Numerical Solution of the Laminar Boundary Layer Equations using the Crank-Nicolson Algorithm

AERSP/ME 525: Turbulence and Applications to CFD: RANS
Computer Assignment Number 2

Brian Knisely

March 26, 2018

Abstract

A finite difference scheme was used to estimate the solution to the laminar boundary layer equations. A second-order accurate Crank-Nicolson scheme in the x-direction was combined with a fourth-order accurate finite difference scheme in the y-direction to march in x starting with the starting profile. For each step in x, a coupled set of algebraic equations were solved explicitly for the x-velocity using an LU decomposition matrix inversion algorithm. After the x-velocity, the y-velocity at a particular step was solved using the continuity equation with an asymmetric finite difference scheme of up to fourth order. The velocity profiles at five locations, displacement thickness, and momentum thickness were computed and compared to the Blasius solution. A code was written in Python to calculate the solution according to the derived scheme. A grid stretched in the y-direction with 151 nodes in the y-direction and 41 nodes in the x-direction was used and the effects of varying the key stretching parameter were investigated.

Introduction

In the field of computational fluid dynamics, the flowfield is governed by partial differential equations which must be solved to determine key parameters like velocity. For most realistic problems, these equations do not have an analytical solution, so engineers are forced to used numerical methods to approximate the solutions. This paper presents a process for numerically solving a the laminar boundary layer equations using a finite difference method, and offers comparisons to the exact Blasius solution.

Mathematical Problem

The given zero pressure gradient boundary layer equations are the following partial differential equations (PDEs) which govern conservation of mass and momentum, respectively.

$$\frac{\partial \tilde{u}}{\partial \tilde{x}} + \frac{\partial \tilde{v}}{\partial \tilde{y}} = 0 \tag{1}$$

$$\tilde{u}\frac{\partial \tilde{u}}{\partial \tilde{x}} + \tilde{v}\frac{\partial \tilde{v}}{\partial \tilde{y}} = \frac{\partial}{\partial \tilde{y}}\left(\tilde{\nu}\frac{\partial \tilde{u}}{\partial \tilde{y}}\right) \tag{2}$$

The x-velocity is u and the y-velocity is v. In these equations, the tildes represent dimensional quantities. The equations can be made nondimensional with the following substitutions:

$$u = \frac{\tilde{u}}{\tilde{U}_\infty} \tag{3}$$

$$v = \frac{\tilde{v}\tilde{L}}{\tilde{U}_\infty \tilde{\delta}} \tag{4}$$

$$x = \frac{\tilde{x}}{\tilde{\delta}} \tag{5}$$

$$y = \frac{\tilde{y}}{\tilde{\delta}} \tag{6}$$

$$\nu = \frac{\tilde{\nu}}{\tilde{\nu}_\infty} \tag{7}$$

$$\frac{1}{RD} = \frac{\tilde{L}\tilde{\nu}_\infty}{\tilde{U}_\infty \tilde{\delta}^2} \tag{8}$$

Substituting these equations into 1 and 2 results in the following nondimensional continuity and momentum equations:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{9}$$

$$u\frac{\partial u}{\partial x} + v\frac{\partial v}{\partial y} = \frac{1}{RD}\frac{\partial}{\partial y}\left(\nu\frac{\partial u}{\partial y}\right) \tag{10}$$

These equations are now nondimensional with x, y, u, and v ranging from 0 to 1. These are coupled partial differential equations which are parabolic in the x-direction. Therefore, an initial condition is needed to initialize the velocity field at x = 0. The given boundary layer profile at x = 0 is easily nondimensionalized to yield

$$u = \begin{cases} sin\left(\frac{\pi y}{2}\right) & \text{for } y \leq 1 \\ 1 & \text{for } y > 1 \end{cases} \tag{11}$$

The other imposed boundary conditions are the no slip condition at the wall, the freestream condition at the upper boundary, and the impermeable surface condition at the wall.

$$u(x, 0) = 0 \tag{12}$$

$$u(x, 1) = 1 \tag{13}$$

$$v(x, 0) = 0 \tag{14}$$

With no given profile for v at x = 0, the starting profile was chosen to be zero. With a first-order derivative for u in 9 and a second-order derivative for u in 10, three boundary conditions for u are needed. With one first-order derivative for v appearing in each of the two nondimensional equations, a two boundary conditions for v are needed. With all five boundary conditions stated, the problem is well-posed. It is crucial that the vertical velocity at the upper boundary is not constrained, because over-constraining the solution would lead to instability in the solution scheme.

Numerical Method

A common approach to numerically solving partial differential equations is to use a finite difference method. In this approach, the domain is subdivided into discrete nodes, at which the differential equations are written as discrete algebraic equations. A uniform, collocated grid is the simplest approach, in which the spacing between nodes in each direction is uniform, and the same grid is used to evaluate each equation. For this analysis, finer resolution is desired near the wall, so the high gradients in velocity near the wall can be resolved. For this case, a stretched grid in y was used.

Grid Stretching

To simplify the solution scheme, a coordinate transformation was used which allowed for the non-uniform grid in y to be converted to a uniform grid in another coordinate system, η. To achieve this result, the discrete points in η were defined to be equally spaced, and a stretching function was used to calculate y. To achieve a physically meaningful result, Equations 9 and 10 and the boundary conditions needed to be transformed from x-y space to x-η space. The full transformation of the equations is available in Appendix 2. Writing the coordinate y as a function of *eta*, the chain rule can be invoked to generically write the derivatives of *eta* with respect to y.

$$y = f(\eta) \tag{15}$$

$$\eta_y = \frac{\partial \eta}{\partial y} = \frac{1}{f'(\eta)} \tag{16}$$

$$\eta_{yy} = \frac{\partial}{\partial y}\left[\frac{1}{f'(\eta)}\right] = \frac{-f''(\eta)}{[f'(\eta)]^2} \tag{17}$$

The nondimensional continuity and x-momentum equations were transformed from x-y space into x-η space, resulting in

$$\frac{\partial u}{\partial x} + \eta_y \frac{\partial v}{\partial \eta} = 0 \tag{18}$$

$$u\frac{\partial u}{\partial x} + \left[v\eta_y - \frac{\nu\eta_y\eta_{yy}}{RD}\right]\frac{\partial u}{\partial \eta} + \frac{-\nu\eta_y^2}{RD}\left(\frac{\partial^2 u}{\partial^2 \eta}\right) = 0 \tag{19}$$

If we choose the stretching function to be

$$y = f(\eta) = y_{max}\frac{\sinh(s\eta)}{\sinh(s)} \tag{20}$$

where s is a constant known as the stretching factor, the derivatives can be determined analytically and substituted into 18 and 19. The derivatives are

$$\eta_y = \frac{\sinh(s)}{sy_{max}\cosh(s\eta)} \tag{21}$$

$$\eta_{yy} = \frac{\sinh(s\eta)\sinh^2(s)}{sy_{max}^2\cosh^3(s\eta)} \tag{22}$$

Discretization Schemes and Stability

A fourth-order central difference scheme was used at internal nodes in the η direction to solve for u at the i+1 position in x. A Crank-Nicolson (second-order) discretization in x was used. A schematic of the computational "stencil" used for internal nodes to solve for u is shown in Figure 1. Near the boundaries, where not enough information was known to use a fourth-order scheme, a second-order central difference scheme was used in η. First the momentum equation was discretized and solved to calculate u at the i+1 location, and then the continuity equation was discretized and solved to calculate v at the i+1 location.

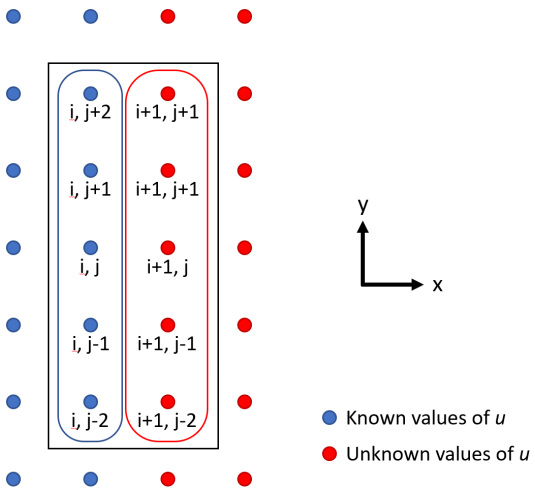


Figure 1: Computational stencil used to solve for u at a specified i+1 location

Using a fourth-order central difference centered at node (i, j) has no stability concerns with the momentum equation, which includes second-order derivatives with respect to η. The main diagonals of the matrix are guaranteed to be nonzero. However, using a fourth-order central difference scheme centered at node (i, j) is unconditionally unstable when the continuity equation is discretized. The main diagonal of the continuity equation is zero because there are no second-order derivatives in the continuity equation, and attempts to invert the matrix result in the solution approaching infinity. As a result, only schemes that were not centered at (i, j) could be used when solving for v.

For internal nodes, a fourth-order scheme centered about (i, j - 1/2) was used to generate the coefficient matrix. For the nodes near boundaries, where a fourth-order scheme could not be used, third-order schemes were used instead. The full derivation of finite difference equations can be found in Appendix 2.

LU Decomposition Matrix Inversion Algorithm

To invert the matrix, an LU decomposition scheme was written specifically for a pentadiagonal matrix. The LU decomposition algorithm was modified to reduce the number of floating-point operations, because the input matrix is known to be pentadiagonal. The full derivation of LU decomposition will not be shown, but the process will be summarized here. In this method, the matrix A is set equal to the matrix product of L, a lower triangular matrix, and U, an upper triangular matrix. Because triangular matrices are much easier to invert than a dense matrix, the LU decomposition is a direct method that allows the exact matrix inverse to be calculated. In general, the process is as follows:

$$\underline{\underline{L}}\,\underline{\underline{U}}\,\underline{u_{i+1}} = \underline{b} \Rightarrow \underline{\underline{U}}\,\underline{u_{i+1}} = \underline{\underline{L}}^{-1}\underline{b} \Rightarrow \underline{u_{i+1}} = \underline{\underline{U}}^{-1}(\underline{\underline{L}}^{-1}\underline{b}) \tag{23}$$

This process is used to solve for the u_{i+1} values in each x-step. The LU decomposition algorithm was implemented in the Python code.

Results

Using a stretching factor of $s = 5$, the finite difference code was executed to calculate the velocity field for the laminar boundary layer. The velocity profiles at various locations downstream are shown in in Figure 2. The Blasius solution as tabulated [1] was made dimensional using physical quantities of the problem. The finite difference solution begins with a starting profile, while the Blasius solution begins at a single point, so an appropriate setback needed to be calculated for the Blasius solution in order to show comparable profiles. Using the given initial boundary layer thickness of 0.005 m, the equation describing boundary layer growth

$$\delta = 5.0\sqrt{\frac{\nu x}{U_\infty}} \tag{24}$$

was rearranged to calculate x as a function of δ. The calculated dimensional setback for the Blasius boundary layer was 27.65 m.

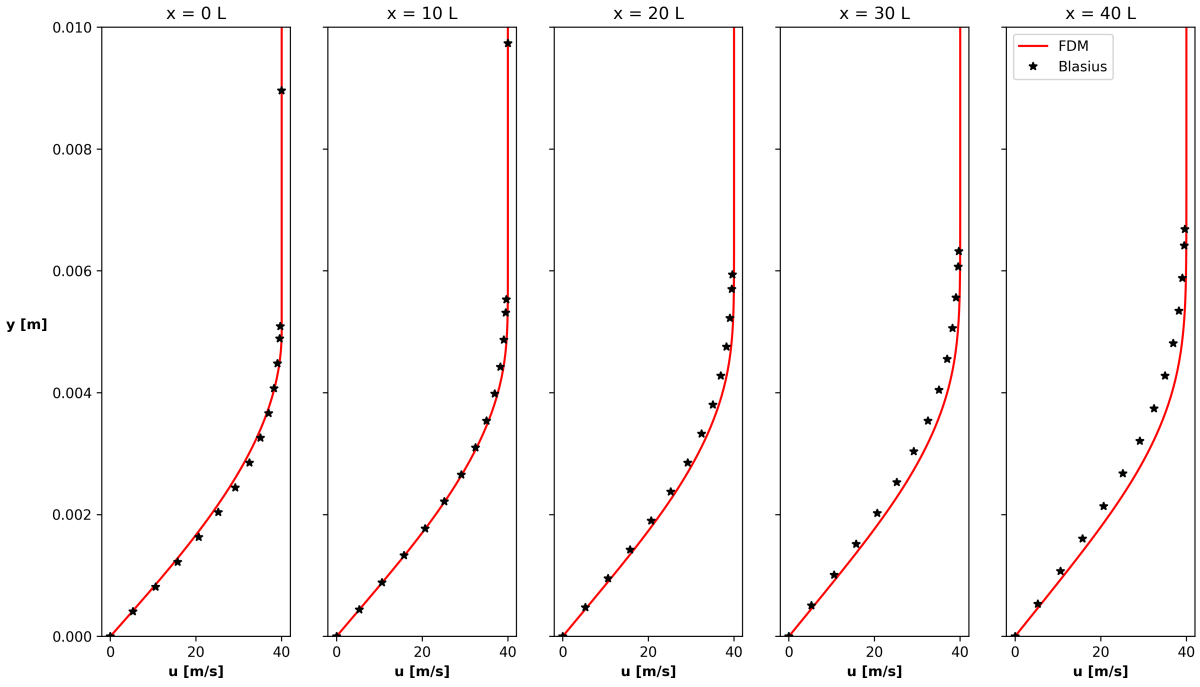


Figure 2: Velocity profiles as computed by the finite difference method and compared to the Blasius solution

The profiles shown in Figure 2 are generally in agreement with the shape defined by the Blasius solution. As the distance downstream increases, the FDM result does a poorer job of matching the Blasius solution. The Blasius solution has a faster growth rate downstream than the FDM result. Improved results could be achieved if an iterative solver was utilized and if the mesh was refined. As the number of nodes approaches infinity, we would expect the solution to converge on the exact solution.

Effect of Stretching Factor

The stretching factor s as used in this stretching function defines the concentration of nodes near the wall. Higher numbers of s correspond to tighter spacing near the wall, while likewise lower numbers of s correspond to larger spacing near the wall. Three stretching factors were compared: s = 1, s = 5, and s = 10. The velocity profiles at x = 40 L are shown in Figure 3. As expected, the s = 1 case has the fewest nodes near the wall, while the s = 5 and s = 10 cases have more nodes near the wall. There is little discernible difference between the three curves. With 151 nodes across the vertical direction of the domain, the stretching may not be necessary to resolve the gradients near the wall. If fewer nodes were used, the stretching would be of more importance to ensure that enough nodes were packed near the wall to resolve the strong gradients near the wall.

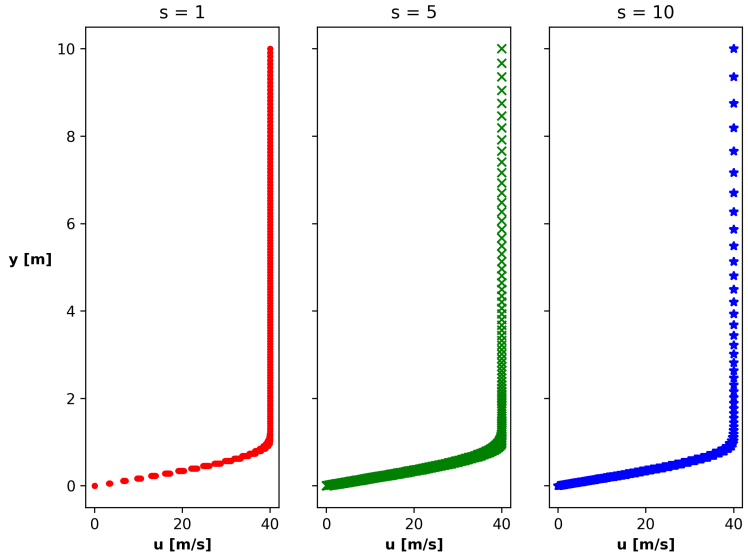


Figure 3: Effect of varying stretching factor s on final velocity profile at x = 40 L

Conclusion

A finite-difference code was written in Python to solve the laminar boundary layer equations for a flat plate. The numerical method used a fourth-order central difference approach in y with the second-order Crank-Nicolson algorithm to march in x, using LU decomposition to solve for a column of x-velocity values simultaneously at each x-step. The continuity equation was solved to calculate the vertical velocity values. With 151 nodes across the domain and a stretching factor of 5, velocity profiles were calculated at five locations along the plate. The profiles were found to match the Blasius solution, while the boundary layer growth rate was not matched exactly. An iterative solution scheme would result in a more accurate growth rate and would be expected to match the Blasius solution more accurately.

References

[1] IIT Kanpur. Chapter 9 : Laminar Boundary Layers. `http://www.nptel.ac.in/courses/112104118/lecture-28/28-8_values_velocity_profile.htm`, 2009. [Online; accessed 15-March-2018].

Appendix 1: Python Code

```
# -*- coding: utf-8 -*-
"""
@author: Brian Knisely

AERSP/ME 525: Turbulence and Applications to CFD: RANS
Computer Project Number 2

The purpose of this code is to use finite difference to solve the laminar
boundary layer equations for a 2D flat plate and compare to the Blasius
solution.

The PDEs with dimensional variables (denoted with ~) are:
    du~/dx~ + dv~/dy~ = 0
    u~ du~/dx~ + v~ du~/dy~ = d/dy~(nu~ du~/dy~)

Note: all derivatives are partial derivatives

The equations are nondimensionalized with:
    x = x~/L~
    y = y~/delta~
    u = u~/Uinf~
    v = v~L~/(Uinf~delta~)
    nu = nu~/nu_inf~
    RD = L~nu_inf~/(Uinf~(delta~)^2)

The resulting nondimensional equations are:
    du/dx + dv/dy = 0
    u du/dx + v du/dy = 1/RD d/dy(nu du/dy)

The boundary conditions (BCs) in nondimensional form are:
    u(x, 0) = 0  (no-slip condition)
    u(x, yMax) = 1  (velocity is equal to freestream at top edge of domain)
    u(0, y <= 1) = sin(pi*y/2)  (starting profile)
    u(0, y > 1) = 1  (starting profile)
    v(x, 0) = 0  (impermeable wall condition)

The Crank-Nicolson Algorithm is to be used to march in the x-direction, using
a uniform grid and central differencing scheme that is fourth-order in y. An LU
decompsition algorithm is used to solve the pentadiagonal matrix for u-values
at each x-step. After computing the u-values, the continuity equation is solved
for the v-values at that step. A fourth-order noncentered scheme is used to
generate the coefficient matrix for v to avoid singularities when inverting the
matrix. The solution is compared to the Blasius solution for a flat plate
boundary layer. The effect of stretching factor is investigated.
"""

# Import packages for arrays, plotting, timing, and file I/O
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt
from math import sin, pi, sinh, cosh, asinh, sqrt
import csv
```

```
import os


def pentaLU(A, b):  # Define LU Decomposition function to solve A*x = b for x
    """
    Function to solve A*x = b for a given a pentadiagonal 2D array A and right-
    hand side 1D array, b. Dims of A must be at least 5 x 5.
    """

    Ny = np.shape(A)[0] + 2  # Extract dims of input matrix A
    yVec = np.zeros([1, Ny - 2])  # initialize y vector for LU decomposition
    xVec = np.zeros([1, Ny - 2])  # initialize x vector for LU decomposition

    lo = np.eye(np.shape(A)[0])  # initialize lower array with identity
    up = np.zeros([np.shape(A)[0], np.shape(A)[0]])  # initialize upper

    # Assign values at beginning of matrix
    # 0th row
    up[0, 0:3] = A[0, 0:3]
    # 1st row
    lo[1, 0] = A[1, 0]/up[0, 0]
    up[1, 3] = A[1, 3]
    up[1, 2] = A[1, 2] - lo[1, 0]*up[0, 2]
    up[1, 1] = A[1, 1] - lo[1, 0]*up[0, 1]

    # Assign values in middle of matrix
    for n in range(2, np.shape(A)[0]-2):
        up[n, n+2] = A[n, n+2]
        lo[n, n-2] = A[n, n-2]/up[n-2, n-2]
        lo[n, n-1] = (A[n, n-1] - lo[n, n-2]*up[n-2, n-1])/up[n-1, n-1]
        up[n, n+1] = A[n, n+1] - lo[n, n-1]*up[n-1, n+1]
        up[n, n] = A[n, n] - lo[n, n-2]*up[n-2, n] - lo[n, n-1]*up[n-1, n]

    # Assign values at end of matrix
    # Second-to-last row
    lo[-2, -4] = A[-2, -4]/up[-4, -4]
    lo[-2, -3] = (A[-2, -3] - lo[-2, -4]*up[-4, -3])/up[-3, -3]
    up[-2, -1] = A[-2, -1] - lo[-2, -3]*up[-3, -1]
    up[-2, -2] = A[-2, -2] - lo[-2, -4]*up[-4, -2] - lo[-2, -3]*up[-3, -2]
    # Last row
    lo[-1, -3] = A[-1, -3]/up[-3, -3]
    lo[-1, -2] = (A[-1, -2] - lo[-1, -3]*up[-3, -2])/up[-2, -2]
    up[-1, -1] = A[-1, -1] - lo[-1, -3]*up[-3, -1] - lo[-1, -2]*up[-2, -1]
    # LU Decomposition is complete at this point

    # Now solve for y vector in lo*yVec = b by forward substitution
    yVec[0, 0] = b[0]  # Calculate 0th element of y vector
    yVec[0, 1] = b[1] - lo[1, 0]*yVec[0, 0]  # Compute 1st element of y vector
    for n in range(2, np.shape(yVec)[1]):
        # Compute nth value of y vector
        yVec[0, n] = b[n] - (lo[n, n-2]*yVec[0, n-2] + lo[n, n-1]*yVec[0, n-1])

    # Solve for x vector in up*xVec = yVec
    xVec[0, -1] = 1/up[-1, -1] * yVec[0, -1]  # Calculate last element of xVec
```

```
    # Calculate second-to-last element of x vector
    xVec[0, -2] = 1/up[-2, -2] * (yVec[0, -2] - up[-2, -1]*xVec[0, -1])
    for n in np.arange(np.shape(xVec)[1]-3, -1, -1):
        # Step backwards from end to beginning to compute x vector
        xVec[0, n] = 1/up[n, n] * (yVec[0, n] - (up[n, n+1]*xVec[0, n+1] +
                                                 up[n, n+2]*xVec[0, n+2]))
    # Output value of x vector from function
    return xVec


# %% Main function
def main(Nx, Ny, method, s):  # Define main function to set up grid and matrix
    #                           and march in x-direction using Crank-Nicolson
    #                           algorithm

    #    Inputs:  Nx = number of nodes in x-direction
    #             Ny = number of nodes in y-direction
    #             method = "lu" or "inv" for matrix inversion
    #             s = stretching factor

    # Inputs for testing
    # Nx = 41
    # Ny = 151
    # method = 'lu'
    # s = 5

    # Define bounds of computational domain
    xMax = 20  # Dimensional distance in m
    yMax = 10  # Scaled by BL height

    # Define given dimensional quantities
    nuInfDim = 1.5e-6  # Dimensional freestream viscosity in m^2/s
    uInfDim = 40  # Dimensional freestream velocity in m/s
    LDim = 0.5  # Length of plate in m
    deltaDim = 0.005  # Initial BL thickness in m

    # Calculate derived nondimensional quantity
    RD = uInfDim*deltaDim**2/(LDim*nuInfDim)

    # Make linear-spaced 1D array of x-values from 0 to 1 with Nx elements
    x = np.linspace(0, xMax, Nx)

    # Make linear-spaced 1D array of eta-values from 0 to 1 with Ny elements
    eta = np.linspace(0, 1, Ny)

    # %% Compute values of y based on eta and ymax
    y = [yMax*sinh(s*et)/sinh(s) for et in eta]

    # Evaluate values of fPrime and fDoublePrime
    fPrime = [s*yMax*cosh(s*et)/sinh(s) for et in eta]
    fDoublePrime = [s**2*yMax*sinh(s*et)/sinh(s) for et in eta]
    # Determine etaY and etaYY
    etaY = [1/fP for fP in fPrime]
    etaYY = [-fDoublePrime[n]/(fPrime[n]**3) for n in range(len(fPrime))]
```

```
# Calculate spacings dx and de; constant with the uniform x-eta grid
dx = x[1] - x[0]
de = eta[1] - eta[0]

# Initialize u- and v-arrays: 2D arrays of zeros
# of dimension [Nx columns] by [Ny rows]
u = np.zeros([Ny, Nx])
v = np.zeros([Ny, Nx])
# u is a 2D array of nondimensional x-velocities at all spatial locations
# v is a 2D array of nondimensional y-velocities at all spatial locations

# %% Apply boundary conditions to u matrix

# Set u(x, 0) to be 0 from no-slip boundary condition
u[0, :] = 0  # Set 0th row, all columns to be 0

# Set u(x, 1) to be 1 from Dirichlet boundary condition
u[-1, :] = 1  # Set last row, all columns to be 1

# Set u(0, eta) to starting profile
etaY1 = asinh(1*sinh(s)/yMax)/s  # Determine eta value corresponding to y=1
for n in range(len(y)):
    # u(0, y <= 1) = sin(pi*y/2)
    if eta[n] <= etaY1:
        u[n, 0] = sin(pi*y[n]/2)
    # u(0, y > 1) = 1
    elif y[n] > etaY1:
        u[n, 0] = 1

# %% Apply boundary condition to v matrix
# Set v(x, eta=0) to be 0 from impermeable wall condition
v[0, :] = 0

# %% Loop over each x-step
for i in range(len(x)-1):

    # Set up matrix and RHS "b" matrix (knowns)
    # for Crank-Nicolson algorithm

    # Initialize matrix A for equation [[A]]*[u] = [b]
    A = np.zeros([Ny - 2, Ny - 2])
    b = np.zeros([Ny - 2, 1])
    # Y-dimension reduced by two because u(x, 0) and u(x, 1)
    # are known already

    # %% Use 2nd-Order-Accurate scheme for first interior nodes

    # at j = 2 (near bottom border of domain)
    # Calculate values of A(eta) and B(eta) at j = 2
    Ae = v[1, i]*etaY[1] - etaY[1]*etaYY[1]/RD
    Be = -etaY[1]**2/RD

    # Populate coefficient matrix
```

```
A[0, 0] = u[1, i]/dx - Be/de**2  # Assign value in location [1, 1]
A[0, 1] = Ae/(4*de) + Be/(2*de**2)  # Assign value in matrix location
b[0] = (u[0, i]*(Ae/(4*de) - Be/(2*de**2)) + u[1, i]*Be/de**2
        + u[1, i]**2/dx + u[2, i]*(-Ae/(4*de) - Be/(2*de**2))
        + u[0, i+1]*(Ae/(4*de) - Be/(2*de**2)))

# %% at j = Ny-1 (near top border of domain)
# Calculate values of A(eta) and B(eta) at j = Ny-1
Ae = v[-2, i]*etaY[-2] - etaY[-2]*etaYY[-2]/RD
Be = -etaY[-2]**2/RD

# Populate coefficient matrix
A[-1, -1] = u[-2, i]/dx - Be/de**2  # Assign value to last diag element
A[-1, -2] = -Ae/(4*de) + Be/(2*de**2)  # Assign value to left of diag
b[-1] = (u[-3, i]*(Ae/(4*de) - Be/(2*de**2)) + u[-2, i]*Be/de**2
         + u[-2, i]**2/dx + u[-1, i]*(-Ae/(4*de) - Be/(2*de**2))
         + u[-1, i+1]*(-Ae/(4*de) - Be/(2*de**2)))

# %% Use 4th-Order-Accurate scheme for j = 3 to j = Ny-2

# Second internal node (j = 3)
# Calculate values of A(eta) and B(eta) at j
Ae = v[2, i]*etaY[2] - etaY[2]*etaYY[2]/RD
Be = -etaY[2]**2/RD

# Store coefficients and value for RHS vector b
A[1, 0] = -Ae/(3*de) + 2*Be/(3*de**2)
A[1, 1] = -5*Be/(4*de**2) + u[2, i]/dx
A[1, 2] = Ae/(3*de) + 2*Be/(3*de**2)
A[1, 3] = -Ae/(24*de) - Be/(24*de**2)
b[1] = (u[0, i]*(-Ae/(24*de) + Be/(24*de**2))
        + u[1, i]*(Ae/(3*de) - 2*Be/(3*de**2))
        + u[2, i]*(5*Be/(4*de**2) + u[2, i]/dx)
        + u[3, i]*(-Ae/(3*de) - 2*Be/(3*de**2))
        + u[4, i]*(Ae/(24*de) + Be/(24*de**2))
        + u[0, i+1]*(-Ae/(24*de) + Be/(24*de**2)))

# %% Loop over internal nodes to compute and store coefficients
for j in range(2, Ny-4):
    # Calculate values of A(eta) and B(eta) at j
    Ae = v[j, i]*etaY[j] - etaY[j]*etaYY[j]/RD
    Be = -etaY[j]**2/RD

    A[j, j-2] = Ae/(24*de) - Be/(24*de**2)
    A[j, j-1] = -Ae/(3*de) + 2*Be/(3*de**2)
    A[j, j] = -5*Be/(4*de**2) + u[j, i]/dx
    A[j, j+1] = Ae/(3*de) + 2*Be/(3*de**2)
    A[j, j+2] = -Ae/(24*de) - Be/(24*de**2)
    b[j] = (u[j-2, i]*(-Ae/(24*de) + Be/(24*de**2))
            + u[j-1, i]*(Ae/(3*de) - 2*Be/(3*de**2))
            + u[j, i]*(5*Be/(4*de**2) + u[j, i]/dx)
            + u[j+1, i]*(-Ae/(3*de) - 2*Be/(3*de**2))
            + u[j+2, i]*(Ae/(24*de) + Be/(24*de**2)))
```

```
# %% Second-to-last internal node (j = Ny-2)
# Calculate values of A(eta) and B(eta) at j
Ae = v[-3, i]*etaY[-3] - etaY[-3]*etaYY[-3]/RD
Be = -etaY[-3]**2/RD

# Store coefficients and value for RHS vector b
A[-2, -4] = Ae/(24*de) - Be/(24*de**2)
A[-2, -3] = -Ae/(3*de) + 2*Be/(3*de**2)
A[-2, -2] = -5*Be/(4*de**2) + u[-3, i]/dx
A[-2, -1] = Ae/(3*de) + 2*Be/(3*de**2)
b[-2] = (u[-5, i]*(-Ae/(24*de) + Be/(24*de**2))
         + u[-4, i]*(Ae/(3*de) - 2*Be/(3*de**2))
         + u[-3, i]*(5*Be/(4*de**2) + u[-3, i]/dx)
         + u[-2, i]*(-Ae/(3*de) - 2*Be/(3*de**2))
         + u[-1, i]*(Ae/(24*de) + Be/(24*de**2))
         + u[-1, i+1]*(Ae/(24*de) + Be/(24*de**2)))

# Perform matrix inversion to solve for u
if method == 'lu':  # if input was for LU decomposition
    u[1:-1, i+1] = pentaLU(A, b)  # call the pentaLU solver

if method == 'inv':  # if input was for built-in inv (for testing)
    u[1:-1, i+1] = (inv(A)@b).transpose()  # solve by inverting matrix

# %% u at x+1 has been solved for, now use continuity to solve for v

# Initialize matrix A and vector b for equation [[A]]*[v] = [b]
A = np.zeros([Ny - 1, Ny - 1])
b = np.zeros([Ny - 1, 1])

# Use third order FDS in eta, 2nd order Crank Nicolson in x
# Use biased 3rd order scheme for node adjacent to bottom boundary
A[0, 0] = -etaY[1]/(4*de)
A[0, 1] = etaY[1]/(2*de)
A[0, 2] = -etaY[1]/(12*de)
b[0] = ((u[j, i] - u[j, i+1])/dx
        - etaY[1]/12*(-v[3, i]+6*v[2, i]-3*v[1, i])/de)

# One up from bottom boundary - now use 4th order scheme about j - 1/2
A[1, 0] = -9*etaY[2]/(16*de)
A[1, 1] = 9*etaY[2]/(16*de)
A[1, 2] = -etaY[2]/(48*de)
b[1] = ((u[2, i]-u[2, i+1])/dx
        - (etaY[2]/48)*(27*v[1, i]-27*v[2, i]+v[3, i])/de)

# Loop over internal nodes - still used 4th order scheme about j - 1/2
for j in range(2, Ny-2):
    A[j, j-2] = etaY[j+1]/(48*de)
    A[j, j-1] = -9*etaY[j+1]/(16*de)
    A[j, j] = 9*etaY[j+1]/(16*de)
    A[j, j+1] = -etaY[j+1]/(48*de)
    b[j] = ((u[j+1, i]-u[j+1, i+1])/dx - (etaY[j+1]/48)
            * (-v[j-2, i]+27*v[j-1, i]-27*v[j, i]+v[j+1, i])/de)
```

```
        # Finally at top boundary use 3rd order scheme about Ny - 1/2
        A[-1, -3] = 1
        A[-1, -2] = -4
        A[-1, -1] = 3
        b[-1] = 0

    # Perform matrix inversion to solve for v
    if method == 'lu':  # if input was for LU decomposition
        v[1:, i+1] = pentaLU(A, b)  # call the pentaLU solver
    if method == 'inv':  # if input was for built-in inv (for testing)
        v[1:, i+1] = (inv(A)@b).transpose()  # solve by inverting matrix

    # output is the u-matrix
    return u, y


# %% Plot results
# Define function to plot versus Blasius
def plotsVsBlasius(u, y):

    # Yes! It is possible to read text files in Python!

    # Read CSV containing Blasius solution and store variables
    with open('readfiles//Blasius.csv', newline='') as csvfile:
        reader = csv.reader(csvfile, delimiter=' ')
        data = [row[0].split(',') for row in reader]
        etaB = [eval(val) for val in data[0]]  # Store similarity variable list
        g = [eval(val) for val in data[2]]  # Store normalized velocity list

    xLoc = [0, 5, 10, 15, 20]  # Dimensional x-locations (m)
    xMax = 20  # Dimensional distance in m
    # Define given dimensional quantities
    nuInf = 1.5e-6  # Dimensional freestream viscosity in m^2/s
    uInf = 40  # Dimensional freestream velocity in m/s
    deltaDim = 0.005  # Initial BL thickness in m
    Nx = len(u[0, :])  # number of grid points in x-direction

    delta99 = []  # initialize to store calculated 99% BL thickness values
    xInds = []  # initialize to store x-indices corresponding to xLocs above
    setbacks = []  # initialize array to store estimates for x~
    # Loop through each x-location and solve for 99% BL thickness and estimate
    # x~ based on each BL thickness
    mrk = 0  # marker to count how many while loops
    for xi in xLoc:
        xInd = round(xi/xMax*(Nx-1))  # find index corresponding to that xLoc
        xInds.append(xInd)
        yInd = 0  # store index corresponding to BL height
        while True:
            if u[yInd, xInd] > 0.99:  # if 99% velocity satisfied
                delta99.append(deltaDim*y[yInd])  # Store BL height (m)
                setbacks.append(delta99[-1]**2*uInf/nuInf
                                / 4.91**2-xLoc[mrk])
                mrk += 1
                break  # stop while loop
```

```
            yInd += 1

    # Dimensionalize velocity from Blasius solution
    uB = [gi*uInf for gi in g]

    # Dimensionalize y-coordinates for numerical solution
    yDim = [deltaDim*yi for yi in y]

    # Initialize figure with 5 subplots/axes
    axs = ['ax' + str(n) for n in range(1, 6)]
    fig, axs = plt.subplots(figsize=(8, 6), ncols=5,
                            sharey='row')

    # Loop through all desired positions and plot numerical result vs Blasius
    for ii in range(5):
        xCorr = xLoc[ii] + 0.005**2 * uInf / nuInf / 4.91**2
        yB = [sqrt(xCorr)*etaBi/sqrt(uInf/nuInf) for etaBi in etaB]
        ax = axs[ii]
        ax.plot(uInf*u[:, xInds[ii]], yDim, 'r', uB, yB, 'k*')
        ax.set_xlabel('u [m/s]', fontweight='bold')
        ax.set_title('x = {0:0.0f} L'.format(xLoc[ii]/0.5))
        if ii == 0:
            ax.set_ylabel('y [m]        ', fontweight='bold', rotation=0)

    plt.legend(['FDM', 'Blasius'])  # add legend
    plt.ylim(ymin=0, ymax=0.01)  # axes limits
    plt.savefig(os.getcwd()+"\\figures\\profiles.png", dpi=320,
                bbox_inches='tight')  # save figure to file
    plt.close()  # close figure

    # Estimate initial position using BL thickness at beginning
    xInitial = 0.005**2*uInf/nuInf/4.91**2
    # Print result to console
    print('Estimated initial value of x˜ is {0:0.2f} m'.format(xInitial))
    # Output value of xInitial
    return xInitial


# %% Calculate displacement thickness and momentum thickness
def thicc(u, y):  # function to compute displacement thickness, momentum
    #               thickness, and shape factor
    nuInf = 1.5e-6  # Dimensional freestream viscosity in m^2/s
    uInf = 40  # Dimensional freestream velocity in m/s
    deltaDim = 0.005  # Initial BL thickness in m
    xInitial = 0.005**2*uInf/nuInf/4.91**2
    Ny = len(y)  # Number of grid points in y-direction
    deltaStar = 0  # initialize displacement thickness
    thetaStar = 0  # initialize momentum thickness
    for i in range(1, Ny):
        uUi = ((1 - u[i, -1]) + (1 - u[i-1, -1]))/2  # find velocity deficit
        dy = (y[i] - y[i-1])*deltaDim  # find dimensional y-height difference
        deltaStar += (uUi)*dy  # add ith contribution
        thetaStar += uUi*(1 - uUi)*dy  # add ith contribution
```

```
    # Calculate Blasius estimates from solution
    deltaStarBlasius = 1.702*sqrt(nuInf*(20+xInitial)/uInf)
    thetaStarBlasius = 0.664*sqrt(nuInf*(20+xInitial)/uInf)

    # Print results to console
    print('Displacement thickness at x = 40 L is {0:0.6f} m'.format(deltaStar))
    print('Estimated Blasius displacement thickness at x = 40 L is {0:0.6f} m'
          .format(deltaStarBlasius))
    print('Momentum thickness at x = 40 L is {0:0.6f} m'.format(thetaStar))
    print('Estimated Blasius momentum thickness at x = 40 L is {0:0.6f} m'
          .format(thetaStarBlasius))


# %% Define function to show effect of stretching on profile
def stretchEffect(Nx, Ny, stretch):
    uInf = 40  # freestream velocity
    ls = ['r.', 'gx', 'b*', 'kd', 'mp', 'yh']  # line styles
    # uuu = np.zeros([Ny, Nx, len(stretch)])  # initialize array to compare u
    axs = ['ax' + str(n) for n in range(1, 1+len(stretch))]  # name axes
    # set up subplots on same figure
    fig, axs = plt.subplots(figsize=(8, 6), ncols=len(stretch), sharey='row')
    # Loop for each stretching factor
    for s in range(len(stretch)):
        u, y = main(Nx, Ny, 'lu', stretch[s])  # get u and y from main
        ax = axs[s]  # set axis number
        ax.plot(u*uInf, y, ls[s])  # plot y vs u
        ax.set_xlabel('u [m/s]', fontweight='bold')  # label x axis
        ax.set_title('s = {0:0.0f}'.format(stretch[s]))  # label title
        if s == 0:  # set y-label for the leftmost subplot
                ax.set_ylabel('y [m]        ', fontweight='bold', rotation=0)
    #    plt.savefig(os.getcwd()+"\\figures\\stretching.png", dpi=320,
    #                bbox_inches='tight')  # save figure to file


# %% Run functions in order
Nx = 41  # number of nodes in x-direction
Ny = 151  # number of nodes in y-direction
stretching = 5  # stretching factor
u, y = main(Nx, Ny, 'lu', stretching)

# Plot results compared to Blasius solution and calculate initial x~ position
plotsVsBlasius(u, y)

# Calculate thicknesses at end and compare to Blasius
thicc(u, y)

# %% Run function to see effect of stretching
stretch = [1, 5, 10]
stretchEffect(Nx, Ny, stretch)
```

Appendix 2: Derivation of Nondimensional Equations and Finite Difference Discretization

$$\frac{\partial \tilde{u}}{\partial \tilde{x}} + \frac{\partial \tilde{v}}{\partial \tilde{y}} = 0$$ Dimensional Continuity

$$\tilde{u}\frac{\partial \tilde{u}}{\partial \tilde{x}} + \tilde{v}\frac{\partial \tilde{u}}{\partial \tilde{y}} = \frac{\partial}{\partial \tilde{y}}\left(\tilde{\nu}\frac{\partial \tilde{u}}{\partial \tilde{y}}\right)$$ Dimensional X-momentum

Nondimensionalize $x = \frac{\tilde{x}}{\tilde{L}}$ $y = \frac{\tilde{y}}{\tilde{\delta}}$ $u = \frac{\tilde{u}}{\tilde{U}_\infty}$ $v = \frac{\tilde{v}\tilde{L}}{\tilde{U}_\infty \tilde{\delta}}$

$$\nu = \frac{\tilde{\nu}}{\tilde{\nu}_\infty} \qquad \frac{1}{RD} = \frac{\tilde{L}\tilde{\nu}_\infty}{\tilde{U}_\infty \tilde{\delta}^2}$$

$$\frac{\partial \tilde{u}}{\partial \tilde{x}} = \frac{\partial (u\tilde{U}_\infty)}{\partial (x\tilde{L})} = \frac{\tilde{U}_\infty}{\tilde{L}}\frac{\partial u}{\partial x}$$

$$\frac{\partial \tilde{v}}{\partial \tilde{y}} = \frac{\partial (v\tilde{U}_\infty \tilde{\delta}/\tilde{L})}{\partial (y\tilde{\delta})} = \frac{\tilde{U}_\infty}{\tilde{L}}\frac{\partial v}{\partial y}$$

Continuity $\rightarrow$ $\frac{\tilde{U}_\infty}{\tilde{L}}\frac{\partial u}{\partial x} + \frac{\tilde{U}_\infty}{\tilde{L}}\frac{\partial v}{\partial y} =$ $\boxed{\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0}$ Nondimensional Continuity

Momentum $\rightarrow$ $u\tilde{U}_\infty \frac{\tilde{U}_\infty}{\tilde{L}}\frac{\partial u}{\partial x} + \frac{v\tilde{U}_\infty \tilde{\delta}}{\tilde{L}}\frac{\partial (u\tilde{U}_\infty)}{\partial (y\tilde{\delta})} = \frac{\partial}{\partial (y\tilde{\delta})}\left(\nu\tilde{\nu}_\infty \frac{\partial (u\tilde{U}_\infty)}{\partial (y\tilde{\delta})}\right)$

$$\frac{\tilde{U}_\infty^2}{\tilde{L}}\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) = \frac{\tilde{\nu}_\infty \tilde{U}_\infty}{\tilde{\delta}^2}\frac{\partial}{\partial y}\left(\nu\frac{\partial u}{\partial y}\right)$$

$$\boxed{u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \frac{1}{RD}\frac{\partial}{\partial y}\left(\nu\frac{\partial u}{\partial y}\right)}$$ Nondimensional X-momentum

Stretch the grid near $y=0$

$$y = f(\eta) \qquad dy = f'(\eta)\, d\eta$$

$$\frac{d\eta}{dy} = \eta_y = \frac{1}{f'(\eta)} \qquad \frac{\partial}{\partial y} = \eta_y \frac{\partial}{\partial \eta} \qquad \frac{\partial^2}{\partial y^2} = \frac{\partial}{\partial y}\left(\eta_y \frac{\partial}{\partial \eta}\right) = \eta_{yy}\frac{\partial}{\partial \eta} + (\eta_y)^2 \frac{\partial^2}{\partial \eta^2}$$

$$\eta_{yy} = \frac{\partial}{\partial y}\left(\frac{1}{f'(\eta)}\right) = \frac{-f''}{(f')^3}$$

We must transform the PDEs from x-y space to x-η space

Nondimensional Continuity in (x,η)

$$\boxed{\frac{\partial u}{\partial x} + \eta_y \frac{\partial v}{\partial \eta} = 0}$$

X-momentum

$$u\frac{\partial u}{\partial x} + v\eta_y \frac{\partial u}{\partial \eta} = \frac{1}{RD}\eta_y \frac{\partial}{\partial \eta}\left(\nu \eta_y \frac{\partial u}{\partial \eta}\right) = \frac{\nu \eta_y}{RD}\left[\eta_y \frac{\partial^2 u}{\partial \eta^2} + \eta_{yy}\frac{\partial u}{\partial \eta}\right]$$

$$\boxed{u\frac{\partial u}{\partial x} + \underbrace{\left[v\eta_y - \frac{\nu \eta_y \eta_{yy}}{RD}\right]}_{A(\eta)}\frac{\partial u}{\partial \eta} + \underbrace{\left[\frac{-\nu(\eta_y)^2}{RD}\right]}_{B(\eta)}\frac{\partial^2 u}{\partial \eta^2} = 0}$$

Nondimensional X-momentum in (x,η)

If we choose $y = f(\eta) = y_{max}\dfrac{\sinh(s\eta)}{\sinh(s)}$

$$\frac{\partial y}{\partial \eta} = f'(\eta) = s\, y_{max}\frac{\cosh(s\eta)}{\sinh(s)} \quad \text{and} \quad \frac{\partial^2 y}{\partial \eta^2} = f''(\eta) = s^2 y_{max}\frac{\sinh(s\eta)}{\sinh(s)}$$

$$\hookrightarrow \eta_y = \frac{\sinh(s)}{s y_{max}\cosh(s\eta)} \qquad \hookrightarrow \eta_{yy} = \frac{\sinh(s\eta)\sinh^2(s)}{s\, y_{max}^2 \cosh^3(s\eta)}$$

2

Discretize Derivatives with FDM

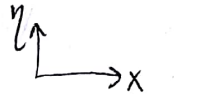
Use 4th-order central difference in η for interior nodes

$$\left.\frac{\partial^2 u}{\partial \eta^2}\right|_{i,j} = \frac{-u_{i,j-2} + 16u_{i,j-1} - 30u_{i,j} + 16u_{i,j+1} - u_{i,j+2}}{12(\Delta\eta)^2} + \mathcal{O}\left((\Delta\eta)^4\right)$$

Use 2nd-order central difference in η for first interior nodes

$$\left.\frac{\partial^2 u}{\partial \eta^2}\right|_{i,j} = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{(\Delta\eta)^2} + \mathcal{O}\left((\Delta\eta)^2\right)$$

Known	solving for
0 $i,j+2$	$i+1,j+2$
0 $i,j+1$	$i+1,j+1$
0 i,j	$i+1,j$
0 $i,j-1$	$i+1,j-1$
0 $i,j-2$	$i+1,j-2$



Use 4th-order central difference for first derivatives for interior nodes

2nd-order near boundaries

$$\left.\frac{\partial u}{\partial \eta}\right|_{i,j} = \frac{u_{i,j-2} - 8u_{i,j-1} + 8u_{i,j+1} - u_{i,j+2}}{12\,\Delta\eta} + \mathcal{O}\left((\Delta\eta)^4\right) = \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta\eta} + \mathcal{O}\left((\Delta\eta)^2\right)$$

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} = \frac{u_{i,j} - u_{i-1,j}}{\Delta x}$$

Interior node - substitute into PDE and use Crank-Nicolson

$$u_{i,j}\left.\frac{\partial u}{\partial x}\right|_{i,j} + A_j\left.\frac{\partial u}{\partial \eta}\right|_{i,j} + B_j\left.\frac{\partial^2 u}{\partial \eta^2}\right|_{i,j} = 0$$

$$u_{i,j}\left[\frac{u_{i+1,j} - u_{i,j}}{\Delta x}\right] + \frac{1}{2}\left\{A_j\left[\frac{u_{i,j-2} - 8u_{i,j-1} + 8u_{i,j+1} - u_{i,j+2}}{12\,\Delta\eta} + \frac{u_{i+1,j-2} - 8u_{i+1,j-1} + 8u_{i+1,j+1} - u_{i+1,j+2}}{12\,\Delta\eta}\right]\right.$$

$$\left. + B_j\left[\frac{-u_{i,j-2} + 16u_{i,j-1} - 30u_{i,j} + 16u_{i,j+1} - u_{i,j+2}}{12(\Delta\eta)^2} + \frac{-u_{i+1,j-2} + 16u_{i+1,j-1} - 30u_{i+1,j} + 16u_{i+1,j+1} - u_{i+1,j+2}}{12(\Delta\eta)^2}\right]\right\}$$

$$= 0$$

3

Rearrange equation to put unknown (i+1) terms on LHS, knowns on RHS

$$\underbrace{\left[\frac{A_j}{24\Delta\eta} - \frac{B_j}{24(\Delta\eta)^2}\right]}_{a_j} u_{i+1,j-2} + \underbrace{\left[\frac{-A_j}{3\Delta\eta} + \frac{2B_j}{3(\Delta\eta)^2}\right]}_{b_j} u_{i+1,j-1} + \underbrace{\left[\frac{-5B_j}{4(\Delta\eta)^2} + \frac{u_{i,j}}{\Delta X}\right]}_{c_j} u_{i+1,j} + \underbrace{\left[\frac{A_j}{3\Delta\eta} + \frac{2B_j}{3(\Delta\eta)^2}\right]}_{d_j} u_{i+1,j+1} + \underbrace{\left[\frac{-A_j}{24\Delta\eta} - \frac{B_j}{24(\Delta\eta)^2}\right]}_{e_j} u_{i+1,j+2}$$

$$= \underbrace{u_{i,j-2}\left[\frac{-A_j}{24\Delta\eta} + \frac{B_j}{24(\Delta\eta)^2}\right] + u_{i,j-1}\left[\frac{A_j}{3\Delta\eta} - \frac{2B_j}{3(\Delta\eta)^2}\right] + u_{i,j}\left[\frac{5B_j}{4(\Delta\eta)^2}\right] + (u_{i,j})^2\frac{1}{\Delta X} + u_{i,j+1}\left[\frac{-A_j}{3\Delta\eta} - \frac{2B_j}{3(\Delta\eta)^2}\right] + u_{i,j+2}\left[\frac{A_j}{24\Delta\eta} + \frac{B_j}{24(\Delta\eta)^2}\right]}_{g_j}$$

Assemble into matrix for x-momentum, internal nodes

$$\begin{bmatrix} \ddots & & & & & \\ a_j & b_j & c_j & d_j & e_j & \\ & a_{j+1} & b_{j+1} & c_{j+1} & d_{j+1} & e_{j+1} \\ & & & & & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ u_{i+1,j-2} \\ u_{i+1,j-1} \\ u_{i+1,j} \\ u_{i+1,j+1} \\ u_{i+1,j+2} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ g_j \\ \vdots \end{bmatrix}$$

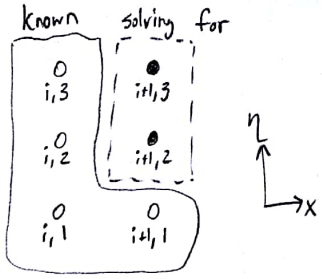
Valid for $j = 4$ to $j = N-3$

For $j = 3$ and $j = N-2$, one term from LHS must be moved to RHS

Next, boundary conditions will be used for $j = 2$ and $j = N-1$

Node adjacent to lower boundary $(j = 2)$

Use 2nd-order central difference in η, Crank-Nicolson in x



$$u_{i,2}\left.\frac{\partial u}{\partial x}\right|_{i,2} + A_2\left.\frac{\partial u}{\partial \eta}\right|_{i,2} + B_2\left.\frac{\partial^2 u}{\partial \eta^2}\right|_{i,2} = 0$$

$$u_{i,2}\left[\frac{u_{i+1,2} - u_{i,2}}{\Delta x}\right] + \frac{1}{2}\left\{A_2\left[\frac{u_{i,3} - u_{i,1}}{2\Delta\eta} + \frac{u_{i+1,3} - u_{i+1,1}}{2\Delta\eta}\right] + B_2\left[\frac{u_{i,1} - 2u_{i,2} + u_{i,3}}{(\Delta\eta)^2} + \frac{u_{i+1,1} - 2u_{i+1,2} + u_{i+1,3}}{(\Delta\eta)^2}\right]\right\} = 0$$

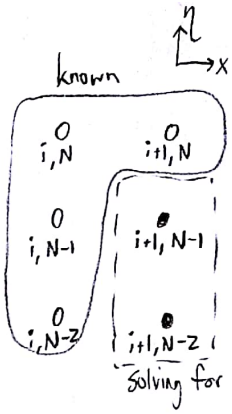
Rearrange with $u_{i+1,2}$ and $u_{i+1,3}$ on LHS, all known terms on RHS

$$\underbrace{\left[\frac{u_{i,2}}{\Delta x} - \frac{B_2}{(\Delta\eta)^2}\right]}_{c_2} u_{i+1,2} + \underbrace{\left[\frac{A_2}{4\Delta\eta} + \frac{B_2}{2(\Delta\eta)^2}\right]}_{d_2} u_{i+1,3}$$

$$= \underbrace{u_{i,1}\left[\frac{A_2}{4\Delta\eta} - \frac{B_2}{2(\Delta\eta)^2}\right] + u_{i,2}\left[\frac{B_2}{(\Delta\eta)^2} + \frac{(u_{i,2})^2}{\Delta x}\right] + u_{i,3}\left[\frac{-A_2}{4\Delta\eta} - \frac{B_2}{2(\Delta\eta)^2}\right] + u_{i+1,1}\left[\frac{A_2}{4\Delta\eta} - \frac{B_2}{2(\Delta\eta)^2}\right]}_{g_2}$$

5

Node adjacent to upper boundary $(j = N-1)$



Use 2nd-order central difference in η, Crank-Nicolson in x

$$u_{i,N-1}\left.\frac{\partial u}{\partial x}\right|_{i,N-1} + A_{N-1}\left.\frac{\partial u}{\partial \eta}\right|_{i,N-1} + B_{N-1}\left.\frac{\partial^2 u}{\partial \eta^2}\right|_{i,N-1} = 0$$

$$u_{i,N-1}\left[\frac{u_{i+1,N-1}-u_{i,N-1}}{\Delta x}\right] + \frac{1}{2}\left\{A_{N-1}\left[\frac{u_{i,N}-u_{i,N-2}}{2\Delta\eta} + \frac{u_{i+1,N}-u_{i+1,N-2}}{2\Delta\eta}\right]\right.$$

$$\left. + B_{N-1}\left[\frac{u_{i,N-2}-2u_{i,N-1}+u_{i,N}}{(\Delta\eta)^2} + \frac{u_{i+1,N-2}-2u_{i+1,N-1}+u_{i+1,N}}{(\Delta\eta)^2}\right]\right\} = 0$$

Rearrange with $u_{i+1,N-1}$ and $u_{i+1,N-2}$ on LHS, all known terms on RHS

$$\underbrace{\left[\frac{u_{i,N-1}}{\Delta x} - \frac{B_{N-1}}{(\Delta\eta)^2}\right]}_{c_{N-1}} u_{i+1,N-1} + \underbrace{\left[\frac{-A_{N-1}}{4\Delta\eta} + \frac{B_{N-1}}{2(\Delta\eta)^2}\right]}_{b_{N-1}} u_{i+1,N-2}$$

$$= \underbrace{u_{i,N-2}\left[\frac{A_{N-1}}{4\Delta\eta} - \frac{B_{N-1}}{2(\Delta\eta)^2}\right] + u_{i,N-1}\left[\frac{B_{N-1}}{(\Delta\eta)^2} + \frac{(u_{i,N-1})^2}{\Delta x}\right] + u_{i,N}\left[\frac{-A_{N-1}}{4\Delta\eta} - \frac{B_{N-1}}{2(\Delta\eta)^2}\right] + u_{i+1,N}\left[\frac{A_{N-1}}{4\Delta\eta} - \frac{B_{N-1}}{2(\Delta\eta)^2}\right]}_{g_{N-1}}$$

After solving for u_{i+1} at all j points, use continuity to solve for v_{i+1} at all j points.

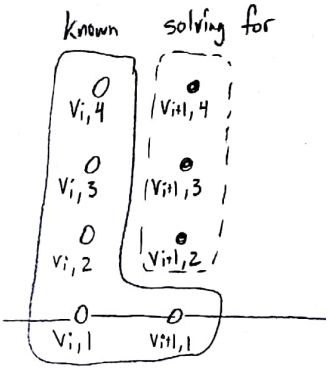
$$\left.\frac{\partial u}{\partial x}\right|_{i,j} + \eta_y \left.\frac{\partial v}{\partial \eta}\right|_{i,j} = 0$$

For stability, - an even-order centered approximation in η can NOT be used, as it would result in zeros along the diagonal of the coefficient matrix and cause the LU decomposition to fail (divide by zero issues.)

Solving for $v_{i+1, j=2}$ (adjacent to bottom boundry nodes)

Use Crank-Nicolson in x and forward-biased 3rd-order in η

$$\left.\frac{\partial v}{\partial \eta}\right|_{i,j} = \frac{-v_{i,j+2} + 6v_{i,j+1} - 3v_{i,j} - 2v_{i,j-1}}{6\Delta\eta} \quad ; \text{ here } j=2$$



Continuity

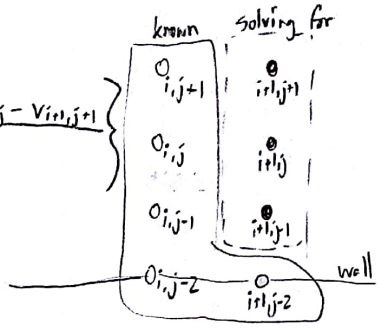
$$\frac{u_{i+1,j} - u_{i,j}}{\Delta x} + \frac{1}{2}\eta_y \left[\frac{-v_{i,j+2} + 6v_{i,j+1} - 3v_{i,j} - 2v_{i,j-1}}{6\Delta\eta} + \frac{-v_{i+1,j+2} + 6v_{i+1,j+1} - 3v_{i+1,j} - 2v_{i+1,j-1}}{6\Delta\eta} \right] = 0$$

($v_{i,j-1}$ = 0 from BC; $v_{i+1,j-1}$ = 0)

$$\underbrace{\left[\frac{-\eta_y}{4\Delta\eta}\right]}_{c_2} v_{i+1,2} + \underbrace{\left[\frac{\eta_y}{2\Delta\eta}\right]}_{d_2} v_{i+1,3} + \underbrace{\left[\frac{-\eta_y}{12\Delta\eta}\right]}_{e_2} v_{i+1,4} = \underbrace{\frac{u_{i,j} - u_{i+1,j}}{\Delta x} - \frac{\eta_y}{12}\left[\frac{-v_{i,4} + 6v_{i,3} - 3v_{i,2}}{\Delta\eta}\right]}_{g_2}$$

Use Crank-Nicolson in x and 4th-order centered at $j-1/2$ for node $j=3$

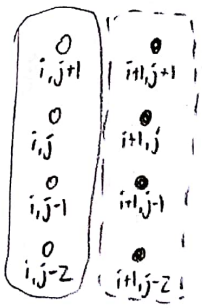
$$0=\left[\frac{u_{i+1,j}-u_{i,j}}{\Delta x}\right]+\frac{\eta_y}{2}\left\{\frac{\cancel{v_{i,j-2}}-27v_{i,j-1}+27v_{ij}-v_{ij+1}}{24\Delta\eta}+\frac{\cancel{v_{i+1,j-2}}-27v_{i+1,j-1}+27v_{i+1,j}-v_{i+1,j+1}}{24\Delta\eta}\right\}$$



$$\underbrace{\left[\frac{-9\eta_{y_3}}{16\Delta\eta}\right]}_{b_3}v_{i+1,2}+\underbrace{\left[\frac{9\eta_{y_3}}{16\Delta\eta}\right]}_{c_3}v_{i+1,3}+\underbrace{\left[\frac{-\eta_{y_3}}{48\Delta\eta}\right]}_{d_3}v_{i+1,4}$$

$$=\underbrace{\frac{u_{i,3}-u_{i+1,3}}{\Delta x}+\frac{\eta_y}{48}\left\{\frac{27v_{i,2}-27v_{i,3}+v_{i,4}}{\Delta\eta}\right\}}_{g_3}$$

Use same "stencil" for nodes $j=4$ to $j=N_y-1$ but one more node is unknown

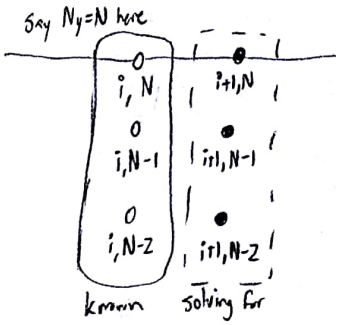


$$\underbrace{\left[\frac{\eta_{y_j}}{48\Delta\eta}\right]}_{a_j}v_{i+1,j-2}+\underbrace{\left[\frac{-9\eta_{y_j}}{16\Delta\eta}\right]}_{b_j}v_{i+1,j-1}+\underbrace{\left[\frac{9\eta_{y_j}}{16\Delta\eta}\right]}_{c_j}v_{i+1,j}+\underbrace{\left[\frac{-\eta_{y_j}}{48\Delta\eta}\right]}_{d_j}v_{i+1,j+1}$$

$$=\underbrace{\frac{u_{i,j}-u_{i+1,j}}{\Delta x}+\frac{\eta_{y_j}}{48}\left\{\frac{-v_{i,j-2}+27v_{i,j-1}-27v_{i,j}+v_{i,j+1}}{\Delta\eta}\right\}}_{g_j}$$

8

Finally at top boundary, use another scheme (3rd order)



backward difference in η about $j = N - 1/2$

$$\left.\frac{\partial v}{\partial \eta}\right|_{i,j} = \frac{3v_{i,N} - 4v_{i,N-1} + v_{i,N-2}}{2\Delta\eta}$$

$$\frac{\overbrace{u_{i+1,j} - u_{i,j}}^{=0}}{\Delta x} + \frac{\eta_y}{2}\left\{ \frac{3v_{i,N} - 4v_{i,N-1} + v_{i,N-2}}{2\Delta\eta} + \frac{3v_{i+1,N} - 4v_{i+1,N-1} + v_{i+1,N-2}}{2\Delta\eta} \right\} = 0$$

($v_{i,N}$, $v_{i,N-1}$, $v_{i,N-2}$ all $= 0$)

$$\underbrace{[1]}_{a_N} v_{i+1,N-2} + \underbrace{[-4]}_{b_N} v_{i+1,N-1} + \underbrace{[3]}_{c_N} v_{i+1,N} = \underbrace{0}_{g_N}$$