Christian Rowsell

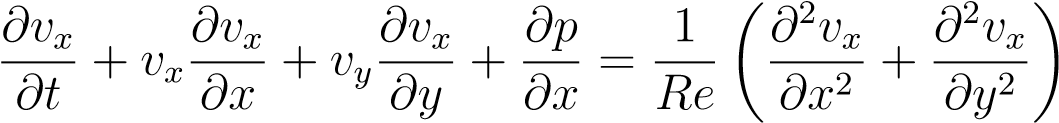
UBC Mechanical Engineering – Fall 2022

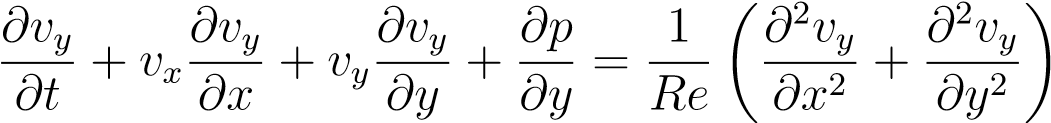
Mech 587 Project 3

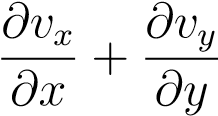
Navier-Stokes Solution using SIMPLE Method

Programming Assignment – 3

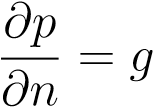
In this assignment, your final objective is to numerically solve the Navier-Stokes equation with **S**emi-**I**mplicit **M**ethod for **P**ressure **L**inked **E**quations (SIMPLE), and verify your code with the lid-driven cavity problem. The Navier-Stokes equations for the time interval (0*,T*) in a domain Ω is given by,

 in (0*,T*) × Ω*,* (1)

 in (0*,T*) × Ω*,* (2)

= 0 in (0*,T*) × Ω*,* (3)

|  |  |  |
| --- | --- | --- |
| *v*(*x,y,*0) = *v*0*,* | *p*(*x,y,*0) = *p*0 | (4) |
| *v*(*x,y,t*) = *vD* | on *∂*Ω*D,* | (5) |

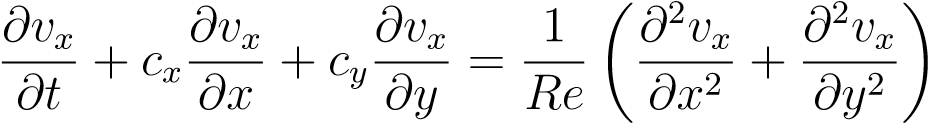
 on *∂*Ω*N,* (6)

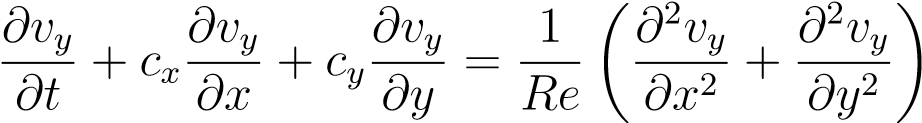
where *∂*Ω*D* and *∂*Ω*N* denote the regions of the boundary over which Dirichlet and Neumann boundary conditions are applied respectively. *v* = (*vx,vy*) and *p* denote the velocity and pressure field over the domain Ω.

# Preparation

The implementation of the Navier-Stokes solver with SIMPLE method can be decomposed into the following steps (There are 40% marks towards these steps! You need to write them together so that you can modify your file toward the Navier-Stokes solver later):

1. 2D advection-diffusion equations Solve the following equations:

 in (0*,T*) × Ω*,*

 in (0*,T*) × Ω*,* • For the spatial discretization, discretize the convective and diffusive terms using the 2*nd* order central differencing scheme

* For the time integration, Use (a) Adams Bashforth method for the convective terms and (b) Trapezoidal method for the diffusive terms.

Check your code with the second problem in project 2. Run the code at any of three grids with difference level of refinement. Report ||*vx* − *ϕe*||2 and ||*vy* − *ϕe*||2, where *ϕe* is the exact solution in project 2. Report the order of convergence. Save the code for your own reference.

Project 2 code was modified so that it can solve both the X and Y momentum equations simultaneously. The two convective velocity terms cx, and cy were taken from project 2, and were kept at constant values of 1.0, and 0.05 respectively. The modified solveConvectionDiffusion code can be seen as follows.

void SolveConvectionDiffusion**(**const Grid **&**G**,** const double tf**,** double dt**)**

**{**

const size\_t Nx **=** G**.**Nx**();**

const size\_t Ny **=** G**.**Ny**();**

// Solve and plot exact solution

Vector phi\_exact**(**Nx**,**Ny**);**

InitializePhiExact**(**phi\_exact**,** G**);**

char fname4**[**20**]** **=** "Phi\_exact.vtk"**;**

storeVTKStructured**(**phi\_exact**,** G**,** fname4**);**

// Solve and plot initial variables for initial conditions

Vector u\_conv**(**Nx**,** Ny**);**

Vector v\_conv**(**Nx**,** Ny**);**

Vector p\_init**(**Nx**,** Ny**);**

InitializeVelCD**(**u\_conv**,**v\_conv**,**G**);**

char fname1**[**30**]** **=** "Init\_Conv.vtk"**;**

storeVTKSolution**(**u\_conv**,** v\_conv**,** p\_init**,** G**,** fname1**);**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Declare variables and begin solving for

X-momentum equation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Vector VelX**(**Nx**,**Ny**);** // X-velocity

InitializePhiCD**(**VelX**,**G**);** // Initialize X velocity

// variables needed for Newton-Raphson iteration in X

Matrix Ax**(**Nx**,**Ny**);** // LHS matrix Ax

Vector Rx**(**Nx**,**Ny**);** // Residual Rx

Vector dVelX**(**Nx**,**Ny**);** // Increment dphi

// variables needed for calculating the residual

Vector fc\_Curr\_X**(**Nx**,**Ny**);** // Forward convection

Vector fc\_Prev\_X**(**Nx**,**Ny**);** // Previous Convection

Vector diffuseX**(**Nx**,**Ny**);** // Diffusive Residual

// Solve First Iteration

computeTransientMatrix**(**Ax**,** G**,** dt**);**

double R0X **=** 0.0**;** // X Residual Norm

computeDiffusion**(**diffuseX**,**VelX**,**G**);**

CDS2**(**fc\_Curr\_X**,** VelX**,** u\_conv**,** v\_conv**,** G**);**

fc\_Prev\_X **=** fc\_Curr\_X**;**

computeResidual**(**Rx**,**fc\_Curr\_X**,**fc\_Prev\_X**,**diffuseX**);**

applyBC**(**Rx**,**dVelX**,**G**,** VelX**);**

R0X **=** Rx**.**L2Norm**();**

printf**(**"Initial X Residual Norm = %14.12e\n"**,** R0X**);**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Declare variables and begin solving for

Y-momentum equation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Vector VelY**(**Nx**,**Ny**);** // Y-velocity

InitializePhiCD**(**VelY**,**G**);** // Initialize Y velocity

// variables needed for Newton-Raphson iteration in Y

Matrix Ay**(**Nx**,**Ny**);** // LHS matrix Ay

Vector Ry**(**Nx**,**Ny**);** // Residual Ry

Vector dVelY**(**Nx**,**Ny**);** // Increment dphi

// variables needed for calculating the residual

Vector fc\_Curr\_Y**(**Nx**,**Ny**);** // Forward convection

Vector fc\_Prev\_Y**(**Nx**,**Ny**);** // Previous Convection

Vector diffuseY**(**Nx**,**Ny**);** // Diffusive Residual

// Solve First Iteration

computeTransientMatrix**(**Ay**,** G**,** dt**);**

double R0Y **=** 0.0**;** // X Residual Norm

computeDiffusion**(**diffuseY**,**VelY**,**G**);**

CDS2**(**fc\_Curr\_Y**,** VelY**,** u\_conv**,** v\_conv**,** G**);**

fc\_Prev\_Y **=** fc\_Curr\_Y**;**

computeResidual**(**Ry**,**fc\_Curr\_Y**,**fc\_Prev\_Y**,**diffuseY**);**

applyBC**(**Ry**,**dVelY**,**G**,** VelY**);**

R0Y **=** Ry**.**L2Norm**();**

printf**(**"Initial Y Residual Norm = %14.12e\n"**,** R0Y**);**

char fileName**[**50**]** **=** "solution\_0.vtk"**;**

storeVTKSolution**(**VelX**,** VelY**,** p\_init**,** G**,** fileName**);**

unsigned long itime **=** 0**;** double time **=** 0.0**;** bool last **=** **false;**

**while(**time **<** tf**)**

**{**

// Update convection terms

fc\_Prev\_X **=** fc\_Curr\_X**;**

fc\_Prev\_Y **=** fc\_Curr\_Y**;**

// Solve Au=b for X and Y Momentum

solveGS**(**dVelX**,**Ax**,**Rx**);**

solveGS**(**dVelY**,**Ay**,**Ry**);**

// Increment solution

VelX **=** VelX **+** dVelX**;**

VelY **=** VelY **+** dVelY**;**

// calculate the diffusion vector for X and Y momentum

computeDiffusion**(**diffuseX**,**VelX**,**G**);**

computeDiffusion**(**diffuseY**,**VelY**,**G**);**

// calculate the convection vector at the current time step

CDS2**(**fc\_Curr\_X**,** VelX**,** u\_conv**,** v\_conv**,** G**);**

CDS2**(**fc\_Curr\_Y**,** VelY**,** u\_conv**,** v\_conv**,** G**);**

// Calculate Residual

computeResidual**(**Rx**,**fc\_Curr\_X**,**fc\_Prev\_X**,**diffuseX**);**

computeResidual**(**Ry**,**fc\_Curr\_Y**,**fc\_Prev\_Y**,**diffuseY**);**

// Apply Boundary Conditions

applyBC**(**Rx**,**dVelX**,**G**,** VelX**);**

applyBC**(**Ry**,**dVelY**,** G**,** VelY**);**

// Calculate Residual Norm

double R1X **=** Rx**.**L2Norm**();**

double R1Y **=** Ry**.**L2Norm**();**

// Check if final time step is reached

time **=** **(**last **==** **true** **?** tf **:(**itime**+**1**)\***dt**);**

**if(**itime **%** 1 **==** 0 **||** last**)**

**{**

std**::**cout**<<**std**::**endl**;**

printf**(**"==================================================================================================\n"**);**

printf**(**"Timestep = %ld, Time = %lf, X-Residual = %14.12e, Y-Residual = %14.12e\n"**,** itime**,** time**,**

dVelX**.**L2Norm**(),** dVelY**.**L2Norm**());**

printf**(**"Max X-Velocity = %14.12e, Min X-Velocity = %14.12e,\n"**,**

VelX**.**GetMax**(),**VelX**.**GetMin**());**

printf**(**"X Residual Norm = %14.12e,\nX Residual Norm Ratio (R/R0X) = %14.12e\n"**,** R1X**,** R1Y**/**R0X**);**

printf**(**"Max Y-Velocity = %14.12e, Min Y-Velocity = %14.12e\n"**,** VelY**.**GetMax**(),**VelY**.**GetMin**());**

printf**(**"Y Residual Norm = %14.12e,\nY Residual Norm Ratio (R/R0X) = %14.12e\n"**,** R1Y**,** R1Y**/**R0Y**);**

printf**(**"==================================================================================================\n"**);**

std**::**cout**<<**std**::**endl**;**

**}**

**if(**last**)**

**break;**

**if((**itime **+** 1**)** **%** 10 **==** 0**)**

**{**

sprintf**(**fileName**,** "solution\_%lu.vtk"**,** itime **+** 1**);**

storeVTKSolution**(**VelX**,** VelY**,** p\_init**,** G**,** fileName**);**

**}**

//Check convergence

**if(**dVelX**.**L2Norm**()** **<** 1e-8 **&&** dVelY**.**L2Norm**()){**

printf**(**"Steady state reached in %lu time steps.\n Final time = %lf.\n"**,**itime**,**itime**\***dt**);**

**break;**

**}**

**++**itime**;**

**if(**itime**\***dt **>** tf**){**

dt **=** tf **-** **(**itime**-**1**)\***dt**;**

last **=** **true;**

**}**

**}**

The code was then ran at 3 different grid sizes, using a 17x17 grid size, a 33x33 grid size, and a 65x65 grid size. The L2Norm error was calculated with respect to the exact solution found in project 2. These values can be found in

|  |  |  |
| --- | --- | --- |
| Grid Size | L2Norm | |
| ||*vx* − *ϕe*||2 | ||*vy* − *ϕe*||2 |
| 17x17 (289 grid points) | 2.558504873729e-02 | 2.55850487372923e-02 |
| 33x33 (1089 grid points) | 6.321108496806e-03 | 6.32110849680624e-03 |
| 65x65 (4225 grid points) | 1.569247085119e-03 | 1.56924708511860e-03 |

Table : L2Norm Error for Solving Convection-Diffusion in X and Y

A Python code was written to plot and report the order of accuracy for both of these values. The output can be seen below.

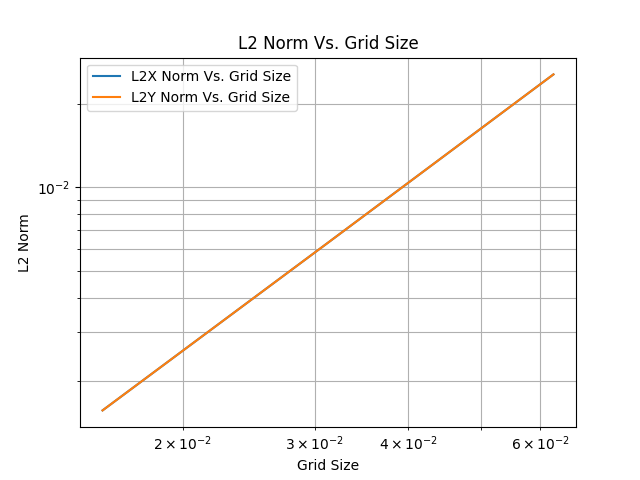


Figure : X and Y Convection Diffusion Order of Accuracy Plot



Figure : X and Y Order of Accuracy Printout

The contour for both solutions can be seen below.

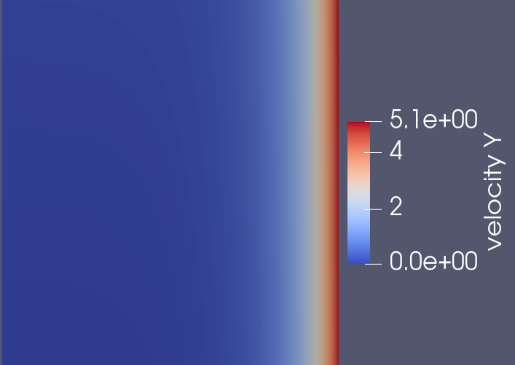
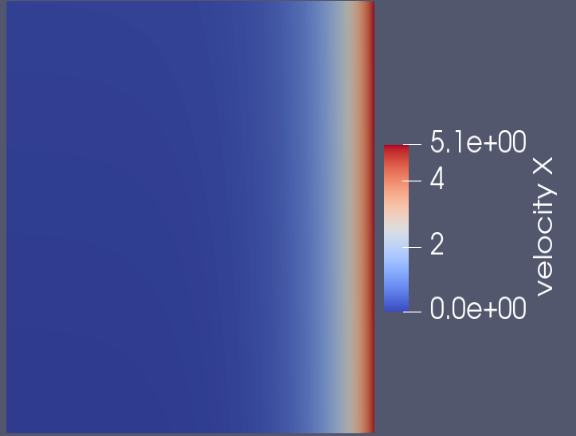
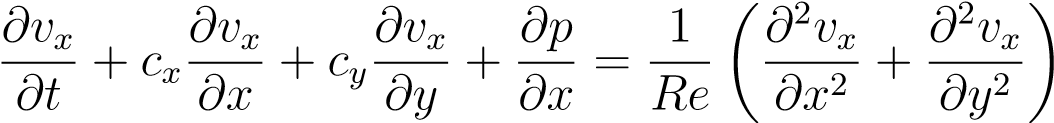
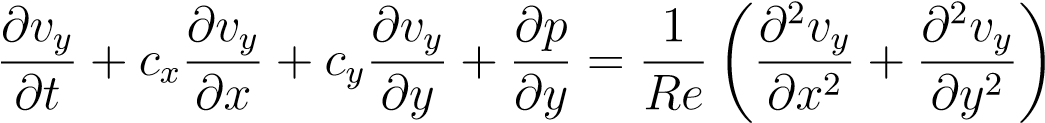


Figure : Project 2 Comparison X Velocity

Figure : Project 2 Comparison Y Velocity

2. Include the pressure gradient

 in (0*,T*) × Ω*,*

 in (0*,T*) × Ω*,*

* Write a function which takes *p* and give *∂p/∂x* and *∂p/∂y* for the interior nodes.
* Use the 2*nd* order central difference scheme for spatial discretization

Given *p* = *x*, check the vector *∂p/∂x*. Similarly, given *p* = *y*, check the vector *∂p/∂y*. Report the calculated value at one of your interior grid point.

A function to calculate the pressure gradient was created. It takes 2 pressure values, and calculates the pressure gradient with respect to x for one, and with respect to y for the other. The code for this can be seen below.

// Calculate the gradient of the pressure for part 1

void CalcGradPressure**(**Vector **&**gradpx**,** Vector **&**gradpy**,** const Vector **&**px**,**const Vector **&**py**,** const Grid **&**G**)**

**{**

unsigned long i**,** j**;**

const unsigned long Nx **=** G**.**Nx**();**

const unsigned long Ny **=** G**.**Ny**();**

const double dx **=** G**.**dx**();**

const double dy **=** G**.**dy**();**

// Interior points only

**for(**i **=** 1**;** i **<** Nx **-** 1**;** i**++)**

**for(**j **=** 1**;** j **<** Ny **-** 1**;** j**++)**

**{**

gradpx**(**i**,** j**)** **=** **(**px**(**i **+** 1**,** j**)** **-** px**(**i **-** 1**,** j**))** **/** **(**2 **\*** dx**);**

gradpy**(**i**,** j**)** **=** **(**py**(**i**,** j **+** 1**)** **-** py**(**i**,** j **-** 1**))** **/** **(**2 **\*** dy**);**

**}**

**}**

The pressure gradients at any point away from the boundary was found to be 1.0 for both the gradient in the x and y directions. This makes sense, as the gradient in the x direction is taken for p=x. The derivative of x with respect to x is 1. The same logic can be applied for the y direction.

3. Pressure correction equation

Solve the following equation:

∆*p* = *s*

where *s* is a vector of source term

• The Laplacian operator is discretized with the 2*nd* order central difference method.

Check the solver with the problem 1 in the project 1. Solve for three grids with different level of refinement. Report the *L*2 norm of the error and the order of convergence.

The code from project 1 was modified, and incorporated into the newly made code. This was then ran at 3 different grid sizes, using 17 points, 33 points and 65 points. The L2Error can be seen below.

|  |  |
| --- | --- |
| Grid Size | L2Norm |
| 17x17 (289 grid points) | 6.048073324687e-04 |
| 33x33 (1089 grid points) | 1.561666503123e-04 |
| 65x65 (4225 grid points) | 3.966651043744e-05 |

# Navier-Stokes solver with SIMPLE method

Adjust the prepared code towards the Navier-Stokes solver with SIMPLE method (see project description).

# Verification with the cavity problem

The lid-driven cavity problem is a widely used test case for benchmarking incompressible flow code. Then fluid contained inside a squared cavity is set into motion by the upper wall, which is sliding at a constant speed (see Fig. 1). Take the the sliding velocity as *U* = 1, density of the incompressible fluid to be *ρ* = 1 and the dynamic viscosity to be *µ* = 0*.*01.

1. Start by using 33 by 33 grid so that you have nodes located at the centerline of the cavity at *x* = 0*.*5 and *y* = 0*.*5. Run the simulation until steady state. Bisecting the length of the intervals until 129 by 129.

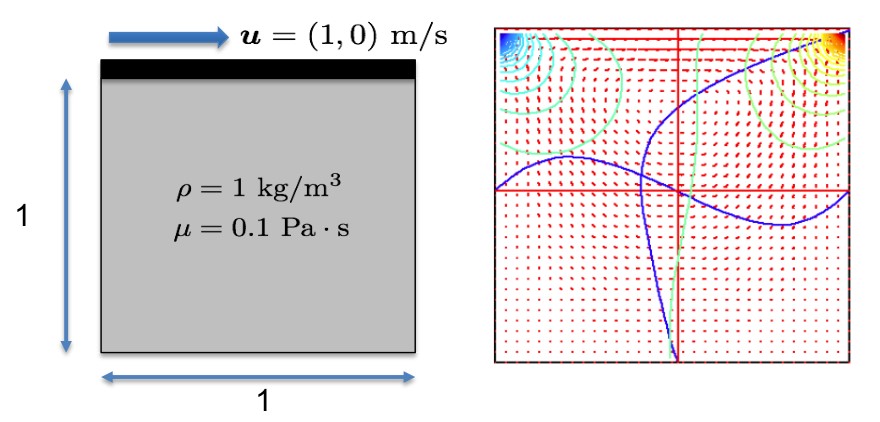


Figure 1: Cavity problem: definition of the problem (left) and demonstration of the solution with pressure contour and velocity at mid-lines (right)

1. Plot the velocity at the centerline against the TABLE I and TABLE II in reference [1] at different grid sides.
2. Report the spatial convergence of the velocity and pressure field.

# References

[1] UKNG Ghia, Kirti N Ghia, and CT Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 48(3):387–411, 1982.