Numerical Solution of the Turbulent Boundary Layer Equations using the Cebeci-Smith Turbulence Model

AERSP/ME 525: Turbulence and Applications to CFD: RANS
Computer Assignment Number 3

Brian Knisely

May 2, 2018

Abstract

A finite difference scheme was used to solve the turbulent 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. To provide closure to the Reynolds Stress terms, the Cebeci-Smith algebraic turbulence model was used. 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. The boundary layer growth and wall shear stress were compared to correlations. A code was written in Python to calculate the solution according to the derived scheme. The grid was stretched in the y-direction; 251 nodes in the y-direction and 141 nodes in the x-direction were used.

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 the case of turbulent flow, the well-known closure problem requires that additional equations be used to describe the turbulent velocity fluctuations. Many widely-used turbulence models involve additional partial differential equations for turbulence parameters like turbulent kinetic energy (k) and rate of viscous dissipation (ϵ), for example in the k-ϵ model. While these two-equation models are relatively robust, the addition of partial differential equations to provide closure means that they are computationally expensive. Algebraic turbulence models such as the Cebeci-Smith model do not add any partial differential equations, and instead use explicit algebraic equations which are much faster to solve. This paper presents a process for numerically solving a the turbulent boundary layer equations using a finite difference method and the Cebeci-Smith turbulence model, and offers comparisons to correlations.

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.

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

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

The x-velocity is u and the y-velocity is v. The kinematic viscosity coefficient is ν and the turbulent viscosity, or eddy viscosity, is ν_T. In these equations, the tildes represent dimensional quantities. The equations can be made nondimensional with the following substitutions:

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

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

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

$$y = \frac{\tilde{y}}{\tilde{\delta}} \quad (6) \qquad \nu = \frac{\tilde{\nu}}{\tilde{\nu}_\infty} \quad (7) \qquad \frac{1}{RD} = \frac{\tilde{L}\tilde{\nu}_\infty}{\tilde{U}_\infty \tilde{\delta}^2} \quad (8) \qquad \nu_T = \frac{\tilde{\nu_T}}{\tilde{\nu}} \quad (9)$$

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

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{10}$$

$$u\frac{\partial u}{\partial x} + v\frac{\partial v}{\partial y} = \frac{1}{RD}\frac{\partial}{\partial y}\left((\nu + \nu_T)\frac{\partial u}{\partial y}\right) \tag{11}$$

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 nondimensionalized to yield

$$u = \begin{cases} sin\left(\frac{\pi y}{2}\right) & \text{for } y \leq 1 \\ 1 & \text{for } y > 1 \end{cases} \tag{12}$$

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.

$$u(x, 0) = 0 \tag{13}$$

$$u(x, 1) = 1 \tag{14}$$

$$v(x, 0) = 0 \tag{15}$$

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 10 and a second-order derivative for u in 11, 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 problem 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. The same stretching function was used as in the previous project and the boundary conditions and governing equations were transformed similarly. The nondimensional continuity and x-momentum equations were transformed from x-y space into x-η space, resulting in

$$\frac{\partial u}{\partial x} + \eta_y \frac{\partial v}{\partial \eta} = 0 \tag{16}$$

$$u\frac{\partial u}{\partial x} + \left[v\eta_y - \frac{1}{RD}\left((\nu + \nu_T)\eta_{yy} + \frac{\partial \nu_T}{\partial \eta}(\eta_y)^2\right)\right]\frac{\partial u}{\partial \eta} + \frac{-\nu\eta_y^2}{RD}\left(\frac{\partial^2 u}{\partial^2 \eta}\right) = 0 \tag{17}$$

As in the previous project, the stretching function was chosen to be

$$y = f(\eta) = y_{max}\frac{\sinh(s\eta)}{\sinh(s)} \tag{18}$$

where s is a constant known as the stretching factor, the derivatives can be determined analytically and substituted into 16 and 17. The derivatives are

$$\eta_y = \frac{\sinh(s)}{sy_{max}\cosh(s\eta)} \tag{19}$$

$$\eta_{yy} = \frac{\sinh(s\eta)\sinh^2(s)}{sy_{max}^2\cosh^3(s\eta)} \tag{20}$$

Discretization Schemes

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. 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.

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.

Eddy Viscosity Calculation using the Cebeci-Smith Turbulence Model

The eddy viscosity is a parameter that describes the effect of turbulent fluctuations on the mean flow. The eddy viscosity is not constant; it is a function of the velocity field. The Cebeci-Smith model involves several calculations to eventually arrive at the eddy viscosity. Eddy viscosities for the inner (ν_{Ti}) and outer (ν_{To}) regions of the boundary layer are calculated, and are combined at the point in which $\nu_{Ti} = \nu_{To}$. To calculate the inner eddy viscosity, the following equations are used:

$$\tilde{u}_\tau = \sqrt{\tilde{\nu}\frac{\partial\tilde{u}}{\partial\tilde{y}}} \tag{21}$$

$$y^+ = \frac{\tilde{y}\tilde{u_\tau}}{\tilde{\nu}} \tag{22}$$

$$\tilde{\ell}_{mix} = \kappa\tilde{y}\left[1 - \exp\left(\frac{-y^+}{A^+}\right)\right] \tag{23}$$

$$\tilde{\nu}_{Ti} = \tilde{\ell}_{mix}^2\left[\left(\frac{\partial\tilde{u}}{\partial\tilde{y}}\right)^2 + \left(\frac{\partial\tilde{v}}{\partial\tilde{x}}\right)^2\right]^{1/2} \tag{24}$$

The parameters κ and A^+ are tuned constants. For this model, κ (also known as the von Kármán constant) was set to 0.4 and A^+ was set to 26. To calculate the outer eddy viscosity, the following equations are used:

$$\tilde{\delta}^* = \int_0^{\tilde{\delta}}\left(1 - \frac{\tilde{u}}{\tilde{U_\infty}}\right)d\tilde{y} \tag{25}$$

$$F_{KLEB} = \left[1 + 5.5\left(\frac{\tilde{y}}{\tilde{\delta}}\right)^6\right]^{-1} \tag{26}$$

$$\tilde{\nu_{To}} = \alpha\tilde{U}_\infty\tilde{\delta}^*F_{KLEB} \tag{27}$$

In these equations, α is another empirical constant which is set to 0.0168 and $\tilde{\delta}$ is the dimensional 99% boundary layer thickness.

LU Decomposition Matrix Inversion Algorithm

To invert the matrix, an LU decomposition scheme was written specifically for a pentadiagonal matrix. The same LU decomposition algorithm was implemented in the Python code as in previous projects.

Results

Using a stretching factor of $s = 5$, the finite difference code was executed to calculate the velocity field for the turbulent boundary layer. The velocity profiles at various locations downstream are shown in in Figure 1. The finite difference solution begins with a laminar starting profile which quickly grows due to the turbulence. The 99% boundary layer height was calculated and used to determine the virtual origin of the turbulent boundary layer. The correlation:

$$\tilde{\delta} = 0.375\tilde{x}Re_{\tilde{x}}^{-1/5} = 0.375\left(\frac{\tilde{U_\infty}}{\tilde{\nu}}\right)^{-1/5}\tilde{x}^{4/5} \tag{28}$$

was rearranged to calculate x as a function of δ. Based on the boundary layer thickness at the final x-step, the virtual origin of the turbulent boundary layer is at **x = 0.23 m**.

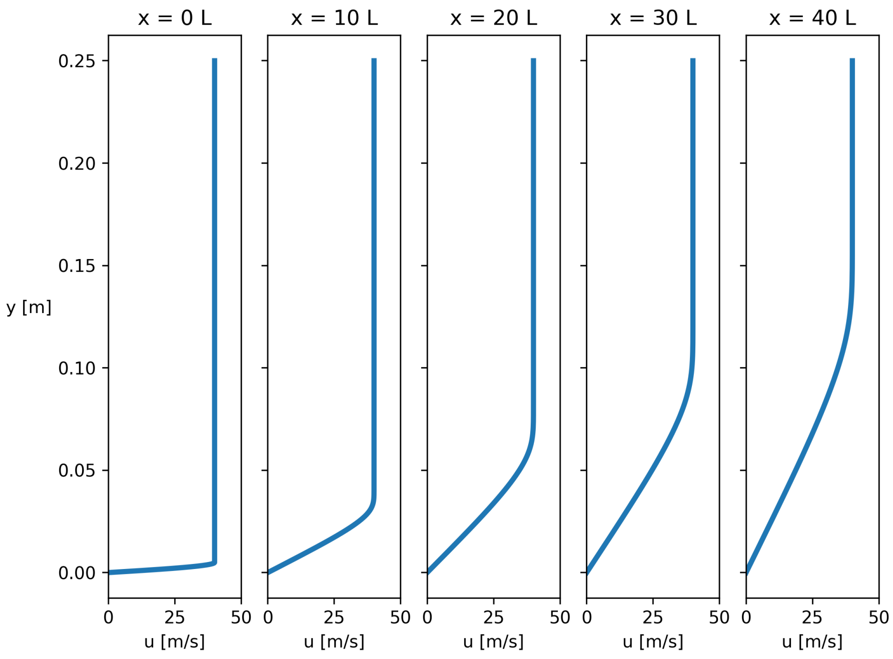

Figure 1: Velocity profiles at five x-locations as computed by the finite difference method

The profiles shown in Figure 1 are not as expected for a turbulent boundary layer. The shape of the boundary layer is indicative of a laminar boundary layer, as evidenced by the nearly-linear behavior near the wall and the lack of "fullness" expected from a turbulent boundary layer. Figure 2 shows the velocity contour and again indicates that the boundary layer does not appear to match the expected behavior for a turbulent boundary layer. The growth rate appears linear from the contour plot. Figure 3 shows the near-wall behavior at the final x-step. Again, the solution exhibits no turbulent behavior with no transition to the "log region".

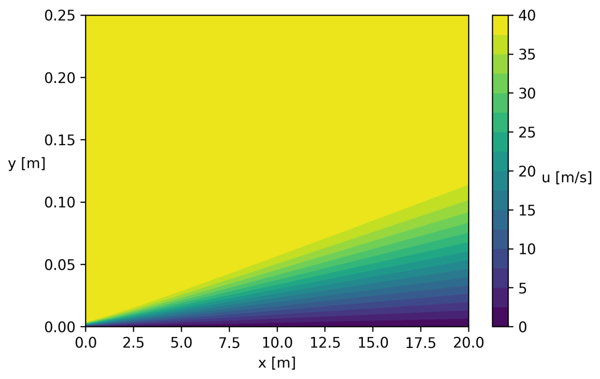

Figure 2: Velocity contour as computed by the finite difference method

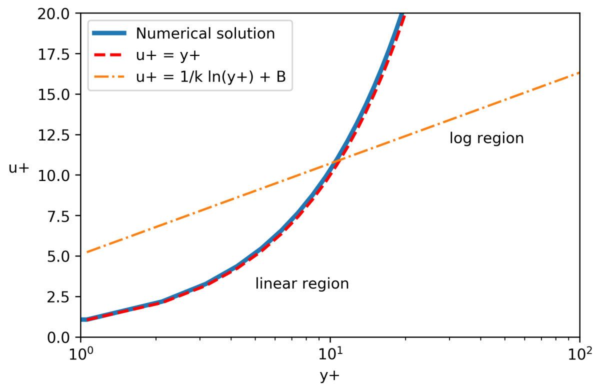

Figure 3: Near-wall velocity profile compared to the Law of the Wall

Figure 4 shows that the boundary layer is growing at the correct order of magnitude, but appears to be growing linearly rather than growing proportional to $x^{4/5}$. Figure 5 shows the skin friction compared to the 1/7 Power Law correlation. Again the boundary layer is exhibiting laminar-like behavior with very low skin friction. The reason for these discrepancies with the solution will be discussed in more detail below.

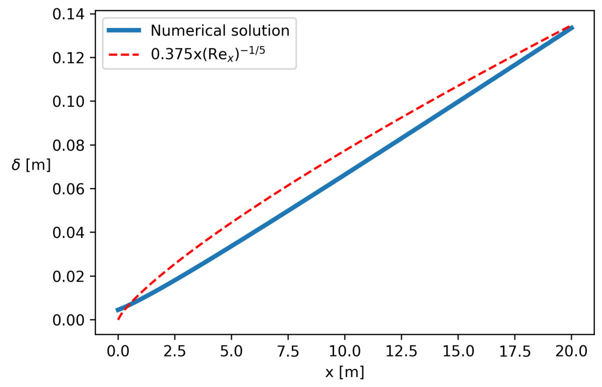

Figure 4: Boundary layer growth

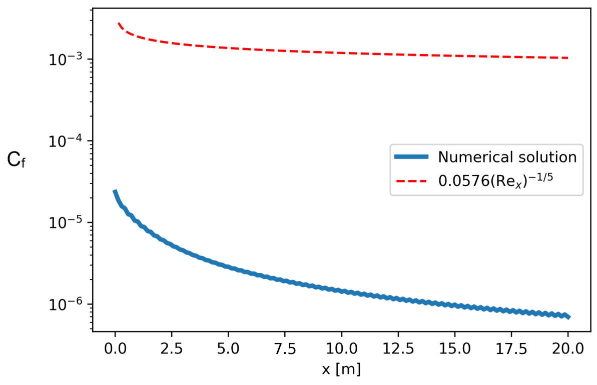

Figure 5: Skin friction compared to 1/7 Power Law Correlation

Sensitivity of Solution to Tuned Constants

The constants κ and α were adjusted to determine their effects on the solution. Each parameter was independently increased or decreased by 50%. Figures 6 and 7 show the effect on the near-wall region of decreasing or increasing κ by 50%, respectively. The change in the boundary layer growth rate was not discernible with the different values of κ. The viscous sublayer behavior is closely matched for all values of κ, so the κ does not appear to heavily impact the flow field. In contrast, α had a significant effect on the boundary layer growth rate but little effect on the near-wall region. Figures 8 and 9 show the effects of decreasing or increasing α by 50%, respectively. The constant α appears to directly scale the boundary layer growth rate.

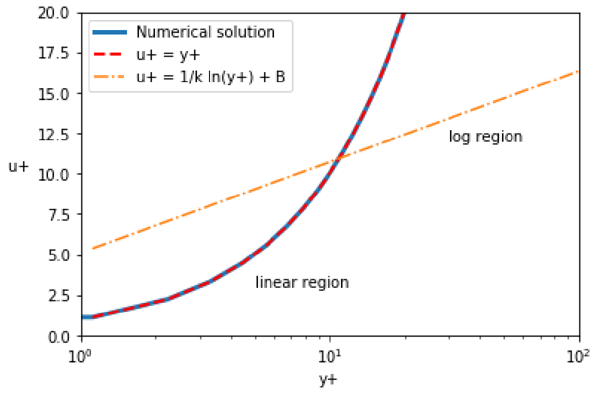

Figure 6: Near-wall velocity profile with κ = 0.2

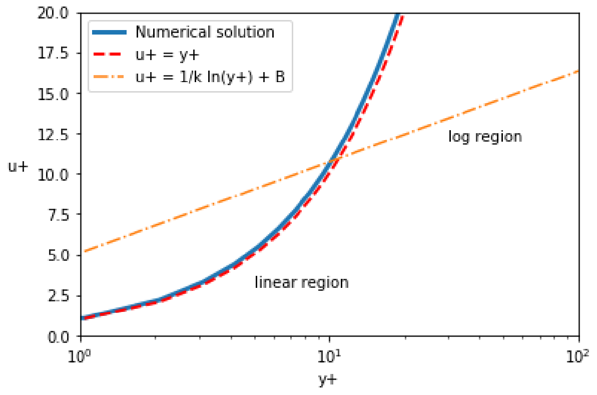

Figure 7: Near-wall velocity profile with κ = 0.6

Considerations with Increasing Reynolds Number

The Reynolds number was increased by a factor of 10 by increasing the freestream velocity from 40 m/s to 400 m/s. Using the same stretching factor (5) and grid resolution in y (251 nodes), it was evident that not enough nodes were concentrated near the wall. Only three nodes were within $y+ = 10$. The stretching factor was increased to 15 to compensate and focus more nodes near the wall. In turn, the grid resolution also had to be increased because not enough nodes were located away from the wall. This simple experiment demonstrated that when dealing with high-Reynolds number flows, increased grid resolution near walls is necessary to resolve the viscous sublayer behavior.

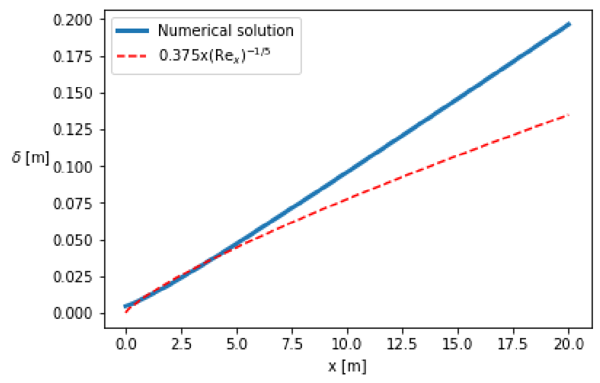

Figure 8: Boundary layer growth with α = 0.0084

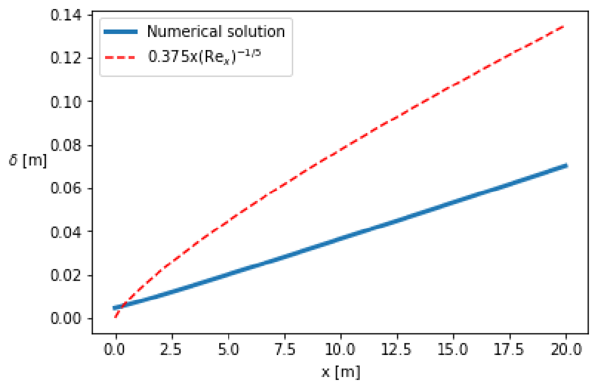

Figure 9: Boundary layer growth with α = 0.0252

In practice, one could instead use wall functions and simply place the node nearest to the wall at a y^+ of approximately 50, and then use the Law of the Wall result to determine velocities closer to the wall. This would reduce complexity in meshing near the walls and save computation time.

Problems with Implementation of Model

The Navier-Stokes equations are nonlinear, but this solution simply solves a linear system. The equations were made linear by using some information from the previous x-step. While this is a simple and computationally inexpensive method, it is also an inaccurate method. Using an established method like the predictor-corrector pattern found in many commercial CFD codes would dramatically increase the accuracy of the solution. For each x-step, the momentum equations would each be solved for one variable and then make corrections using the pressure Poisson equation (in place of continuity). This method would not only increase accuracy but also increase stability as well.

Stability was a major concern with this solution scheme. In particular, the $\partial \nu_T / \partial \eta$ term in (17) caused significant problems when attempting to march through the solution. A problem arose because of velocity fluctuations in $\partial u / \partial y$ in the boundary layer. These small fluctuations in velocity caused large spikes in the derivative of eddy viscosity. The solution would quickly grow out of control, resulting in velocities on the order of 1e20. Attempts to adjust stretching factor, grid resolution, and freestream velocity did not allow the solution to converge. In the end, the $\partial \nu_T / \partial \eta$ was set to zero in order to produce any results at all. It is clear that this term is essential for resolving the near-wall log behavior and to slow the growth of the boundary layer to match that of correlations.

Conclusion

A finite-difference code was written in Python to solve the turbulent boundary layer equations for a flat plate. The numerical method used a fourth-order central difference approach in y, the second-order Crank-Nicolson algorithm to march in x, the Cebeci-Smith turbulence model for eddy viscosity, and LU decomposition to solve the linear systems. After stability problems with the $\partial \nu_T / \partial \eta$ term in (17), that term was neglected. The resulting velocity profiles were characteristically laminar, indicating that the neglected term is important in resolving the turbulent profile. The near-wall behavior did not match the Law of the Wall, with the numerical solution deciding to stay in the viscous region instead of transitioning to the log region. The growth rate did not match the expected growth rate for a turbulent boundary layer, but grew to the correct order of magnitude by the end of the domain. The effect of constants κ and α was investigated. The solution is not strongly affected by changes in κ, while the boundary layer grows proportionately with α.

Appendix 1: Python Code

```
# -*- coding: utf-8 -*-
"""
Created on Thu April 4 19:06:50 2018

@author: Brian

AERSP/ME 525: Turbulence and Applications to CFD: RANS
Computer Project Number 3

The purpose of this code is to use finite difference to solve the turbulent
boundary layer equations for a 2D flat plate and compare to correlations.
The Cebeci-Smith algebraic turbulence model is used.

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

Note: all derivatives are partial derivatives

The boundary conditions (BCs) in nondimensional form are:
    u(x, 0) = 0  (no-slip condition)
    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.
"""

# 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, exp, log
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


# %% Function to calculate eddy viscosity
def eddyViscosity(u, v, x, y, i, uInfDim):
```

```
# inputs:   u = nondimensional x-velocity
#           v = nondimensional y-velocity
#           x = *dimensional* x-coordinates
#           y = nondimensional y-coordinates
#           i = index of x-step currently being evaluated

# Coefficients and dimensional constants
Aplus = 26.
alpha = 0.0168
kappa = 0.4
nuInfDim = 1.5e-6  # Dimensional freestream viscosity in m^2/s
LDim = 0.5  # Length of plate in m
deltaDim = 0.005  # Initial BL thickness in m

# Format inputs for use in this function
ui = u[:, i]*uInfDim  # current dimensional x-velocity
uiNorm = u[:, i]  # current nondimensional x-velocity
# Convert list of nondimensional y locations to array
# of dimensional y-values
yDim = deltaDim*np.array(y)
vDim = v*uInfDim*deltaDim/LDim

# Initialize arrays
dudy = np.zeros(len(y))
dvdx = np.zeros(len(y))

# Compute friction velocity (scalar) with dimensional quantities
# and 3rd order Finite Difference Method
uTau = (nuInfDim*(-ui[2]+4*ui[1]-3*ui[0]) / (yDim[2]-yDim[0]))**(1/2)

# Compute yPlus (vector)
yPlus = yDim*uTau/nuInfDim

# Compute mixingLength (vector)
mixingLength = [kappa*yDim[i]*(1-exp(-yPlus[i]/Aplus))
                for i in range(len(y))]

# Compute nuTi (vector) for all y-values
# using 3rd-order scheme in y and 1st-order scheme in x
# First calculate u-gradient in y using current x-step
dudy[0] = (-ui[2]+4*ui[1]-3*ui[0]) / (yDim[2]-yDim[0])
for j in range(1, len(y)-1):
    dudy[j] = (-ui[j+1]+4*ui[j]-3*ui[j-1]) / (yDim[j+1]-yDim[j-1])
dudy[-1] = (-ui[-1]+4*ui[-2]-3*ui[-3]) / (yDim[-1]-yDim[-3])
# Next calculate v-gradient in x using current and previous x-steps
# First-order backward difference
for j in range(0, len(y)):
    dvdx[j] = (vDim[j, i] - vDim[j, i-1])/(x[i]-x[i-1])
# Compute nuTi using calculated derivatives
nuti = [mixingLength[i]**2*sqrt(dudy[i]**2+dvdx[i]**2)
        for i in range(len(y))]

# Compute delta99 (scalar) and deltaStar (scalar)
deltaStar = 0  # initialize scalars
```

```
    delta99 = 0  # initialize scalars
    idx = np.argmin(abs(uiNorm-0.99))
    # linear interpolate between nearest points to get better estimate for d99
    if uiNorm[idx] < 0.99:
        delta99 = yDim[idx+1]-(uiNorm[idx+1]-0.99)*(yDim[idx+1]-yDim[idx])/(
                  uiNorm[idx+1]-uiNorm[idx])
    else:
        delta99 = yDim[idx]-(uiNorm[idx]-0.99)*(yDim[idx]-yDim[idx-1])/(
                  uiNorm[idx]-uiNorm[idx-1])
    # integrate from y = 0 to t = delta99 to calculate deltaStar (scalar)
    for j in range(1, idx):
        dy = (y[j] - y[j-1])*deltaDim  # calculate dimensional y-difference
        integrand = ((1 - u[j, i]) + (1 - u[j-1, i]))/2
        deltaStar += integrand*dy  # add sections using trapezoidal integration

    # Compute F_KLEB (vector)
    fKleb = (1 + 5.5*((yDim/delta99)**6))**-1

    # Compute nuto (vector) for all y-values
    nuto = alpha*uInfDim*deltaStar*fKleb

    # Combine nuti and nuto into a single nut variable (vector)
    nut = np.zeros(len(nuti))
    zone = 'inner'  # initialize variable to set whether in inner or outer zone
    for n in range(len(nuti)):
        if nuti[n] > nuto[n]:  # check whether inner is greater than outer
            zone = 'outer'  # if so, begin assigning outer values
        if zone == 'inner':
            nut[n] = nuti[n]
        else:
            nut[n] = nuto[n]

    # Numerically differentiate to calculate dnut/dy (vector)
    # using 3rd order finite difference method
    dnutdy = np.zeros(len(nut))
    dnutdy[0] = (-nut[2]+4*nut[1]-3*nut[0]) / (yDim[2]-yDim[0])
    for j in range(len(nut)-1):
        dnutdy[j] = (-nut[j+1]+4*nut[j]-3*nut[j-1]) / (yDim[j+1]-yDim[j-1])
    dnutdy[-1] = (-nut[-1]+4*nut[-2]-3*nut[-3]) / (yDim[-1]-yDim[-3])

    return nut, dnutdy  # output eddy viscosity vector and its derivative


# %% Main function
def main(Nx, Ny, method, s, uInfDim):
    # 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
    #             uInfDim = dimensional freestream velocity in m/s
    # Inputs for testing
    # Nx = 141
    # Ny = 251
    # method = 'lu'
    # s = 5
    # uInfDim = 40

    # Define bounds of computational domain
    xMax = 20  # Dimensional distance in m
    yMax = 50  # nondimensional, Scaled by BL height

    # Define given dimensional quantities
    nuInfDim = 1.5e-6  # Dimensional freestream viscosity in m^2/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 = np.array([1/fP for fP in fPrime])
    etaYY = np.array([-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

    # Compute turbulent viscosity based on previous x-step velocities
    nut, dnutdy = eddyViscosity(u, v, x, y, i, uInfDim)  # call function
    # Nondimensionalize eddy viscosity and derivative
    nutNorm = nut/nuInfDim
    dnutde = nuInfDim*deltaDim/etaY * dnutdy

    # %% 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
    j = 1  # note python indices start at 0, not 1
    Ae = dx/(2*u[j, i]) * (v[j, i]*etaY[j] - 1/RD
                           * ((1+nutNorm[j])*etaYY[j]
                           + dnutde[j]*etaY[j]**2))
    Be = -dx/(2*u[j, i]) * (1+nutNorm[j]) * etaY[j]**2/RD

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

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

    # Second internal node (j = 3)
    # Calculate values of A(eta) and B(eta) at j
    j = 2
    Ae = dx/(2*u[j, i]) * (v[j, i]*etaY[j] - 1/RD
                           * ((1+nutNorm[j])*etaYY[j]
```

```python
                        + dnutde[j]*etaY[j]**2))
Be = -dx/(2*u[j, i]) * (1+nutNorm[j]) * etaY[j]**2/RD

# Store coefficients and value for RHS vector b
A[1, 0] = (-2*Ae/de + 4*Be/de**2)/3
A[1, 1] = 1 - 5/2*Be/de**2
A[1, 2] = 2/3*Ae/de+4/3*Be/de**2
A[1, 3] = -Ae/(12*de) - Be/(12*de**2)
b[1] = ((2/3*Ae/de - 4/3*Be/(de**2))*u[j-1, i]
        + (1 + 5/2*Be/(de**2))*u[j, i]
        + (-2/3*Ae/de - 4/3*Be/(de**2))*u[j+1, i]
        + (Ae/(12*de) + Be/(12*(de**2)))*u[j+2, i])

# %% Loop over internal nodes to compute and store coefficients
for j in range(3, Ny-3):
    # Calculate values of A(eta) and B(eta) at j
    Ae = dx/(2*u[j, i]) * (v[j, i]*etaY[j] - 1/RD
                            * ((1+nutNorm[j])*etaYY[j]
                            + dnutde[j]*etaY[j]**2))
    Be = -dx/(2*u[j, i]) * (1+nutNorm[j]) * etaY[j]**2/RD
    A[j-1, j-3] = Ae/(12*de) - Be/(12*de**2)
    A[j-1, j-2] = -2/3*Ae/de + 4/3*Be/de**2
    A[j-1, j-1] = 1 - 5/2*Be/de**2
    A[j-1, j] = 2/3*Ae/de+4/3*Be/de**2
    A[j-1, j+1] = -Ae/(12*de) - Be/(12*de**2)
    b[j-1] = ((-Ae/(12*de)+Be/(12*(de**2)))*u[j-2, i]
              + (2/3*Ae/de-4/3*Be/de**2)*u[j-1, i]
              + (1+5/2*Be/de**2)*u[j, i]
              + (-2/3*Ae/de-4/3*Be/de**2)*u[j+1, i]
              + (Ae/(12*de)+Be/(12*de**2))*u[j+2, i])

# %% Second-to-last internal node (j = Ny-2)
# Calculate values of A(eta) and B(eta) at j
j = Ny-3
Ae = dx/(2*u[j, i]) * (v[j, i]*etaY[j] - 1/RD
                        * ((1+nutNorm[j])*etaYY[j]
                        + dnutde[j]*etaY[j]**2))
Be = -dx/(2*u[j, i]) * (1+nutNorm[j]) * etaY[j]**2/RD

# Store coefficients and value for RHS vector b
A[-2, j-3] = Ae/(12*de) - Be/(12*de**2)
A[-2, j-2] = -2/3*Ae/de + 4/3*Be/de**2
A[-2, j-1] = 1 - 5/2*Be/de**2
A[-2, j] = 2/3*Ae/de+4/3*Be/de**2
b[-2] = ((-Ae/(12*de)+Be/(12*de**2))*u[j-2, i]
         + (2/3*Ae/de-4/3*Be/de**2)*u[j-1, i]
         + (1+5/2*Be/de**2)*u[j, i]
         + (-2/3*Ae/de - 4/3*Be/de**2)*u[j+1, i]
         + (Ae/(6*de) + Be/(6*de**2)))

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

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

for j in range(1, Ny):

    if i == 0:  # if first iteration of x-step loop, use first-order x

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

        # One up from bottom boundary - use 4th order about j - 1/2
        elif j == 2:
            A[j-1, j-2] = -9*etaY[2]/(8*de)
            A[j-1, j-1] = 9*etaY[2]/(8*de)
            A[j-1, j] = -etaY[2]/(24*de)
            b[j-1] = ((u[j, i+1] - u[j, i] + u[j-1, i+1] - u[j-1, i])
                      / (-2*dx))

        # At top boundary use 3rd order scheme about Ny - 1/2
        elif j == Ny-1:
            A[j-1, j-3] = 1/2
            A[j-1, j-2] = -2
            A[j-1, j-1] = 3/2
            b[j-1] = 0

        # Loop over internal nodes - use 4th order scheme about j - 1/2
        else:
            A[j-1, j-3] = etaY[j+1]/(24*de)
```

```
                    A[j-1, j-2] = -9*etaY[j+1]/(8*de)
                    A[j-1, j-1] = 9*etaY[j+1]/(8*de)
                    A[j-1, j] = -etaY[j+1]/(24*de)
                    b[j-1] = ((u[j, i+1] - u[j, i] + u[j-1, i+1] - u[j-1, i])
                              / (-2*dx))
            else:
                # Use third order FDS in eta, 2nd order Crank Nicolson in x
                # Use biased 3rd order for node adjacent to bottom boundary
                if j == 1:
                    A[j-1, j-1] = -etaY[j]/(2*de)
                    A[j-1, j] = etaY[j]/(de)
                    A[j-1, j+1] = -etaY[j]/(6*de)
                    b[j-1] = -1/dx*(3/2*u[j, i+1] - 2*u[j, i] + 1/2*u[j, i-1])

                # One up from bottom boundary - use 4th order about j-1/2
                elif j == 2:
                    A[j-1, j-2] = -9*etaY[2]/(8*de)
                    A[j-1, j-1] = 9*etaY[2]/(8*de)
                    A[j-1, j] = -etaY[2]/(24*de)
                    b[j-1] = -1/dx*(3/2*u[j, i+1] - 2*u[j, i] + 1/2*u[j, i-1]
                                    + 3/2*u[j-1, i+1]-2*u[j-1, i]
                                    + 1/2*u[j-1, i-1])

                # At top boundary use 3rd order scheme about Ny - 1/2
                elif j == Ny-1:
                    A[j-1, j-3] = 1/2
                    A[j-1, j-2] = -2
                    A[j-1, j-1] = 3/2
                    b[j-1] = 0

                # Loop over internal nodes - use 4th order scheme about j - 1/2
                else:
                    A[j-1, j-3] = etaY[j+1]/(24*de)
                    A[j-1, j-2] = -9*etaY[j+1]/(8*de)
                    A[j-1, j-1] = 9*etaY[j+1]/(8*de)
                    A[j-1, j] = -etaY[j+1]/(24*de)
                    b[j-1] = -1/dx*(3/2*u[j, i+1] - 2*u[j, i] + 1/2*u[j, i-1]
                                    + 3/2*u[j-1, i+1]-2*u[j-1, i]
                                    + 1/2*u[j-1, i-1])

    # 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, v-matrix, and nondimensional x- and y-vectors
    return u, v, x, y


# %% Define function to plot velocity contour
def velocityContour(u, x, y, uInfDim):
    deltaDim = 0.005  # dimensional starting BL height
    plt.figure()
    plt.contourf(x, np.array(y)*deltaDim, u*uInfDim,
                 levels=np.linspace(0, uInfDim, num=17))
    cbar = plt.colorbar()
    cbar.set_label('           u [m/s]', rotation=0)
    plt.xlabel('x [m]')
    plt.ylabel('y [m]     ', rotation=0)
    # save figure to file
    # plt.savefig(os.getcwd()+"\\figures\\turbVelocityContour.png", dpi=320,
    #             bbox_inches='tight')
    plt.close()


# %% Define function to plot velocity profiles along plate
def velocityProfiles(x, y, u, uInfDim):
    deltaDim = 0.005  # dimensional starting BL height
    yDim = deltaDim*np.array(y)  # dimensional y-value

    # get indices corresponding to five locations on plate
    xInds = [int(xInd) for xInd in np.linspace(0, len(x)-1, 5)]
    xLoc = x[xInds]

    # 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
    for ii in range(5):
        ax = axs[ii]
        ax.plot(uInfDim*u[:, xInds[ii]], yDim, linewidth=3)
        ax.set_xlabel('u [m/s]')
        ax.set_title('x = {0:0.0f} L'.format(xLoc[ii]/0.5))
        if ii == 0:
            ax.set_ylabel('y [m]        ', rotation=0)
        ax.set_xlim([0, 50])

    # save figure to file
    # plt.savefig(os.getcwd()+"\\figures\\turbVelocityProfiles.png", dpi=320,
    #              bbox_inches='tight')
    plt.close()


# %% Define function to calculate 99% BL thickness for all x-locations
#    and plot result compared to correlation
def delta99(u, x, y, uInfDim):
    nuInfDim = 1.5e-6  # dimensional viscosity
    deltaDim = 0.005  # dimensional starting BL height
    # Compute ith component of delta99
    delta = np.zeros(np.shape(u)[1])  # initialize array
    for i in range(np.shape(u)[1]):
        uiNorm = u[:, i]
        idx = np.argmin(abs(uiNorm-0.99))
        # linear interpolate between nearest points to get better estimate
        if uiNorm[idx] < 0.99:
            delta[i] = (y[idx+1]-(uiNorm[idx+1]-0.99)
                        * (y[idx+1]-y[idx])
                        / (uiNorm[idx+1]-uiNorm[idx]))
        else:
            delta[i] = (y[idx]-(uiNorm[idx]-0.99)*(y[idx]-y[idx-1])
                        / (uiNorm[idx]-uiNorm[idx-1]))

    Rex = uInfDim*x/nuInfDim  # calculate Reynolds Number based on x-distance
    # replace 0s with very small number to prevent divide by zero
    Rex[Rex == 0] = 1e-16
    deltaAn = 0.375*x*(Rex)**(-1/5)/deltaDim
    plt.figure()
    plt.plot(x, delta*deltaDim, linewidth=3)
    plt.plot(x, deltaAn*deltaDim, 'r--')
    plt.xlabel('x [m]')
    plt.ylabel(r'$\delta$ [m]       ', rotation=0)
    plt.legend(['Numerical solution', r'0.375x(Re$_x)^{-1/5}$'])
    # plt.savefig(os.getcwd()+"\\figures\\turbDeltaGrowth.png", dpi=320,
    #             bbox_inches='tight')
    plt.close()

    # Return dimensional BL height at final profile (x = 20 m)
    return delta[-1]*deltaDim


# %% Define function to calculate skin friction as function of x
#    and compare to correlation
def skinFriction(u, x, y, uInfDim):
    nuInfDim = 1.5e-6  # dimensional viscosity
    deltaDim = 0.005  # dimensional starting BL height
    yDim = deltaDim*np.array(y)  # dimensional y-value
    cf = np.zeros(len(u[0, :]))
    for i in range(len(u[0, :])):
        ui = u[:, i]*uInfDim  # current dimensional x-velocity
        muDuDyOverRho = (nuInfDim*(-ui[2]+4*ui[1]-3*ui[0]) / (yDim[2]-yDim[0]))
        cf[i] = muDuDyOverRho / (0.5*uInfDim**2)

    Rex = x*uInfDim/nuInfDim  # Reynolds number along plate
    # Replace Re=0 with very small number to avoid divide by zero error
    Rex[Rex == 0] = 1e-20
    cfCorr = 0.0576*Rex**(-1/5)  # Use Cf correlation from 1/7 power law
    plt.figure()
    plt.semilogy(x, cf, linewidth=3)
    plt.semilogy(x, cfCorr, 'r--')
    plt.xlabel('x [m]')
    plt.ylabel(r'C$_f$    ', rotation=0)
    plt.legend(['Numerical solution', r'0.0576(Re$_x)^{-1/5}$'])
    # save figure
    # plt.savefig(os.getcwd()+"\\figures\\turbSkinFriction.png", dpi=320,
    #             bbox_inches='tight')
    plt.close()


# %% Define function to compare end profile to the "Law of the Wall"
```

```python
def lawOfTheWall(u, y, uInfDim):
    ui = u[:, -1]*uInfDim  # final column of x-velocity values (dimensional)
    nuInfDim = 1.5e-6  # dimensional viscosity
    deltaDim = 0.005
    yDim = np.array(y)*deltaDim
    # inputs are u-matrix and y-vector (both nondimensional)
    print("haha")
    uStar = (nuInfDim*(-ui[2]+4*ui[1]-3*ui[0]) / (yDim[2]-yDim[0]))**(1/2)

    # Compute yPlus (vector)
    yPlus = yDim*uStar/nuInfDim
    uPlus = ui/uStar

    plt.figure()
    plt.semilogx(yPlus, uPlus, linewidth=3)
    yPlus[yPlus == 0] = 1e-20
    uPlusLin = yPlus
    uPlusLog = [1/0.41 * log(yPlusi) + 5.1 for yPlusi in yPlus]
    plt.semilogx(yPlus[1:], uPlusLin[1:], '--r', linewidth=2)
    plt.semilogx(yPlus[1:], uPlusLog[1:], '-.')
    plt.xlabel('y+')
    plt.ylabel('u+      ', rotation=0)
    plt.xlim([1, 100])
    plt.ylim([0, 20])
    plt.text(30, 12, 'log region')
    plt.text(5, 3, 'linear region')
    plt.legend(['Numerical solution', 'u+ = y+', 'u+ = 1/k ln(y+) + B'])
    # save figure to file
    # plt.savefig(os.getcwd()+"\\figures\\turbLawOfTheWall.png", dpi=320,
    #             bbox_inches='tight')
    plt.close()


# %% Define function to determine virtual origin of turbulent BL
def virtualOrigin(x, uInfDim, deltaEnd):
    nuInfDim = 1.5e-6
    xOrig = (deltaEnd/0.375 * (nuInfDim/uInfDim)**-0.2)**(5/4)
    return xOrig


# %% Run functions in order

# Define constants
Nx = 141  # number of nodes in x-direction
Ny = 251  # number of nodes in y-direction
s = 5  # stretching factor
uInfDim = 40  # dimensional freestream velocity in m/s

# Run main function
u, v, x, y = main(Nx, Ny, 'lu', s, uInfDim)

u[u > 1] = 1  # correct numerical noise for values slightly above 1

# Plot velocity contour
```

```python
velocityContour(u, x, y, uInfDim)

# Plot velocity profiles at various x-locations
velocityProfiles(x, y, u, uInfDim)

# Plot 99% BL thickness growth and return BL height at final profile
deltaEnd = delta99(u, x, y, uInfDim)

# Calculate skin friction as function of x and compare to correlation
skinFriction(u, x, y, uInfDim)

# Compare near-wall velocity profile with the Law of the Wall solution
lawOfTheWall(u, y, uInfDim)

xOrig = virtualOrigin(u, uInfDim, deltaEnd)
print('Turbulent BL origin is at {0:.2f} meters.'.format(20-xOrig))
```