HiperLife Tutorial: Cavity flow problem

LaCàN

November 20, 2024

1 Problem Definition

This example has become a standard benchmark test for incompressible flows. Figure 1 shows a schematic representation of the problem statement. It models a plane flow of an isothermal fluid in a square lid-driven cavity. The upper side of the cavity moves in its own plane at unit speed, while the other sides are fixed.

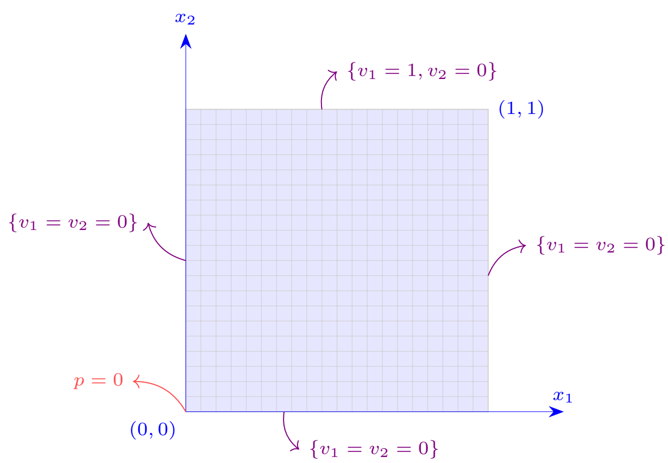


Figure 1: Geometry, boundary conditions and computational domain used for the analysis.

There is a discontinuity in the boundary conditions at the two upper corners of the cavity. Two cases can be envisioned: the two upper corners are either considered as belonging to the top mobile side (leaky cavity), or they are assumed to belong to the fixed vertical walls (non-leaky). The former case is adopted here. It introduces a singularity in the pressure field precisely at those two upper corners. Finally, it should be noticed that Dirichlet boundary conditions are imposed on every boundary in this example. This implies that pressure is known up to a constant at an arbitrary point, the lower left corner of the cavity, the value $p = 0$ is prescribed. Here, we solve the lid-driven cavity for the Stokes problem and the standard Galerkin formulation.[1].

2 Governing Equation

In this section we present the governing equations (continuity and momentum) which are in terms of $(\mathbf{v}, P)$ for isotropic, Newtonian, viscous, incompressible fluids in the presence of body forces:

$$\begin{aligned} & -\nabla \cdot \mathbf{v} = 0\,, \\ & \rho\Big(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}\Big) + \nabla P - \mu \nabla \cdot \Big[(\nabla \mathbf{v}) + (\nabla \mathbf{v}^T)\Big] = \rho \mathbf{f}\,. \end{aligned} \tag{1}$$

where $\mathbf{v}$ represents the velocity vector, ρ is the density, μ the fluid viscosity and $\mathbf{f}$ is the body force vector measured per unit mass. P is the hydrostatic pressure. The boundary conditions for the flow problem are given by

$$\begin{aligned} \mathbf{v} &= \bar{\mathbf{v}} \quad \text{on } \Gamma_D \,, \\ \mathbf{t} \equiv \hat{\mathbf{n}} \cdot \boldsymbol{\sigma} &= \hat{\mathbf{t}} \quad \text{on } \Gamma_N \,. \end{aligned} \tag{2}$$

where $\hat{\mathbf{n}}$ is the unit normal to the boundary and $\hat{\mathbf{t}}$ is the traction. The Cauchy stress tensor $\boldsymbol{\sigma}$ can be define as

$$\boldsymbol{\sigma} = 2\mu\mathbf{D} - P\mathbf{I} \tag{3}$$

where $\mathbf{D} = \frac{1}{2}[(\nabla\mathbf{v}) + (\nabla\mathbf{v}^T)]$ and $\mathbf{I}$ is the unit tensor.

3 Weak Form

The starting point for the development of the finite element models of Eq. (1) is their weak forms. Here we consider steady flow ($\frac{Dv}{dt} = 0$) two-dimensional case. The variation formulation of our model problem can be introduced as find $(\mathbf{v}, p) \in W$ such that

$$\mathcal{F}(\mathbf{v}, P; \mathbf{u}, q) = 0 \quad \forall (\mathbf{u}, q) \in \hat{W} \,. \tag{4}$$

where $W = V \times P$ is a mixed function space, and

$$\mathcal{F}(\mathbf{v}, p; \mathbf{u}, q) = \int_\Omega \mathbf{u}\nabla P - \mathbf{u}\mu\nabla \cdot \Big[(\nabla\mathbf{v}) + (\nabla\mathbf{v}^T)\Big] - q\nabla \cdot \mathbf{v} - \mathbf{u}\rho\mathbf{f} \; \mathrm{d}\Omega \,. \tag{5}$$

and

$$\begin{aligned} \hat{W} &= \{\mathbf{u} \in H^1(\Omega) : \mathbf{u} = 0 \text{ on } \Gamma\}, \\ W &= \{\mathbf{u} \in H^1(\Omega) : \mathbf{u} = 0 \text{ on } (x = 0, x = 1, y = 0), u_2 = 1 \text{ on } y = 1\} \,. \end{aligned} \tag{6}$$

where $(\mathbf{u}, q)$ is a test functions, which will be equated, in the our FE model to the interpolation function used for $(\mathbf{v}, P)$. Applying integration by part, and using the definition of stress, we can rewrite the weak form as following

$$\begin{aligned} 0 &= -\int_{\Omega^e} q\nabla \cdot \mathbf{v} \mathrm{d}V \,, \\ 0 &= \int_{\Omega^e} \mathbf{u}\nabla P - \mathbf{u}\mu\nabla \cdot \Big[(\nabla\mathbf{v}) + (\nabla\mathbf{v}^T)\Big] - \mathbf{u}\rho\mathbf{f}\mathrm{d}V \\ &= \int_{\Omega^e} \Big\{\nabla \cdot (\mathbf{u}P) - P\nabla\mathbf{u}\Big\}\mathrm{d}V + \int_{\Omega^e} \Big\{\mu\nabla\mathbf{u} \cdot \Big[(\nabla\mathbf{v}) + (\nabla\mathbf{v}^T)\Big] - \nabla \cdot \Big(\mathbf{u}\mu\Big[(\nabla\mathbf{v}) + (\nabla\mathbf{v}^T)\Big]\Big)\Big\}\mathrm{d}V - \int_{\Omega^e} \mathbf{u}\rho\mathbf{f}\mathrm{d}V \\ &= \int_{\Gamma^e} \mathbf{u}P\mathbf{I} \cdot \mathbf{n}\mathrm{d}S - \int_{\Omega^e} P\nabla\mathbf{u}\mathrm{d}V - \int_{\Gamma^e} (2\mathbf{u}\mu\mathbf{D}) \cdot \mathbf{n}\mathrm{d}S + \int_{\Omega^e} \mu\nabla\mathbf{u} \cdot \Big[(\nabla\mathbf{v}) + (\nabla\mathbf{v}^T)\Big]\mathrm{d}V - \int_{\Omega^e} \mathbf{u}\rho\mathbf{f}\mathrm{d}V \\ &= -\int_{\Gamma^e} \mathbf{u}\hat{\mathbf{t}}\mathrm{d}S - \int_{\Omega^e} P\nabla\mathbf{u}\mathrm{d}V + \int_{\Omega^e} \mu\nabla\mathbf{u}\Big[(\nabla\mathbf{v}) + (\nabla\mathbf{v}^T)\Big]\mathrm{d}V - \int_{\Omega^e} \mathbf{u}\rho\mathbf{f}\mathrm{d}V \,. \end{aligned} \tag{7}$$

4 Finite Element Model

Since we are developing the Ritz-Galerkin finite element models, the choice of the weight functions is restricted to the spaces of approximation functions used for the pressure and velocity fields. Suppose that the dependent variables (v_i, P) are approximated by expansions of the form

$$\begin{aligned} v_i(\mathrm{x}, t) &= \sum_{m=1}^{M} \psi_m(\mathrm{x})\mathbf{v}_i^m(t) = \mathbf{\Psi}^T\mathbf{v}_i \,, \\ p(\mathrm{x}, t) &= \sum_{n=1}^{N} \phi_n(\mathrm{x})P^n(t) = \mathbf{\Phi}^T\mathbf{P} \,. \end{aligned} \tag{8}$$

where $\mathbf{\Psi}$ and $\mathbf{\Phi}$ are (column) vectors of interpolation (or shape) functions, $\mathbf{v}_i = \{v_1, v_2\}^T$ and $\mathbf{P}$ are vectors of nodal values of velocity components and pressure, respectively, and the superscript $(\cdot)^T$ denotes a transpose of the enclosed vector or matrix. Substitution of these equation into Eq. (4) results in the following finite element equations.

Continuity:

$$-\Big[\int \mathbf{\Phi}\frac{\partial \mathbf{\Psi}}{\partial x_i}\mathrm{d}V\Big]\mathbf{v}_i = 0\,. \tag{9}$$

Momentum:

$$\Big[\int_{\Omega^e}\mu\frac{\partial \mathbf{\Psi}}{\partial x_j}\frac{\partial \mathbf{\Psi}^T}{\partial x_j}\mathrm{d}V\Big]\mathbf{v}_i + \Big[\int_{\Omega^e}\mu\frac{\partial \mathbf{\Psi}}{\partial x_j}\frac{\partial \mathbf{\Psi}^T}{\partial x_i}\mathrm{d}V\Big]\mathbf{v}_j - \Big[\int_{\Omega^e}\mathbf{\Phi}^T\frac{\partial \mathbf{\Psi}}{\partial x_i}\mathrm{d}V\Big]\mathbf{P} = \int_{\Omega^e}\mathbf{\Psi}\rho f_i\mathrm{d}V + \int_{\Gamma^e}\mathbf{\Psi}t_i\mathrm{d}S\,. \tag{10}$$

The above equations can be written symbolically in matrix form as

$$\begin{aligned} -\mathbf{Q}^T\mathbf{v} &= \mathbf{0}\,,\\ \mathbf{K}\mathbf{v} - \mathbf{Q}\mathbf{P} &= \mathbf{F}\,. \end{aligned} \tag{11}$$

By combining continuity and momentum equations into one, Eq. (8) has the following explicit matrix form:

$$\begin{Bmatrix}\mathbf{F_1}\\ \mathbf{F_2}\\ 0\end{Bmatrix} = \begin{bmatrix} 2\mathbf{K}_{11}+\mathbf{K}_{22} & \mathbf{K}_{12} & -\mathbf{Q}_1\\ \mathbf{K}_{21} & \mathbf{K}_{11}+2\mathbf{K}_{22} & -\mathbf{Q}_2\\ -\mathbf{Q}_1^T & -\mathbf{Q}_2^T & \mathbf{0}\end{bmatrix}\begin{Bmatrix}\mathbf{v_1}\\ \mathbf{v_2}\\ \mathbf{P}\end{Bmatrix}\,. \tag{12}$$

The coefficient matrices shown in Eq. (9) are defined by

$$\mathbf{K}_{ij} = \int_{\Omega^e}\mu\frac{\partial \mathbf{\Psi}}{\partial x_i}\frac{\partial \mathbf{\Psi}^T}{\partial x_j}\mathrm{d}V\,, \quad \mathbf{Q}_i = \int_{\Omega^e}\frac{\partial \mathbf{\Psi}}{\partial x_i}\mathbf{\Phi}^T\mathrm{d}V\,, \quad \mathbf{F}_i = \int_{\Omega^e}\rho\mathbf{\Psi}f_i\mathrm{d}V + \int_{\Gamma^e}\mathbf{\Psi}t_i\mathrm{d}S\,. \tag{13}$$

5 Choice of Elements

There are lots of elements available for using in mixed finint elements model, but here for sake of simplicity we choose $Q2Q1$ elements. The quadratic quadrilateral elements shown in Figure 2 are known to give reliable solutions for velocity and pressure fields.

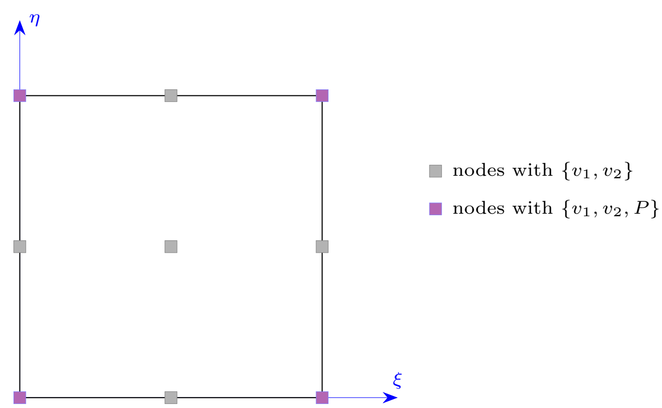


Figure 2: Quadratic quadrilateral element used for the mixed finite element model.

6 Implementation

In this section, we present the implementation of our solution in the Hiperlife. The program is divided into three separate files, main part which we create our problem by the Hiperlife headers, auxiliary header where we introduce parameters and declare our functions, and at last auxiliary file, where we define some functions which provide required matrices like the Hessian and Jacobian.

6.1 CavityFlow.cpp

```cpp
/*
* Incompressible stokes flow: Cavity flow problem
*/
// cpp headers
#include <iostream>
#include <fstream>
#include <cmath>

// hiperlife headers
#include "hl_Core.h"
#include "hl_Parser.h"
#include "hl_Tensor.h"
#include "hl_TypeDefs.h"
#include "hl_DOFsHandler.h"
#include "hl_HiPerProblem.h"
#include "hl_FillStructure.h"
#include "hl_ParamStructure.h"
#include "hl_DistributedMesh.h"
#include "hl_StructMeshGenerator.h"
#include "hl_GlobalBasisFunctions.h"
#include "hl_LinearSolver_Direct_MUMPS.h"
#include "hl_NonlinearSolver_NewtonRaphson.h"
#include "hl_LinearSolver_Iterative_AztecOO.h"

// Header to auxiliary functions
#include "AuxCavityFlow.h"

// ------------------------------------------------------------------------//
/// ----------------    MAIN FUNCTION    ---------------------------------///
// ------------------------------------------------------------------------//

int main(int argc, char** argv)
{
        using namespace std;
        using namespace hiperlife;
        using namespace hiperlife::Tensor;

        // ----------------------------------------------------------------//
        /// *****                 INITIALIZATION                   *****///
        // ----------------------------------------------------------------//

        // Initialize MPI
        hiperlife::Init(argc, argv);

        // ----------------------------------------------------------------//
        /// *****                   DATA INPUT                     *****///
        // ----------------------------------------------------------------//

        // Put parameters in the user structure
        SmartPtr<ParamStructure> paramStr = CreateParamStructure<CavityParams>();

        // Data
        paramStr->setRealParameter(CavityParams::rho, 1.0);
        paramStr->setRealParameter(CavityParams::mu, 0.1);
        paramStr->setRealParameter(CavityParams::f1, 0.0);
        paramStr->setRealParameter(CavityParams::f2, 0.0);

        double rho = paramStr->getRealParameter(CavityParams::rho);
        double mu = paramStr->getRealParameter(CavityParams::mu);
        double f1 = paramStr->getRealParameter(CavityParams::f1);
        double f2 = paramStr->getRealParameter(CavityParams::f2);


        // analysis parameter
        ElemType elemType = ElemType::Square;// Triang or Square
        int n = 10;// number of elements in x and y direction
```

```
    // ------------------------------------------------------------------//
    /// *****                    MESH CREATION                     *****///
    // ------------------------------------------------------------------//

    // Create a structural mesh
    SmartPtr<StructMeshGenerator> StrMesh = Create<StructMeshGenerator>();
    StrMesh->setNDim(3);
    StrMesh->setBasisFuncType(BasisFuncType::Lagrangian);
    StrMesh->setBasisFuncOrder(1);
    StrMesh->setElemType(elemType);
    StrMesh->genSquare(n,1.0);

    //------------------Distributed Mesh-----------------------------------//
    // For Pressure
    SmartPtr<DistributedMesh> disMeshPress = Create<DistributedMesh>();

    disMeshPress->setMesh(StrMesh);
    disMeshPress->setBalanceMesh(true);
    disMeshPress->setElementLocatorEngine(ElementLocatorEngine::BoundingVolumeHierarchy);
    disMeshPress->Update();

    // For Velocity
    SmartPtr<DistributedMesh> disMeshVeloc = Create<DistributedMesh>();

    disMeshVeloc->setMeshRelation(MeshRelation::pRefin, disMeshPress);
    disMeshVeloc->setPRefinement(1);
    disMeshVeloc->setBalanceMesh(true);
    disMeshVeloc->setElementLocatorEngine(ElementLocatorEngine::BoundingVolumeHierarchy);
    disMeshVeloc->Update();

    cout << "---check meshv/p files to see the meshes---" << endl;
    disMeshVeloc->printFileLegacyVtk("meshv");
    disMeshPress->printFileLegacyVtk("meshp");

    // ------------------------------------------------------------------//
    /// *****                DOFsHANDLER CREATION                  *****///
    // ------------------------------------------------------------------//

    // DOFHandler
    // For Velocity
    SmartPtr<DOFsHandler> dhandV = Create<DOFsHandler>(disMeshVeloc);
    dhandV ->setNameTag("dhandV");
    dhandV ->setNumDOFs(2);
    dhandV->setDOFs({"vx","vy"});
    dhandV->Update();

    // For Pressure
    SmartPtr<DOFsHandler> dhandP = Create<DOFsHandler>(disMeshPress);
    dhandP->setNameTag("dhandP");
    dhandP ->setNumDOFs(1);
    dhandP->setDOFs({"p"});
    dhandP->Update();
    cout << "---DOFsHandler for Velocity and Pressure successfully created---" << endl;


    // ---------------- Boundary conditions ------------------------ //
    //------------------------------------------------------------------//
    // Set boundary conditions for the velocity
    //velocities are zero everywhere except at Ymax vx=1
    dhandV->setBoundaryCondition(0, MAxis::Xmin, 0.0);
    dhandV->setBoundaryCondition(1, MAxis::Xmin, 0.0);

    dhandV->setBoundaryCondition(0, MAxis::Xmax, 0.0);
    dhandV->setBoundaryCondition(1, MAxis::Xmax, 0.0);

    dhandV->setBoundaryCondition(0, MAxis::Ymin, 0.0);
    dhandV->setBoundaryCondition(1, MAxis::Ymin, 0.0);

```

```
        dhandV->setBoundaryCondition(0, MAxis::Ymax, 1.0);
        dhandV->setBoundaryCondition(1, MAxis::Ymax, 0.0);

        // Set boundary conditions for the pressure
        // Set initial value for the pressure at (0,0) p=0
        dhandP->setBoundaryCondition(0,0,IndexType::Local,0.0);

        // Update
        dhandV->UpdateGhosts();
        dhandP->UpdateGhosts();

        // ----------------------------------------------------------------//
        /// *****               HIPERPROBLEM CREATION              *****///
        // ----------------------------------------------------------------//

        SmartPtr<HiPerProblem> hiperProbl = Create<HiPerProblem>();
        hiperProbl->setParameterStructure(paramStr);
        hiperProbl->setDOFsHandlers({dhandV, dhandP});
        hiperProbl->setIntegration("IntegCavity", {"dhandV", "dhandP"});

        if (elemType==ElemType::Square)
        hiperProbl->setCubatureGauss("IntegCavity", 4);
        else if (elemType==ElemType::Triang)
        hiperProbl->setCubatureGauss("IntegCavity", 3);

        hiperProbl->setElementFillings("IntegCavity", LS);
        hiperProbl->Update();
        // ----------------------------------------------------------------//
        /// *****                SOLVER CREATION                   *****///
        // ----------------------------------------------------------------//
        // Create linear solver direct
        SmartPtr<MUMPSDirectLinearSolver> dirsolver = Create<MUMPSDirectLinearSolver>();
        dirsolver->setHiPerProblem(hiperProbl);
        dirsolver->setVerbosity(MUMPSDirectLinearSolver::Verbosity::Extreme);
        dirsolver->setDefaultParameters();
        dirsolver->Update();

        // Solve
        dirsolver->solve();
        hiperProbl->UpdateGhosts();

        // Update Solution
        dirsolver->UpdateSolution();

        // ----------------------------------------------------------------//
        /// *****              Post Processing              *****///
        // ----------------------------------------------------------------//

        // Print solution
        dhandP->printFileLegacyVtk("CavityP");
        dhandV->printFileLegacyVtk("CavityV");

        // mpi finilizing
        hiperlife::Finalize();
        return 0;
}
```

6.2 AuxCavityFlow.h

```
#ifndef AUXCavity_H
#define AUXCavity_H

// C headers
#include <iostream>

// hiperlife headers
```

```
#include "hl_Core.h"
#include "hl_Parser.h"
#include "hl_TypeDefs.h"
#include "hl_DOFsHandler.h"
#include "hl_HiPerProblem.h"
#include "hl_FillStructure.h"
#include "hl_ParamStructure.h"
#include "hl_DistributedMesh.h"
#include "hl_StructMeshGenerator.h"
#include "hl_GlobalBasisFunctions.h"
#include "hl_NonlinearSolver_NewtonRaphson.h"
#include "hl_LinearSolver_Iterative_AztecOO.h"

struct CavityParams
{
        enum RealParameters
        {
                rho,
                mu,
                f1,
                f2,
        };
        HL_PARAMETER_LIST DefaultValues
        {
                {"rho,", 1.0},
                {"mu,", 0.1},
                {"f1,", 0.0},
                {"f2,", 0.0},
        };
};

void LS(hiperlife::FillStructure& fillStr);

#endif
```

6.3 AuxCavityFlow.cpp

```
// Header to cpp
#include <fstream>
#include <iostream>
#include <string>

// Header to auxiliary functions
#include "AuxCavityFlow.h"

// Hiperlife headers
#include "hl_Core.h"
#include "hl_ParamStructure.h"
#include "hl_Parser.h"
#include "hl_TypeDefs.h"
#include "hl_GlobalBasisFunctions.h"
#include "hl_StructMeshGenerator.h"
#include "hl_DistributedMesh.h"
#include "hl_FillStructure.h"
#include "hl_DOFsHandler.h"
#include "hl_HiPerProblem.h"
#include "hl_LinearSolver_Iterative_AztecOO.h"
#include "hl_NonlinearSolver_NewtonRaphson.h"

using namespace std;
using namespace hiperlife;
using namespace hiperlife::Tensor;


// Cavity flow

```

```cpp
void LS(hiperlife::FillStructure& fillStr)
{
        using namespace std;
        using namespace hiperlife;
        using hiperlife::Tensor::tensor;

        double rho  = fillStr.getRealParameter(CavityParams::rho);
        double mu = fillStr.getRealParameter(CavityParams::mu);
        double f1 = fillStr.getRealParameter(CavityParams::f1);
        double f2 = fillStr.getRealParameter(CavityParams::f2);
        ttl::tensor<double,1>  F{f1,f2};


        //-------------------------------------------------------------------//
        // -------------------------- INPUT DATA ---------------------------//
        //-------------------------------------------------------------------//

        //---------------------------Velocity-related----------------------//
        // Dimensions
        SubFillStructure& subFill = fillStr["dhandV"];
        int pDim = subFill.pDim;
        int eNN  = subFill.eNN;
        int numDOFs = subFill.numDOFs;

        // Shape functions and derivatives at Gauss points
        double jac{};
        ttl::wrapper<double,2> nborCoords(subFill.nborCoords.data(), eNN, pDim);
        ttl::wrapper<double,2> nborDOFs(subFill.nborDOFs.data(), eNN, numDOFs);
        ttl::tensor<double,1,false>  bf(subFill.nborBFs(), eNN);
        ttl::tensor<double,2> Dbf_g(eNN,pDim);
        GlobalBasisFunctions::gradients(Dbf_g, jac, subFill);

        //---------------------------Pressure-related----------------------//
        // Dimensions
        SubFillStructure& subFill_p = fillStr["dhandP"];
        int nDim_p = subFill_p.nDim;
        int eNN_p  = subFill_p.eNN;
        int numDOFs_p = subFill_p.numDOFs;

        // Shape functions and derivatives at Gauss points
        ttl::wrapper<double,2> nborCoords_p(subFill_p.nborCoords.data(),eNN_p,nDim_p);
        ttl::wrapper<double,1> nborDOFs_p(subFill_p.nborDOFs.data(),eNN_p);
        ttl::tensor<double,1,false>  bf_p(subFill_p.nborBFs(),eNN_p);

        //---------------------------------------------------------------------------------//
        // ------------------------------ OUTPUT DATA ------------------------------------//
        //---------------------------------------------------------------------------------//
        ttl::wrapper<double,2> Bk0(fillStr.Bk(0).data(),eNN,numDOFs);
        ttl::wrapper<double,1> Bk1(fillStr.Bk(1).data(),eNN_p);

        ttl::wrapper<double,4> Ak00(fillStr.Ak(0,0).data(),eNN,numDOFs,eNN,numDOFs);
        ttl::wrapper<double,3> Ak01(fillStr.Ak(0,1).data(),eNN,numDOFs,eNN_p);
        ttl::wrapper<double,3> Ak10(fillStr.Ak(1,0).data(),eNN_p,eNN,numDOFs);
        ttl::wrapper<double,2> Ak11(fillStr.Ak(1,1).data(),eNN_p,eNN_p);

        //------------------------------- EQUATIONS ------------------------------------//
        //-------------------------------------------------------------------------------//
        for (int i=0;i < eNN;i++)
        {
                for (int j=0;j < eNN;j++)
                {
                        Ak00(i,0,j,0)+=jac*mu*(2.0*Dbf_g(i,0)*Dbf_g(j,0)+Dbf_g(i,1)*Dbf_g(j,1));
                        Ak00(i,0,j,1)+=jac*mu*Dbf_g(i,1)*Dbf_g(j,0);
                        Ak00(i,1,j,0)+=jac*mu*Dbf_g(i,0)*Dbf_g(j,1);
                        Ak00(i,1,j,1)+=jac*mu*(2.0*Dbf_g(i,1)*Dbf_g(j,1)+Dbf_g(i,0)*Dbf_g(j,0));
                }
                for (int j=0;j < eNN_p;j++)
                {
```

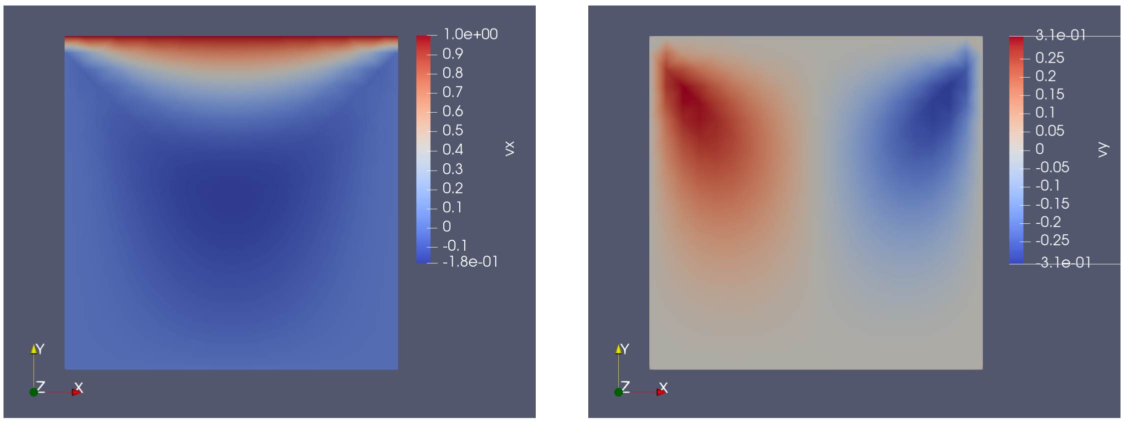
```
                    Ak01(i,0,j)-=jac*Dbf_g(i,0)*bf_p(j);
                    Ak01(i,1,j)-=jac*Dbf_g(i,1)*bf_p(j);

                    Ak10(j,i,0)-=jac*bf_p(j)*Dbf_g(i,0);
                    Ak10(j,i,1)-=jac*bf_p(j)*Dbf_g(i,1);
                }
        }

}
```

7 Results

In this section, we present the results of our solution. Figure 3 shows the velocities distribution in our domain.



(a) velocity in x-direction

(b) velocity in y-direction

Figure 3: velocity of flow in cavity

Pressure contour is presented in Figure 4.

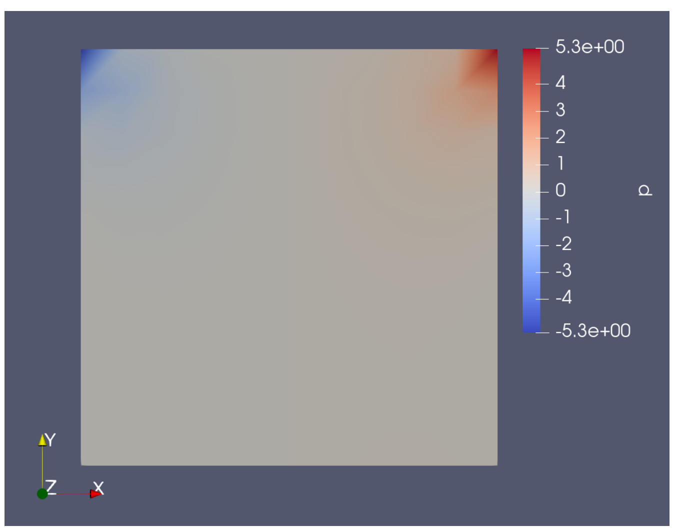


Figure 4: Pressure distribution in the domain.

References

[1] Jean Donea and Antonio Huerta. *Finite element methods for flow problems*. John Wiley & Sons, 2003.