ME317 - Computational Fluid Dynamics
Assignment 2
Faculty: Dr. A Perumal
Name: Drishika Nadella
Number: 181ME222
Date of submission: 10/04/2021

ME317 Computational Fluid Dynamics

Assignment 2 - Drishika Nadella (181ME222)

This assignment has been solved using the Anaconda Jupyter Notebook Environment in Python3.

Question 1

The upward velocity of a rocket is given at three different times in Table 1.

Time t (s)	Velocity v (m/s)
5	106.8
8	177.2
12	279.2

The velocity data is approximated as a polynomial as:

$$v(t) = a_1 t^2 + a_2 t + a_3$$

where $5 \leq t \leq 12$.

The coefficients a_1, a_2 and a_3 for the above given expression are given as:

$$\begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 106.8 \\ 177.2 \\ 279.2 \end{bmatrix}$$

Find the velocity at t = 6 seconds.

In [1]:

```
# Importing the necessary libraries
import numpy as np
import sys
import timeit
```

Let us first solve this problem analytically, so we know what answers to expect.

In [2]:

```
# Initializing the variables

# Initializing the coefficient matrix
A = np.array([[25, 5, 1], [64, 8, 1], [144, 12, 1]])

#Initializing the RHS matrix
B = np.array([106.8, 177.2, 279.2])
```

In [3]:

```
# Solving the system of linear equations analytically

final = np.linalg.inv(A).dot(B)
print(final)
```

```
[ 0.29047619 19.69047619  1.08571429]
```

Therefore, the expected solutions are $a_1 = 0.29047619$, $a_2 = 19.69047619$ and $a_3 = 1.08571429$.

Now, solving the system of equations numerically:

Let us try to solve this problem using the Gauss-Seidel method.

Gauss-Seidel Method

According to the Gauss Seidel method, the system of equations will be as follows:

$$a_1^{k+1} = \frac{106.8 - 5a_2^k - a_3^k}{25}$$
$$a_2^{k+1} = \frac{177.2 - 64a_1^{k+1} - a_3^k}{8}$$
$$a_3^{k+1} = \frac{279.2 - 144a_1^{k+1} - 12a_2^{k+1}}{1}$$

We use the most recent iteration of the three variables while finding the values for the other variables.

In [4]:

```
# Initializing the variables

nt = 50                                          # Maximum number of iterations possible
error = 1E-6                                     # Maximum permissible error
c = [[25, 5, 1], [64, 8, 1], [144, 12, 1]]       # Coefficient matrix
d = [106.8, 177.2, 279.2]                        # RHS Matrix
a = [1, 2, 5]                                    # Initial guesses for the answers a1, a2
and a3
```

In [5]:

```
def GaussSeidel(nt, error, a, c, d):

    """
    This function solves the system of linear equations using the Gauss Seidel method a
s explained above.
    Here, nt = maximum number of iterations
    error = maximum permissible error of the answers to the true values
    a = matrix for the initial guesses of a1, a2, a3
    c = coefficient matrix
    d = RHS matrix
    The function prints out the a1, a2, a3 values every 10 iterations.
    """

    for n in range(nt+1):

        # x, y, z store the old values of a1, a2, a3
        x = a[0]
        y = a[1]
        z = a[2]

        # Performing Gauss Seidel iteration for all three variables
        # Note how a2 calculation uses the most recent a1, and a3 calculation uses the
 most recent a1 and a2.
        a[0] = (d[0] - c[0][1]*y - c[0][2]*z)/c[0][0]
        a[1] = (d[1] - c[1][0]*a[0] - c[1][2]*z)/c[1][1]
        a[2] = (d[2] - c[2][0]*a[0] - c[2][1]*a[1])/c[2][2]

        # Checking for convergence
        if max(abs(x-a[0]), abs(y-a[1]), abs(z-a[2])) < error:
            print("\n")
            print("Finally, a1 = %f, a2 = %f, a3 = %f"% (a[0], a[1], a[2]))
            print("It took %d iterations"% (n+1))
            break

        # Prints out the values every 10 iterations
        elif (n+1)%10==0:
            print("Iteration number: ", n+1)
            print("a1 = %f \n a2 = %f \n a3 = %f \n"% (a[0], a[1], a[2]))
            print("\n")
```

In [6]:

```
# Calling the Gauss Seidel function
GaussSeidel(nt, error, a, c, d)

# Printing the expected analytical result
print("Expected result: a1 = %f \n a2 = %f \n a3 = %f"%(final[0], final[1], final[2]))
```

```
Iteration number:  10
a1 = 988378.793288
 a2 = -5671869.896305
 a3 = -74263828.277822



Iteration number:  20
a1 = 1509210001999.948975
 a2 = -8660719631421.269531
 a3 = -113397604710658.234375



Iteration number:  30
a1 = 2304496976556495104.000000
 a2 = -13224536134145269760.000000
 a3 = -173153131014392086528.000000



Iteration number:  40
a1 = 3518865040598500695343104.000000
 a2 = -20193282245095887279751168.000000
 a3 = -264397178905033435592523776.000000



Iteration number:  50
a1 = 537315140783944686477941879603 2.000000
 a2 = -308342495868161955914789179883 52.000000
 a3 = -403722807687086019444887800250368.000000



Expected result: a1 = 0.290476
 a2 = 19.690476
 a3 = 1.085714
```

We can see that even after 50 iterations, the values diverge drastically from the expected result and the errors compound. This is because Gauss-Seidel solver is an iterative solver. Iterative solvers require the "diagonal dominance" condition to be satisfied. The diagonal dominance condition is:

- The coefficient of the diagonal of the matrix must be at least equal to the sum of the other coefficients in that row
- At least one row with a diagonal coefficient greater than the sum of the other coefficients in that row

Let us test the diagonal dominance condition on the rows of the coefficient matrix.

- Row 1

$25 \geq 5 + 1$ condition is satisfied.

- Row 2

$8 \geq 64 + 1$ condition is not satisfied.

- Row 3

$1 \geq 144 + 12$ condition is not satisfied.

Therefore, rows 2 and 3 fail the first condition in the diagonal dominance requirement. Therefore, iterative solvers such as the Gauss-Seidel method are **not valid** for the problem.

We need to use direct solvers to solve the problem.

Gauss Elimination Method

The Gauss Elimination method is a type of direct solver that obtains the solutions to a system of linear equations. Since it is a direct solver, it does not require diagonal dominance.

The steps involved in the Gauss Elimination method are:

- Swapping two rows
- Multiplying a row by a nonzero number
- Adding a multiple of one row to another row

These operations are performed until the lower left-hand corner of the matrix is filled with zeros, as much as possible.

In [7]:

```
def Gauss(a, n):
    """
    This function solves a set of linear equations using the Gauss Elimination method.
    Here, a = augmented matrix that contains the coefficient and the RHS matrices
    n = number of unknowns
    The function returns the three unknowns a1, a2, a3
    """
    # Defining the array that will contain the solution
    x = np.zeros(n)

    # Gaussian Elimination method
    for i in range(n):
        for j in range(i+1, n):
            ratio = a[j][i]/a[i][i]

            for k in range(n+1):
                a[j][k] = a[j][k] - ratio * a[i][k]

    x[n-1] = a[n-1][n]/a[n-1][n-1]

    # Back substitution
    for i in range(n-2,-1,-1):
        x[i] = a[i][n]

        for j in range(i+1,n):
            x[i] = x[i] - a[i][j]*x[j]

        x[i] = x[i]/a[i][i]

    return x
```

In [8]:

```
init = np.array([[25, 5, 1, 106.8], [64, 8, 1, 177.2], [144, 12, 1, 279.2]])
results = Gauss(init, 3)
print("The final results are: a1 = %f, a2 = %f, a3 = %f"% (results[0], results[1], resu
lts[2]))
```

```
The final results are: a1 = 0.290476, a2 = 19.690476, a3 = 1.085714
```

We can see that the above results we obtained with the Gauss Elimination method gave us the same results as the analytical method. Now, we need to find the value of velocity v at time t = 6 seconds.

In [9]:

```
t = 6
v = results[0]*t**2 + results[1]*t + results[2]
print("The velocity v at time %d seconds is %f m/s"%(t,v))
```

```
The velocity v at time 6 seconds is 129.685714 m/s
```

Computing CPU time to execute the code

The CPU time to execute the code can be done using the timeit module in Python. It requires me to put the code whose execution time needs to be computed in a string. The final result is printed below.

In [10]:

```
mysetup = "import numpy as np"

mycode = """
def Gauss(a, n):
    x = np.zeros(n)

    for i in range(n):
        for j in range(i+1, n):
            ratio = a[j][i]/a[i][i]

            for k in range(n+1):
                a[j][k] = a[j][k] - ratio * a[i][k]

    x[n-1] = a[n-1][n]/a[n-1][n-1]

    for i in range(n-2,-1,-1):
        x[i] = a[i][n]

        for j in range(i+1,n):
            x[i] = x[i] - a[i][j]*x[j]

        x[i] = x[i]/a[i][i]

    return x

init = np.array([[25, 5, 1, 106.8], [64, 8, 1, 177.2], [144, 12, 1, 279.2]])
results = Gauss(init, 3)
print(results)
"""

print("The time taken for the Gauss Elimination code to execute is: %f"% (timeit.timeit
(setup = mysetup, stmt = mycode, number = 1)))
```

```
[ 0.29047619 19.69047619  1.08571429]
The time taken for the Gauss Elimination code to execute is: 0.000609
```

Results and Discussion

We can see that the velocity at time t=6 seconds is 129.685714 m/s.

Therefore, we can see that for the given question, the Gauss-Seidel code is invalid as it does not satisfy the diagonal dominance condition, but the Gauss Elimination code is **valid**, since it is a direct solver that does not require the diagonal dominance condition to be satisfied.

We checked for the validity of the code by comparing the coefficient values obtained by the Gauss Elimination method with the coefficient values obtained by the analytical method at the beginning. They both match, thereby confirming the validity of the code.

The Gauss Elimination algorithm took around 0.0006 seconds to execute.

Question 2

The temperature T(x) that arises due to steady-state heat conduction in a bar 30 cm long is governed by the following ODE, if uniform temperature is assumed across any cross section:

$$\frac{\partial^2 T}{\partial x^2} - GT = 0$$

where T is the temperature difference from the ambient medium, which is at 20°C, x is the axial coordinate distance, and G is a constant that depends on the surface heat transfer rate. This equation may be replaced by a finite difference approximation, using the second order central difference scheme, as

$$(T_{i+1} - 2T_i + T_{i-1})/\Delta x^2 = GT_i$$

where x = $i\Delta x$. Considering 30 subdivisions of the length of the rod, with $\Delta x = 1cm$ find the temperature differences T_i, where i = 1, 2, . . ., 29.

The temperatures differences T_0 and T_{30}, at x = 0 and x = 30 cm, respectively, are given as 100°C, and the constant G as $(0.071)^2 cm^{-2}$.

Write a code and solve using Thomas Algorithm. Present your results both in tabular form and in graphical form (T(x) vs x). Compare the numerical results T(x) vs x with analytical results (Validation). Comment on the validity of the developed code.

In [3]:

```
# Importing the necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import math
import pandas as pd
import timeit
```

In [4]:

```
nx=31          #number of points
dx=30/(nx-1)   #division of points
T=20           #ambient temperature
G=(0.071)**2   #constant

xarr=np.linspace(0,30,nx) #creating the x array from 0 to 30
print(xarr)

T_comp=np.zeros(nx) #creating an array for the numerical solution
T_comp[0]=100       #giving initial conditions
T_comp[-1]=100
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17.
 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30.]
```

The Analytical Solution

$$\frac{\partial^2 T}{\partial x^2} - GT = 0$$

Comparing it with the equation: $a\frac{\partial^2 y}{\partial x^2} + b\frac{\partial y}{\partial x} + cy = 0$

Here, $a = 1$, $b = 0$ and $c = -G = -0.005041$.

Solving the quadratic equation:

$$ar^2 + br + c = 0$$

i.e.

$$r^2 - 0.005041 = 0$$

we get $r = 0.071$ and $r = -0.071$.

The solution becomes $T = c_1 e^{0.071x} + c_2 e^{-0.071x}$.

Given initial conditions are $y(0) = 100$ and $y(30) = 100$.

Substituting the initial conditions in the solution, we get $c_1 = 10.6215$ and $c_2 = 89.3785$.

Therefore, the solution is

$$T = 10.6215e^{0.071x} + 89.3785e^{-0.071x}$$

In [5]:

```
#Analytical solution

T_analytical = np.asarray([10.6215*math.exp(0.071*x)+89.3785*math.exp(-0.071*x) for x i
n xarr])
print(T_analytical)        #array of analytical solutions from x=0 to x=30
```

```
[100.          94.65570972  89.78877935  85.37466438  81.39110393
  77.81800841  74.6373583   71.83311319  69.39113094  67.29909636
  65.54645906  64.12438028  63.02568833  62.24484235  61.77790446
  61.62251984  61.77790485  62.24484312  63.02568949  64.12438184
  65.54646102  67.29909873  69.39113374  71.83311642  74.63736199
  77.81801258  81.39110859  85.37466956  89.78878507  94.65571602
 100.00000691]
```

Numerical Solution using TDMA:

Discretizing the equation as Central Difference in space, we get:

$$(T_{i+1} - 2T_i + T_{i-1})/\Delta x^2 = GT_i$$

where $\Delta x = 1$ and $G = 0.071^2$. This gives:

$$T_{i+1} - (2 + G)T_i + T_{i-1} = 0$$

Comparing this equation with the following equation:

$$aT_{i+1} + bT_i + cT_{i-1} = d$$

$a = 1, b = -(2 + G), c = 1, d = 0$ for the interior nodes. For the exterior nodes, $d = 100$.

Let A be the lower diagonal matrix, B is the main diagonal matrix, C is the upper diagonal matrix and D is the right hand side of the system. $a_0 = 0$ and $c_30 = 0$ i.e. $A[0] = 0$ and $C[30] = 0$.

Applying the boundary conditions, $T_0 = 100$, so $c_0 = 0$ and $b_0 = 1$ and $T_{30} = 100$, so $a_{30} = 0$ and $b_{30} = 1$.

In [6]:

```
#Implicit numerical solution using TDMA

#Lower Diagonal matrix
a=[1]*29+[0]

#Middle Diagonal matrix
b=[1]+[-2-G]*29+[1]

#Upper Diagonal Matrix
c=[0]+[1]*29

#Right hand side of the system
d=[100]+[0]*29+[100]

def TDMAAlgo(a,b,c,d):
    """
    This function solves a set of equations for the temperature distribution of a 1-D a
rray using the Thomas Algorithm.

    Here, a is the lower diagonal of the coefficient matrix.
    b is the main diagonal of the coefficient matrix.
    c is the upper diagonal of the coefficient matrix.
    d is the RHS column matrix.

    This function solves the set of equations using TDMA, which uses a series of forwar
d elimination and back substitutions.
    It finally returns p, the solved set of temperatures.
    """
    n = len(d)                  # Finds the total number of equations

    # Initializing some arrays
    w = np.zeros(n-1,float)
    g = np.zeros(n, float)
    p = np.zeros(n,float)

    # Steps to make the main diagonal element 1
    w[0] = c[0]/b[0]
    g[0] = d[0]/b[0]

    # Steps to make the lower diagonal element 0
    for i in range(1,n-1):
        w[i] = c[i]/(b[i] - a[i-1]*w[i-1])
    for i in range(1,n):
        g[i] = (d[i] - a[i-1]*g[i-1])/(b[i] - a[i-1]*w[i-1])

    # Getting the final RHS matrix element
    p[n-1] = g[n-1]

    # Back substitution
    for i in range(n-1,0,-1):
        p[i-1] = g[i-1] - w[i-1]*p[i]
    return p

T_comp=TDMAAlgo(a,b,c,d)
print(T_comp)
```

```
[100.          94.65738374  89.79193535  85.37912811  81.39671705
  77.82462684  74.64485058  71.84135901  69.40001973  67.30852595
  65.55633444  64.13461242  63.03619299  62.255539    61.78871518
  61.63336827  61.78871518  62.255539    63.03619299  64.13461242
  65.55633444  67.30852595  69.40001973  71.84135901  74.64485058
  77.82462684  81.39671705  85.37912811  89.79193535  94.65738374
 100.        ]
```

Let us print the output values in tabular form for better visualization.

In [7]:

```
df = pd.DataFrame(list(zip(T_analytical, T_comp)), index=range(0,31), columns =['Analyt
ical Values', 'Numerical Values'])
print(df)
```

```
    Analytical Values  Numerical Values
0          100.000000        100.000000
1           94.655710         94.657384
2           89.788779         89.791935
3           85.374664         85.379128
4           81.391104         81.396717
5           77.818008         77.824627
6           74.637358         74.644851
7           71.833113         71.841359
8           69.391131         69.400020
9           67.299096         67.308526
10          65.546459         65.556334
11          64.124380         64.134612
12          63.025688         63.036193
13          62.244842         62.255539
14          61.777904         61.788715
15          61.622520         61.633368
16          61.777905         61.788715
17          62.244843         62.255539
18          63.025689         63.036193
19          64.124382         64.134612
20          65.546461         65.556334
21          67.299099         67.308526
22          69.391134         69.400020
23          71.833116         71.841359
24          74.637362         74.644851
25          77.818013         77.824627
26          81.391109         81.396717
27          85.374670         85.379128
28          89.788785         89.791935
29          94.655716         94.657384
30         100.000007        100.000000
```

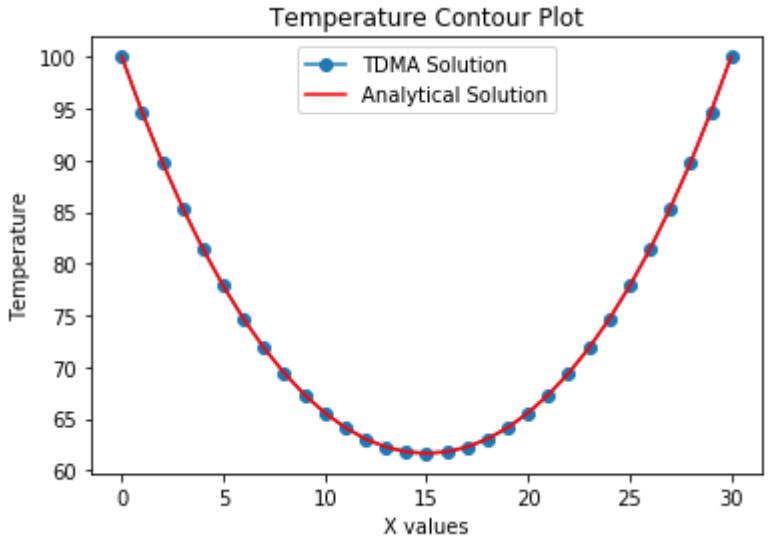
We can see that the analytical and the numerical results that we obtain through TDMA closely agree with each other. This can be further confirmed if we plot a graph depicting the numerical and analytical results together.

In [8]:

```
#Plotting the graph
plt.plot(xarr,T_comp, marker='o', label="TDMA Solution")
plt.plot(xarr, T_analytical, color='red', label='Analytical Solution')
plt.title("Temperature Contour Plot")
plt.xlabel("X values")
plt.ylabel("Temperature")
plt.legend()
plt.plot()
```

Out[8]:

```
[]
```



Measuring the CPU time taken

Let us measure the time taken by the CPU to execute the numerical code.

The CPU time to execute the code can be done using the timeit module in Python. It requires me to put the code whose execution time needs to be computed in a string. The final result is printed below.

In [9]:

```
mysetup = "import numpy as np"

mycode = """

G = (0.071)**2

#Implicit numerical solution using TDMA

#Lower Diagonal matrix
a=[1]*29+[0]

#Middle Diagonal matrix
b=[1]+[-2-G]*29+[1]

#Upper Diagonal Matrix
c=[0]+[1]*29

#Right hand side of the system
d=[100]+[0]*29+[100]

def TDMAAlgo(a,b,c,d):
    n = len(d)                  # Finds the total number of equations

    # Initializing some arrays
    w = np.zeros(n-1,float)
    g = np.zeros(n, float)
    p = np.zeros(n,float)

    # Steps to make the main diagonal element 1
    w[0] = c[0]/b[0]
    g[0] = d[0]/b[0]

    # Steps to make the lower diagonal element 0
    for i in range(1,n-1):
        w[i] = c[i]/(b[i] - a[i-1]*w[i-1])
    for i in range(1,n):
        g[i] = (d[i] - a[i-1]*g[i-1])/(b[i] - a[i-1]*w[i-1])

    # Getting the final RHS matrix element
    p[n-1] = g[n-1]

    # Back substitution
    for i in range(n-1,0,-1):
        p[i-1] = g[i-1] - w[i-1]*p[i]
    return p

T_comp=TDMAAlgo(a,b,c,d)
print(T_comp)

"""

print("The time taken for the TDMA code to execute is: %f seconds"% (timeit.timeit(setu
p = mysetup, stmt = mycode, number = 1)))
```

```
[100.          94.65738374  89.79193535  85.37912811  81.39671705
  77.82462684  74.64485058  71.84135901  69.40001973  67.30852595
  65.55633444  64.13461242  63.03619299  62.255539    61.78871518
  61.63336827  61.78871518  62.255539    63.03619299  64.13461242
  65.55633444  67.30852595  69.40001973  71.84135901  74.64485058
  77.82462684  81.39671705  85.37912811  89.79193535  94.65738374
 100.        ]
The time taken for the TDMA code to execute is: 0.001194 seconds
```

Results and Discussions

Therefore, we can see from the graph that the numerical solution obtained by TDMA closely agrees with the analytical solution. Hence, the code developed is **valid**.

We can see that the temperature decreases until the centre of the bar, and then increases again towards the end. The temperature distribution is symmetric about the vertical axis.

The CPU time taken for the TDMA code to execute is about 0.001194 seconds.

ME317 CFD Assignment 2

Question 3

Introduction

In this question, we find the flow fields around a square cylinder obstacle. We find the pressure and velocity fields for laminar flow around the obstacle, with Reynold's number Re = 36. The analysis is performed using ANSYS Student Version 2021.

Governing Equations

Governing Equation

The governing equations used are the mass and momentum conservation equations from the Navier-Stokes equation.

Continuity: $\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} + \frac{\partial(\rho w)}{\partial z} = 0$

here, t = time
ρ = density
$u, v, w \rightarrow$ x, y, z components of velocities

Since the flow here is steady and velocity along y and z direction is 0 m/s.

$\frac{\partial \rho}{\partial t} = 0, \quad \frac{\partial \rho_v}{\partial y} = \frac{\partial \rho_w}{\partial z} = 0$

Continuity equation used: $\frac{\partial \rho_u}{\partial x} = 0$

Momentum: $\rho \frac{\partial \vec{V}}{\partial t} + \rho(\vec{V}.\nabla)\vec{V} = -\nabla p + \rho r + \mu \nabla^2 \vec{V}$

where $\vec{V} = u\hat{i} + v\hat{j} + w\hat{k}$

$(u, v, w) \rightarrow$ velocities along x, y, z directions

ρ = density
μ = viscosity
r = body force taken

Since this analysis is only 2D, $w = 0$ m/s

So, $\vec{V} = u\hat{i} + v\hat{j}$

In our problem, we have taken velocity along y-axis as zero.

$v = 0$ m/s

Hence, $\vec{V} = u\hat{i}$

∴ Momentum equation: -

$\rho \frac{d\vec{u}}{dt} + \rho(\vec{u}.\nabla)\vec{u} = -\nabla p + \rho r = \mu \nabla^2 \vec{u}$.

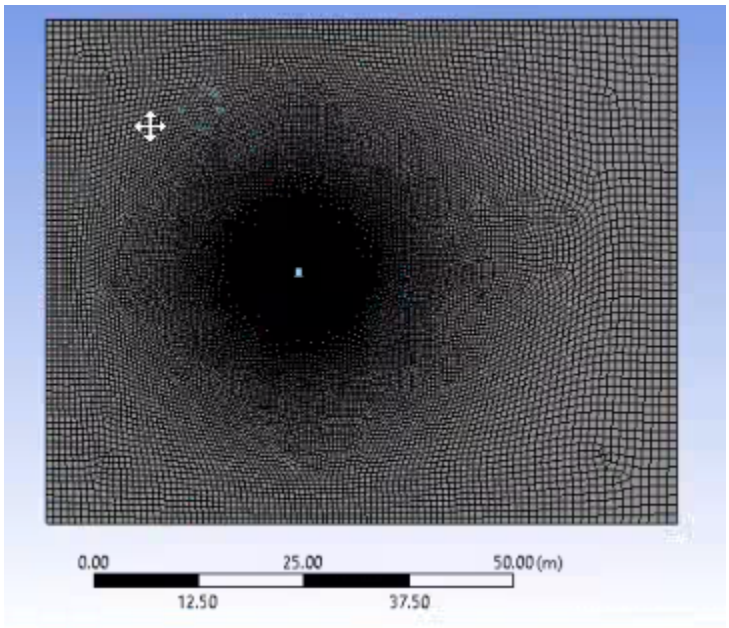
Pre-processing

We assume the density ρ and viscosity μ to be 1. The characteristic length of the cylinder is taken as 1. Reynold's number is assigned as 36. The inlet velocity is also 36m/s based on the below formula:

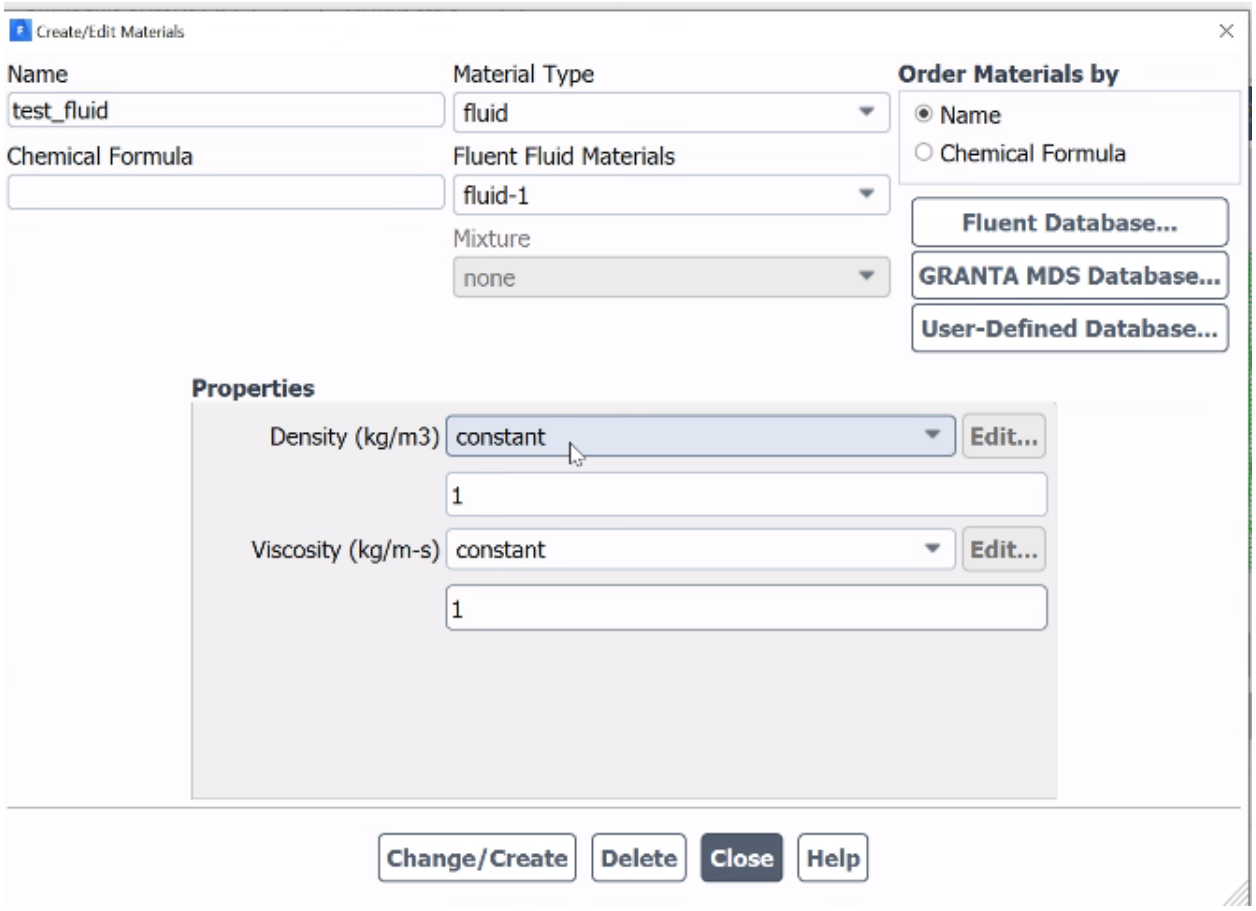
$Re = \rho\mu v/L$

The initialization of the square cylinder is done such that it is closer to the inlet wall of the domain compared to the outlet wall. The height of the domain is 30 times that of the obstacle.

For the meshing, there are 50 divisions made around the obstacle edges, so as to create a refined mesh around the cylinder. Face meshing is enabled.



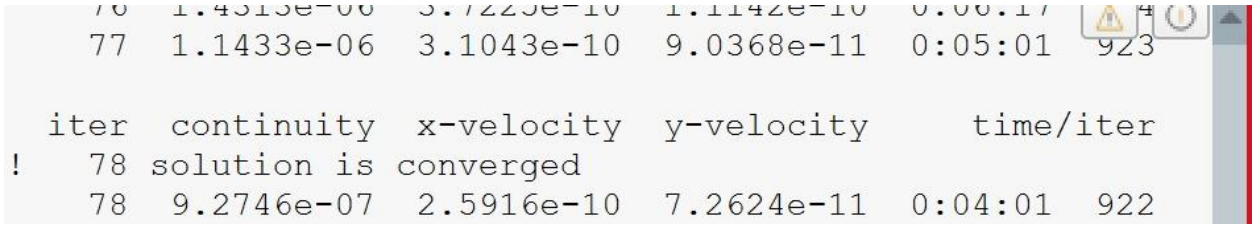
The obstacle wall, inlet, outlet and the domain walls are added as named selections.
In Fluent, laminar flow is selected. In the materials dialog box, a new test fluid with density and viscosity 1 are defined.



This test fluid is selected in the Fluid dialog box.

The inlet wall is given a velocity of 36 m/s.

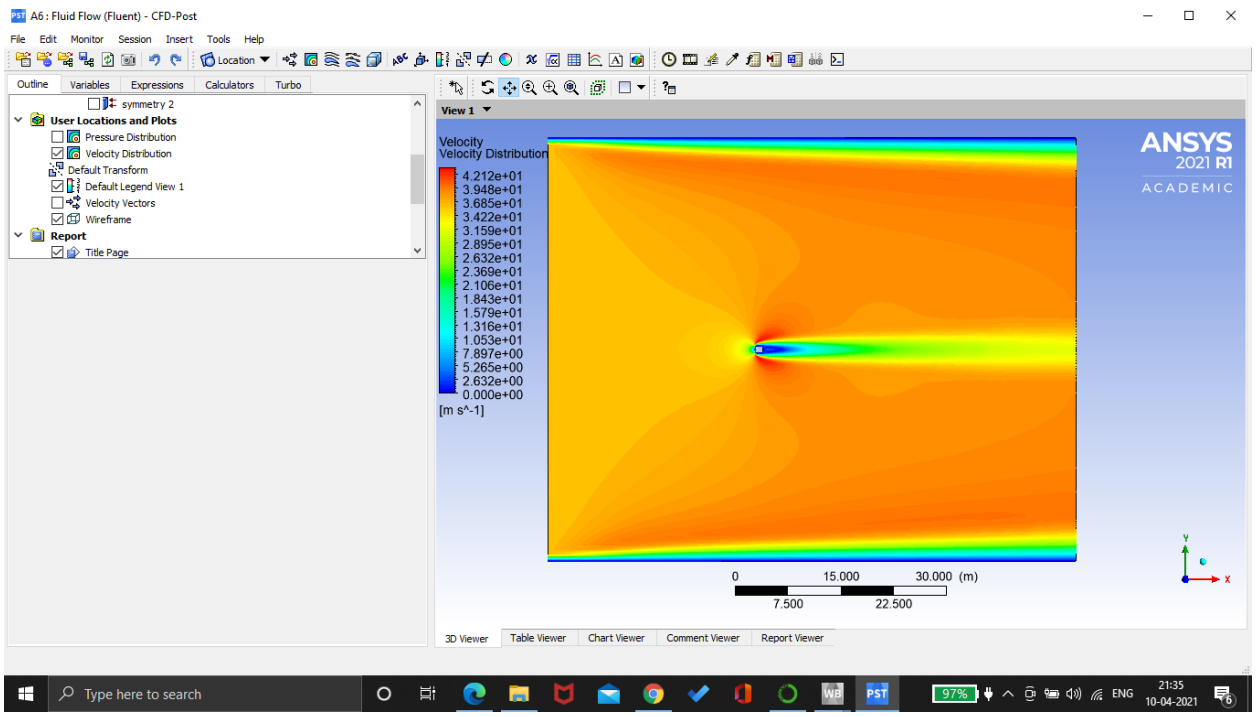
An error of 10^{-6} is taken in the residuals dialog box and a maximum of 1000 iterations is given.

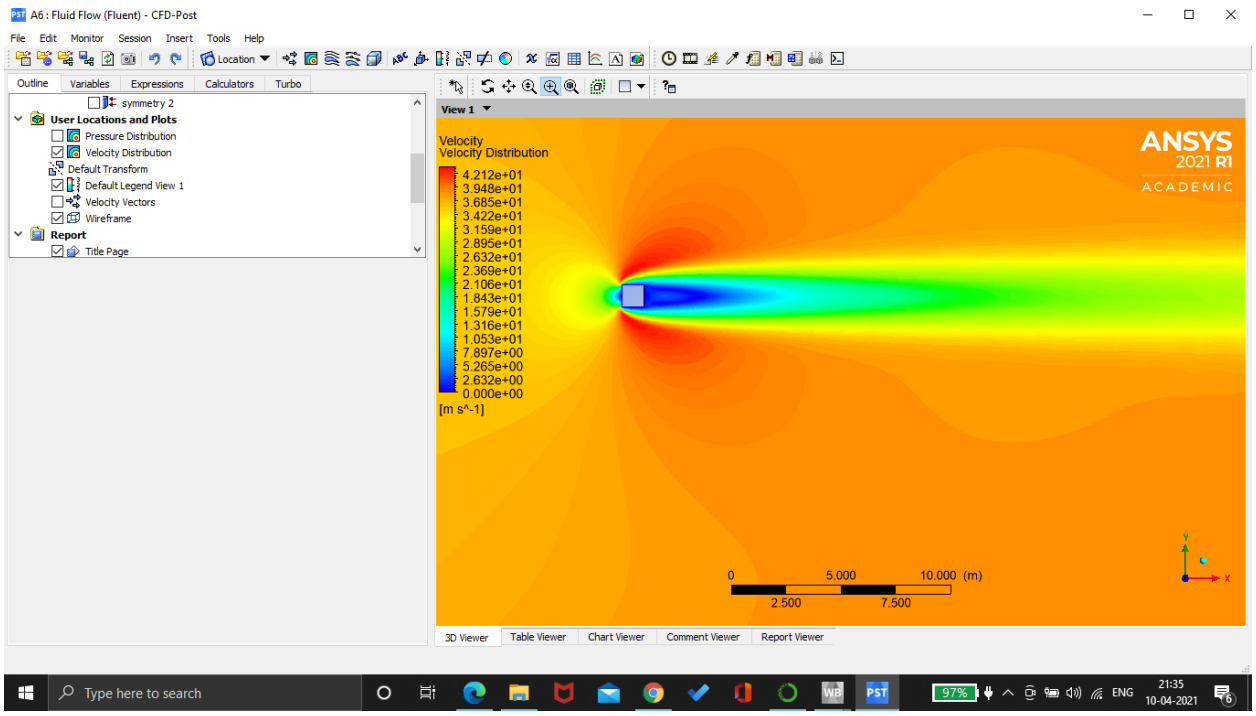


The solution is converged at the 78th iteration.

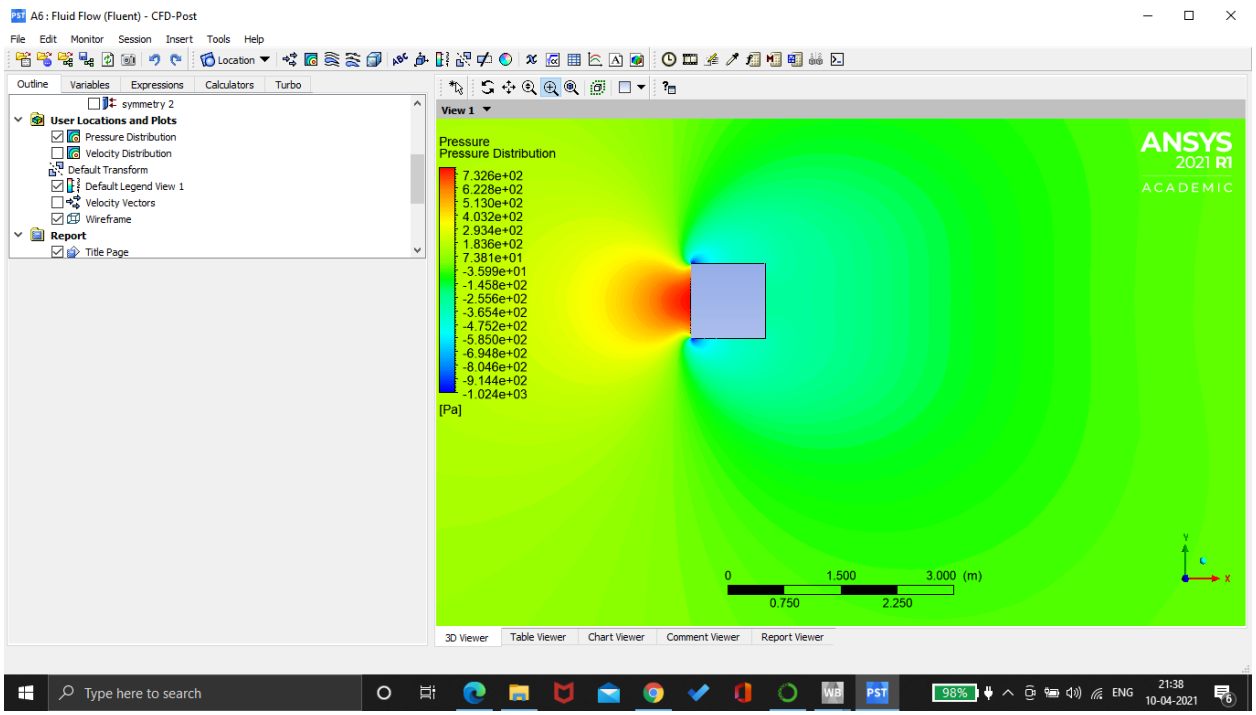
Post-processing

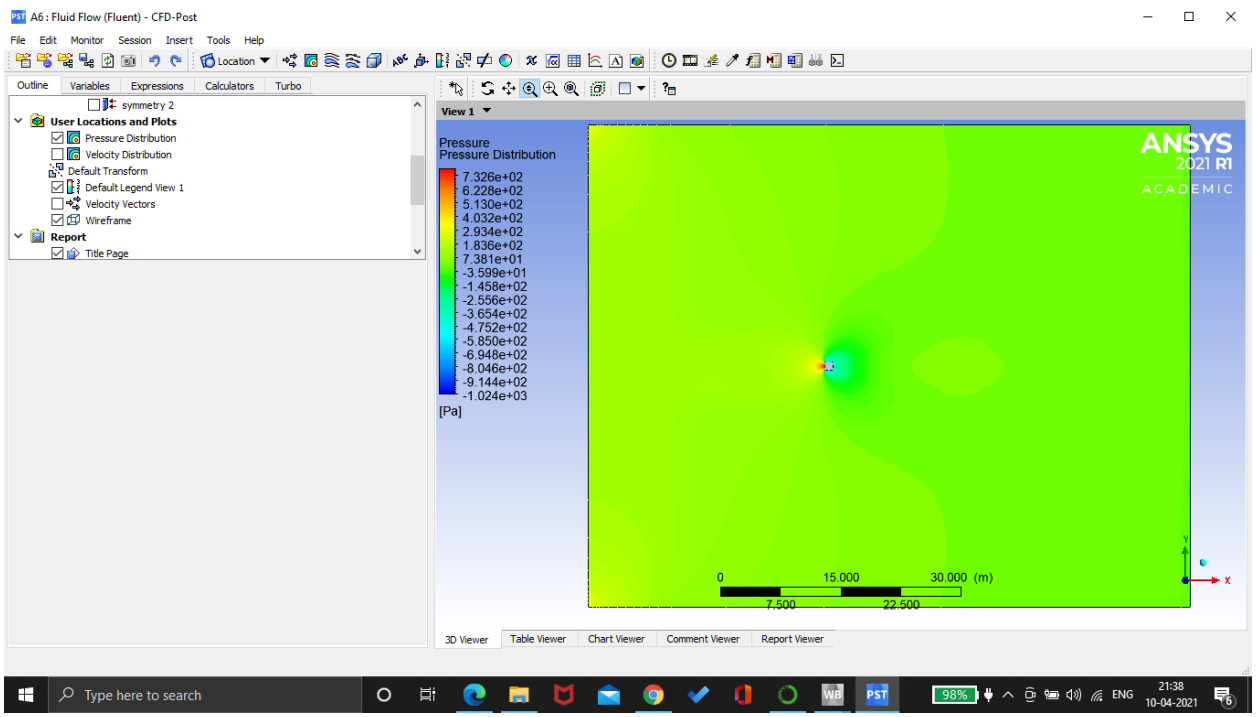
Velocity Contour Plots





Pressure contour plots





Discussion

From the pressure contour plots, we can see that there is a high pressure region in front of the square cylinder, and a low pressure ‘wake’ behind the cylinder. The lowest pressure is at the leading edge of the obstacle. The lowest pressure is -1.024e+03 Pa. The highest pressure is 732 Pa and it is acting on the front edge of the obstacle.

In the velocity contour plots, we can see that the velocity at the leading edge of the cylinder is the highest at 42 m/s. Behind the cylinder, the velocity is 0. Far from the obstacle, we can see that the velocity is at a constant of 36 m/s which is the inlet velocity.

Performing such analyses provides a simulated environment of the laminar flow around several types of shapes. This has a large number of applications in aerospace and other mechanical industries.