Numerical Solution of a Parabolic Partial Differential Equation using the Crank-Nicolson Algorithm

AERSP/ME 525: Turbulence and Applications to CFD: RANS
Computer Assignment Number 1

Brian Knisely

February 19, 2018

Abstract

A finite difference scheme was used to estimate the solution to a parabolic partial differential equation. 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 initial condition. For each step in x, a coupled set of algebraic equations were solved explicitly using an LU decomposition matrix inversion algorithm. The same partial differential equation was also solved analytically using separation of variables. A code was written in Python to calculate the solution according to the derived scheme and compute the error compared to the exact, analytical solution. A coarse grid of 21 x 21 nodes resolved the solution to a high degree of accuracy. Decreasing grid spacing in x reduced error, while grid spacing in y had no effect on error.

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, pressure, and temperature. 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 parabolic partial differential equation using a finite difference method, and offers comparisons to the exact, analytical solution.

Mathematical Problem

The given partial differential equation (PDE) and prescribed boundary conditions are

$$\frac{\partial u}{\partial x} - \frac{\partial^2 u}{\partial^2 y} = y \tag{1}$$

$$u(x,0) = 0 \quad (2) \qquad u(x,1) = 1 \quad (3) \qquad u(0,y) = y. \quad (4)$$

This problem statement is characteristically a linear, parabolic 2nd-order PDE of u in two dimensions, x and y. The PDE is nonhomogeneous with only one homogeneous boundary condition.

Analytical Solution

A useful method of solving PDEs in more than one dimension is to use separation of variables. To use separation of variables, the PDE must be homogeneous with at least two homogeneous boundary conditions. Because the given PDE is linear, we can use superposition to separate u into two separate variables, U and v, as shown below.

$$u(x,y) = U(y) + v(x,y) \tag{5}$$

When substituting U and v into the original PDE, the new PDE of two variables is:

$$\frac{\partial v}{\partial x} - \frac{d^2 U}{dy^2} - \frac{\partial^2 u}{\partial^2 y} = y \tag{6}$$

By setting -$\frac{d^2U}{dy^2}$ to be equal to y, the equations for U and v can become separated and uncoupled. We now have two equations: one PDE for v and one ordinary differential equation (ODE) for U. To make the boundary conditions with v homogeneous, we choose to impose the following boundary conditions on U:

$$U(0) = 0 \qquad (7) \qquad\qquad U(1) = 1. \qquad (8)$$

The ODE for U can now be solved directly. The full derivation of the U solution is shown in Appendix 2. The solution to the ODE is

$$U(y) = \frac{-y^3}{6} + \frac{7y}{6}. \tag{9}$$

Using the imposed boundary conditions on U, the appropriate boundary conditions can be directly solved for v. The remaining PDE and boundary conditions for v are

$$\frac{\partial v}{\partial x} - \frac{\partial^2 v}{\partial^2 y} = 0 \tag{10}$$

$$v(x, 0) = 0 \quad (11) \qquad v(x, 1) = 0 \quad (12) \qquad v(0, y) = y^3/6 - y/6. \quad (13)$$

This is a 2nd-order homogeneous PDE with two homogeneous boundary conditions, so the separation of variables method may be used to solve for the analytical solution. The full derivation of the analytical solution is available in Appendix 2; the solution method will be summarized here. The single variable *v(x, y)* is assumed to be composed of two independent variables, *F(x)* and *G(y)*. The two variables are simply multiplied together to compose v. After substituting F and G into the PDE for v and rearranging terms, we find the left-hand side, which is a function of x only, is equal to the right-hand side, which is a function of y only. Since x and y can vary independently, this result is only possible if each term is equal to the same constant, which is introduced as -λ^2.

$$\frac{1}{F(x)}\frac{dF}{dx} = \frac{1}{G(y)}\frac{d^2G}{dy^2} = -\lambda^2 = constant \tag{14}$$

Now, each ODE is solved independently. The solution of each ODE is shown below.

$$G(y) = c_1 cos(\lambda y) + c_2 sin(\lambda y) \tag{15}$$

$$F(x) = c_3 e^{-\lambda^2 x} \tag{16}$$

Each coefficient c with a subscript is an unknown constant. These solutions to *F(x)* and *G(y)* can be multiplied to form *v(x, y)*, where the three c constants are converted into two new unknown constants, A and B.

$$v(x, y) = [Acos(\lambda y) + Bsin(\lambda y)]e^{-\lambda^2 x} \tag{17}$$

Now, boundary conditions must be applied to solve for the three unknowns: A, B, and λ. Using the boundary condition in (11), we find that $A = 0$. Using the boundary condition in (12), we find that there are infinitely many solutions for λ.

$$\lambda_n = n\pi, n = 1, 2, 3, ... \tag{18}$$

We now write the result for v as an infinite sum, assuming that the solution is a linear combination of all solutions.

$$v(x, y) = \sum_{n=1}^{\infty} B_n sin(n\pi y)e^{-(n\pi)^2 x} \tag{19}$$

This is a Fourier sine series. To solve for the unknown constants B_n, we use the final boundary condition from (13). By multiplying both sides by sin(m π y), integrating from 0 to 1, and recognizing the orthogonality of the sine function, we can solve discretely for B_n. The unknown coefficients are found to be

$$B_n = \frac{2(-1)^n}{(n\pi)^3}. \tag{20}$$

Therefore, the final analytical result for v is known, and the final expression for *u(x, y)* is

$$u(x, y) = \frac{-y^3}{6} + \frac{7y}{6} + \sum_{n=1}^{\infty} \frac{2(-1)^n}{(n\pi)^3} sin(n\pi y)e^{-(n\pi)^2 x}. \tag{21}$$

In the computer code, a non-infinite number of 100 terms were used to calculate the summation.

Numerical Method

A common approach to numerically solve a partial differential equation is to use a finite difference method, in which the computational domain is divided into a framework of discrete points, or nodes. In general the nodes may be arranged in any positions. For simplicity a uniform, Cartesian grid was selected; the spacing between nodes in the x-direction, Δx, was constant, and the spacing between nodes in the y-direction, Δy, was also constant. The computational domain for this problem was chosen to be 0 ≤ x ≤ 1 and 0 ≤ y ≤ 1. Coordinates in the x-direction are indexed with i, and coordinates in the y-direction are indexed with j. When N_x nodes are used in the x-direction and N_y points are used in the y-direction, the indices range from $1 \leq i \leq N_x$ and $1 \leq j \leq N_y$.

Beginning with the given PDE, finite difference approximations were constructed to convert derivatives into algebraic expressions. A full derivation of the finite difference construction is found in Appendix 3, but will be summarized here. For nodes that were centrally-located in the domain which had at least two nodes above and below, a fourth-order central differencing scheme was used. In this scheme, second-order derivatives in y at some internal point *(i, j)* are evaluated as

$$\left.\frac{\partial^2 u}{\partial y^2}\right|_{i,j} = \frac{-u_{i,j-2} + 16u_{i,j-1} - 30u_{i,j} + 16u_{i,j+1} - u_{i,j+2}}{12(\Delta y)^2} + O((\Delta y)^4). \tag{22}$$

The $O((\Delta y)^4)$ term indicates that there are remaining terms in this equation, with the leading-order truncation error of order $(\Delta y)^4$. All high-order terms are dropped in this approximation, so the derivative is estimated, not solved exactly. This scheme is consistent, so as Δy approaches zero, the error in solution also approaches zero. To use this scheme, a total of five nodes in the y-direction are used to calculate the derivative.

Combining the Crank-Nicolson scheme in x with the fourth-order approximation in y allows for the PDE of two directions to be computed. The Crank-Nicolson scheme effectively utilizes a forward differencing scheme, starting with an initial condition, to solve for successive values in one dimension. In this case, all values of *u(y)* at x = 0 are known from (4). From this column of nodes, a system of algebraic equations can be formed to describe the values at the next column (in the positive x-direction). A diagram showing the computational "stencil" for internal nodes is shown in Figure 1.

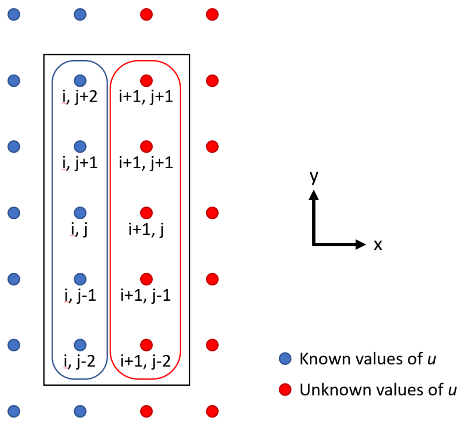


Figure 1: The computational stencil for internal nodes. The Crank-Nicolson scheme uses information at the i^{th} column to calculate all values at *i+1*.

These equations are assembled into a matrix equation $\underline{\underline{A}}\,\underline{u_{i+1}} = \underline{b}$ and the matrix equation is solved by inverting the coefficient matrix A and multiplying the result by b.

$$\underline{u_{i+1}} = \underline{\underline{A}}^{-1}\underline{b} \tag{23}$$

For the first internal node near a boundary, in which there are fewer than two nodes above or below, a second-order central differencing scheme was used. The computational stencil for a boundary node is shown in Figure 2.

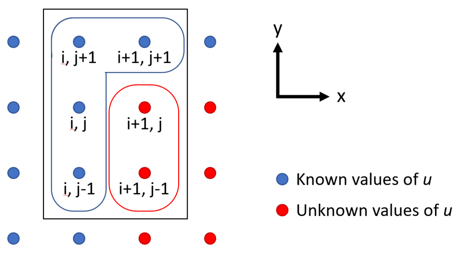


Figure 2: The computational stencil for nodes near boundaries.

A code was written in Python to systematically loop through all nodes in the y-direction and to assemble the coefficient matrix. The full Python code is available in Appendix 1. For each x-step, the coefficient matrix must be inverted to solve for the values of *u* at the *i+1* step.

The form of the *A* matrix is pentadiagonal, because there are five unknowns for each location of the stencil. While standard matrix inversion algorithms exist, a code designed specifically to solve a pentadiagonal matrix would theoretically reduce the number of computations and save calculation time.

LU Decomposition Matrix Inversion Algorithm

To invert the matrix, an LU decomposition scheme was written specifically for a pentadiagonal matrix. The widely-known 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:

$$\underline{\underline{L}}\,\underline{\underline{U}}\,\underline{u_{i+1}} = \underline{b} \Rightarrow \underline{\underline{U}}\,\underline{u_{i+1}} = \underline{\underline{L}}^{-1}\underline{b} \Rightarrow \underline{u_{i+1}} = \underline{\underline{U}}^{-1}(\underline{\underline{L}}^{-1}\underline{b}) \tag{24}$$

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

Initially, a coarse grid with 21 nodes in the x-direction and 21 nodes in the y-direction was used to calculate results. A contour plot of *u* as a function of x and y is shown in Figure 3, and a 1-dimensional plot of *u(y)* at *x* = 0.05, 0.1, 0.2, 0.5, and 1.0 is shown in Figure 4. The boundary conditions are satisfied by this result. At low values of x, u(y) is changing the most dramatically. As x increases, the profile of u(y) settles to a slightly upward-curving shape. At x = 0.5, the curve changes minimally though the remainder of the computational domain; the curves for x = 0.5 and x = 1.0 are indistinguishable. The absolute error in the numerical solution, defined as the absolute value of the numerical solution minus the analytical solution, is shown in Figures 5 and 6.

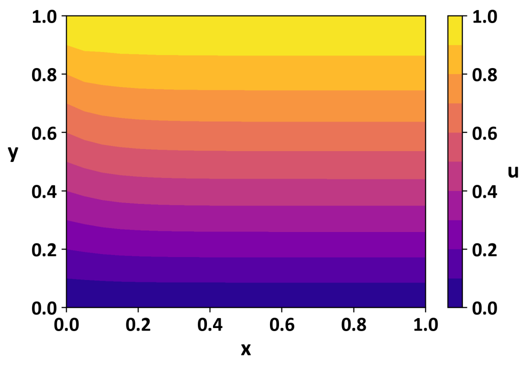


Figure 3: Numerical solution of *u* with N_x = 21 and N_y = 21.

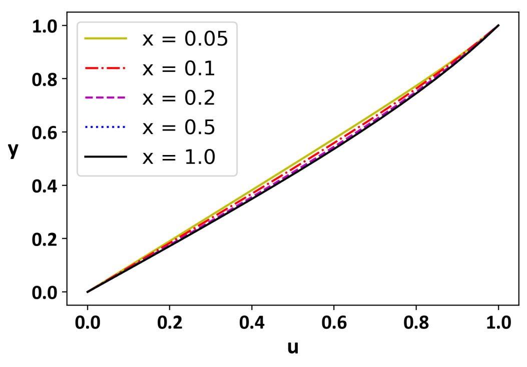


Figure 4: Numerical solution of *u* with N_x = 21 and N_y = 21 at x = 0.05, 0.1, 0.2, 0.5, and 1.0.

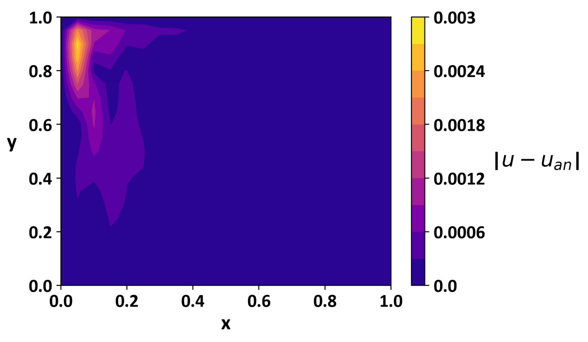


Figure 5: Error in numerical solution of *u* with N_x = 21 and N_y = 21.

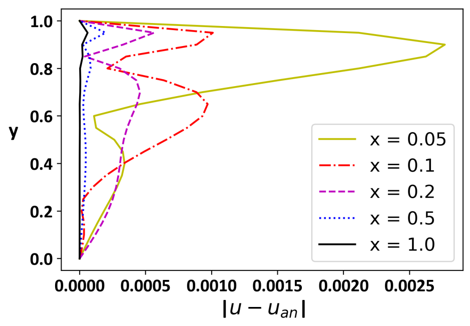


Figure 6: Error in numerical solution of *u* with N_x = 21 and N_y = 21 at x = 0.05, 0.1, 0.2, 0.5, and 1.0.

The maximum relative error is approximately 0.3%, so even with a coarse grid the results are quite accurate. The regions with the the steepest gradients (near x = 0.05 and y = 0.9) have the highest error, as expected. As the solution increases in x, the error decreases and the numerical solution is in very good agreement with the analytical solution.

Effect of Grid Spacings Δx and Δy

The effect of grid spacings Δx and Δy were studied by first varying Δx while Δy was held constant, then varying Δy while Δx was held constant, and finally decreasing both Δx and Δy together as the same values. Figure 7 shows that decreasing Δx decreases error up until a certain limit. Eventually the accuracy of the solution is limited by the finite-precision numbers used by the computer, and the error no longer increases. Also as Δx decreases, the computation time increases exponentially. Figure 8 shows that decreasing Δy does not decrease error. This would lead one to conclude that Δx is the key factor that determines error. Similarly to Figure 7, the computation time increases exponentially as Δy decreases. Figure 9 shows the effect on error and computation time as both Δx and Δy are decreased together. The error is comparable to Figure 7, in which only Δx was changed, while the computational effort is orders of magnitude higher. The lack of improvement in error with dramatically increased computation time indicates that the lowest error should be pursued by decreasing Δx only.

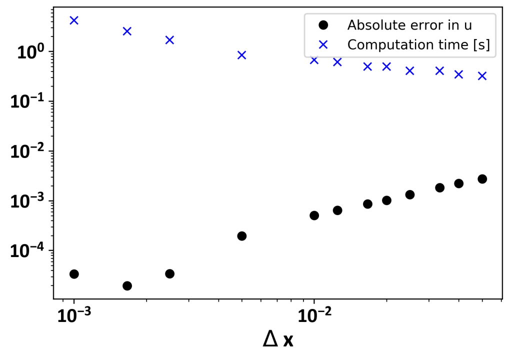


Figure 7: Effect of varying Δx on error and computation time with Δy held constant at 0.05.

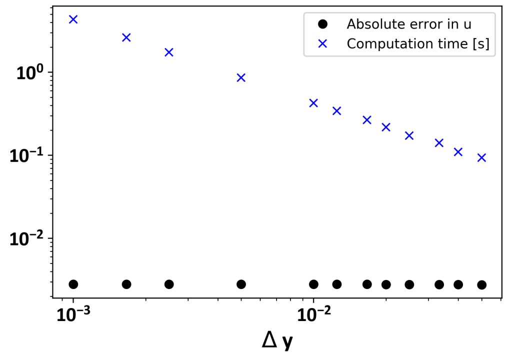


Figure 8: Effect of varying Δy on error and computation time with Δx held constant at 0.05.

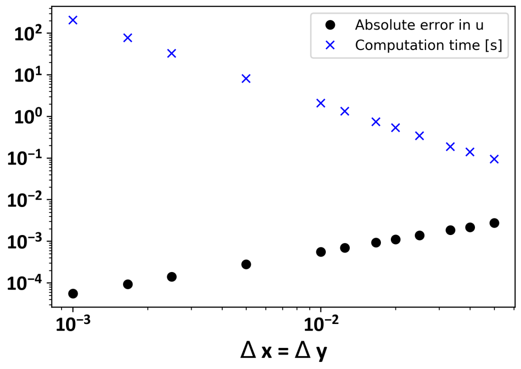


Figure 9: Effect of varying Δx and Δy together on error and computation time

Conclusion

A finite-difference code was written in Python to solve a linear, non-homogeneous parabolic partial differential equation. First, the analytical solution was solved using superposition, separation of variables, and orthogonality to result in an infinite Fourier sine series solution. The numerical method used a fourth-order central difference approach in y with a Crank-Nicolson algorithm to march in x, utilizing LU decomposition to solve for a column of u-values simultaneously at each x-step. A coarse grid was found to resolve the solution to a high degree of accuracy. The areas of the solution with the steepest gradients had the largest errors in the numerical solution. Decreasing x-spacing Δx had a direct effect on reducing error in the solution, while decreasing Δy had no effect on error. Decreasing either spacing resulted in significant increases to computation time.

Appendix 1: Python Code

```python
"""
Mon, Feb 19, 2018

@author: Brian F. Knisely

AERSP/ME 525: Turbulence and Applications to CFD: RANS
Computer Project Number 1

The purpose of this code is to use finite difference to solve a parabolic
partial differential equation.

The PDE is (all partial derivatives): du/dx - d^2(u)/dy^2 = y

Subject to the following boundary conditions (BCs):
    u(x, 0) = 0
    u(x, 1) = 1
    u(0, y) = y

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.

Developed in Spyder 3.2.6 for Windows
"""

# Import packages for arrays, plotting, and timing
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt
import time


def analytic(Nx, Ny):  # Define function to output analytic solution
    """
    This function solves this PDE (all derivatives are partial):

    du/dx - d^2(u)/dy^2 = y

    using the Fourier sine series solution with N = 100 summation terms.
    """
    # Import functions as necessary
    from math import sin, exp, pi

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

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

    # Initialize solution as 2D array of zeros
    # of dimension [Nx columns] by [Ny rows]
    u = np.zeros([Ny, Nx])

    nTerms = 100  # choose number of terms for finite sum in solution
    # Loop to solve for u at each (x, y) location
    for i in range(len(x)):  # loop over all x values
        for j in range(len(y)):  # loop over all y values
            v = 0  # set v function to zero initially
            for n in range(1, nTerms+1):  # loop over all eigvalues from 1 to N
                # calculate the Bn for the nth eigenvalue
                Bn = 2*(-1)**n / (n*pi)**3
                # add terms to v one at a time
                v = v + Bn * sin(n*pi*y[j]) * exp(-(n*pi)**2*x[i])
            U = -y[j]**3/6 + 7*y[j]/6  # solution to U(y)
            # Store result in the jth row, ith column
            u[j, i] = U + v  # Add solutions of U and v together to get u(x, y)
    return u


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. Shape 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


def main(Nx, Ny, method):  # Define main function to set up grid and A 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 = choose 'lu' or 'inv' matrix inversion

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

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

    # Calculate spacings dx and dy; these are constant with the uniform grid
    dx = x[1] - x[0]
    dy = y[1] - y[0]

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

    # Apply boundary conditions to u matrix

    # Set u(x, 0) to be 0 from Dirichlet 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, y) to be y from Dirichlet boundary condition
    u[:, 0] = y  # Set all rows, 0th column to be y

    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 = 1 (near bottom border of domain)
        A[0, 0] = 1/dx + 1/dy**2  # Assign value in matrix location [1, 1]
        A[0, 1] = -1/(2*dy**2)  # Assign value in matrix location [1, 2]

        # Assign right-hand side of equation (known values) for 0th value in b
        b[0] = (y[1] + 1/(2*dy**2)*u[0, i] + (-1/dy**2 + 1/dx)*u[1, i]
                + 1/(2*dy**2)*u[2, i] + 1/(2*dy**2)*u[0, i+1])

        # at j = Ny-2 (near top border of domain)
        A[-1, -1] = 1/dx + 1/dy**2  # Assign value to last diagonal element
        A[-1, -2] = -1/(2*dy**2)  # Assign value to left of last diag element

        # Assign right-hand side of equation (known values) for last value in b
        b[-1] = (y[-2] + 1/(2*dy**2)*u[-3, i] + (-1/dy**2 + 1/dx)*u[-2, i]
                 + 1/(2*dy**2)*u[-1, i] + 1/(2*dy**2)*u[-1, i+1])

        # # # Use 4th-Order-Accurate scheme for j = 2 to j = Ny-3
        # Store coefficients for second internal node
        A[1, 0] = -2/(3*dy**2)
        A[1, 1] = 5/(4*dy**2) + 1/dx
        A[1, 2] = -2/(3*dy**2)
        A[1, 3] = 1/(24*dy**2)
        b[1] = (y[2] + -1/(24*dy**2)*u[0, i] + 2/(3*dy**2)*u[1, i]
                + (1/dx-5/(4*dy**2))*u[2, i] + 2/(3*dy**2)*u[3, i]
                + (-1/(24*dy**2))*u[4, i] + (-1/(24*dy**2))*u[0, i+1])

        # Store coefficients for second-from-last internal node
        A[-2, -1] = -2/(3*dy**2)
        A[-2, -2] = 5/(4*dy**2) + 1/dx
        A[-2, -3] = -2/(3*dy**2)
        A[-2, -4] = 1/(24*dy**2)
        b[-2] = (y[-3] + -1/(24*dy**2)*u[-5, i] + 2/(3*dy**2)*u[-4, i]
                 + (1/dx-5/(4*dy**2))*u[-3, i] + 2/(3*dy**2)*u[-2, i]
                 + (-1/(24*dy**2))*u[-1, i] + (-1/(24*dy**2))*u[-1, i+1])

        # Loop over internal nodes to compute and store coefficients
        for j in range(2, Ny-4):
            A[j, j-2] = 1/(24*dy**2)
            A[j, j-1] = -2/(3*dy**2)
            A[j, j] = 5/(4*dy**2) + 1/dx
            A[j, j+1] = -2/(3*dy**2)
            A[j, j+2] = 1/(24*dy**2)
            b[j] = (y[j+1] + -1/(24*dy**2)*u[j-1, i] + 2/(3*dy**2)*u[j, i]
                    + (1/dx-5/(4*dy**2))*u[j+1, i] + 2/(3*dy**2)*u[j+2, i]
                    + (-1/(24*dy**2))*u[j+3, i])

        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

    # output is the u-matrix
    return u
    # End of main function


def makePlots(u):  # Display results spatially

    # Form x and y arrays
    Ny, Nx = np.shape(u)
    # Make linear-spaced 1D array of x-values from 0 to 1 with Nx elements
    x = np.linspace(0, 1, Nx)
    # Make linear-spaced 1D array of y-values from 0 to 1 with Ny elements
    y = np.linspace(0, 1, Ny)

    # Create contour plot of u vs x and y
    plt.figure(figsize=(6, 4))
    plt.contourf(x, y, u, cmap='plasma', levels=np.linspace(0., 1., 11))
    cbar = plt.colorbar()
    fs = 17  # Define font size for figures
    fn = 'Calibri'  # Define font for figures
    plt.xlabel('x', fontsize=fs, fontname=fn, fontweight='bold')
    plt.ylabel('y' + '     ', fontsize=fs, rotation=0, fontname=fn,
               fontweight='bold')
    plt.xticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    plt.yticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    cbar.ax.set_ylabel('    u', rotation=0, fontname=fn, fontsize=fs,
                       weight='bold')
    cbar.ax.set_yticklabels([round(cbar.get_ticks()[n], 2)
                            for n in range(len(cbar.get_ticks()))],
                            fontsize=fs-2, fontname=fn, weight='bold')
    # plt.savefig("contour"+str(Nx)+'_'+str(Ny)+".png", dpi=320,
    #             bbox_inches='tight')  # save figure
    # plt.close()  # close figure

    # Look at solution at selected x-locations
    xLocs = [0.05, 0.1, 0.2, 0.5, 1.0]
    lines = ['y-', 'r-.', 'm--', 'b:', 'k-']  # define line styles
    # Loop for each x-location to make plots at each location comparing exact
    # solution with analytical solution
    legStr = []  # initialize value to store strings for legend
    plt.figure()  # create new figure
    for n in range(len(xLocs)):
        col = np.argmin(abs(x-xLocs[n]))  # extract value closest to given xLoc
        plt.plot(u[:, col], y, lines[n])
        legStr.append('x = {}'.format(xLocs[n]))  # append value to leg string
    plt.legend(legStr, fontsize=fs-2)
    plt.xlabel('u', fontsize=fs, fontname=fn, fontweight='bold')
    plt.ylabel('y' + '     ', fontsize=fs, rotation=0, fontname=fn,
               fontweight='bold')
    plt.xticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    plt.yticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    # plt.savefig("curves"+str(Nx)+'_'+str(Ny)+".png", dpi=320,
    #             bbox_inches='tight')  # save figure
    # plt.close()  # close figure
    # End of makePlots function


def errorPlots(u, u_an):  # function to plot error in u vs u_an
    # Form x and y arrays
    Ny, Nx = np.shape(u)
    # Make linear-spaced 1D array of x-values from 0 to 1 with Nx elements
    x = np.linspace(0, 1, Nx)
    # Make linear-spaced 1D array of y-values from 0 to 1 with Ny elements
    y = np.linspace(0, 1, Ny)
    # Create contour plot of error (u-u_an) vs x and y
    plt.figure(figsize=(6, 4))
    cmax = round(np.amax(abs(u-u_an)), 3)  # maximum absolute contour value
    lev = np.linspace(0, cmax, 11)
    plt.contourf(x, y, abs(u-u_an), cmap='plasma',
                 levels=lev)
    cbar = plt.colorbar()  # make colorbar
    fs = 17  # Define font size for figures
    fn = 'Calibri'  # Define font for figures
    # Label axes and set tick styles
    plt.xlabel('x', fontsize=fs, fontname=fn, fontweight='bold')
    plt.ylabel('y' + '     ', fontsize=fs, rotation=0, fontname=fn,
               fontweight='bold')
    plt.xticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    plt.yticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    cbar.ax.set_ylabel('                   |${u-u_{an}}$|', rotation=0,
                       fontname=fn, fontsize=fs, weight='bold')
    cbar.ax.set_yticklabels([round(cbar.get_ticks()[n], 4)
                            for n in range(len(cbar.get_ticks()))],
                            fontsize=fs-2, fontname=fn, weight='bold')
    # plt.savefig("err_contour"+str(Nx)+'_'+str(Ny)+".png", dpi=320,
    #             bbox_inches='tight')  # save figure
    # plt.close()  # close figure

    # Look at error in solution at selected x-locations
    xLocs = [0.05, 0.1, 0.2, 0.5, 1.0]
    lines = ['y-', 'r-.', 'm--', 'b:', 'k-']
    # Loop for each x-location to make plots at each location comparing exact
    # solution with analytical solution
    legStr = []
    plt.figure()  # create new figure
    for n in range(len(xLocs)):
        col = np.argmin(abs(x-xLocs[n]))  # extract value closest to given xLoc
        plt.plot(abs(u[:, col]-u_an[:, col]), y, lines[n])
        legStr.append('x = {}'.format(xLocs[n]))

    plt.legend(legStr, fontsize=fs-2)
    plt.xlabel('|${u-u_{an}}$|', fontsize=fs, fontname=fn, fontweight='bold')
    plt.ylabel('y' + '     ', fontsize=fs, rotation=0, fontname=fn,
               fontweight='bold')
    plt.xticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    plt.yticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    # plt.savefig("err_curves"+str(Nx)+'_'+str(Ny)+".png", dpi=320,
    #             bbox_inches='tight')  # save figure
    # plt.close()  # close figure
    # End of errorPlots function


def plotDxDyEffects():  # function to plot effects of dx and dy spacings

    # Data collected on variation of dx values (with dy constant at 0.05)
    dxes = [0.05, 0.04, 0.0333333, 0.025, 0.02, 0.0166667, 0.0125, 0.01, 0.005,
            0.0025, 0.00166667, 0.001]  # dx values themselves
    dxerrs = [0.00276807, 0.00222227, 0.00183598, 0.00132455, 0.00101785,
              0.000861093, 0.000647488, 0.000507671, 0.00019631, 3.41267e-05,
              1.97199e-05, 3.37169e-05]  # maximum error in u
    dxt = [0.3222, 0.343764, 0.406265, 0.406255, 0.499992, 0.499991, 0.609383,
           0.671899, 0.843771, 1.70315, 2.55095, 4.21325]  # computation time
    # Create plot to show effects of dx on error and computation time
    plt.figure()
    plt.loglog(dxes, dxerrs, 'ko', dxes, dxt, 'bx')  # plot on log-log axes
    fs = 17  # Define font size for figures
    fn = 'Calibri'  # Define font for figures
    # Label axes and set tick styles
    plt.xlabel('$\Delta$ x', fontsize=fs, fontname=fn, fontweight='bold')
    # plt.ylabel('y' + '     ', fontsize=fs, rotation=0, fontname=fn,
    #            fontweight='bold')
    plt.xticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    plt.yticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    plt.legend(['Absolute error in u', 'Computation time [s]'])
    # plt.savefig("dxEffect.png", dpi=320, bbox_inches='tight')

    # Data collected on variation of dy values (with dx constant at 0.05)
    # dy values themselves
    dyes = [0.05, 0.04, 0.0333333, 0.025, 0.02, 0.0166667, 0.0125, 0.01, 0.005,
            0.0025, 0.00166667, 0.001]
    # maximum error in u
    dyerrs = [0.00276807, 0.00278632, 0.00278221, 0.00278486, 0.00279274,
              0.00279906, 0.002803, 0.00280316, 0.00280319, 0.00280319,
              0.0028033, 0.00280335]
    # computation time
    dyt = [0.0937653, 0.109376, 0.140636, 0.171877, 0.218763, 0.265623,
           0.34375, 0.425119, 0.859382, 1.73438, 2.61386, 4.33241]
    # Create plot to show effects of dy on error and computation time
    plt.figure()
    plt.loglog(dyes, dyerrs, 'ko', dyes, dyt, 'bx')  # plot on log-log axes
    # Label axes and set tick styles
    plt.xlabel('$\Delta$ y', fontsize=fs, fontname=fn, fontweight='bold')
    # plt.ylabel('y' + '     ', fontsize=fs, rotation=0, fontname=fn,
    #            fontweight='bold')
    plt.xticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    plt.yticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    plt.legend(['Absolute error in u', 'Computation time [s]'])
    # plt.savefig("dyEffect.png", dpi=320, bbox_inches='tight')

    # Data collected on varying both dx and dy together
    dxdyes = [0.05, 0.04, 0.0333333, 0.025, 0.02, 0.0166667, 0.0125, 0.01,
              0.005, 0.0025, 0.00166667, 0.001]  # dx and dy values themselves
    dxdyerrs = [0.00276807, 0.00218433, 0.00185296, 0.00139747, 0.00110844,
                0.000932838, 0.00069616, 0.000560447, 0.000280287, 0.000140149,
                9.34256e-05, 5.60506e-05]
    dxdyt = [0.0937514, 0.140621, 0.187512, 0.343739, 0.536466, 0.75001,
             1.34378, 2.09376, 8.223, 32.7938, 77.9015, 207.268]
    # Create plot to show effects of dy on error and computation time
    plt.figure()
    # plot on log-log axes
    plt.loglog(dxdyes, dxdyerrs, 'ko', dxdyes, dxdyt, 'bx')
    # Label axes and set tick styles
    plt.xlabel('$\Delta$ x = $\Delta$ y', fontsize=fs, fontname=fn,
               fontweight='bold')
    plt.xticks(fontsize=fs-2, fontname=fn, fontweight='bold')
    plt.yticks(fontsize=fs-2, fontname=fn, fontweight='bold')
```

```
    plt.legend(['Absolute error in u', 'Computation time [s]'])
    # plt.savefig("dxdyEffect.png", dpi=320, bbox_inches='tight')  # save fig
    # End of plotDxDyEffects function


# Run functions in order
t0 = time.time()  # begin timer
Nx = 21  # number of nodes in x-direction
Ny = 21  # number of nodes in y-direction

# execute main numerical solver to calculate u, use LU solver
u = main(Nx, Ny, 'lu')

# execute analytic solver to calculate u_analytic
u_an = analytic(Nx, Ny)

# run function to make plots of results for u
makePlots(u)

# run function to make plots of errors compared to analytic solution
errorPlots(u, u_an)

# calculate difference in time from current to when code started (elapsed time)
elapsed = time.time() - t0

# run function to make plots showing the effects of dx and dy spacing
plotDxDyEffects()

print('dx = {0:0.6}'.format(1/(Nx-1)))  # print current dx spacing
print('dy = {0:0.6}'.format(1/(Ny-1)))  # print current dy spacing
print('Max error in u is {0:0.6}'.format(np.amax(abs(u-u_an))))  # print error
print('Elapsed time is {0:0.4}'.format(elapsed) + ' s.')  # print elapsed time
```

Appendix 2: Derivation of Analytical Solution

PDE: $\frac{\partial u}{\partial x} - \frac{\partial^2 u}{\partial y^2} = y$

BCs: $u(x,0) = 0$

$u(x,1) = 1$

$u(0,y) = y$

Make problem homogeneous (Superposition)

$u(x,y) = U(y) + v(x,y)$

$$\frac{\partial v}{\partial x} - \frac{d^2 U}{dy^2} - \frac{\partial^2 v}{\partial y^2} = y$$

Let $\frac{d^2U}{dy^2} = -y \rightarrow \frac{dU}{dy} = \frac{-y^2}{2} + c_1 \rightarrow U(y) = \frac{-y^3}{6} + c_1 y + c_2$

Choose $\begin{cases} U(0) = 0 \rightarrow c_2 = 0 \\ U(1) = 1 = \frac{-1}{6} + c_1 \rightarrow \underline{c_1 = \frac{7}{6}} \end{cases}$ $\Big\}\ U(y) = \frac{-y^3}{6} + \frac{7}{6}y$

Now determine boundary conditions for v

$u(x,0) = U(0) + v(x,0) = 0 \longrightarrow \underline{v(x,0) = 0}$

$u(x,1) = U(1) + v(x,1) = 1 \longrightarrow \underline{v(x,1) = 0}$

$u(0,y) = U(y) + v(0,y) = y$

$\underline{v(0,y) = \frac{y^3}{6} - \frac{y}{6} = \frac{y}{6}(y^2-1)}$

PDE for v: $\frac{\partial v}{\partial x} - \frac{\partial^2 v}{\partial y^2} = 0$

Use separation of variables.

Assert that $v(x,y) = F(x)\,G(y)$

Substitute $\frac{\partial}{\partial x}\big(F(x)G(y)\big) = \frac{\partial^2}{\partial y^2}\big(F(x)G(y)\big)$

$$G(y)\frac{dF}{dx} = F(x)\frac{d^2G}{dy^2}$$

$$\frac{1}{F(x)}\frac{dF}{dx} = \frac{1}{G(y)}\frac{d^2G}{dy^2} = -\lambda^2 = \text{constant}$$

We now have two ODEs:

$$\frac{d^2G}{dy^2} + \lambda^2 G = 0 \quad \text{and} \quad \frac{dF}{dx} + \lambda^2 F = 0$$

General solutions of each:

$$G(y) = c_1\cos(\lambda y) + c_2\sin(\lambda y) \qquad F(x) = c_3\exp(-\lambda^2 x)$$

Combine solutions for general solution to v

$$v = FG = [A\cdot\cos(\lambda y) + B\cdot\sin(\lambda y)]\exp(-\lambda^2 x)$$

Use BC: $v(x,0) = 0 = [A\cdot\cos(0) + B\,\cancel{\sin(0)}^{0}]\exp(-\lambda^2 x)$

$\hookrightarrow A = 0$

Now $v(x,y) = B\sin(\lambda y)\exp(-\lambda^2 x)$

Use BC: $v(x,1) = 0 = B\sin(\lambda)\exp(-\lambda^2 x)$

For non-trivial solution, $\sin(\lambda) = 0 \longrightarrow \lambda_n = n\pi, \quad n = 1, 2, 3, \ldots$

Presume that the solution is a linear combination of all possible solutions

$$v(x,y) = \sum_{n=1}^{\infty} B_n \sin(\lambda_n y)\exp(-\lambda_n^2 x)$$

Use last BC to solve for unknown B_n values

$$v(0,y) = \frac{y}{6}(y^2-1) = \sum_{n=1}^{\infty} B_n \sin(n\pi y) \cancel{\exp(0)}^{1}$$

$$\frac{y}{6}(y^2-1) = \sum_{n=1}^{\infty} B_n \sin(n\pi y)$$

multiply by $\sin(m\pi y)$ and convert sum into integral

$$\int_0^1 \frac{y}{6}(y^2-1)\sin(m\pi y)\,dy = B_n \underbrace{\int_0^1 \sin(n\pi y)\sin(m\pi y)\,dy}$$

Orthogonality: integral is zero unless $n=m$

$$\int_0^1 \frac{y}{6}(y^2-1)\sin(n\pi y)\,dy = B_n \int_0^1 \sin^2(n\pi y)\,dy$$

Evaluate each integral

$$\frac{1}{6}\int_0^1 \left[y^3 \sin(n\pi y) - y\sin(n\pi y)\right]dy$$

$$= \left[\frac{\left[\pi^2 n^2(3y^2-1)-6\right]\sin(n\pi y) + n\pi y\left(6-(n\pi)^2(y^2-1)\right)\cos(n\pi y)}{6n^4\pi^4}\right]_0^1$$

$$= \frac{(\pi^2 n^2 - 3)\cancel{\sin(n\pi)}^{0} + 3n\pi \cancel{\cos(n\pi)}^{\substack{1 \text{ if } n \text{ is even} \\ -1 \text{ if } n \text{ is odd}}}}{3n^4\pi^4}$$

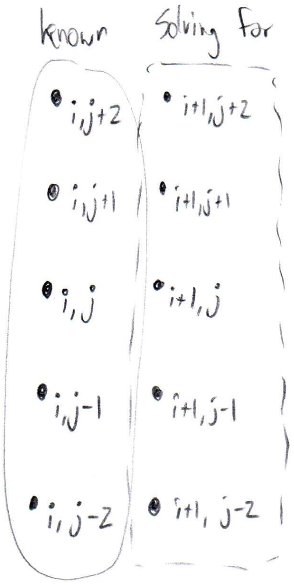
$$= \frac{(-1)^n}{(n\pi)^3}$$

$$\int_0^1 \sin^2(n\pi y)\,dy = \left[\frac{y}{2} - \frac{\sin(2n\pi y)}{4\pi n}\right]_0^1 = \frac{1}{2} - \frac{\sin(2n\pi)\to 0}{4n\pi} = \frac{1}{2}$$

$$\therefore B_n = \frac{2\cdot(-1)^n}{(n\pi)^3}$$

$$u(x,y) = U(y) + v(x,y) = \boxed{\frac{-y^3}{6} + \frac{7y}{6} + \sum_{n=1}^{\infty}\frac{2(-1)^n}{(n\pi)^3}\sin(n\pi y)\exp(-(n\pi)^2 x)}$$

Appendix 3: Derivation of Crank-Nicolson Algorithm with 4th-Order accuracy in y



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

2nd order $$\left.\frac{\partial^2 u}{\partial y^2}\right|_{i,j} \approx \frac{u_{i,j-1}-2u_{i,j}+u_{i,j+1}}{(\Delta y)^2}$$

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

$$\frac{\partial u}{\partial x} - \frac{\partial^2 u}{\partial y^2} - y = 0$$

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

$$-y_{i,j} \approx 0$$

$$\frac{u_{i+1,j}}{\Delta x} - \frac{u_{i,j}}{\Delta x} + \left\{ \frac{u_{i,j-2}}{24\Delta y^2} - \frac{2u_{i,j-1}}{3\Delta y^2} + \frac{5u_{i,j}}{4\Delta y^2} - \frac{2u_{i,j+1}}{3\Delta y^2} + \frac{u_{i,j+2}}{24\Delta y^2} \right\} + \left\{ \frac{u_{i+1,j-2}}{24\Delta y^2} - \frac{2}{3}\frac{u_{i+1,j-1}}{\Delta y^2} + \frac{5u_{i+1,j}}{4\Delta y^2} - \frac{2u_{i+1,j+1}}{3\Delta y^2} + \frac{u_{i+1,j+2}}{24\Delta y^2} \right\} - y_{i,j} = 0$$

$$\underbrace{\left(\frac{1}{24\Delta y^2}\right)}_{a_j} u_{i+1,j-2} + \underbrace{\left(\frac{-2}{3\Delta y^2}\right)}_{b_j} u_{i+1,j-1} + \underbrace{\left(\frac{5}{4\Delta y^2} + \frac{1}{\Delta x}\right)}_{c_j} u_{i+1,j} + \underbrace{\left(\frac{-2}{3\Delta y^2}\right)}_{d_j} u_{i+1,j+1} + \underbrace{\left(\frac{1}{24\Delta y^2}\right)}_{e_j} u_{i+1,j+2}$$

$$= \underbrace{y_{i,j} + \left(\frac{-1}{24\Delta y^2}\right)u_{i,j-2} + \left(\frac{2}{3\Delta y^2}\right)u_{i,j-1} + \left(\frac{1}{\Delta x} - \frac{5}{4\Delta y^2}\right)u_{i,j} + \left(\frac{2}{3\Delta y^2}\right)u_{i,j+1} + \left(\frac{-1}{24\Delta y^2}\right)u_{i,j+2}}_{f_j}$$

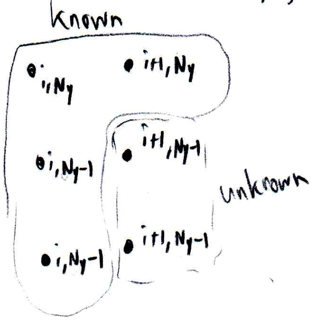
$$\begin{bmatrix} \ddots & & & & & \\ a_j & b_j & c_j & d_j & e_j & \\ & a_{j+1} & b_{j+1} & c_{j+1} & d_{j+1} & e_{j+1} \\ & & & & & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ u_{i+1,j-2} \\ u_{i+1,j-1} \\ u_{i+1,j} \\ u_{i+1,j+1} \\ u_{i+2,j+2} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\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

2nd-Order for first interior points

$$\frac{u_{i+1,j} - u_{i,j}}{\Delta x} - \frac{1}{2}\left\{\left[\frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2}\right] + \left[\frac{u_{i+1,j-1} - 2u_{i+1,j} + u_{i+1,j+1}}{\Delta y^2}\right]\right\} - y_{i,j} = 0$$

$$\left(\frac{-1}{2\Delta y^2}\right)u_{i+1,j-1} + \left(\frac{1}{\Delta x} + \frac{1}{\Delta y^2}\right)u_{i+1,j} + \left(\frac{-1}{2\Delta y^2}\right)u_{i+1,j+1}$$

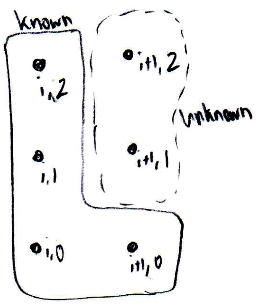
$$= y_{i,j} + \left(\frac{1}{2\Delta y^2}\right)u_{i,j-1} + \left(\frac{-1}{\Delta y^2} + \frac{1}{\Delta x}\right)u_{i,j} + \left(\frac{1}{2\Delta y^2}\right)u_{i,j+1}$$



If $j = N_y - 1$:

$$\left(\frac{-1}{2\Delta y^2}\right)u_{i+1,N_y-2} + \left(\frac{1}{\Delta x} + \frac{1}{\Delta y^2}\right)u_{i+1,N_y-1} =$$

$$y_{i,N_y-1} + \left(\frac{1}{2\Delta y^2}\right)u_{i,N_y-2} + \left(\frac{-1}{\Delta y^2} + \frac{1}{\Delta x}\right)u_{i,N_y-1} + \left(\frac{1}{2\Delta y^2}\right)u_{i,N_y} + \left(\frac{1}{2\Delta y^2}\right)u_{i+1,N_y}$$



If $j = 1$

$$\left(\frac{1}{\Delta x} + \frac{1}{\Delta y^2}\right)u_{i+1,1} + \left(\frac{-1}{2\Delta y^2}\right)u_{i+1,2}$$

$$= y_{i,1} + \left(\frac{1}{2\Delta y^2}\right)u_{i,0} + \left(\frac{-1}{\Delta y^2} + \frac{1}{\Delta x}\right)u_{i,1} + \left(\frac{1}{2\Delta y^2}\right)u_{i,2} + \left(\frac{1}{2\Delta y^2}\right)u_{i+1,0}$$