Discontinuous Galerkin Methods for Partial Differential Equations

Pratik Aghor (2011A4TS296P) Abhishek Kumar (2011A4PS303P)

25^{th} April, 2014

Contents

Acknowledgement

We would like to thank Dr. Sangita Yadav for her tremendous patience with our questions, her insistence on perfection and the extra effort that she made, to make sure that we understood concepts well enough.

We would also like to thank the Department of Mathematics, for giving us an opportunity to do a study project in discontinuous galerkin methods.

Lasty, we would like to thank Dr. Sai Jagan Mohan, for his encouragement and interest in our project, and for introducing us to the world of numerical methods for computation.

Pratik Aghor

Abhishek Kumar

Chapter 1

Introduction

1.1 Notations and Preliminaries

The notations used throughout the report are as follows:

- Domain is Ω which is a bounded, open set in $\mathbb{R}^2$ with Lipschitz continuous boundary $\partial\Omega$
- Γ_D on $\partial\Omega$ is the Dirichlet condition prescribed boundary
- Γ_N on $\partial\Omega$ is the Neumann condition prescribed boundary
- $\Gamma_N \cup \Gamma_D = \partial\Omega$ and $\Gamma_N \cap \Gamma_D = \phi$
- P_h is a partition of domain Ω, it numbers N_e partitions in this domain
- $\Omega = \bigcup_{K_i \in P_h} K_i$, $K_i \cap K_j = \phi$, $i \neq j$
- Set of edges E_h =set of$\gamma_l, l = 1, 2...N_\gamma$
- $E_h = E_{h,D} \cup E_{h,N} \cup E_{h,int}$

Chapter 2

DGM formulations for the Poisson Model Problem

2.1 Toy Problem

The toy problem or model problem used here is the Poisson equation. The equation and the boundary conditions are as follows.

2.1.1 Toy Problem: Poisson's Equation

$-\Delta u + cu = f$ in Ω

Boundary conditions are :

- $u = u_o$ on Γ_D
- $\vec{n} \cdot \nabla u = g$ on Γ_N

Here, $f \epsilon L^2$ represents the load scalar and c is a positive constant over the domain Ω

2.2 Derivation of Weak Formulation

- We now derive the weak form or the variational form of the toy problem.
- To do so, multiply PDE by test function v and integrate over Ω
- $\int_\Omega (\nabla \cdot \nabla u + cu)vdx = \int_\Omega fvdx$
- Unlike classical FEM approach, we will first decompose these integrals into their element contributions and then integrating by parts, we get the following:

$$\sum_{K \in P_h} \int_K (\nabla u \cdot \nabla v + cuv)dx - \sum_{K \in P_h} \int_{\partial K} ((\vec{n}) \cdot \nabla u)vds = \sum_{K \in P_h} \int_K fvdx$$

- Boundary integral is defined on each boundary element as follows:

$$\sum_{K \in P_h} \int_{\partial K} (\vec{n} \cdot \nabla u)vds = \int_{\Gamma_D} (\vec{n} \cdot \nabla u)vds + \int_{\Gamma_N} (\vec{n} \cdot \nabla u)vds +$$

$$\sum_{\gamma_{ij} \in E_{h,int}} \int_{\gamma_{ij}} (\vec{n} \cdot \nabla u)_i v_i + (\vec{n} \cdot \nabla u)_j v_j ds$$

Where v_i and v_j denote the restrictions of v on the elements K_i and K_j respectively. In the similar way, $(\vec{n} \cdot \nabla u)_i$ and $(\vec{n} \cdot \nabla u)_j$ represent the restrictions of the flux $(\vec{n} \cdot \nabla u)$ on the elements K_i and K_j respectively.

2.3 Simplification of the terms:

We can simplify the boundary terms and the simplification is done as follows:

- To simplify, we use the identity given below:
- $ac - bd = 1/2(a + b)(c - d) + 1/2(a - b)(c + d)$

Observing the similarity between the LHS of the above identity and

$$\vec{n} \cdot (\nabla u)_i v_i - \vec{n} \cdot (\nabla)_j v_j$$

We can write

- $$\vec{n}\cdot(\nabla u)_i v_i - \vec{n}\cdot(\nabla u)_j v_j = 1/2(\vec{n}\cdot(\nabla u)_i + \vec{n}\cdot(\nabla u)_j)(v_i - v_j) + 1/2(\vec{n}\cdot(\nabla u)_i - \vec{n}\cdot(\nabla u)_j)(v_i + v_j) = \langle \vec{n}\cdot\nabla u\rangle[v] + [vecn\cdot\nabla u]\langle v\rangle$$
- Where:Jump is
$$[v] = v_i - v_j$$
and
- Average is
$$\langle v\rangle = \frac{v_i + v_j}{2}$$

For an edge lying on Γ_D has

$$[v] = v$$

and

$$\langle v\rangle = v$$

Allowing us to combine interior and Dirichlet boundary conditions in one term :

$$\int_{\Gamma_{int}\cup\Gamma_D} \langle \vec{n} \cdot \nabla u\rangle[v] + [\vec{n} \cdot \nabla u]\langle v\rangle ds$$

Using this in the weak form is reduced to

$$\sum_{K\in P_h} \int_K (\nabla u \cdot \nabla v + cuv)dx - \int_{\Gamma_{int}\cup\Gamma_D} \langle \vec{n} \cdot \nabla u\rangle ds = \sum_{K\in P_h} \int_K fvdx + \int_{\Gamma_N} gvds$$

- Introducing a bilinear form
$$B(u, v) = \sum_{K\in P_h} \int_K (\nabla u \cdot \nabla v + cuv)dx$$
- We also define a bilinear form for the boundaries Γ_D and Γ_{int} as
$$J(u, v) = \int_{\Gamma_D\cup\Gamma_{int}} \langle \vec{n} \cdot \nabla u\rangle[v]ds$$
- Defining a linear form $F(v) = \sum_{K\in P_h} \int_K fvdx + \int_{\Gamma_N} gvds$

Therefore, a general discontinuous weak formulation of the Poisson Equation hence reads as:

$$B(u, v) - J(u, v) = F(v), \forall v \in H^2(P_h)$$

2.4 Weak Formulations and FEM Discretizations

For $u \in H^1(\Omega) \cap H^2(P_h)$, as the solution we seek is continuous over the domain, jump of u, i.e. [u] vanishes on every internal boundary γ_{ij},

$$\int_{\gamma_{ij}} v[u]ds = 0, \forall v \in L^2(\gamma_{ij})$$

Follows that :

$$\int_{\Gamma_{int}} \langle \vec{n} \cdot \nabla v \rangle [u]ds = 0, \forall v \in H^2(P_h)$$

Consequently, The Diritchlet condition can also be applied.

$$\int_{\Gamma_D} (\vec{n} \cdot \nabla v)uds = \int_{\Gamma_D} (\vec{n} \cdot \nabla v)u_0 ds, \forall v \in H^2(P_h)$$

Therefore, Introducing a new linear form

- $$J_0(v) = \int_{\Gamma_D} (\vec{n} \cdot \nabla v)u_0 ds, \forall v \in H^2(P_h)$$
- A simple observation leads to: $u = u_0$ on Γ_D,

$$J(u, v) = J_0(v), \forall v \in H^2(P_h)$$

2.5 Global Element Method (GEM):

Introducing a new bilinear form:$B_-(u, v) = B(u, v) - J(u, v) - J(v, u)$
And a new linear form: $F_-(v) = F(v) - J_0(v)$
Therefore, GEM defines the problem in the following manner:
Find u such that :

$$B_-(u, v) = F_-(v), \forall v \in V^{hp}$$

Advantage of GEM:

- It defines the symmetric problem. Notice that though $B(.,.)$ is symmetric $J(.,.)$is not. Therefore, LHS of the original weak formulation is not symmetric. But the LHS of GEM formulation is made symmetric by subtracting $J(v, u)$ term. Now if we flip u and v there is no effect whatsoever.
  Disadvantages of GEM:
- Bilinear form is not guaranteed to be semi-positive definite, which means, its eigenvalues may assume negative values. This may cause the formulation to go unconditionally unstable (e.g. while dealing with time-dependent problems).

The corresponding FEM Discretization of GEM is: Find finding $u_h \in V^{hp}$ such that :

$$B_-(u_h, v) = F_-(v), \forall v \in H^2(P_h)$$

2.6 Symmetric Interior Penalty Galerkin Method (SIPG):

We expect our solution to be a continuous function over the domain. But we are implementing discontinuous method. To resolve the above paradox and to ensure the continuity of the solution, we introduce following Penalty terms: (They were first added by Arnold and Wheeler):

- Let σ be penalty parameter depending on length of edges γ_{ij} and γ and the polynomial degree used in elements i.e $\sigma = \sigma(h, p)$

$$J^{\sigma}(u, v) = \int_{\Gamma_i nt \cup \Gamma_D} \sigma[u][v] ds$$

and

$$J_0^{\sigma}(v) = \int_{\Gamma_D} \sigma u_0 v ds$$

- $B_-(u, v)^{\sigma} = B(u, v) - J(u, v) - J(v, u) + J^{\sigma}(u, v)$
- $F_-^{\sigma}(v) = F(v) - J_0(v) + J_0(v)^{\sigma}$

The SIPG defines the problem as:
Find u such that :

$$B_-(u, v)^{\sigma} = F_-^{\sigma}(v), \forall v \epsilon H^2(P_h)$$

The corresponding FEM discretization of SIPG reads: find $u_h \in V^{hp}$ such that :

$$B_-(u_h, v)^{\sigma} = F_-^{\sigma}(v), \forall v \in V^{hp}$$

2.7 Non-Symmetric Interior Penalty Galerkin Method (NIPG)

We begin NIPG by introducing bilinear forms as follows:

- $B_-(u, v)^{\sigma} = B(u, v) - J(u, v) + J(v, u) + J^{\sigma}(u, v)$
- $F_+^{\sigma}(v) = F(v) + J_0(v) + J_0(v)^{\sigma}$

The NIPG defines the problem as: Find u such that:

$$B_+(u, v)^{\sigma} = F_+^{\sigma}(v), \forall v \epsilon H^2(P_h)$$

The corresponding FEM analogue is:

$$B_+(u_h, v)^{\sigma} = F_+^{\sigma}(v), \forall v \in V^{hp}$$

Chapter 3

Numerical Implementation

3.1 MATLAB Implementation

In this chapter, we have documented the efforts towards the numerical implementation of DGM on the toy problem. We first discretize the unit square into triangles as per the convention in the figure given below:

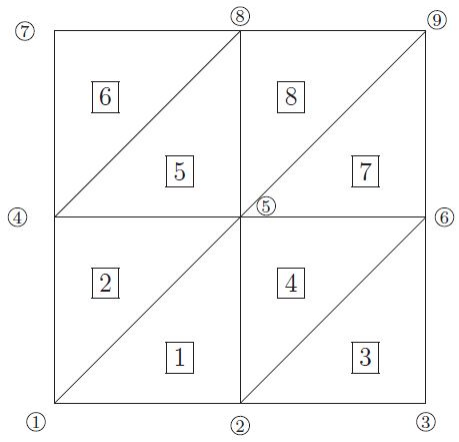


Figure 3.1: Triangulation Convention

3.2 Matlab file to generate co-ordinates of a mesh

```
function coordinate= coord(n)
n=input(' no of node points on 0 to 1 line=');
h=1/(n-1);
coord=sparse(n^2,2);
for i=1:n
    for j=1:n
        coord(n*(i-1)+j,:)=[(j-1)*h,(i-1)*h];
    end
end
coord+0
```

3.3 Matlab file to generate edge to node function

```
function node=edge2node_func2(n)
n=input('enter the no of nodes on the 0 to 1 line');
h=1/(n-1);
edge2node=sparse(n^2+(2*(n-1)^2)-1,2);
for j=1:n-1
```

```
    for i=1:n-1
        edge2node((j-1)*(n+2*(n-1))+i,:)=[(j-1)*n+i,(j-1)*n+i+1];
            if rem(i,2)==0 && rem(j,2)~=0
            edge2node (n+((j-1)*(n+2*(n-1))+i-1)*2-(j-1)*(n+2*(n-1)),:)=[(j-1)*n+i,n+(j-1)*n+i];
            edge2node(n+((j-1)*(n+2*(n-1))+i-1)*2+1-(j-1)*(n+2*(n-1)),:)=[(j-1)*n+i,n+(j-1)*n+i+1];

            elseif rem(i,2)~=0 && rem(j,2)~=0
            edge2node((n+2)+((j-1)*(n+2*(n-1))+i-2)*2-(j-1)*(n+2*(n-1)),:)=[(j-1)*n+i,n+(j-1)*n+i];
            edge2node((n+2)+((j-1)*(n+2*(n-1))+i-2)*2+1-(j-1)*(n+2*(n-1)),:)=[(j-1)*n+i,n+(j-1)*n+i+1];

            elseif rem(i,2)==0 && rem(j,2)==0
            edge2node (n+((j-1)*(n+2*(n-1))+i-1)*2-(j-1)*(n+2*(n-1)),:)=[(j-1)*n+i,n+(j-1)*n+i];
            edge2node(n+((j-1)*(n+2*(n-1))+i-1)*2+1-(j-1)*(n+2*(n-1)),:)=[(j-1)*n+i,n+(j-1)*n+i+1];

            elseif rem(i,2)~=0 && rem(j,2)==0
            edge2node((n+2)+((j-1)*(n+2*(n-1))+i-2)*2-(j-1)*(n+2*(n-1)),:)=[(j-1)*n+i,n+(j-1)*n+i];
            edge2node((n+2)+((j-1)*(n+2*(n-1))+i-2)*2+1-(j-1)*(n+2*(n-1)),:)=[(j-1)*n+i,n+(j-1)*n+i+1];

            end
    end
        edge2node(j*(n+2*(n-1)),:)=[j*n,(j+1)*n];
end

for i=1:n-1
    edge2node((n-1)*(n+2*(n-1))+i,:)=[(n-1)*n+i,(n-1)*n+i+1];
end
    edge2node+0
```

3.4 Matlab file to generate element to node function

```
function element2node_func(n)
n=input('enter the no of nodes on the 0 to 1 line');
h=1/(n-1);
element2node=sparse(2*((n-1)^2),3);
%
% x^2 + e^{\pi i}
for j=1:n-1
    for i=1:n-1
        element2node(2*(j-1)*(n-1)+(2*i-1),:)=[j*n+i,(j-1)*n+i,j*n+i+1];
        element2node(2*(j-1)*(n-1)+(2*i),:)=[(j-1)*n+i+1,j*n+i+1,(j-1)*n+i];
    end
end
element2node+0
```

```
>> coord_new
 no of node points on 0 to 1 line= 3

ans =

         0         0
    0.5000         0
    1.0000         0
         0    0.5000
    0.5000    0.5000
    1.0000    0.5000
         0    1.0000
    0.5000    1.0000
    1.0000    1.0000
```

Figure 3.2: Output of *coord_new.m*

```
>> edge2node_func2
enter the no of nodes on the 0 to 1 line 3

ans =

     1     2
     2     3
     1     4
     1     5
     2     5
     2     6
     3     6
     4     5
     5     6
     4     7
     4     8
     5     8
     5     9
     6     9
     7     8
     8     9
```

Figure 3.3: Output of $edge2node\_func2.m$

```
>> element2node_func
enter the no of nodes on the 0 to 1 line 3

ans =

     4     1     5
     2     5     1
     5     2     6
     3     6     2
     7     4     8
     5     8     4
     8     5     9
     6     9     5
```

Figure 3.4: Output of $element2node\_func.m$

Chapter 4

Implementation in freeFEM++

4.0.1 Code for SIPG

```
macro dn(u) (N.x*dx(u)+N.y*dy(u) ) //  def the normal derivative
macro gradx(u) (dx(u))//
macro grady(u) (dy(u))//

int i=1;
for (i=1;i<6;i++)
{
mesh Th = square(2^i,2^i); // unite square
plot (Th);
fespace Vh(Th,P2dc);     // Discontinous P2 finite element
real pena=4*(2^i); // a paramater to add penalisation
varf Ans(u,v)=
int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
+intalledges(Th)(
//  loop on all  edge of all triangle
// the edge are see nTonEdge times so we / nTonEdge
// remark: nTonEdge =1 on border edge and =2 on internal
// we are in a triange th normal is the exterior normal
// def: jump = external - internal value; on border exter value =0
//      mean = (external + internal value)/2, on border just internal value
(jump(v)*mean(dn(u)) +  jump(u)*mean(dn(v))
+ pena*jump(u)*jump(v) ) / nTonEdge
);
func f=2*(x*(1-x)+y*(1-y));
func g=0;
Vh u,v;
problem A(u,v,solver=UMFPACK) = Ans
- int2d(Th)(f*v)
- int1d(Th)(g*dn(v)  + pena*g*v)
;
A; // solve  DG
//////////////////
//error estimate
real gg= int2d(Th)(dx(u)*dx(u)+dy(u)*dy(u))
+intalledges(Th)(
( -jump(u)*mean(dn(u)) -  jump(u)*mean(dn(u))
```

```
+ pena*jump(u)*jump(u) ) / nTonEdge
);
real mm=  int2d(Th)(f*u)
+ int1d(Th)(g*dn(u)  + pena*g*u);
cout << "residual"<< gg-mm << endl;
Vh uu = (x*(1-x)*y*(1-y));
Vh gradxu=gradx(u);
Vh gradyu=grady(u);
Vh gradxuu=gradx(uu);
Vh gradyuu=grady(uu);
plot(u,cmm="SIPG Solution residual"+(gg-mm),wait=1,value=1,fill=1,ps="SIPGu"+i+".jpg");
plot(gradxu,cmm="SIPG gradx(u)",wait=1,dim=2,fill=1,ps="SIPGGux"+i+".jpg");
plot(gradyu,cmm="SIPG grady(u)",wait=1,dim=2,fill=1,ps="SIPGGuy"+i+".jpg");
plot(uu,cmm="analytical",fill=1,ps="analyu"+".jpg");
plot(gradxuu,cmm="SIPG gradx(uu)",wait=1,dim=2,fill=1,ps="SIPGGuux"+i+".jpg");
plot(gradyuu,cmm="SIPG grady(uu)",wait=1,dim=2,fill=1,ps="SIPGGuuy"+i+".jpg");


Vh error=u-(x*(1-x)*y*(1-y));
Vh gradxer=gradx(u)-gradx(uu);
Vh gradyer=grady(u)-grady(uu);
real er= error[].max;
real ergx=gradxer[].max;
real ergy=gradyer[].max;
//ofstream file("sol_error.txt",append);
//file << error[] <<endl;
cout<<"error="<<er<<endl;
plot(error,cmm="Error plot"+"max er"+er,wait=1,dim=2,fill=1,ps="SIPGEr"+i+".jpg");
plot(gradxer,cmm="Error plot gradx(u)"+"max er"+ergx,wait=1,dim=2,fill=1,ps="SIPGErgx"+i+".jpg");
plot(gradyer,cmm="Error plot grady(u)"+"max er"+ergy,wait=1,dim=2,fill=1,ps="SIPGErgy"+i+".jpg");/////////
}
```

4.0.2 Code for NIPG

```
macro dn(u) (N.x*dx(u)+N.y*dy(u) ) //  def the normal derivative
macro gradx(u) (dx(u))//
macro grady(u) (dy(u))//

int i=1;
for (i=1;i<6;i++)
{
mesh Th = square(2^i,2^i); // unite square
plot (Th);
fespace Vh(Th,P2dc);     // Discontinous P2 finite element
real pena=4*(2^i); // a paramater to add penalisation
varf Ans(u,v)=
int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
+intalledges(Th)(
//  loop on all  edge of all triangle
// the edge are see nTonEdge times so we / nTonEdge
```

```
// remark: nTonEdge =1 on border edge and =2 on internal
// we are in a triange th normal is the exterior normal
// def: jump = external - internal value; on border exter value =0
//      mean = (external + internal value)/2, on border just internal value
( jump(v)*mean(dn(u)) -  jump(u)*mean(dn(v))
+ pena*jump(u)*jump(v) ) / nTonEdge
);
func f=2*(x*(1-x)+y*(1-y));
func g=0;
Vh u,v;
problem A(u,v,solver=UMFPACK) = Ans
- int2d(Th)(f*v)
- int1d(Th)(g*dn(v)  + pena*g*v)
;
A; // solve  DG
//////////////////
//error estimate
real gg= int2d(Th)(dx(u)*dx(u)+dy(u)*dy(u))
+intalledges(Th)(
( -jump(u)*mean(dn(u)) + jump(u)*mean(dn(u))
+ pena*jump(u)*jump(u) ) / nTonEdge
);
real mm=  int2d(Th)(f*u)
+ int1d(Th)(g*dn(u)  + pena*g*u);
cout << "residual"<< gg-mm << endl;
Vh uu = (x*(1-x)*y*(1-y));
Vh gradxu=gradx(u);
Vh gradyu=grady(u);
Vh gradxuu=gradx(uu);
Vh gradyuu=grady(uu);
plot(u,cmm="NIPG Solution residual"+(gg-mm),wait=1,value=1,fill=1,ps="NIPGu"+i+".jpg");
plot(gradxu,cmm="NIPG gradx(u)",wait=1,dim=2,fill=1,ps="NIPGGux"+i+".jpg");
plot(gradyu,cmm="NIPG grady(u)",wait=1,dim=2,fill=1,ps="NIPGGuy"+i+".jpg");
plot(uu,cmm="analytical",fill=1,ps="analyu"+".jpg");
plot(gradxuu,cmm="NIPG gradx(uu)",wait=1,dim=2,fill=1,ps="NIPGGuux"+i+".jpg");
plot(gradyuu,cmm="NIPG grady(uu)",wait=1,dim=2,fill=1,ps="NIPGGuuy"+i+".jpg");


Vh error=u-(x*(1-x)*y*(1-y));
Vh gradxer=gradx(u)-gradx(uu);
Vh gradyer=grady(u)-grady(uu);
real er= error[].max;
real ergx=gradxer[].max;
real ergy=gradyer[].max;
//ofstream file("sol_error.txt",append);
//file << error[] <<endl;
cout<<"error="<<er<<endl;
plot(error,cmm="Error plot"+"max er"+er,wait=1,dim=2,fill=1,ps="NIPGEr"+i+".jpg");
plot(gradxer,cmm="Error plot gradx(u)"+"max er"+ergx,wait=1,dim=2,fill=1,ps="NIPGErgx"+i+".jpg");
plot(gradyer,cmm="Error plot grady(u)"+"max er"+ergy,wait=1,dim=2,fill=1,ps="NIPGErgy"+i+".jpg");/////////
}
```

Chapter 5

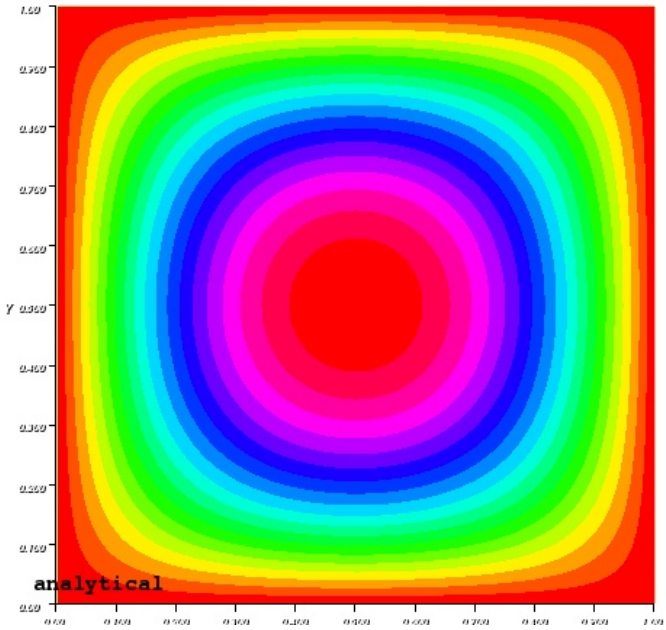
Results for Laplace Equation

Note : i in the following report refers to the exponent of the number of nodes on an edge (2^i, as was written on our code) Analytically the solution for the equation that we have solved is :

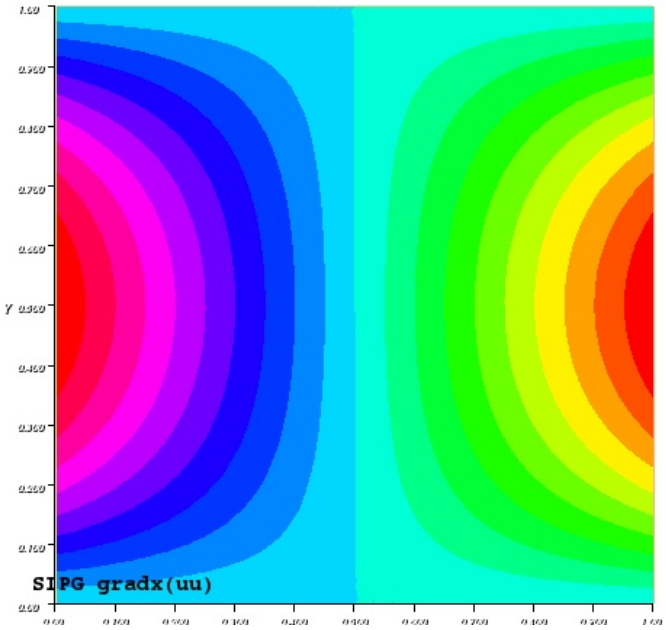
$$u = x(1-x)y(1-y)$$

Also the analytical solution for the ∇u is :

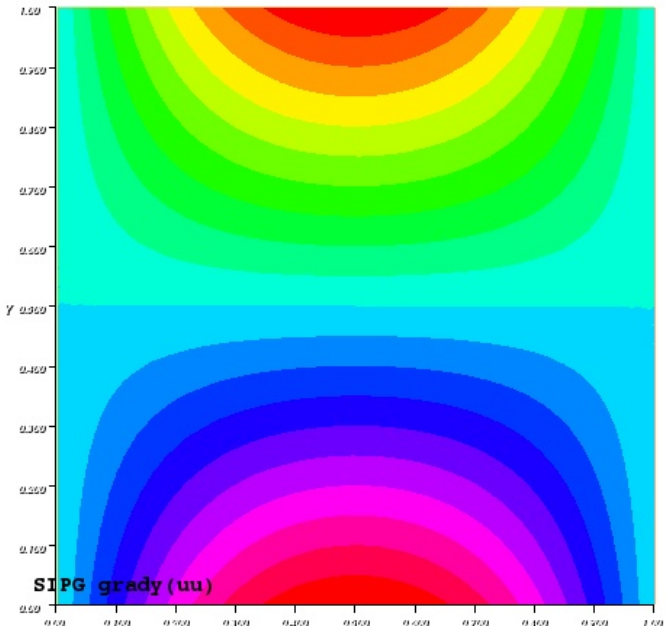
$$\nabla u = y(1-y)(1-2x)\vec{i} + x(1-x)(1-2y)\vec{j}$$



(a) The analytical solution of u

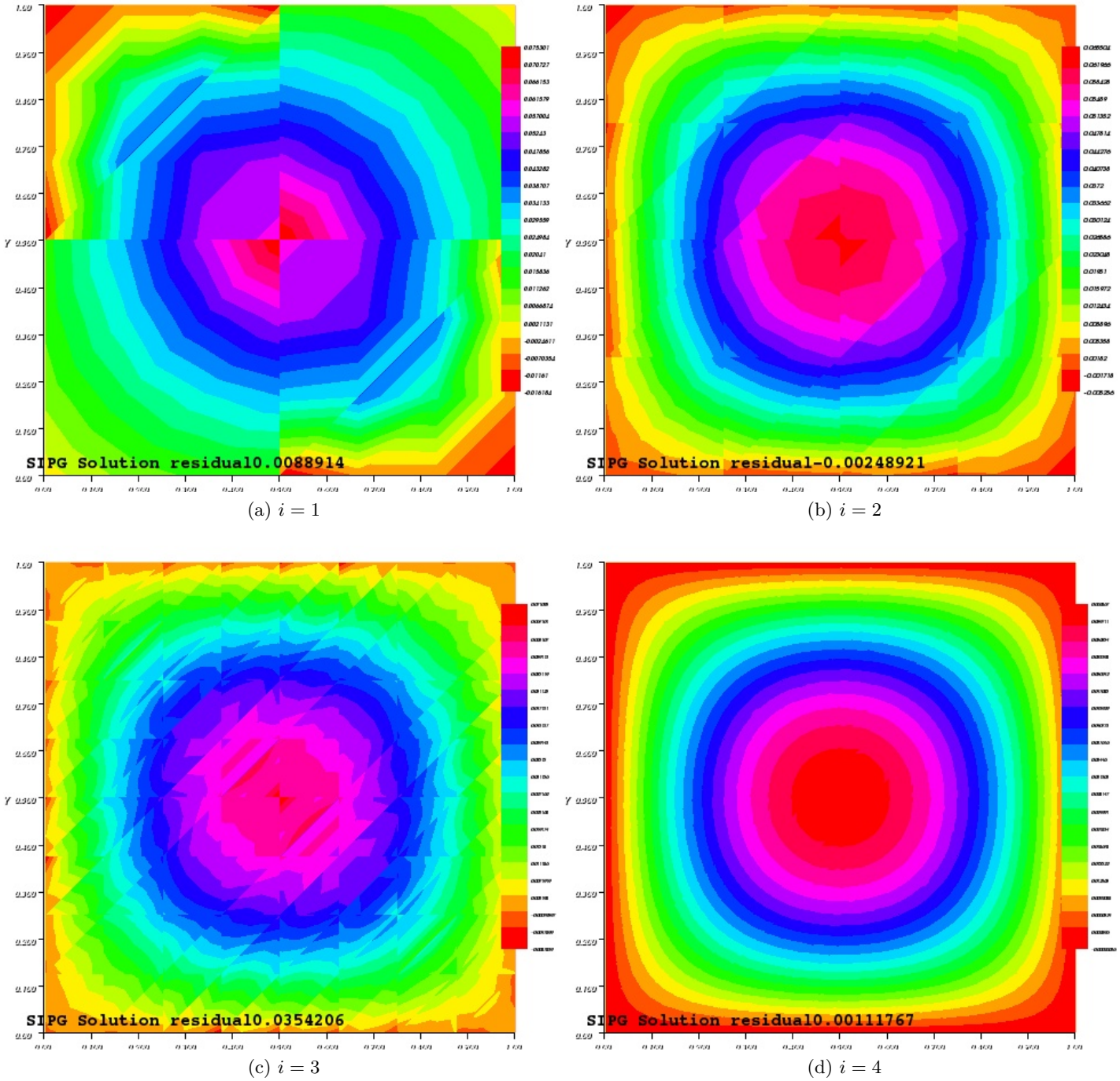


(b) The analytical solution of component x of ∇u



(c) The analytical solution of component y of ∇u

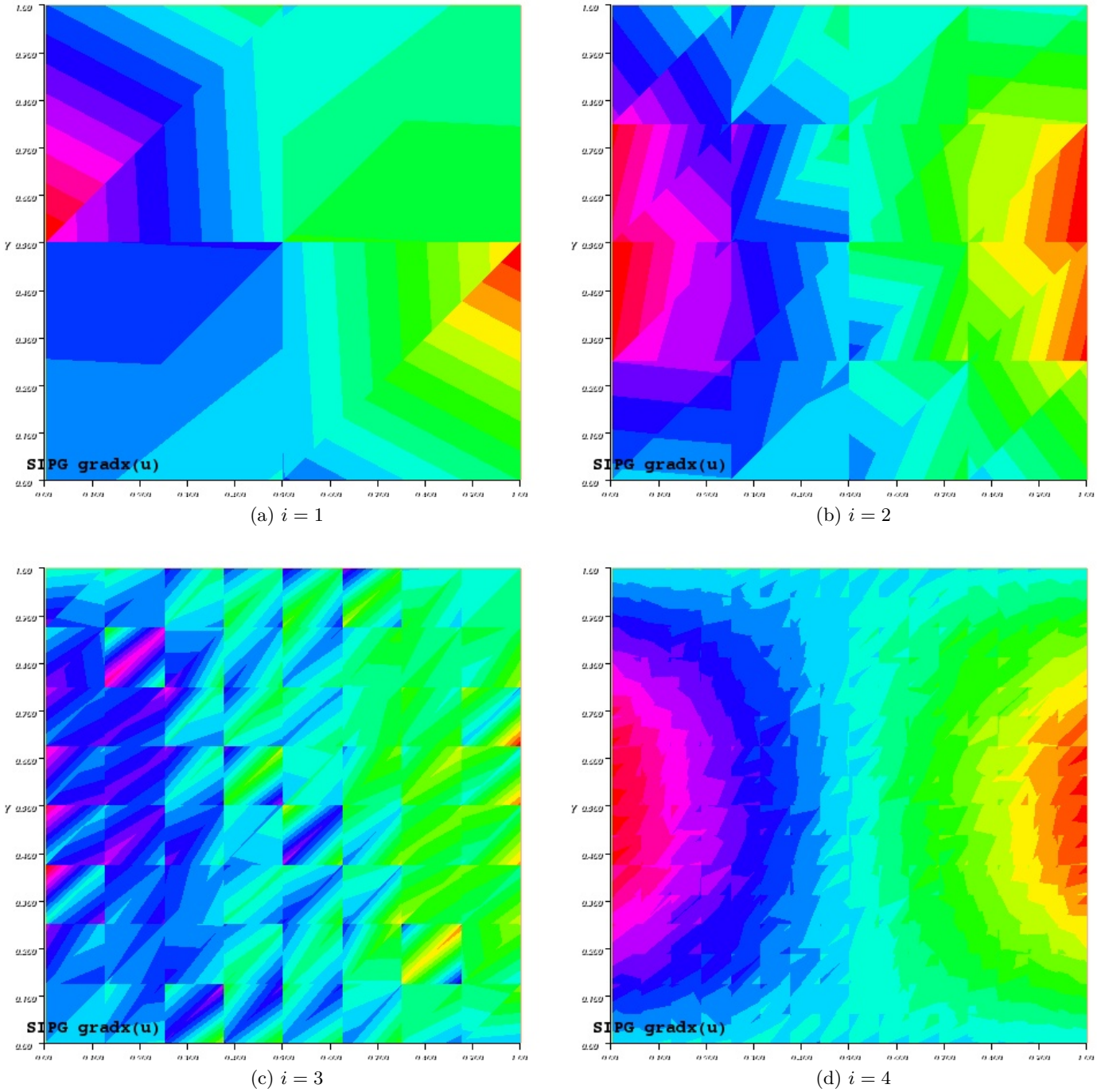
Figure 5.1: Analytical Solutions of our functions



(a) $i = 1$ (b) $i = 2$

(c) $i = 3$ (d) $i = 4$

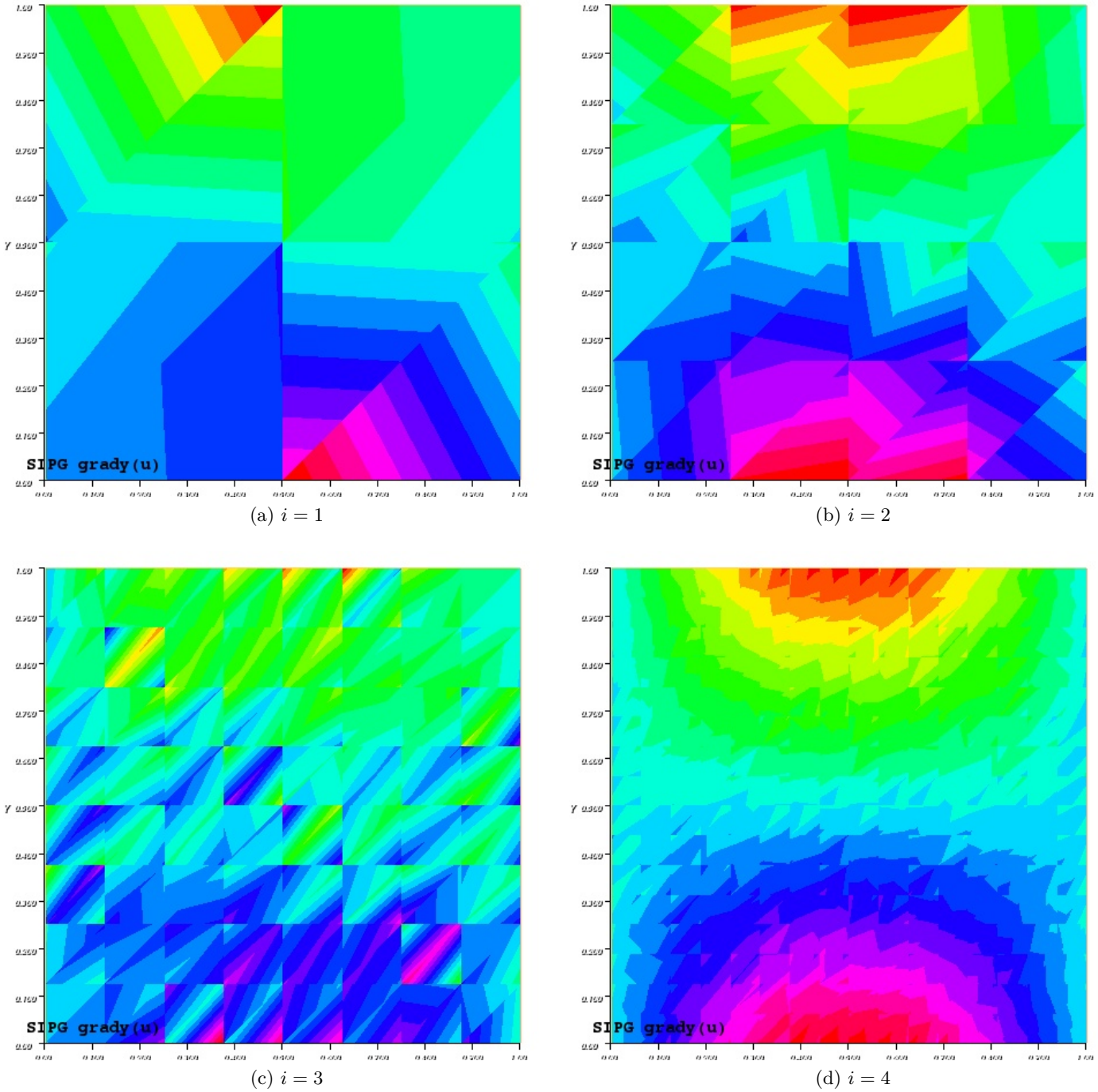
Figure 5.2: Plots of function u by SIPG Method



(a) $i = 1$ (b) $i = 2$

(c) $i = 3$ (d) $i = 4$

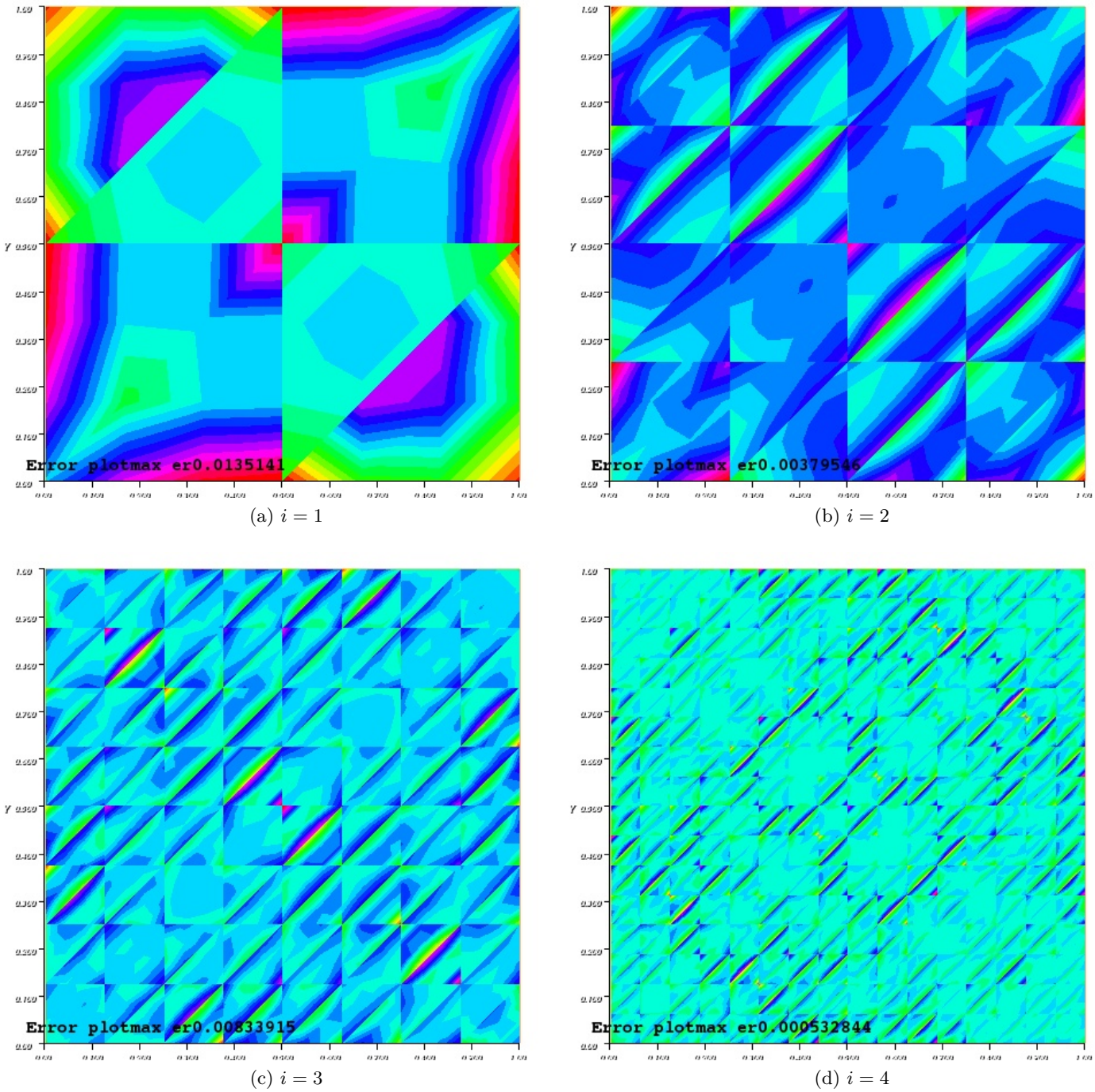
Figure 5.3: Plots of function component x of ∇u by SIPG Method



(a) $i = 1$ (b) $i = 2$

(c) $i = 3$ (d) $i = 4$

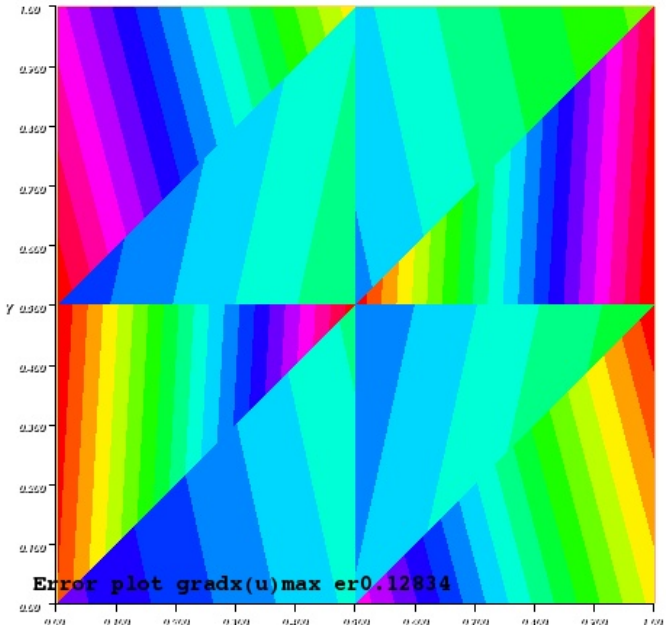
Figure 5.4: Plots of function component y of ∇u by SIPG Method



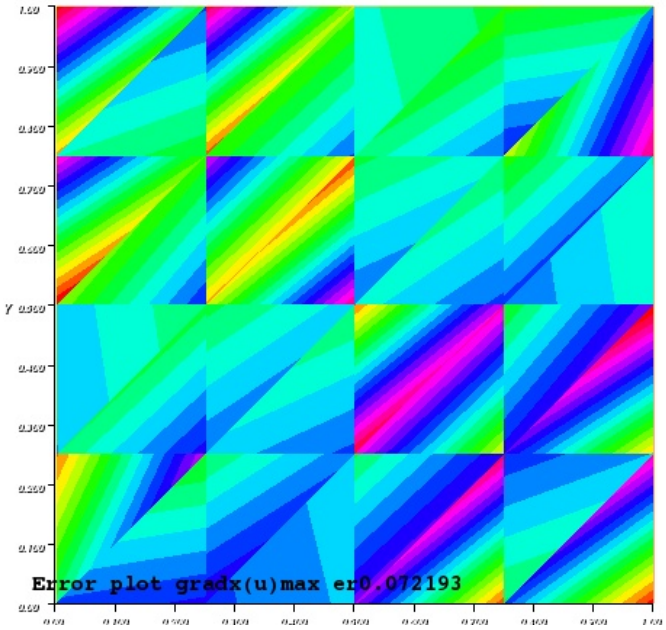
(a) $i = 1$ (b) $i = 2$

(c) $i = 3$ (d) $i = 4$

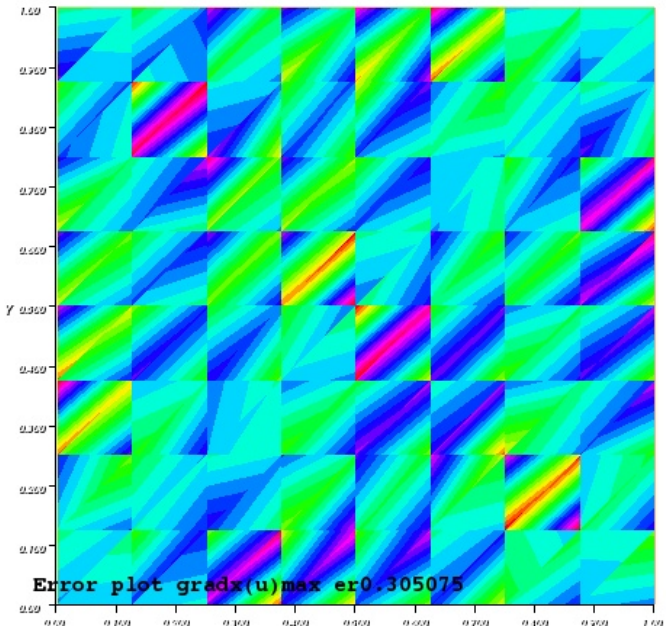
Figure 5.5: Plots of Error for u by SIPG Method



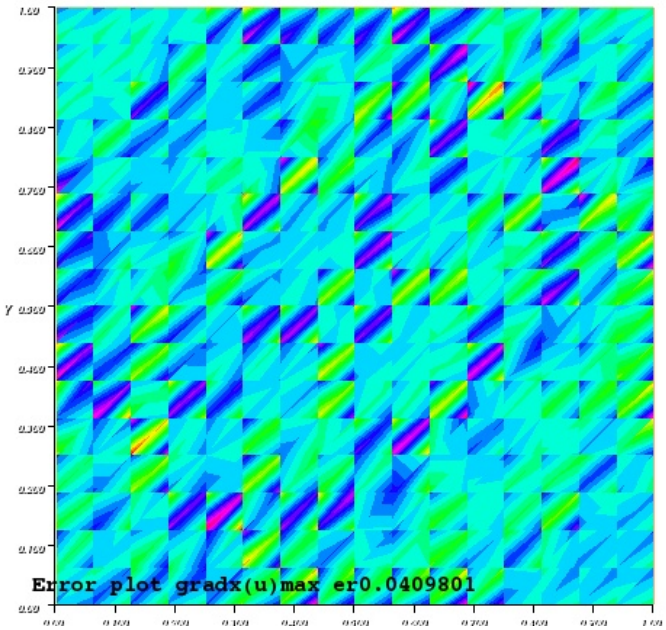
(a) $i = 1$



(b) $i = 2$

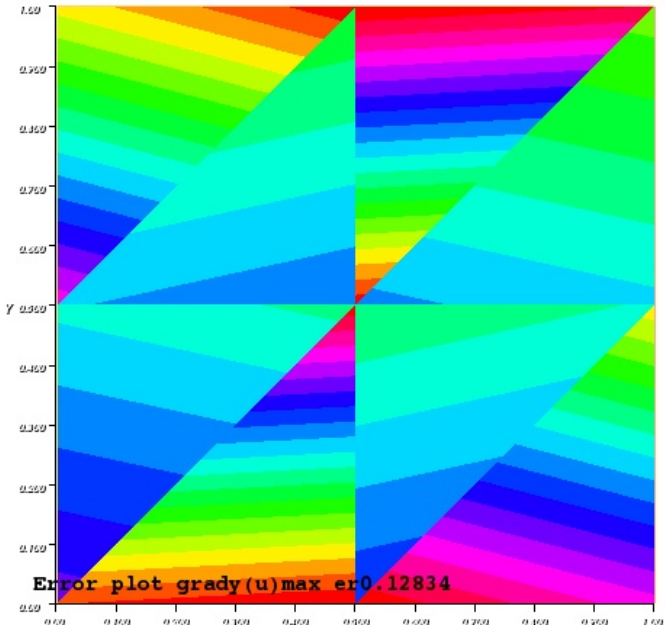


(c) $i = 3$

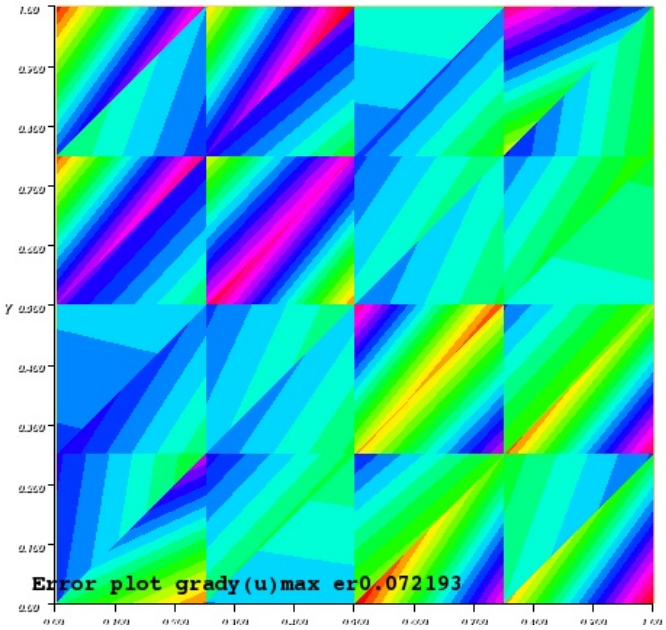


(d) $i = 4$

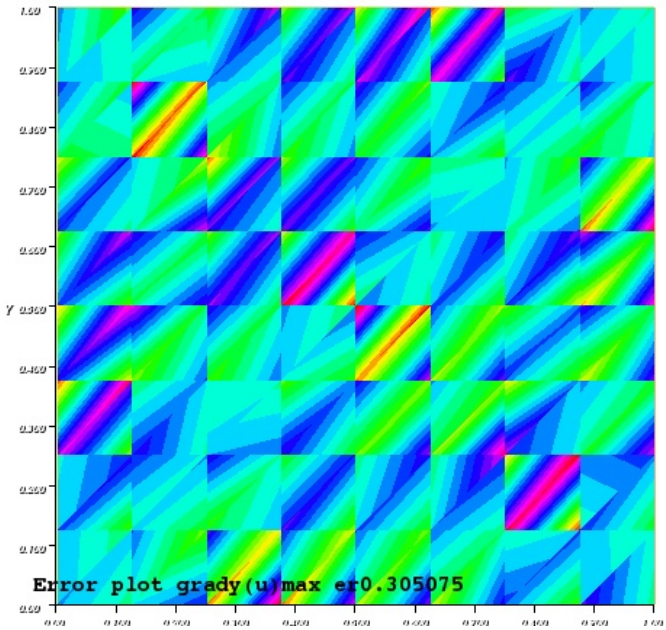
Figure 5.6: Plots of Error for component x of $\nabla u$by SIPG Method



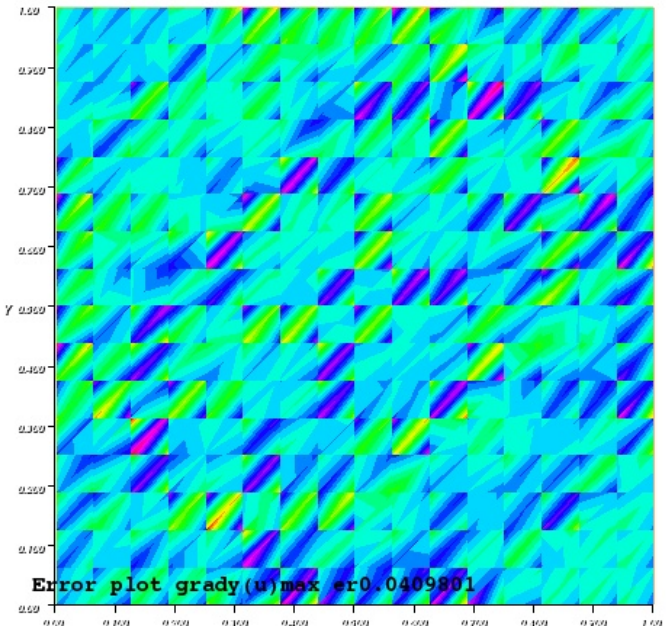
(a) $i = 1$



(b) $i = 2$

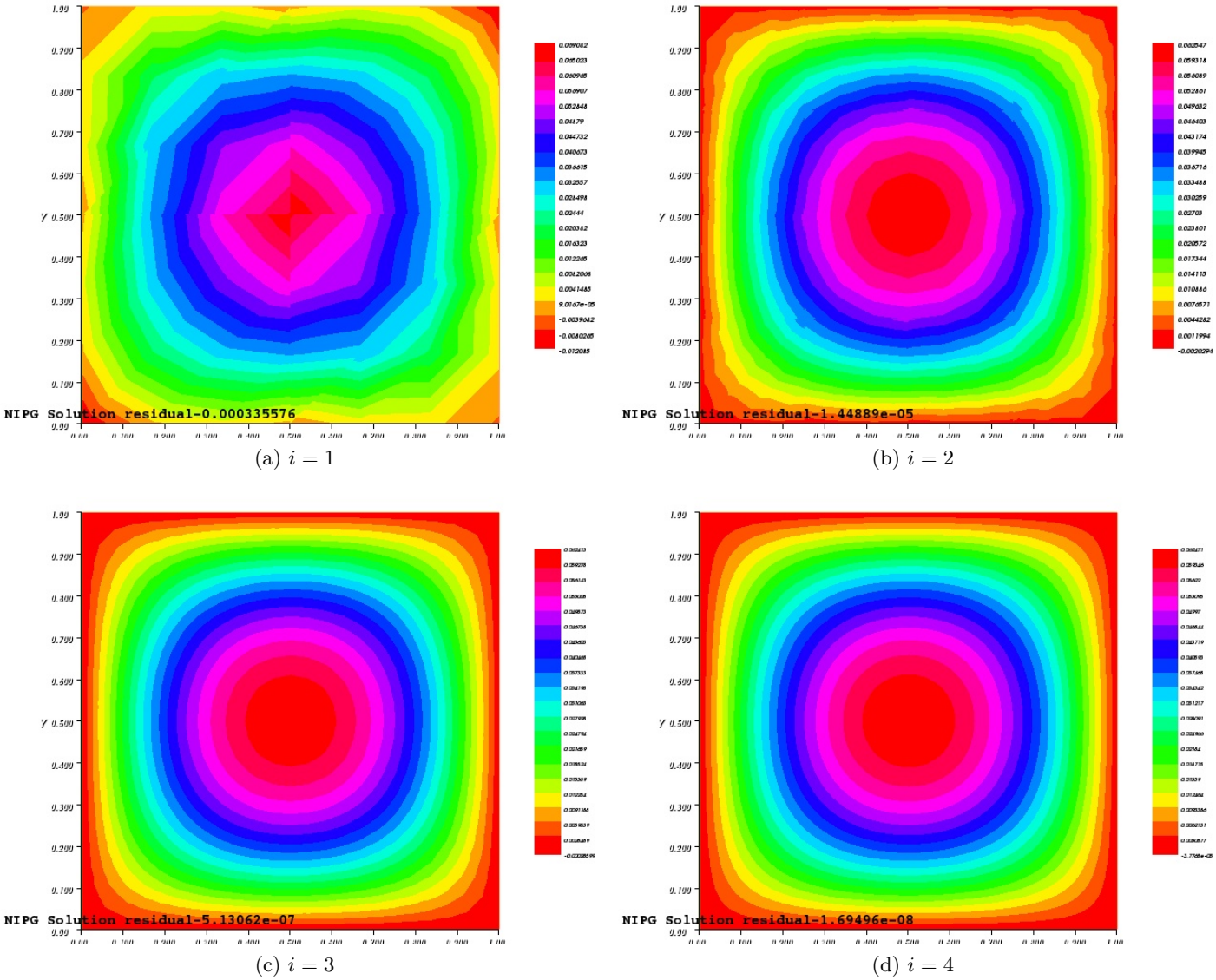


(c) $i = 3$



(d) $i = 4$

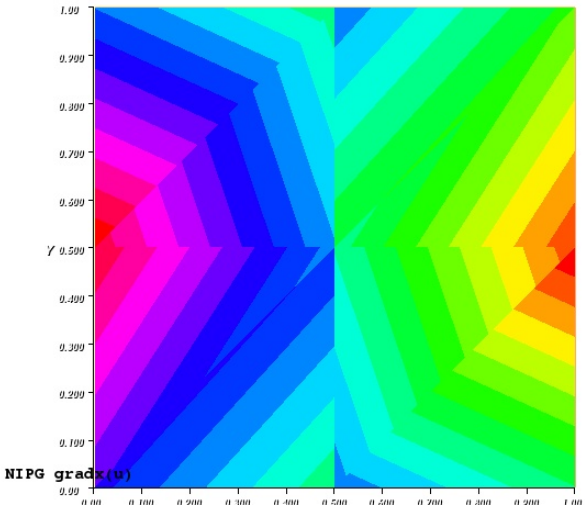
Figure 5.7: Plots of Error of component y of ∇u by SIPG Method



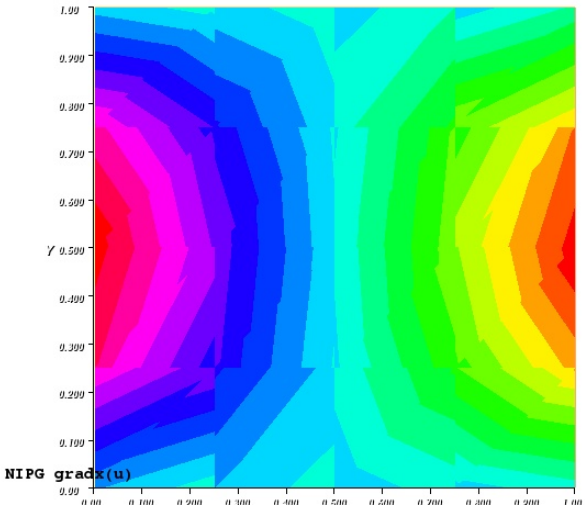
(a) $i = 1$ (b) $i = 2$

(c) $i = 3$ (d) $i = 4$

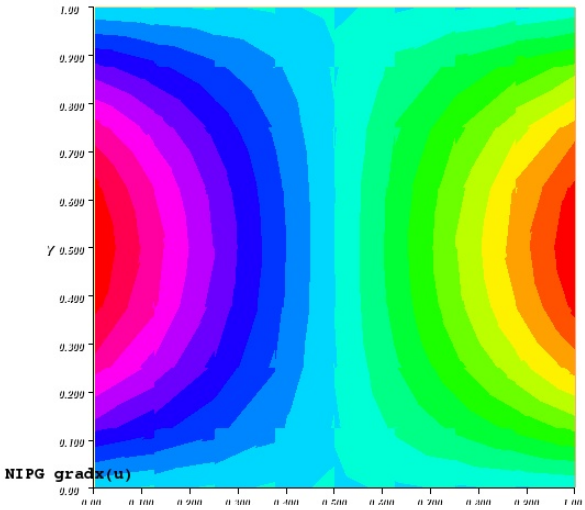
Figure 5.8: Plots of function u by NIPG Method



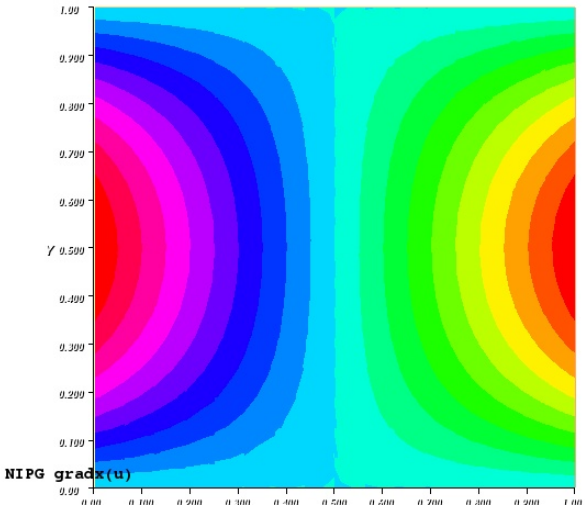
(a) $i = 1$



(b) $i = 2$

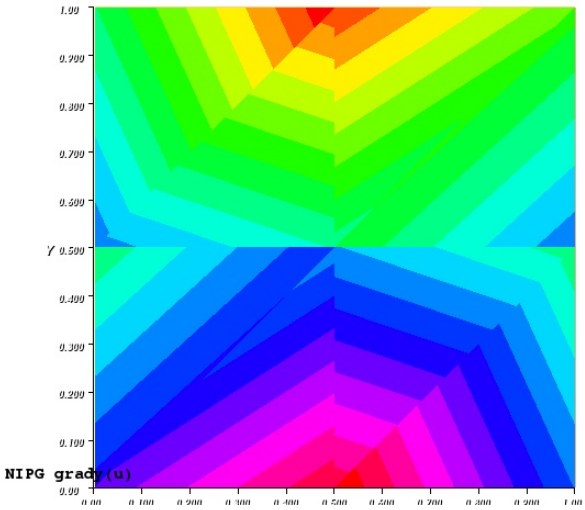


(c) $i = 3$

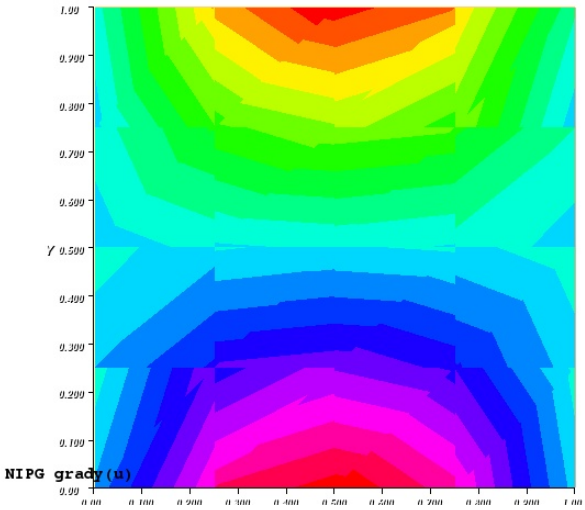


(d) $i = 4$

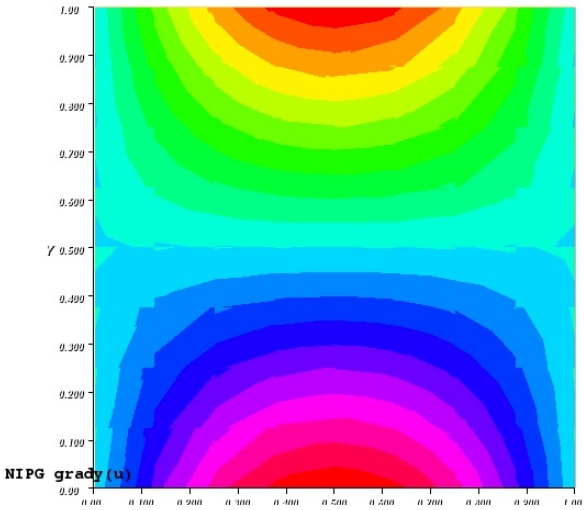
Figure 5.9: Plots of function component of x ∇u by NIPG Method



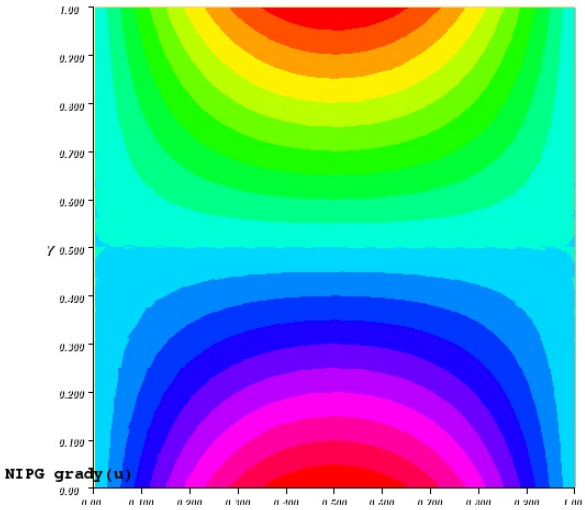
(a) $i = 1$



(b) $i = 2$

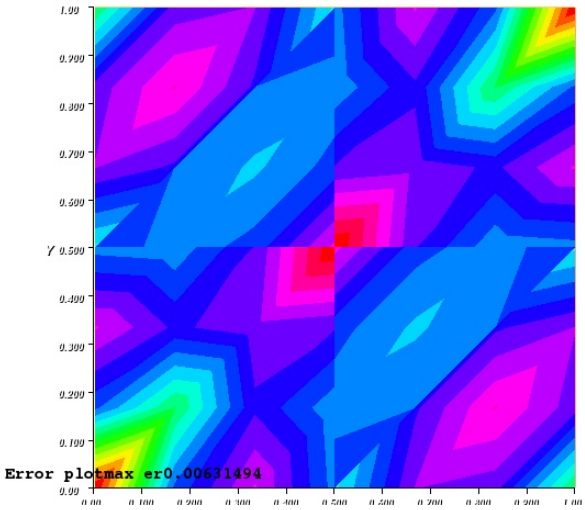


(c) $i = 3$

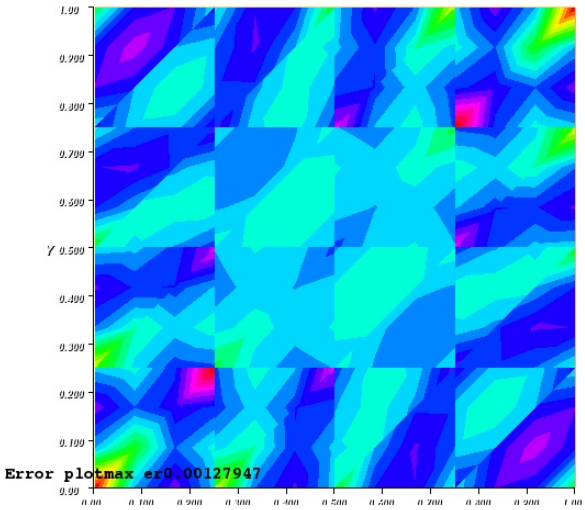


(d) $i = 4$

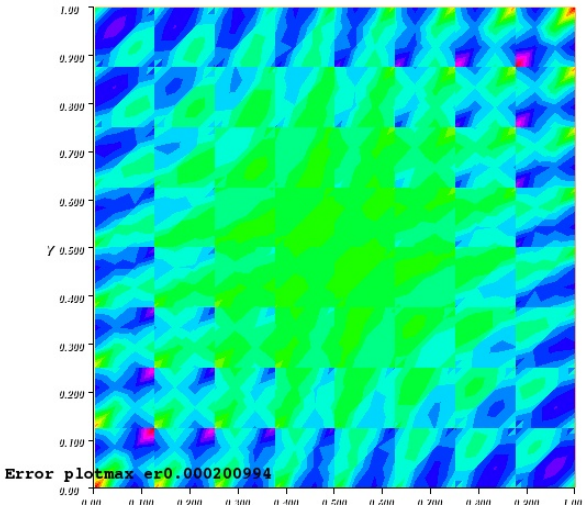
Figure 5.10: Plots of function component of y ∇u by NIPG Method



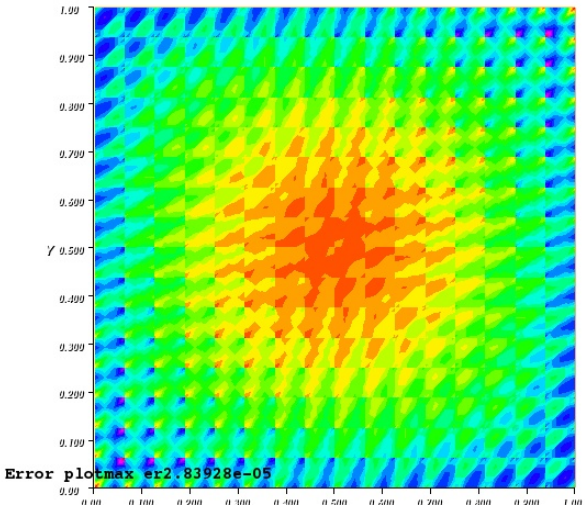
(a) $i = 1$



(b) $i = 2$

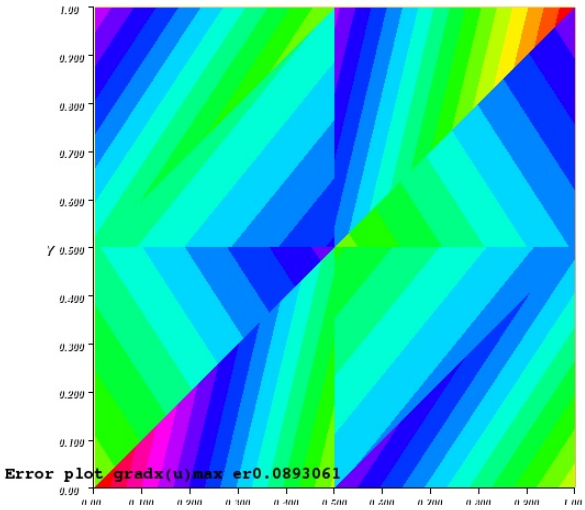


(c) $i = 3$

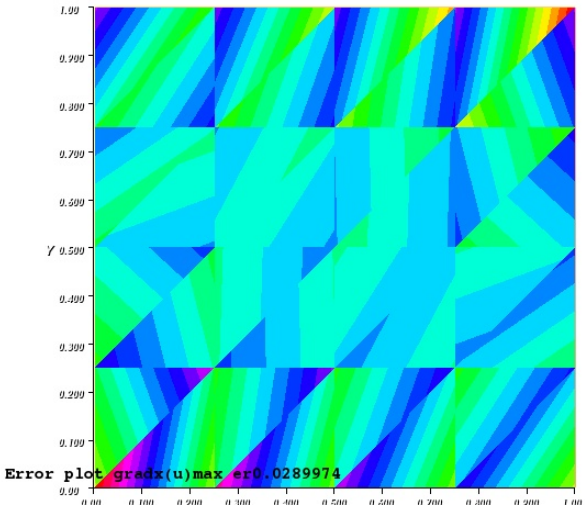


(d) $i = 4$

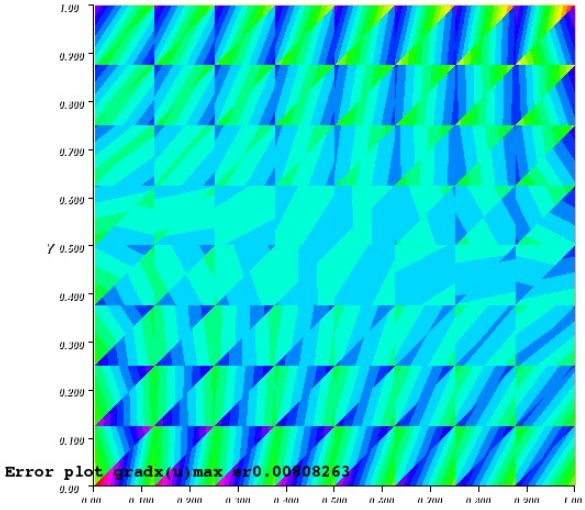
Figure 5.11: Plots of Error of u by NIPG Method



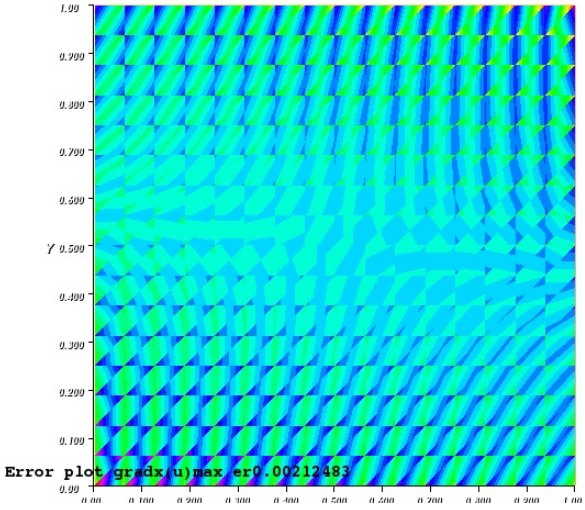
(a) $i = 1$



(b) $i = 2$

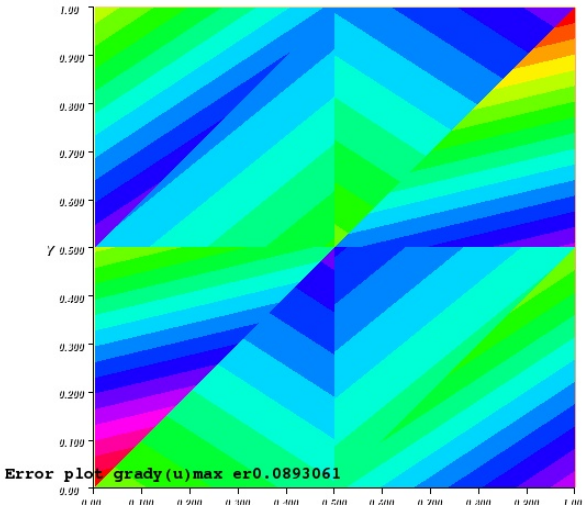


(c) $i = 3$

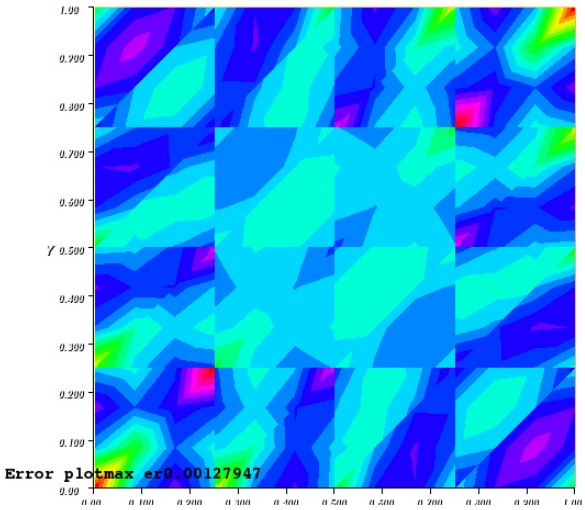


(d) $i = 4$

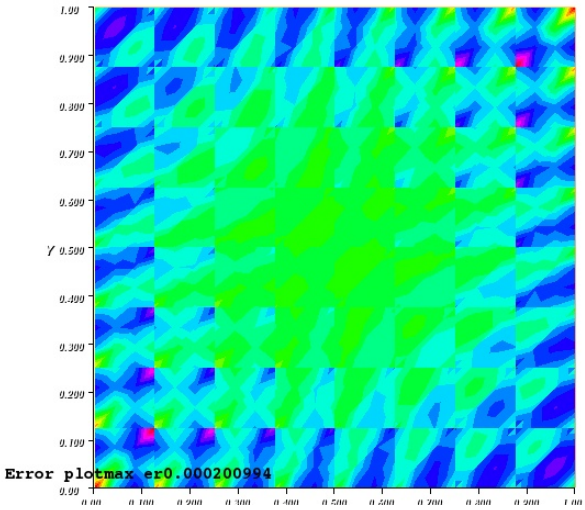
Figure 5.12: Plots of Error of component x of ∇u by NIPG Method



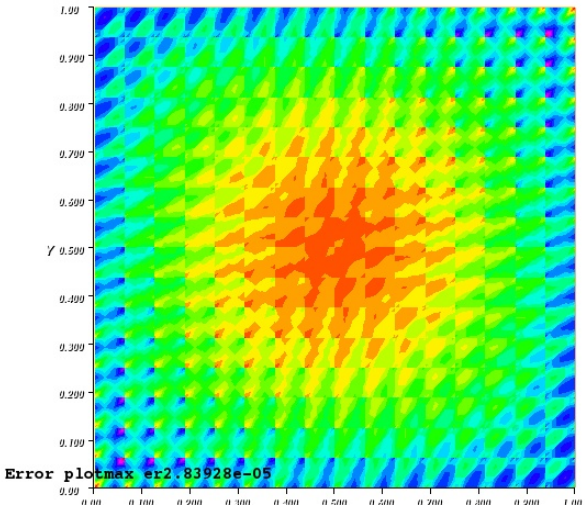
(a) $i = 1$



(b) $i = 2$



(c) $i = 3$

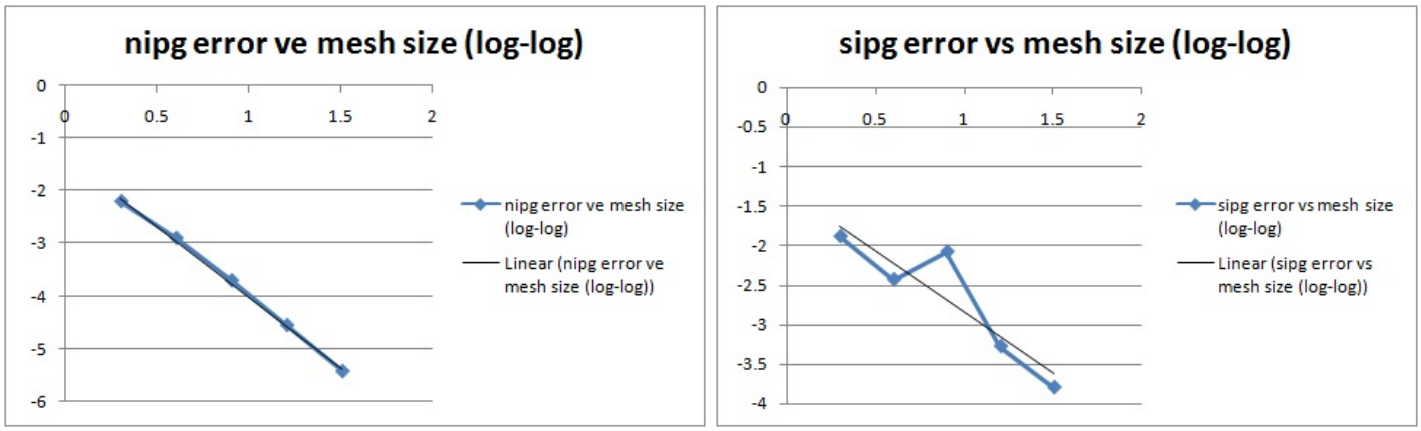


(d) $i = 4$

Figure 5.13: Plots of Error of component y of ∇u by NIPG Method

Chapter 6

Error Plots



(a) Plots for NIPG

(b) Plots for SIPG

Figure 6.1: Plots of convergence of error

References

- Bahriawati, C., and C. Carstensen. "Three MATLAB implementations of the lowest-order Raviart-Thomas MFEM with a posteriori error control." Comput. Methods Appl. Math. 5, no. 4 (2005): 333-361.

- Prudhomme, S., F. Pascal, J. T. Oden, and A. Romkes. "Review of a priori error estimation for discontinuous Galerkin methods." (2000).

- Hecht, Frédéric, Olivier Pironneau, A. Le Hyaric, and K. Ohtsuka. "FreeFem++ manual." (2005).