>
#include<iomanip>
#include<iostream>

double XYcircle(double x, double y,unsigned int x_or_y){
    double r = 1.0;
    double theta = 0.0;
    double Pi = 4.0*atan(1.0);
    if (0==y){
        theta = Pi/2.0*x;
        if (x_or_y == 1)
            return r*cos(theta);
        else
            return r*sin(theta);
    }
    if (1==x){
        theta = Pi/2+Pi/2*y;
        if (x_or_y == 1)
            return r*cos(theta);
        else
            return r*sin(theta);
    }
    if (1==y){
        theta=Pi+Pi/2*(1.0-x);
        if (x_or_y == 1)
            return r*cos(theta);
        else
            return r*sin(theta);
    }
    if (0==x){
        theta = 3.0*Pi/2.0+Pi/2.0*(1.0-y);
        if (x_or_y == 1)
            return r*cos(theta);
        else
            return r*sin(theta);
    }
}
#endif
```

2.7. makefile

```
# Generic make file for LaTeX: requires GNU make
#
# This makefile provides four targets: dvi, ps, pdf and clean.
# The default is "pdf".
# To make a dvi file, type "make␣dvi"
# To make a ps file, type "make␣ps".
# To make a pdf file, type "make␣pdf" or simply "make".
# To remove all files generated by make, type "make␣clean".
#
#
#

TEXFILE =  Main_MS.tex


.PHONY: dvi ps pdf clean

pdf:    $(TEXFILE:.tex=.pdf)
ps:     $(TEXFILE:.tex=.ps)
dvi:    $(TEXFILE:.tex=.dvi)

%.dvi: %.tex
        ( \
        \latex $<; \
        while \grep -q "Rerun␣to␣get␣cross-references␣right."
$(<:.tex=.log); \
        do \
                \latex $<; \
        done \
        )

%.ps: %.dvi
        \dvips -q -t a4 $<

%.pdf: %.ps
        \ps2pdf -dPDFSETTINGS=/prepress $<

clean:
        @\rm -f \
        $(TEXFILE:.tex=.aux) \
        $(TEXFILE:.tex=.log) \
        $(TEXFILE:.tex=.out) \
        $(TEXFILE:.tex=.dvi) \
        $(TEXFILE:.tex=.ps)
```

3. Numerical Examples.

3.1. Example 1. In this example, we cluster the grids along the centre lines of a circular physical domain. Figure 4 shows the grid in the parameter space for concentrating grids at the centre lines of the physical space. Grid density in the physical space is determined by grid density in the parameter space. Figure 5 shows the converged grid in the physical space. For generating this grid lines **26** and **27** in Subsection 2.1 are used.

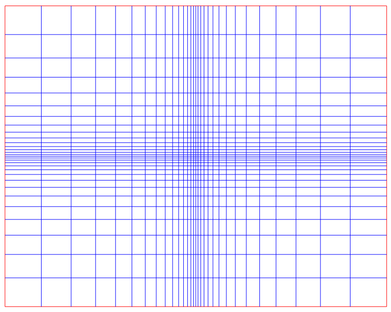
Fig. 4. A grid in the parameter space

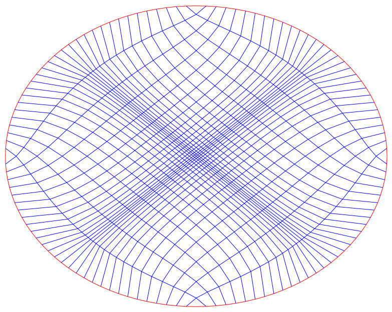
Fig. 5. Adapted grid by elliptic system

3.2. Example 2. See Figure 6 for grids in the parameter space and Figure 7 for the converged grids in the physical space. For generating the grids lines **29** and **30** of subsection 2.1 are used.

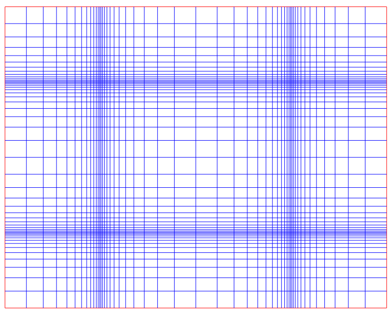
Fig. 6. A grid in the parameter space

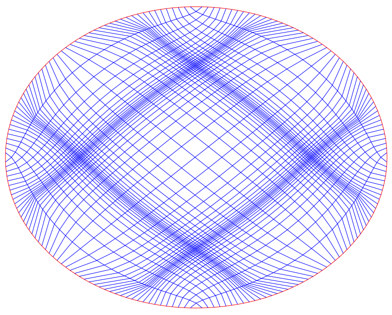
Fig. 7. Adapted grid by elliptic system

3.3. Example 3. In this example, we are interested in concentrating grids at the boundary of the physical space. Figure 8 shows the grid in the parameter space for concentrating grids at the boundary of the physical domain. The converged grids in the physical space is shown in Figure 9. For generating the grids, lines **32** and **33** of subsection 2.1 are used.

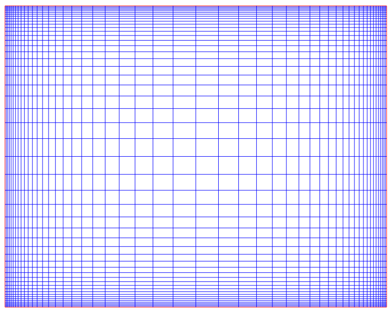
Fig. 8. A grid in the parameter space

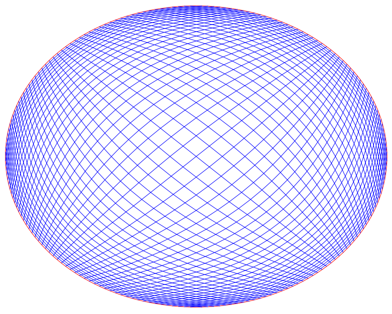
Fig. 9. Adapted grid by elliptic system

4. Conclusions.

An elliptic system for generating adaptive quadrilateral meshes in curved domains has been presented. A C++ implementation of the presented technique is also given. Three examples are reported for demonstrating the effectiveness of the technique and the implementation. Since the quadrilateral meshes are very extensively used for numerical simulations, and grid adaptation is required for capturing many important phenomenon such as the boundary layers, this software package is a very useful tool.

The software package can also be downloaded from the author's web-site.

REFERENCES

[1] Khattri, S. K. Grid generation and adaptation by functionals. *Mat. apl. comput.*, **26** (2007), No 2, 235–249.

[2] Khattri S. K. Analyzing Finite Volume for Single-Phase Flow in Porous Media. *Journal of Porous Media*, **10** (2007), No 2, 109–123.

[3] Khattri S. K. Nonlinear elliptic problems with the method of finite volumes. *Differ. Equ. Nonlinear Mech.,* Article ID 31797, 16 p., electronic only, 2006, doi:10.1155/DENM/2006/31797.

[4] Khattri S. K. An effective quadrilateral mesh adaptation. *Journal of Zhejiang University – Science A,* **7** (2007), No 12, 1862–1775.

[5] Khattri S. K. A New Smoothing Algorithm for Quadrilateral and Hexahedral Meshes. Lecture Notes in Computer Science, vol. **3992**, 2006, 239–246.

[6] Knupp P. M. Jacobian-weighted elliptic grid generation. *SIAM J. Sci. Comput.*, **17** (1996), 1475–1490.

[7] Khattri S. K. Which Meshes Are Better Conditioned: Adaptive, Uniform, Locally Refined or Locally Adjusted? *Lecture Notes in Computer Science,* **3992** (2006), 102–105.

[8] Winslow A. M. Numerical solution of the quasilinear poisson equation in a nonuniform triangle mesh. *J. Comput. Physics*, **1** (1966) No 2, 149–172.

[9] Lee S. H., B. K. Soni. The enhancement of an elliptic grid using appropriate control functions. *Appl. Math. Comput.*, **159** (2004), 809–821.

[10] Codd A. L., T. A. Manteuffel, S. F. McCormick, J. W. Ruge. Multilevel first-order system least squares for elliptic grid generation. *SIAM J. Numer. Anal.*, **41** (2003), 2210–2232 (electronic).

[11] Spiridonov V., A. Zhedanov. Classical biorthogonal rational functions on elliptic grids. *C. R. Math. Acad. Sci. Soc. R. Can.*, **22** (2000), 70–76.

[12] Golik W. L. Parallel solvers for planar elliptic grid generation equations. *Parallel Algorithms Appl.*, **14** (2000), 175–186.

[13] Mathur J. S., S. K. Chakrabartty. An approximate factorization scheme for elliptic grid generation with control functions. *Numer. Methods Partial Differential Equations*, **10**, (1994), 703–713.

[14] Soni B. K. Elliptic grid generation system: control functions revisited – I. *Appl. Math. Comput.*, **59** (1993), 151–163.

[15] Knupp, P. M. A robust elliptic grid generator. *J. Comput. Phys.*, **100** (1992), 409–418.

[16] Findling A., U. Herrmann. In: Proceedings of Numerical grid generation in computational fluid dynamics and related fields, Barcelona, 1991. North-Holland, Amsterdam, 781–792.

[17] Matsuno K., H. A. Dwyer. Adaptive methods for elliptic grid generation. *J. Comput. Phys.*, **77** (1998), 40–52.

[18] Cao W., W. Huang, R. D. Russell. A study of monitor functions for two-dimensional adaptive mesh generation. *SIAM J. Sci. Comput.*, **20** (1999), 1978–1994.

[19] Cao W., R. Carretero-González, Huang, W., Russell, R. D. Variational mesh adaptation methods for axisymmetrical problems. *SIAM J. Numer. Anal.*, **41** (2003), No 1, 235–257.

[20] Wu J.-S., K.-C. Tseng, C.-H. Kuo. The direct simulation Monte Carlo method using unstructured adaptive mesh and its application. *International Journal for Numerical Methods in Fluids*, **38** (2002), No 4, 351–375.

[21] Ortega F. The General Mesh Viewer. GMV is an easy to use, 3-D scientific visualization tool designed to view simulation data from any type of structured or unstructured mesh. Freely available at `http://laws.lanl.gov/XCM/gmv/GMVHome.html`.

[22] Khattri S. K., I. Aavatsmark. Numerical convergence on adaptive grids for control volume methods. *Numer. Methods Partial Differ. Equations,* **24** (2008), No 2, 465–475, `http://dx.doi.org/10.1002/num.20280`.

Sanjay Kumar Khattri
Stord Haugesund Engineering College
Norway
e-mail: `sanjay.khattri@hsh.no`
url: `http://ans.hsh.no/home/skk`

Received April 27, 2009
Final Accepted September 16, 2009