Adaptive Quadrilateral Mesh in Curved Domains

Sanjay Kumar Khattri
Department of Mathematics, University of Bergen, Norway
sanjay@mi.uib.no
www.mi.uib.no/~sanjay

Abstract: Nonlinear elliptic system for generating adaptive quadrilateral meshes in curved domains is presented. Presented technique has been implemented in the C++ language. The included software package can write the converged meshes in the GMV and Matlab formats. Since, grid adaptation is required for numerically capturing important characteristics of a process such as boundary layers. So, the presented technique and the software package can be a useful tool.

Keywords: Adaptation, C++, coupled elliptic system, grid generation

AMS Subject Classifications: 65M50, 65M20, 35K05

Computing Classification System codes: G.1.0, G.4 (Mathematical Software), G.1.7 (Numerical Analysis)

1 Introduction

Quadrilateral grids are extensively used for numerical simulation. Accuracy of a simulation is strongly depend on the grid quality. Here, quality means orthogonality at the boundaries and quasi-orthogonality within the critical regions, smoothness, bounded aspect ratios and solution adaptive behaviour. Grid adaptation is used for increasing the efficiency of numerical schemes by focusing the computational effort where it is needed. In this article, we review the elliptic grid generation system for generating adaptive quadrilateral meshes. The presented scheme has been implemented in the C++ language.

For meshing a domain into non-simplex elements (quadrilaterals in 2D and hexahedrals 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 [1; 2 and references therein] such as elliptic system. We are looking for a vector mapping, \( \mathcal{F}_k(\hat{k}) = (x, y)' \), from a unit square in the reference space (\( \hat{k} = [0, 1] \times [0, 1] \)) to a physical space (\( k \)); i.e. \( \mathcal{F}_k: \hat{k} \to k \) (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 $\xi$ and $\eta$ coordinates ($\xi \in [0, 1]$ and $\eta \in [0, 1]$). We are using the following elliptic system for defining the mapping $\mathscr{F}_k = (x, y)^T$

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

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

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

$$P = g_{22} P_{11}^1 - 2g_{12} P_{12}^1 + g_{11} P_{22}^1, 
$$

$$Q = g_{22} P_{11}^2 - 2g_{12} P_{12}^2 + g_{11} P_{22}^2. 
$$

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

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

For generating grids in the physical space, the elliptic system (1-2) is solved for the coordinates $(x, y)$ on a unit square in the computational space by the method of Finite Differences. Boundary of the physical domain is specified as the Dirichlet boundary condition on the unit square in the computational space. In the Figure 1, $g_1 (= r_\xi)$ and $g_2 (= r_\eta)$ are the covariant base vectors at the point $(x_i, y_j)$. Figure 2 shows a finite difference stencil around the point $(\xi_i, \eta_j)$ in the computational space. A finite difference approximation of $x_\xi$ and $x_\eta$ at the 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, $y_\xi$ and $y_\eta$ can be defined. Here, we are assuming that the grid in the computational space is uniform. However, grid in the physical space can be compressed or stretched. Terms $P_{ij}^k$ ($i = 1, 2$ and $j = 1, 2$ and $k = 1, 2$ and $P_{12}^1 = P_{21}^2$) in the equations (1-2) are determined through another mapping $\mathscr{F}_1$. The mapping $\mathscr{F}_1$ is shown in the Figure 3. This mapping maps a unit square in the computational space to a unit square in the parameter space. For defining the mapping $\mathscr{F}_1: \hat{k} \rightarrow k_1$, the boundary and internal grid points of the parameter space are mapped to the reference space. The Jacobian matrix $T$ of the mapping $\mathscr{F}_1$ and the vectors $P_{11}$, $P_{12}$ and $P_{22}$ are given as follows

$$
T = \begin{pmatrix} 
s_\xi & s_\eta \\
t_\xi & t_\eta \end{pmatrix}, \quad P_{11} = -T^{-1} \begin{pmatrix} s_\xi s_\eta \\
t_\xi t_\eta \end{pmatrix}, 
$$

$$
P_{22} = -T^{-1} \begin{pmatrix} s_\eta s_\xi \\
t_\eta t_\xi \end{pmatrix}, \quad P_{12} = -T^{-1} \begin{pmatrix} s_\eta t_\xi \\
t_\eta t_\xi \end{pmatrix}. 
$$

The terms $P_{ij}^1$ ($i, j = 1, 2$) are the first component of the vector $P_{ij}$ and the terms $P_{ij}^2$ are the
second component of the vector $P_{i,j}$. It should be noted that the vectors $P_{11}$, $P_{12}$ and $P_{22}$ 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 the vectors $P_{11}$, $P_{22}$, $P_{12}$ and the Jacobian $T$ are given in the Table 1. We are using the stencil shown in the Figure 2.

2 C++ Implementation

We have implemented the presented technique in the C++ language for generating adaptive grids. The package can write meshes in the Matlab and GMV [13] formats. It consists of one Domain class (see the subsections 2.2 and 2.3). Domain class is used for expressing unit square in the computational space (see line no. 045 in the subsection 2.1), unit square in the parameter space (see line no. 022 in the subsection 2.1) and the physical domain (see line no. 038 in the subsection 2.1). The physical domain is defined in the file functions.h (see the subsection 2.6). For clustering grids in the parameter space different functions are defined in the domain class (see the line numbers 026, 027, 029, 030, 032, 033 in the subsection 2.2).

The coupled elliptic system are linearised by the method of Finite Difference and the resulting system is solved by the SOR relaxation (see the subsection 2.5). The SOR algorithm
Figure 3: Mapping $F_1$ from a unit square ($\hat{k}$) in the reference space to a unit square in the parameter space ($k_1$).

Table 1: Finite difference approximation of continuous operators.

\[
\begin{align*}
    s_{\xi} &= \frac{s(i+1,j) - s(i,j)}{2\Delta\xi}, & s_{\xi\xi} &= \frac{s(i+1,j) - 2s(i,j) + s(i-1,j)}{\Delta\xi^2} \\
    t_{\xi} &= \frac{t(i,j+1) - t(i,j-1)}{2\Delta\eta}, & t_{\xi\eta} &= \frac{t(i,j+1) - 2t(i,j) + t(i,j-1)}{\Delta\eta^2} \\
    s_{\eta\eta} &= \frac{s(i,j+1) - 2s(i,j) + s(i,j-1)}{\Delta\eta^2}, & t_{\eta\eta} &= \frac{t(i+1,j) - 2t(i,j) + t(i-1,j)}{\Delta\eta^2} \\
    s_{\eta\xi} &= \frac{s(i+1,j+1) + s(i-1,j-1) - s(i-1,j+1) - s(i+1,j-1)}{4\Delta\xi\Delta\eta}, & t_{\eta\xi} &= \frac{t(i+1,j+1) + t(i-1,j-1) - t(i-1,j+1) - t(i+1,j-1)}{4\Delta\xi\Delta\eta}
\end{align*}
\]
The algorithm proceeds as follows. Generate grids in the computational and parameter spaces. Compute the matrix $T$ and vectors $P_{ij}$ for defining the mapping $F_1$ (from computational space to parameter space). An initial grid, $r_{old}$, in the physical region is generated (say by Transfinite Interpolation). This information is then passed to the SOR solver (see the line number 043 in the subsection 2.1).

### 2.1 main.cpp

```cpp
//+++++++++++++++++
#include <iostream>
#include <iomanip>
#include <vector>
#include <iterator>
#include <fstream>
#include <sstream>
#include <map>
#include "domain.h"
#include "write_matlab.h"
#include "matrix.h"
#include "sor_solver.cpp"

//+++++++++++++++++

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);

  return EXIT_SUCCESS;
}
```

The overall algorithm proceeds as follows. Generate grids in the computational and parameter spaces. Compute the matrix $T$ and vectors $P_{ij}$ for defining the mapping $F_1$ (from computational space to parameter space). An initial grid, $r_{old}$, in the physical region is generated (say by Transfinite Interpolation). This information is then passed to the SOR solver (see the line number 043 in the subsection 2.1).
2.2 domain.h

```cpp
#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(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();

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

2.3 domain.cpp

```cpp
#include "domain.h"
#ifndef FUNCTIONS
#define FUNCTIONS
#endif
#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);
    for (int j = 0; j < ydim; ++j)
    {
        for (int i = 0; i < xdim; ++i)
        {
            int no = i+j*xdim;
            xcoords[no] = x(i, j);
        }
    }
    return xcoords;
}

std::vector<double> Domain::YCOORDS()
{
    ycoords.resize(xdim*ydim);
    for (int j = 0; j < ydim; ++j)
    {
        for (int i = 0; i < xdim; ++i)
        {
            int no = i+j*xdim;
            ycoords[no] = y(i, j);
        }
    }
    return ycoords;
}

void Domain::Bound_Cluster_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_Cluster_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 (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)));
            }
        }
    }
}
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)));
}
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;
} } 
//==============
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);
}
\[ y(i,j) = h \eta^2 + (1.0 - \eta^2) \times \text{Eriksson}(\eta / (1.0 - \eta^2)) \]

```cpp
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);
            }
        }
    }
}
```

```cpp
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);
            }
        }
    }
}
```

```cpp
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);
}
```

```cpp
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);
}
```

```cpp
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;
            }
        }
    }
    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(i-1,j) + zeta1 * xt(i+1,j) + (1.0-eta1) * yt(i,j) - (1.0-zeta1) * eta1 * xt(0,j); // TFI
            yt(i,j) = (1.0-eta1) * yt(i,j) + (1.0-zeta1) * xt(i,j) - zeta1 * (1.0-eta1) * yt(i-1,j);
        }
    }
    x = xt; y = yt;
}
```

```cpp
void Domain::Matlab_Writer() {
}
```
std::vector<double> x1 = XCOORDS();
std::vector<double> y1 = YCOORDS();
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){
  int no = i + j * xdim;
  outfile << "x1(" << i+1 << "," << j+1 << ")=" << x1[no] << "; ";
  outfile << "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";
for(i=2:m-1, plot(x1(i,:),y1(i,:),'b'); end\n";
for(j=2:n-1, plot(x1(:,j),y1(:,j),'b'); end\n";
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;
    }
    //writing y coord
    for(int i = 0 ; i < xdim ; ++i){
        for(int j = 0 ; j < ydim ; ++j){
            int no = i + j * xdim;
            outFile << ycoords[no] << " ";
        }
        outFile << std::endl;
    }
    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)) + 1;
            outFile << "quad 4 " << std::endl;
            outFile << no << " " << no+1 << " "
            << no+1 << " " << no+std::endl;
        }
    }
    outFile << "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(x1[i-1]); x2 = x1[i+1];
    x1 = x(i,j-1); x2 = x(i,j+1);
369  y1 = y(i+1,j); y2 = y(i-1,j);
370  double g22 = std::pow((x2-x1)/2.0/del_eta,2) +
371    std::pow((y2-y1)/2.0/del_omega,2);
372  return g22;
373 }
374
375 double Domain::X_xi(unsigned int i , unsigned int j)
376 {
377  double x_xi;
378  x_xi = (x(i+1,j)-x(i-1,j))/(2.0/del_xi);
379  return x_xi;
380 }
381
382 double Domain::X_eta(unsigned int i , unsigned int j)
383 {
384  double x_eta;
385  x_eta = (x(i,j+1)-x(i,j-1))/(2.0/del_eta);
386  return x_eta;
387 }
388
389 double Domain::X_xieta(unsigned int i , unsigned int j)
390 {
391  double x_xieta;
392  x_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);
393  return x_xieta;
394 }
395
396 void Domain::Fill_del_xi_eta(double xi,double eta)
397 {
398  del_xi = xi; del_eta = eta;
399 }
400
401 std::vector<double> & Domain::P11(unsigned int i , unsigned int j)
402 {
403  //x-t coordinate
404  //compute the jacobian at the point
405  double x_xi = (x(i+1,j)-x(i-1,j))/(2.0/del_xi);
406  double x_eta = (x(i,j+1)-x(i,j-1))/(2.0/del_eta);
407  double det = x_xi*det - x_eta*del_xieta;
408  double TI_10 = -x_eta/det; double TI_12 = -x_xi/det;
409  double TI_11 = -x_xieta/del_xi + x_etaeta/del_eta;
410  double x_xi11 = (x(i+1,j)-2.0*x(i,j)+x(i-1,j))/(del_xi2+del_eta2);
411  double x_xi12 = (y(i+1,j)-2.0*y(i,j)+y(i-1,j))/(del_xi2+del_eta2);
412  double x_xi1eta = (y(i+1,j)-2.0*y(i,j)+y(i-1,j))/(del_xi2+del_eta2);
413  return P11[0] = -(s*xi+TI_11); P11[1] = -(s*xi+TI_12);
414  return P11;
415 }
416
417 std::vector<double> & Domain::P22(unsigned int i , unsigned int j)
418 {
419  //x-t coordinate
420  //compute the jacobian at the point
421  double x_xi = (x(i+1,j)-x(i-1,j))/(2.0/del_xi);
422  double x_eta = (x(i,j+1)-x(i,j-1))/(2.0/del_eta);
423  double det = -s*eta + t*xieta;
424  double TI_20 = -s/eta; double TI_21 = -t/xi;
425  double TI_22 = -y/xieta + x/etaeta;
426  double x_xi11 = (x(i+1,j)-2.0*x(i,j)+x(i-1,j))/(del_xi2+del_eta2);
427  double x_xi12 = (y(i+1,j)-2.0*y(i,j)+y(i-1,j))/(del_xi2+del_eta2);
428  double x_xi1eta = (y(i+1,j)-2.0*y(i,j)+y(i-1,j))/(del_xi2+del_eta2);
429  return P22[0] = -(s*eta+TI_21); P22[1] = -(s*eta+TI_22);
430  return P22;
431 }
432
433 std::vector<double> & Domain::P12(unsigned int i , unsigned int j)
434 {
435  //x-t coordinate
436  //compute the jacobian at the point
437  double x_xi = (x(i+1,j)-x(i-1,j))/(2.0/del_xi);
438  double x_eta = (x(i,j+1)-x(i,j-1))/(2.0/del_eta);
439  double det = -s*eta + t*xieta;
440  double TI_10 = -x_eta/det; double TI_11 = -x_xi/det;
441  double TI_12 = -x_xieta/del_xi + x_etaeta/del_eta;
442  double x_xi11 = (x(i+1,j)-2.0*x(i,j)+x(i-1,j))/(del_xi2+del_eta2);
443  double x_xi12 = (y(i+1,j)-2.0*y(i,j)+y(i-1,j))/(del_xi2+del_eta2);
444  double x_xi1eta = (y(i+1,j)-2.0*y(i,j)+y(i-1,j))/(del_xi2+del_eta2);
445  return P12[0] = -(s*xi+TI_11); P12[1] = -(s*xi+TI_12);
446  return P12;
447 }
2.4 matrix.cpp

2.5 sor_solver.cpp
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 xdim = xdim1;
unsigned 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" << std::endl;
}
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 xi = physical.Xxi(i, j);
                double eta = physical.Yeta(i, j);
                double xieta = physical.Xxieta(i, j);
                double g = std::pow(xi * eta - yi * xi, 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]) * 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]) * yi +
                    (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 * xeta + tmpx;
                double lhsy = 2.0 * (g22 / (del_xi * del_xi) + g11 / (del_eta * del_eta));
                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 * yeta + tmpy;
                physical.XCOORD(i, j) = physical.XCOORD(i, j) +
                    w * (rhsx / lhsx / physical.XCOORD(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;
        if(mesh_resid < std::endl) {
            fout << "Iteration = " << iter << " Residual = " << mesh_resid << std::endl;
        }
        return true;
    }
#}
2.6 functions.h

```c
# 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);
    }
    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 + Pi/2 * (1.0 - y);
        if (x_or_y == 1)
            return r * cos(theta);
        else
            return r * sin(theta);
    }
}
#endif
```

2.7 makefile

```bash
CXX = g++
SRC = main.cpp
domain.cpp
OBJ = main.o
OBJ = domain.o
ELL_MESH: $(OBJ)
ELL_MESH: $(OBJ) $(LIB)
clean:
rm -f ellmesh
```

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 the grid density in the parameter space. Figure 5 shows the converged grid in the physical space. For generating this grid the lines 027 and 028 in the subsection 2.1 are used.
3.2 Example 2

See the Figure 6 for grids in the parameter space and the Figure 7 for the converged grids in the physical space. For generating the grids the lines 030 and 031 of the subsection 2.1 are used.

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 the Figure 9. For generating the grids the lines 033 and 034 of the subsection 2.1 are used.
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. Thus, this software package can be a very useful tool.

Our package is freely available at www.mi.uib.no/~sanjay. The authors want to mention that they donot know of any freely available software package that can generate adaptive quadrilateral meshes.

References

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

[2] Winslow, A. M., 1967, Equipotential zoning of two dimensional meshes. J. Comput. Physics, 1.

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

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

[5] Spiridonov, V. and Zhedanov, A., 2000, Classical biorthogonal rational functions on elliptic grids. C. R. Math. Acad. Sci. Soc. R. Can., 22, 70–76.
[6] Golik, W. L., 2000, Parallel solvers for planar elliptic grid generation equations. *Parallel Algorithms Appl.*, 14, 175–186.

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

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

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

[10] Findling, A. and Herrmann, U., 1991, In the proceedings of Numerical grid generation in computational fluid dynamics and related fields (Barcelona, 1991), 781–792, North-Holland, Amsterdam.

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

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

[13] 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](http://laws.lanl.gov/XCM/gmv/GMVHome.html)
\[ F_1 : \hat{K} \leftrightarrow K_1 \]
\[ \text{log(Iteraions)} \]
\[ \text{log(Residual)} \]

- SOR, \( \omega = 1.90 \)
- Gauss–Seidel
