WEST MICHIGAN AVIATION ACADEMY

# AP Calculus Final Project: Conjugate Gradient Method Applied to Discretized Incompressible Navier-Stokes Equations

Sebastian Grabill, Nicholas Grabill
April 22, 2025

## 1 INTRODUCTION

The problem I solve and seek to improve on is a lid-driven cavity flow in two-dimensions for in-compressible Navier-Stokes equations. The use of a discritized Navier-Stokes equations allows for streamlining of the computational resources in fluid modeling. The Navier-Stokes equations describe the flow of fluids and for an in-compressible fluid these equations are given by:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p = \nu\Delta\mathbf{u} + \mathbf{F}, \qquad x \in \Omega,\ t > 0 \tag{1.1}$$

$$\nabla\cdot\mathbf{u} = 0, \qquad x \in \bar{\Omega},\ t > 0 \tag{1.2}$$

$$B(\mathbf{u}, p) = \mathbf{g}, \qquad x \in \partial\Omega,\ t > 0 \tag{1.3}$$

$$\mathbf{u}(\mathbf{x}, 0) = \mathbf{f}(\mathbf{x}), \qquad x \in \Omega,\ t = 0, \tag{1.4}$$

where $\mathbf{u} = (u, v)$ is the velocity, $p$ is the kinematic pressure, $\nu$ is the kinematic viscosity, and $\mathbf{F}$ is a force per volume. I considered these equations over some bounded domain, $\Omega$, with boundary conditions given by $B$ and initial distribution of velocities of $\mathbf{f}$. We consider the divergence of the momentum equation which adds the following equation:

$$\Delta p + J(\nabla\mathbf{u}) - \alpha\nabla\cdot\mathbf{u} = \nabla\cdot\mathbf{F} \quad x \in \Omega,\ t > 0, \tag{1.5}$$

where I added the $\alpha\nabla\cdot\mathbf{u}$ to control for numerically created divergence, and for the dimensional case that we consider $J(\nabla\mathbf{u}) = u_x^2 + u_y^2 + 2u_y u_x$. Through this addition the problem reduces to solving three Poisson Equations: two in velocity, and another in pressure.

For the pressure boundary condition we consider the so called "curl-curl" boundary condition:

$$\frac{\partial p}{\partial t} = \mathbf{n} \cdot \left( -\frac{\partial \mathbf{g}}{\partial t} + (\mathbf{g} \cdot \nabla)\mathbf{u} - \nu \nabla \times \nabla \times \mathbf{u} \right), \quad x \in \partial\Omega,\ t > 0, \tag{1.6}$$

which enforces the in-compressibility on the boundary, is stable, and does not cause time step restriction.

I now follow the discretization the Navier-Stokes equations suggested in Reference, with the forcing term **F** set to zero. This yields:

$$\left( \frac{\mathbf{I}}{\Delta t} - \frac{1}{2} L_I \right) \mathbf{U}^{n+1} = \frac{\mathbf{U}^n}{\Delta t} + \frac{3}{2} L_E \mathbf{U}^n - \frac{1}{2} L_E \mathbf{U}^{n-1} + \frac{3}{2} L_I \mathbf{U}^n \tag{1.7}$$

where $L_I$ is the viscous term, $L_E$ is the nonlinear explicit term using Adams-Bashforth. Let $\mathscr{L}_D$ denote the matrix corresponding to a centered finite difference discretization of $\Delta u$ with Dirichlet boundary conditions and uniform grid spacing $h_x$ and $h_y$, then I solved for the velocities with the system

$$\left( \frac{\mathbf{I}}{\Delta t} - \frac{1}{2} \mathscr{L}_D \right) \mathbf{W}^{n+1} = \mathbf{h} \tag{1.8}$$

where **W** is the grid function for the velocity under consideration and **h** incorporates the right hand side from above and boundary conditions. We note that this is symmetric, positive definite. With $\mathbf{U}^{n+1}$ known I solve for the pressure of the system, with a modification for uniqueness give

$$\begin{pmatrix} \mathscr{L}_N & \mathbf{r} \\ \mathbf{r}^T & 0 \end{pmatrix} \begin{pmatrix} \mathbf{P}^{n+1} \\ \beta \end{pmatrix} = \begin{pmatrix} \mathbf{h} \\ 0 \end{pmatrix}, \tag{1.9}$$

where the matrix on the left is non-singular.

In the provided code, equations 1.8 and 1.9 are solved via dense linear algebra methods from **LAPACK** (Linear Algebra PACKage). The subroutine **DGETRF** computes the LU decomposition of a matrix, which is utilized by subroutine **DGETRS** to provide solutions to the above systems. I replaced these methods with the iterative conjugate gradient (**CG**) method given in Reference [**?**]:

**Algorithm 1:** Conjugate Gradient Method to solve $Ax = b$

**Input:** Symmetric, Positive Definite Matrix A, vector b
**Output:** solution $x_n$ after a stopping condition
Set $x_0 = 0$;
Set $r_0 = b$;
Set $p_0 = r_0$;
**for** $n = 1 \ldots$ **do**
    Set $\alpha_n = (r_{n-1}^T r_{n-1}) / p_{n-1}^T A p_{n-1})$;
    Set $x_n = x_{n-1} + \alpha_n p_{n-1}$;
    Set $r_n = r_{n-1} - \alpha_n A p_{n-1}$;
    Set $\beta_n = (r_n^T r_n) / (r_{n-1}^T r_{n-1})$;
    Set $p_n = r_n + \beta_n p_{n-1}$;
**end**

## 2 FORTRAN IMPLEMENTATION

Listing 1: Conjugate Gradient Subroutine in FORTRAN 90

```
subroutine conjugateGradient(TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO)
      ! This routine computes solution to Ax = b
      ! via the conjugate gradient method.

      character, intent(in) :: TRANS     ! Form of system of equations
      integer, intent(in) :: N           ! Order of matrix A
      integer, intent(in) :: NRHS        ! Columns of matrix B
      integer, intent(in) :: LDA         ! The leading dimension of array A.
      real(dp), intent(in) :: A(LDA,N)   ! The matrix A.
      integer, intent(in) :: IPIV(n)     ! The pivot indices from DGETRF.
      integer, intent(in) :: LDB         ! The leading dimension of array B.
      real(dp), intent(inout) :: B(LDB)  ! Entry: Matrix B; Exit: Solution
      integer, intent(inout) :: INFO     ! 0 : Success | -i : illegal i-th arg
      integer :: i                       ! for initial setup
      ! for final iterating
      real(dp) :: a_k, b_k, res_norm_new, res_norm_old, res_norm_0
      x_k = 0_dp                              ! Approximation to Solution
      r_k = b - matmul(A, x_k)                ! Residual Error
      p_k = r_k                               ! Search Direction
      res_norm_old = dot_product(r_k, r_k)
      res_norm_0 = res_norm_old
      i = 0
      do while(sqrt(res_norm_old / res_norm_0) > cg_tol)
        q_k = matmul(A, p_k)
        a_k = res_norm_old / (dot_product(p_k,q_k)) ! step length
        x_k = x_k + a_k * p_k                       ! approximate solution
        if(mod(i,100).eq.0) then              ! Every 100 iterations
          r_k = b - matmul(A, x_k)            ! reset residual,
        else
          r_k = r_k - a_k * q_k               ! else compute residual
        end if
        res_norm_new = dot_product(r_k, r_k)
        b_k = res_norm_new / res_norm_old
        p_k = r_k + b_k * p_k                 ! new search direction
        res_norm_old = res_norm_new
        i = i + 1
      end do
      INFO = i
      B = x_k
    end subroutine conjugateGradient
```

This method, though, performs frequent and expensive matrix multiplications. To see the true potential in time gain from iterative methods over direct methods, we must implement a matrix free variation of the conjugate gradient routine utilizing the velocity Laplacian routine:

Listing 2: Matrix Free Conjugate Gradient Subroutine in FORTRAN 90

```
subroutine conjugateGradient_matrixfree_uv(B, N, Nx, Ny, hx, hy, k, nu)
      ! This routine computes solution to Ax = b
      ! by the matrix free conjugate gradient method

      integer, intent(in) :: N, Nx, Ny ! N: The order of A; Nx, Ny: grid dims
      real(dp), intent(inout) :: B(N) ! Entry: RHS matrix B. Exit: Sol
      real(dp), intent(in) :: hx, hy, k, nu ! hx, hy: grid size in x, y; k: t-step
      integer :: i ! for initial setup

      ! for final iterating
      real(dp) :: a_k, b_k, res_norm_new, res_norm_old, res_norm_0
      x_k = 0_dp
      CALL apply_velocity_laplacian(a_x, x_k, Nx, Ny, hx, hy)
      a_x = -0.5d0*nu*a_x + (1.0d0/k)*x_k
      r_k = b - a_x
      p_k = r_k
      res_norm_old = dot_product(r_k, r_k)
      res_norm_0 = res_norm_old
      i = 0
      do while(sqrt(res_norm_old / res_norm_0) > cg_tol)
        CALL apply_velocity_laplacian(q_k, p_k, Nx, Ny, hx, hy)
        q_k = -0.5d0*nu*q_k + (1.0d0/k)*p_k
        a_k = res_norm_old / (dot_product(p_k,q_k)) ! step length
        x_k = x_k + a_k * p_k ! approximate solution
        if(mod(i,100).eq.0) then
          CALL apply_velocity_laplacian(a_x, x_k, Nx, Ny, hx, hy)
          a_x = -0.5d0*nu*a_x + (1.0d0/k)*x_k
          r_k = b - a_x
        else
          r_k = r_k - a_k * q_k ! residual
        end if
        res_norm_new = dot_product(r_k, r_k)
        b_k = res_norm_new / res_norm_old ! improvement this step
        p_k = r_k + b_k * p_k ! search direction
        res_norm_old = res_norm_new
        i = i + 1
      end do
      B = x_k
    end subroutine conjugateGradient_matrixfree_uv
```

These are applied to both the velocity and pressure, although an entirely different module for the pressure was needed, but it contains essentially the same FORTRAN 90 code as in the velocity module. This is necessary because the problem sizes are different in the velocity and pressure cases, but the Conjugate Gradient algorithm is nonetheless the same.

## 3 Results & Discussion

For this section take $N_x$ to be the number of gridpoints in the $x$ direction, $N_y$ to be the number of gridpoints in the $y$ direction, $N$ to be the number of timesteps, $n$ to be the time step, and $Re$ to be the Reynold's number. These parameters are used in Incompressible Navier-Stokes (INS) simulations described in this section.

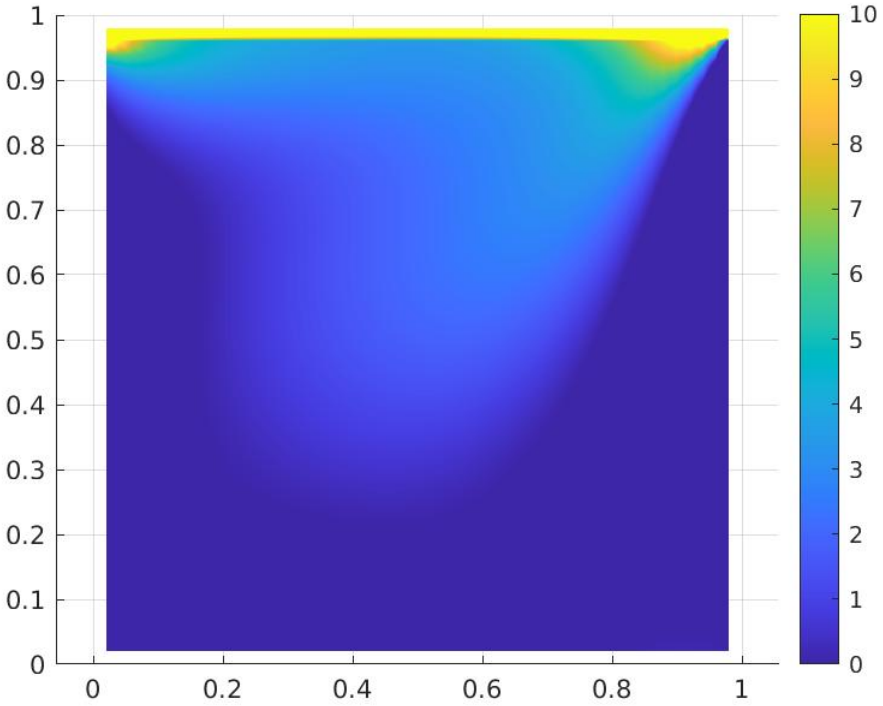


Figure 3.1: INS Simulation with $N_x = N_y = 50$, $N = 5000$, $n = 0.01$, and $Re = 100$. This is the representation of the system at timestep 5000.

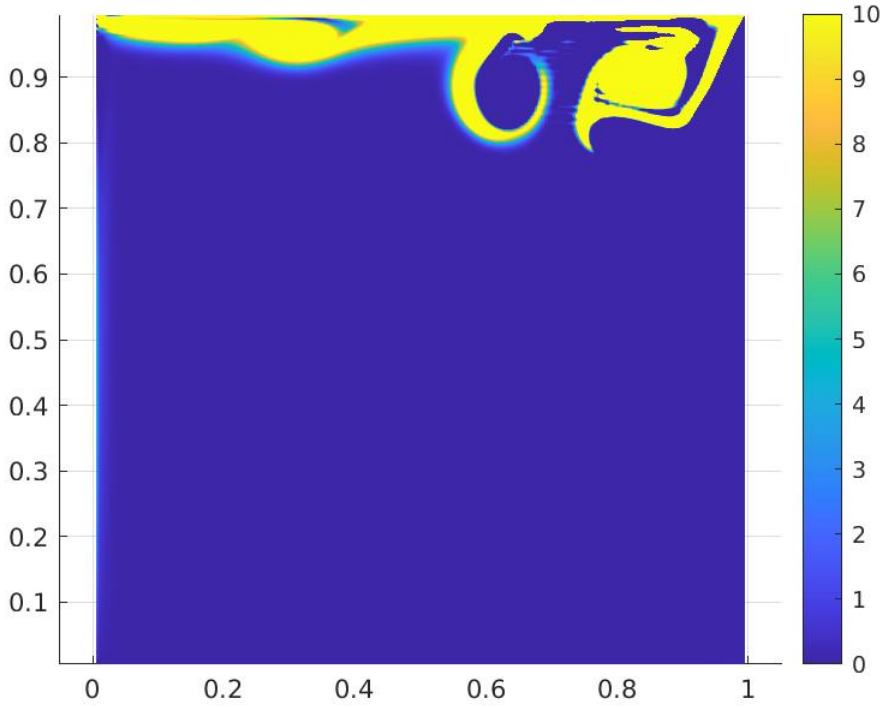


Figure 3.2: INS Simulation with $N_x = N_y = 200$, $N = 5000$, $n = 0.00025$, and $Re = 8000$. This is the representation of the system at timestep 4300. Note: Blowup at timestep 4400.

When my simulation described in Figure 3.1 was run using FORTRAN implementation in Listing 1, I found that it took approximately 879 seconds to run 5000 timesteps. When the same simulation was run using the FORTRAN implementation in Listing 2, I found that it took approximately 12 seconds to run 5000 timesteps. Therefore, our simulation speed experienced a speedup of 7300% between the two **CG** methods in Listings 1 and 2.

## 4 Conclusions & Future Work

My effort demonstrated the benefits of implementing the conjugate gradient iterative method over dense linear algebra for an in-compressible Navier-Stokes discrete simulation. Without the performance benefits of **CG**, and the further performance gain of Matrix Free **CG**, I would not have been able to assess more complicated problems involving larger Reynolds numbers, or simulations discretized over finer mesh.

If time allowed, future work would entail a fuller exploration and analysis of high Reynolds number situations. We encountered problems on the MSU HPCC when increase mesh size, likely due to memory availability. Implementing code changes that reduce memory footprints would be potential avenue for code development.

## 5 ACKNOWLEDGEMENTS

Nicholas Grabill gave raw code that was expanded upon, provided insight into use of PDEs and Navier-Stokes equations for applications in discretized incompressible Navier-Stokes problems.

I used Fortran to run the simulation code and MATLAB to do the data handling. I ran these simulations locally and with Nicholas Grabill's HPCC access at MSU.

# 6 REFERENCES

(1) Lloyd Trefethen, David Bau, 1995. "Numerical Linear Algebra" Society for Industrial and Applied Mathematics. 1997.