Lab Report on Computational Fluid Dynamics(CFD)

A report submitted in partial fulfillment of requirements for the degree of BE in Chemical Science and Engineering

Submitted By:

Jeevan Sapkota 0280119-20

DEPARTMENT OF CHEMICAL SCIENCE AND ENGINEERING
SCHOOL OF ENGINEERING
KATHMANDU UNIVERSITY

April, 2024

Contents

1 Introduction

Computational fluid dynamics (CFD) is a method for analyzing the behavior of a fluid in motion or analyzing the movement of a fluid. In order to depict how a gas or liquid travels and how it affects objects as it passes by, computational fluid dynamics (CFD) uses applied mathematics, physics, and computer software. The Navier-Stokes equations serve as the basis for computational fluid dynamics. These equations explain the relationship between a flowing fluid's velocity, pressure, temperature, and density.

1.1 Navier-Stokes Equations

The Navier-Stokes equations are fundamental equations in fluid mechanics that describe the motion of fluid substances. They provide a mathematical framework for understanding how the velocity, pressure, temperature, and density of a fluid are related in a flowing system. The equations are named after Claude-Louis Navier and Sir George Gabriel Stokes, who made significant contributions to the understanding of fluid dynamics in the 19th century.

Continuity Equation The continuity equation represents the principle of mass conservation in fluid flow. It states that the rate of change of mass within a control volume is equal to the net rate of flow of mass into or out of the control volume. Mathematically, it can be expressed as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{1}$$

where: - ρ is the fluid density, - $\mathbf{v}$ is the velocity vector, - $\frac{\partial}{\partial t}$ denotes the partial derivative with respect to time, - $\nabla \cdot$ represents the divergence operator.

In simpler terms, the continuity equation states that the rate of change of mass density with respect to time plus the divergence of the mass flux density $\rho\mathbf{v}$ is equal to zero. This equation ensures that mass is conserved within the fluid flow field.

Navier-Stokes Equation (Momentum Equation) The Navier-Stokes equation describes the conservation of momentum for a fluid element and is derived from Newton's second law of motion. It takes into account the effects of viscosity and pressure gradients on fluid flow. The general form of the Navier-Stokes equation for incompressible flow is:

$$\rho \left(\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v} \right) = -\nabla p + \mu \nabla^2 \mathbf{v} + \mathbf{f} \tag{2}$$

where: - $\mathbf{v}$ is the velocity vector, - p is the pressure, - (μ is the dynamic viscosity of the fluid, - ∇ represents the gradient operator, - ∇^2 denotes the Laplacian operator, - $\mathbf{f}$ represents external body forces acting on the fluid (such as gravity or electromagnetic forces).

The Navier-Stokes equation accounts for the acceleration of fluid particles, the effects of pressure gradients, and the dissipative effects of viscosity. It is a vector equation, with components in each direction of the flow.

Together, the continuity equation and the Navier-Stokes equation form a set of partial differential equations that govern the behavior of fluid flow in various physical systems. These equations are fundamental in computational fluid dynamics (CFD) for simulating and analyzing fluid flow phenomena in engineering and scientific applications.

1.2 12 Steps to Solve Navier-Stokes Equation

1. **1-D Linear Convection**: Solve the linear convection equation with a step-function initial condition and appropriate boundary conditions.
2. **1-D Nonlinear Convection**: Solve the nonlinear convection equation with the same initial condition and boundary conditions as in step 1.
3. **1-D Diffusion Equation**: Solve the diffusion equation with the step-function initial condition and appropriate boundary conditions.
4. **1-D Burgers' Equation**: Solve Burgers' equation with a saw-tooth initial condition and periodic boundary conditions.
5. **2-D Linear Convection**: Extend the solution to linear convection to two dimensions, using a square function initial condition and appropriate boundary conditions.
6. **2-D Nonlinear Convection**: Solve the two-dimensional nonlinear convection equation with the same initial condition and boundary conditions as in step 5.
7. **2-D Diffusion Equation**: Extend the solution to diffusion only to two dimensions, using the same initial condition and boundary conditions as in step 5.
8. **2-D Burgers' Equation**: Solve Burgers' equation in two dimensions with the same initial condition and boundary conditions as in step 5.
9. **2-D Laplace Equation**: Solve the Laplace equation with zero initial condition and both Neumann and Dirichlet boundary conditions.
10. **2-D Poisson Equation**: Solve the Poisson equation in two dimensions.
11. **Cavity Flow**: Solve the Navier-Stokes equation for cavity flow in two dimensions.
12. **Channel Flow**: Solve the Navier-Stokes equation for channel flow in two dimensions.

1.3 Step 1: 1-D Linear Convection

The 1-D Linear Convection equation is the simplest, most basic model that can be used to learn something about CFD. It is surprising that this little equation can teach us so much! Here it is:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{3}$$

With given initial conditions (understood as a *wave*), the equation represents the propagation of that initial *wave* with speed c, without change of shape. Let the initial condition be

$$u(x, 0) = u_0(x) \tag{4}$$

. Then the exact solution of the equation is

$$u(x, t) = u_0(x - ct) \tag{5}$$

.

Discretizing this equation in both space and time, using the Forward Difference scheme for the time derivative and the Backward Difference scheme for the space derivative, we obtain:

$$\frac{u_i^{n+1}-u_i^n}{\Delta t}+c\frac{u_i^n-u_{i-1}^n}{\Delta x}=0 \tag{6}$$

Where n and $n+1$ are two consecutive steps in time, while $i-1$ and i are two neighboring points of the discretized x coordinate. If there are given initial conditions, then the only unknown in this discretization is u_i^{n+1}. We can solve for our unknown to get an equation that allows us to advance in time, as follows:

$$u_i^{n+1}=u_i^n-c\frac{\Delta t}{\Delta x}(u_i^n-u_{i-1}^n) \tag{7}$$

Now let's try implementing this in Python.

We'll start by importing a few libraries to help us out.

- `numpy` is a library that provides a bunch of useful matrix operations akin to MATLAB
- `matplotlib` is a 2D plotting library that we will use to plot our results
- `time` and `sys` provide basic timing functions that we'll use to slow down animations for viewing

1.4 Step 2: 1-D Nonlinear Convection

The 1D convection equation is:

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}=0 \tag{8}$$

Instead of a constant factor c multiplying the second term, now we have the solution u multiplying it. Thus, the second term of the equation is now *nonlinear*. Using the same discretization as in Step 1 — forward difference in time and backward difference in space. Here is the discretized equation.

$$\frac{u_i^{n+1}-u_i^n}{\Delta t}+u_i^n\frac{u_i^n-u_{i-1}^n}{\Delta x}=0 \tag{9}$$

Solving for the only unknown term, u_i^{n+1}, yields:

$$u_i^{n+1}=u_i^n-u_i^n\frac{\Delta t}{\Delta x}(u_i^n-u_{i-1}^n) \tag{10}$$

1.5 Step 3: 1-D Diffusion Equation

The one-dimensional diffusion equation is:

$$\frac{\partial u}{\partial t}=\nu\frac{\partial^2 u}{\partial x^2} \tag{11}$$

The first thing to notice is that —unlike the previous two simple equations studied— this equation has a second-order derivative.

Discretizing:$\frac{\partial^2 u}{\partial x^2}$

The second-order derivative can be represented geometrically as the line tangent to the curve given by the first derivative. The second-order derivative is discretized with a Central Difference scheme:

a combination of Forward Difference and Backward Difference of the first derivative. Considering the Taylor expansion of u_{i+1} and u_{i-1} around u_i:

$$u_{i+1} = u_i + \Delta x \frac{\partial u}{\partial x}\bigg|_i + \frac{\Delta x^2}{2}\frac{\partial^2 u}{\partial x^2}\bigg|_i + \frac{\Delta x^3}{3!}\frac{\partial^3 u}{\partial x^3}\bigg|_i + O(\Delta x^4) \tag{12}$$

$$u_{i-1} = u_i - \Delta x \frac{\partial u}{\partial x}\bigg|_i + \frac{\Delta x^2}{2}\frac{\partial^2 u}{\partial x^2}\bigg|_i - \frac{\Delta x^3}{3!}\frac{\partial^3 u}{\partial x^3}\bigg|_i + O(\Delta x^4) \tag{13}$$

Adding these two expansions, the odd-numbered derivative terms cancel each other out. Neglecting any terms of $O(\Delta x^4)$ or higher, the sum of these two expansions can be rearranged to solve for the second-derivative:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + O(\Delta x^2) \tag{14}$$

Back to Step 3 The discretized version of the diffusion equation in 1D:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \nu \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} \tag{15}$$

For the initial condition, the only unknown is u_i^{n+1}, so re-arranging the equation for solving the unknown:

$$u_i^{n+1} = u_i^n + \frac{\nu \Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n) \tag{16}$$

The above discrete equation allows writing a program to advance a solution in time. An initial condition is needed. Continuing the use of the hat function, at $t = 0$, $u = 2$ in the interval $0.5 \leq x \leq 1$ and $u = 1$ everywhere else.

1.6 Step 4: 1-D Burgers' Equation

Burgers' equation in one spatial dimension is:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \tag{17}$$

This equation combines non-linear convection and diffusion, offering valuable insights despite its simplicity.

Discretization of Burgers' equation can be achieved using the methods detailed in Steps 1 to 3. Employing forward difference for time, backward difference for space, and a 2nd-order method for the second derivatives, we arrive at:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^n \frac{u_i^n - u_{i-1}^n}{\Delta x} = \nu \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} \tag{18}$$

Upon obtaining the initial condition, the only unknown is u_i^{n+1}. Time stepping can be performed as follows:

$$u_i^{n+1} = u_i^n - u_i^n \frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) + \nu \frac{\Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n) \tag{19}$$

Initial and Boundary Conditions To examine some interesting properties of Burgers' equation, different initial and boundary conditions are utilized compared to previous steps.

The initial condition for this problem is:

$$u = -\frac{2\nu}{\phi}\frac{\partial \phi}{\partial x} + 4 \tag{20}$$

$$\phi = \exp\left(\frac{-x^2}{4\nu}\right) + \exp\left(\frac{-(x-2\pi)^2}{4\nu}\right) \tag{21}$$

This has an analytical solution given by:

$$u = -\frac{2\nu}{\phi}\frac{\partial \phi}{\partial x} + 4\phi = \exp\left(\frac{-(x-4t)^2}{4\nu(t+1)}\right) + \exp\left(\frac{-(x-4t-2\pi)^2}{4\nu(t+1)}\right) \tag{22}$$

The boundary condition is:

$$u(0) = u(2\pi)$$

This is referred to as a *periodic* boundary condition.

1.7 Step 5: 2-D Linear Convection

The partial differential equation (PDE) governing 2-D Linear Convection is written as:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} + c\frac{\partial u}{\partial y} = 0 \tag{23}$$

This is the exact same form as with 1-D Linear Convection, except that we now have two spatial dimensions to account for as we step forward in time.

Again, the timestep will be discretized as a forward difference, and both spatial steps will be discretized as backward differences.

With 1-D implementations, i subscripts were used to denote movement in space (e.g., $u_i^n - u_{i-1}^n$). Now that there are two dimensions to account for, a second subscript, j, is added to account for all the information in the regime.

Here, i will again be used as the index for the x values, and the j subscript will be added to track the y values.

With that in mind, the discretization of the PDE should be relatively straightforward:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} + c\frac{u_{i,j}^n - u_{i-1,j}^n}{\Delta x} + c\frac{u_{i,j}^n - u_{i,j-1}^n}{\Delta y} = 0 \tag{24}$$

As before, solve for the only unknown:

$$u_{i,j}^{n+1} = u_{i,j}^n - c\frac{\Delta t}{\Delta x}(u_{i,j}^n - u_{i-1,j}^n) - c\frac{\Delta t}{\Delta y}(u_{i,j}^n - u_{i,j-1}^n) \tag{25}$$

This equation will be solved with the following initial conditions:

$$u(x,y) = \begin{cases} 2 \text{ for} & 0.5 \leq x, y \leq 1 \\ 1 \text{ for} & \text{everywhere else} \end{cases}$$

and boundary conditions:

$$u = 1 \text{ for } \begin{cases} x = 0,\ 2 \\ y = 0,\ 2 \end{cases}$$

2 Objectives

1. To provide a detailed description of the computational fluid dynamics (CFD) modeling process employed in solving the first five steps of 12-step Navier-Stokes equations.
2. To demonstrate the application of numerical techniques for solving each step of the Navier-Stokes equations.
3. To analyze the results obtained from the simulations and interpret the physical implications of the obtained data.
4. To present a comprehensive overview of the methodologies used for each step of the modeling process.
5. To showcase the effectiveness of the implemented numerical techniques in accurately simulating fluid dynamics phenomena.

3 Methodology

3.1 Step 1: 1-D Linear Convection

The initial condition is u(x, 0) = u_0(x). The exact solution is then obtained as shown in Equation 5. Employing the Forward Difference scheme for the time derivative and the Backward Difference method for the space derivative, the supplied equations are discretized in both space and time. Equation 6 allows us to solve for the unknown to enable time evolution. Necessary libraries such as `matplotlib`, `scipy`, and `numpy` are imported, and variables and constants are defined accordingly. Using `np.array`, velocity and (u) arrays are generated. An operation on Equation 6 is performed for each element of the array (u), and the outcome is stored in a new array called (u_n), serving as the answer for the following time-step. Matplotlib subplots are used to plot the required plots, displaying the velocity results (u) using the calculated variable and the space grid created with `np.linspace`.

3.2 Step 2: 1-D Nonlinear Convection

The 1-D nonlinear convection equation differs from linear convection in that the nonlinear convection wave moves with a variable speed instead of a constant speed. The second term is now multiplied by the solution (u) rather than a constant factor (c), rendering the equation's second term nonlinear. Equation 7 is derived using the same discretization as in Step 1, employing forward difference in time and backward difference in space. Matplotlib subplots are used to plot the required plots following the same steps as before.

3.3 Step 3: 1-D Diffusion Equation

Equation 8 represents the one-dimensional diffusion equation. Geometrically, the second-order derivative can be illustrated as the line tangent to the curve produced by the first derivative. Discretizing the second-order derivative with a Central Difference scheme, which combines the Forward Difference and Backward Difference of the first derivative, is undertaken. Equation 15 presents the discretized diffusion equation in 1D. Initialization is performed, and necessary plots are created using Matplotlib subplots, displaying the calculated variable against the space grid created with `np.linspace`.

3.4 Step 4: 1-D Burger's Equation

Burgers' equation appears in one spatial dimension as shown in Equation 17, combining diffusion and non-linear convection. Discretization is conducted using techniques described in Steps 1 through 3, utilizing forward difference for time, backward difference for space, and a second-order technique for the second derivatives. Notably, periodic boundary conditions are applied in Step 4. The problem setup is completed, and necessary plots are generated using Matplotlib subplots, showcasing the calculated variable against the space grid created with `np.linspace`.

3.5 Step 5: 2-D Linear Convection

Step 5 involves discretizing the 2D linear convection equation and applying the initial and boundary layer conditions to solve them. Implementation utilizes various array functions from the numpy library, and a for loop is employed to iterate across a number of time steps. Finally, Axes3D from mpl_toolkits.mplot3d is utilized to plot the required results.

4 Results and Discussion

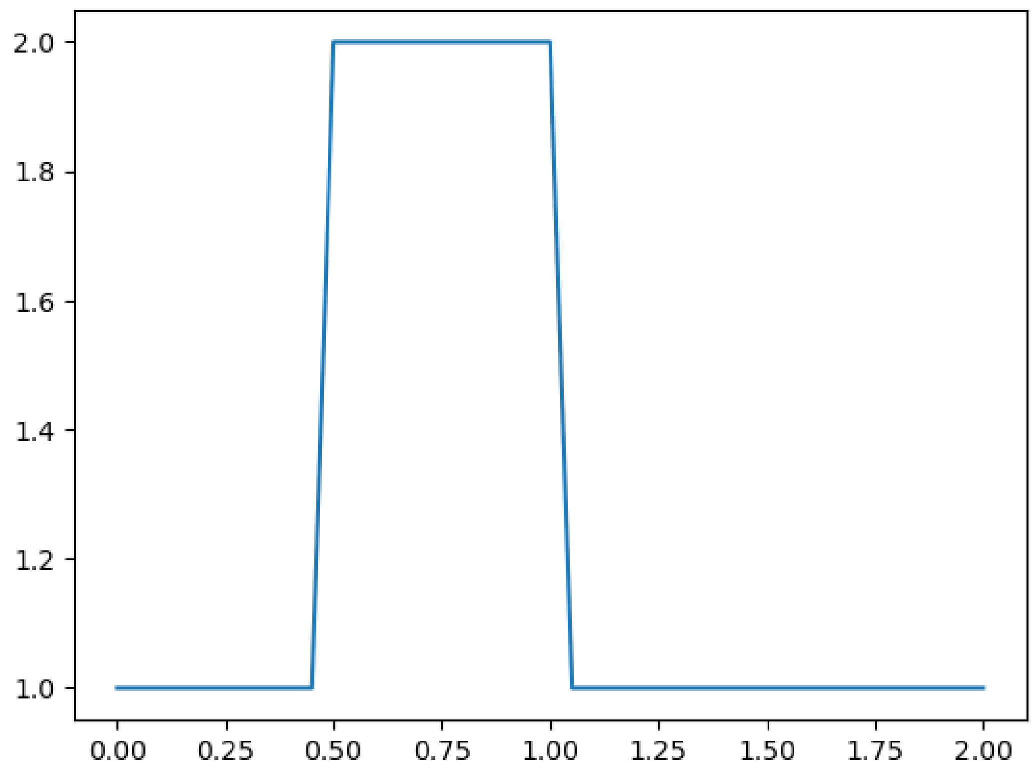
In this section, we present the results and discuss the findings obtained from each step of the computational fluid dynamics (CFD) simulations. For each step, we examine the behavior of the fluid flow and analyze how different parameters and boundary conditions affect the solutions. Through these discussions, we gain insights into the underlying physics and numerical methods employed in solving the Navier-Stokes equations.

4.1 Step 1: 1-D Linear Convection

[63]:
```
import numpy                        #here we load numpy
from matplotlib import pyplot, cm   #here we load matplotlib
import time, sys                    #and load some utilities
from celluloid import Camera
import sympy
from sympy import init_printing
init_printing(use_latex=True)
```

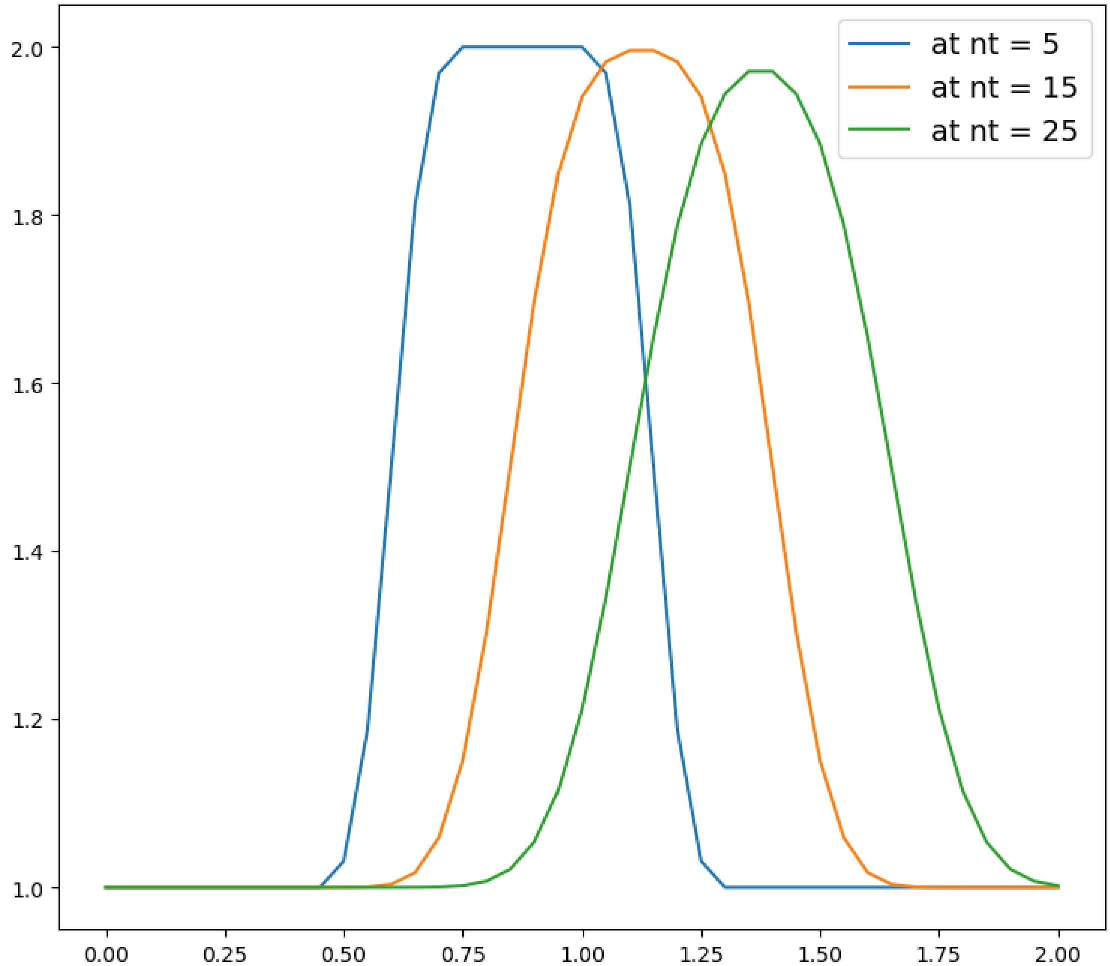
```python
from sympy.utilities.lambdify import lambdify

nx = 41
dx = 2 / (nx-1)
nt = 25    #nt is the number of timesteps we want to calculate
dt = .025  #dt is the amount of time each timestep covers (delta t)
c = 1      #assume wavespeed of c = 1

u = numpy.ones(nx)      #numpy function ones()
u[int(.5 / dx):int(1 / dx + 1)] = 2  #setting u = 2 between 0.5 and 1 as per our
↪I.C.s
print(u)

pyplot.plot(numpy.linspace(0, 2, nx), u);
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```



A hat function was obtained when the initial boundary condition given by the problem statement was placed as shown below

[64]:
```
fig = pyplot.figure(figsize=(9,8))

def convec(nt):
    u = numpy.ones(nx) #numpy function ones()
    u[int(.5 / dx):int(1 / dx + 1)] = 2 #setting u = 2 between 0.5 and 1 as per␣
 ↪our I.C.s
    un = numpy.ones(nx) #initialize a temporary array

    for n in range(nt): #loop for values of n from 0 to nt, so it will run nt␣
 ↪times
        un = u.copy() ##copy the existing values of u into un

        for i in range(1, nx):
            u[i] = un[i] - c * dt / dx * (un[i] - un[i-1])

    pyplot.plot(numpy.linspace(0, 2, nx), u, label = "at nt = {}".format(nt))
    pyplot.legend(fontsize=14);

convec(5)
convec(15)
convec(25)
```

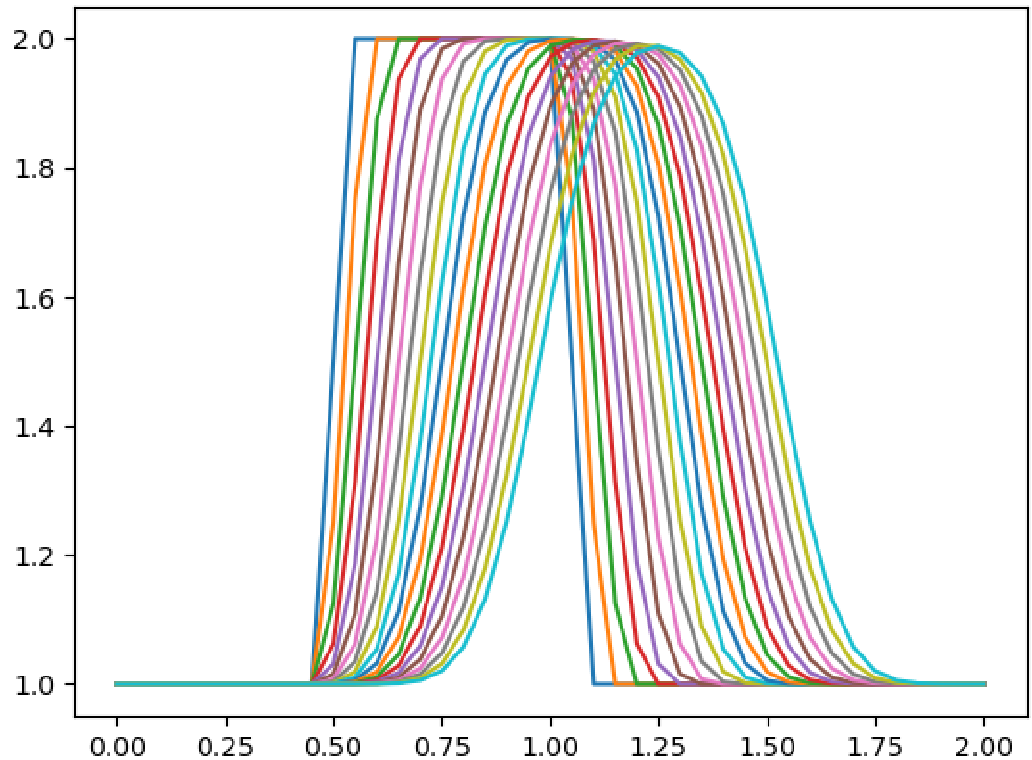


When the initial boundary conditions were set and placed in the model equation, a set of convection plots are obtained at timestep of 25 units. As shown above, when the timestep for which the equation solved is increases, the graph of the convection also moves towards right. As well as moving along the x-axis, the width of the plot also decreases, narrowing the width of the bulk and diverting away from the shape of a hat function to a parabolic shape because of the linear system.

4.2 Step 2: 1-D Nonlinear Convection

[65]:
```
nx = 41
dx = 2 / (nx - 1)
nt = 20    #nt is the number of timesteps we want to calculate
dt = .025  #dt is the amount of time each timestep covers (delta t)
c=1
u = numpy.ones(nx)      #as before, we initialize u with every value equal to 1.
u[int(.5 / dx) : int(1 / dx + 1)] = 2  #then set u = 2 between 0.5 and 1 as per
 ↪our I.C.s
```

```
un = numpy.ones(nx)
for n in range(nt):
    un = u.copy()
    for i in range(1, nx):
        for i in range(nx):
            u[i] = un[i] - c * dt / dx * (un[i] - un[i-1])




    pyplot.plot(numpy.linspace(0, 2, nx), u) ##Plot the results
```

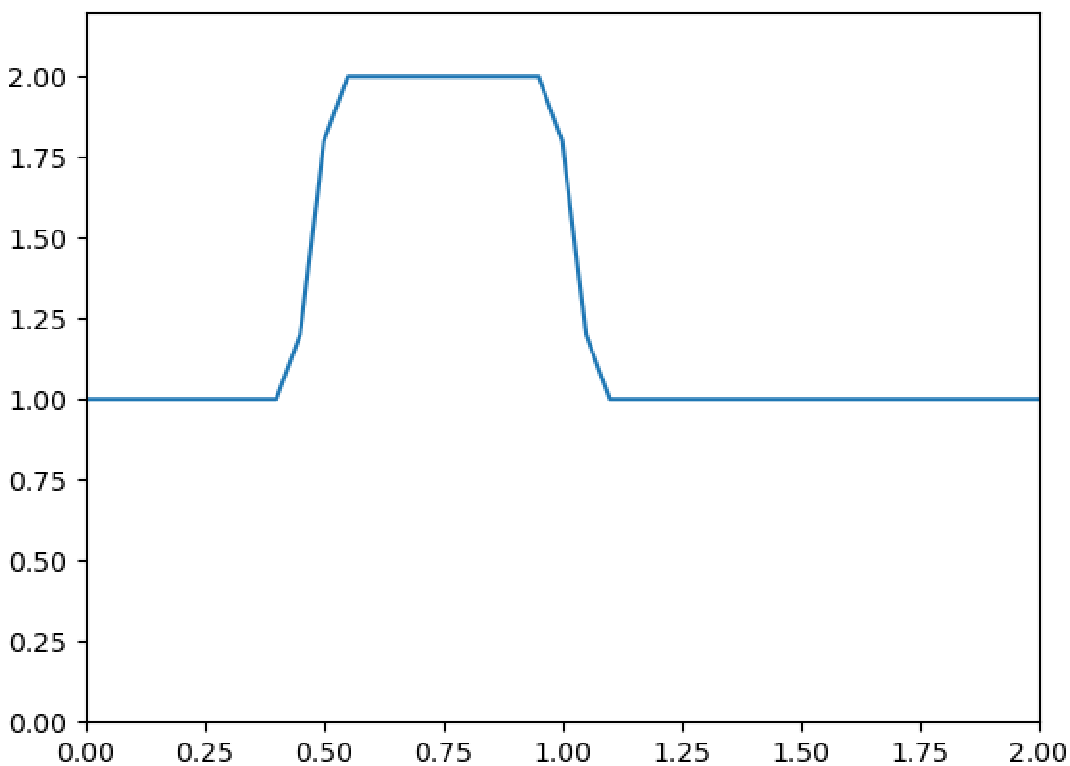


In the nonlinear convection, the convection plots move along x-axis with the timestep value being increased as shown in figure above, similar to the linear convection. However, the shape of the plots becomes irregular along the x-axis, indicating the nonlinearity property of the plot.

4.3 Step 3: 1-D Diffusion

[66]:
```
nu = 0.3
sigma = .2 #sigma is a parameter, we'll learn more about it later
dt = sigma * dx**2 / nu #dt is defined using sigma ... more later!
```

```
u = numpy.ones(nx)      #a numpy array with nx elements all equal to 1.
u[int(.5 / dx):int(1 / dx + 1)] = 2  #setting u = 2 between 0.5 and 1 as per our␣
↪I.C.s

un = numpy.ones(nx) #our placeholder array, un, to advance the solution in time

fig=pyplot.figure()
camera=Camera(fig)
ax= pyplot.axes()
ax.set_ylim(0,2.2)
ax.set_xlim(0,2)

for n in range(nt):  #iterate through time
    un = u.copy() ##copy the existing values of u into un
    for i in range(1, nx - 1):
        u[i] = un[i] + nu * dt / dx**2 * (un[i+1] - 2 * un[i] + un[i-1])
    ax.plot(numpy.linspace(0, 2, nx), u)
    pyplot.pause(0.01)
    camera.snap()
animation=camera.animate()
animation.save("avash2.gif",writer="PillowWriter",fps=100)
pyplot.show()
```



```
MovieWriter PillowWriter unavailable; using Pillow instead.
```

In the diffusion model, instead of moving along a path, the material diffuses on other material. As in figure above, it can be observed that as the timestep is increased, the cone shape of the plot diffuses out to the whole x-axis. While diffusing, the shape of the graph is retended as there is no nonlineratiy in the equation. Along with this, the peak of the material reduces and width increases accordingly as increase in timestep value.

4.4 Step 4: 1-D Burger's Equation

Utilizing SymPy for Time Savings In Burgers' Equation, evaluating the initial condition can be complex and prone to manual errors. The derivative $\frac{\partial \phi}{\partial x}$, while manageable, is susceptible to mistakes in manual computation. Therefore, we leverage SymPy, a symbolic math library for Python, to simplify this task. SymPy provides symbolic math functionality akin to Mathematica, with the added advantage of easy integration into Python calculations. By utilizing SymPy, we streamline the process and minimize the risk of errors in our calculations.

[67]:
```python
from sympy import init_printing
init_printing(use_latex=True)

x, nu, t = sympy.symbols('x nu t')
phi = (sympy.exp(-(x - 4 * t)**2 / (4 * nu * (t + 1))) + sympy.exp(-(x - 4 * t -
↪2 * sympy.pi)**2 / (4 * nu * (t + 1))))
phi
```

[67]:

$$e^{-\frac{(-4t+x-2\pi)^2}{4\nu(t+1)}} + e^{-\frac{(-4t+x)^2}{4\nu(t+1)}}$$

[68]:
```python
phiprime = phi.diff(x)
phiprime
```

[68]:

$$-\frac{(-8t+2x)\,e^{-\frac{(-4t+x)^2}{4\nu(t+1)}}}{4\nu\,(t+1)} - \frac{(-8t+2x-4\pi)\,e^{-\frac{(-4t+x-2\pi)^2}{4\nu(t+1)}}}{4\nu\,(t+1)}$$

[69]:
```python
from sympy.utilities.lambdify import lambdify

u = -2 * nu * (phiprime / phi) + 4
u
```

[69]:

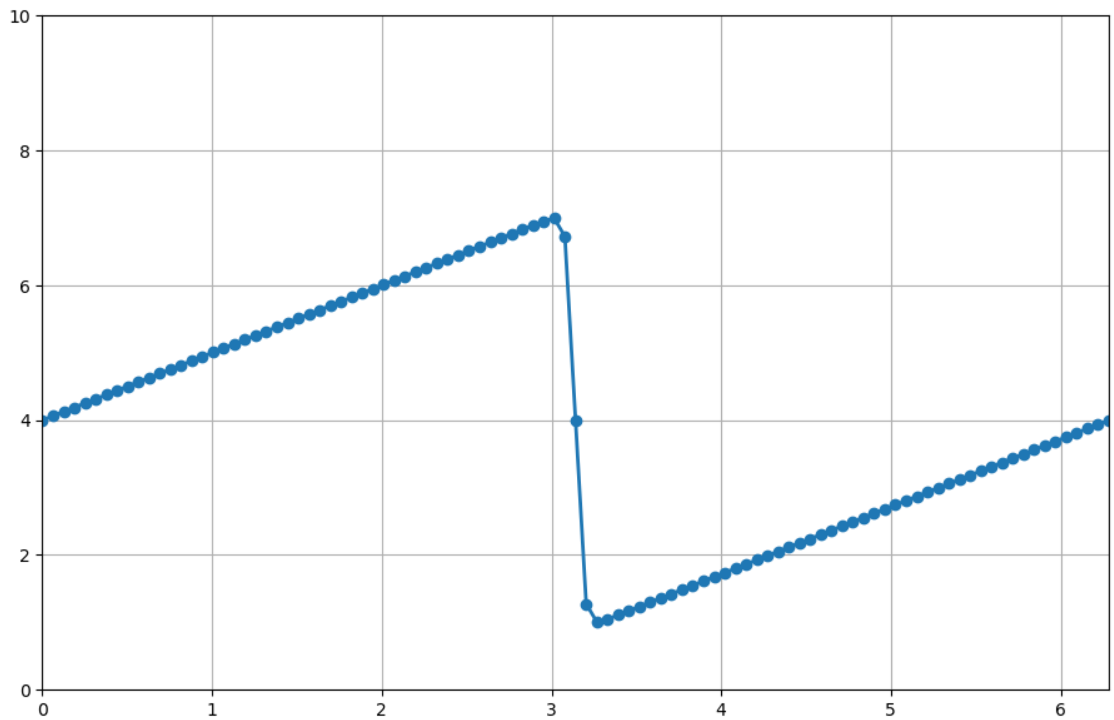
$$-\frac{2\nu\left(-\frac{(-8t+2x)e^{-\frac{(-4t+x)^2}{4\nu(t+1)}}}{4\nu(t+1)} - \frac{(-8t+2x-4\pi)e^{-\frac{(-4t+x-2\pi)^2}{4\nu(t+1)}}}{4\nu(t+1)}\right)}{e^{-\frac{(-4t+x-2\pi)^2}{4\nu(t+1)}} + e^{-\frac{(-4t+x)^2}{4\nu(t+1)}}} + 4$$

[70]:
```python
ufunc = lambdify((t, x, nu), u)
print(ufunc(1, 4, 3))
```

```
3.49170664206445
```

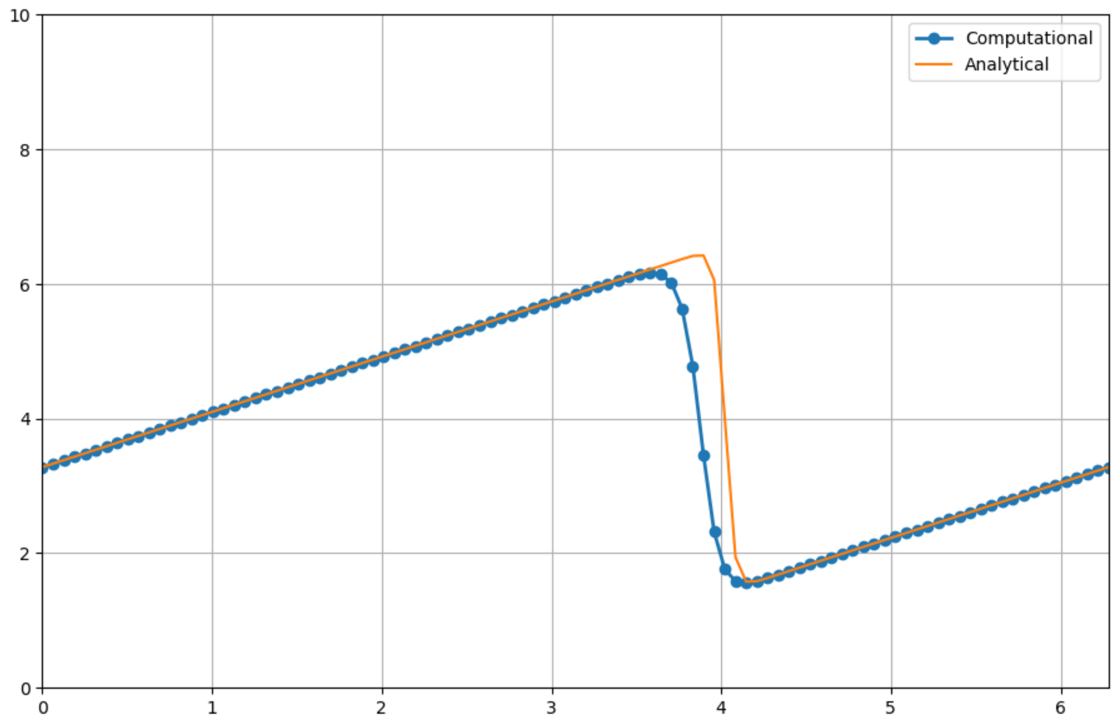
Back to Burgers' Equation

[71]:
```
nx = 101
nt = 50
dx = 2 * numpy.pi / (nx - 1)
nu = .07
dt = dx * nu
x = numpy.linspace(0, 2 * numpy.pi, nx)
un = numpy.empty(nx)
t = 0
u = numpy.asarray([ufunc(t, x0, nu) for x0 in x])
pyplot.figure(figsize=(11, 7), dpi=100)
pyplot.plot(x, u, marker='o', lw=2)
pyplot.xlim([0, 2 * numpy.pi])
pyplot.ylim([0, 10])
pyplot.grid();
```



[72]:
```
for n in range(nt):
    un = u.copy()
    for i in range(1, nx-1):
        u[i] = un[i] - un[i] * dt / dx *(un[i] - un[i-1]) + nu * dt / dx**2␣
↪*(un[i+1] - 2 * un[i] + un[i-1])
    u[0] = un[0] - un[0] * dt / dx * (un[0] - un[-2]) + nu * dt / dx**2 * (un[1]␣
↪- 2 * un[0] + un[-2])
    u[-1] = u[0]
```

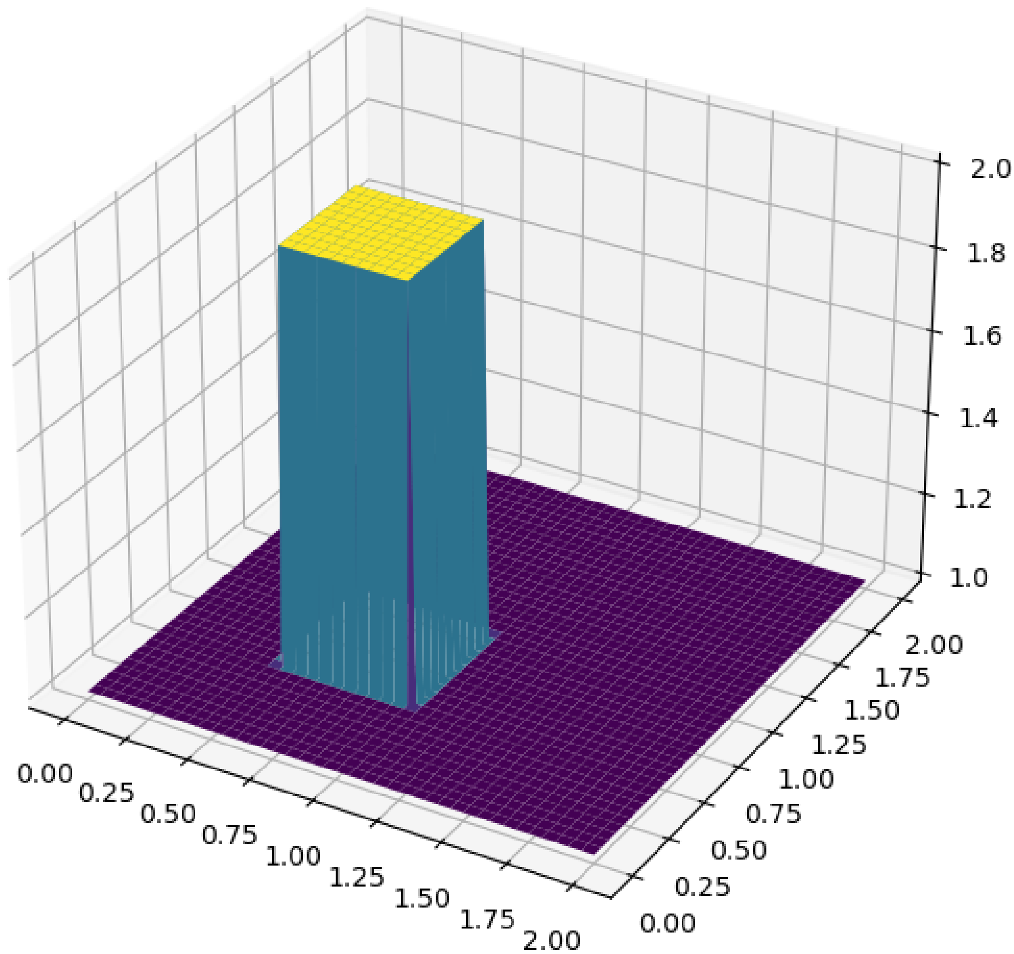
```
u_analytical = numpy.asarray([ufunc(nt * dt, xi, nu) for xi in x])

pyplot.figure(figsize=(11, 7), dpi=100)
pyplot.plot(x,u, marker='o', lw=2, label='Computational')
pyplot.plot(x, u_analytical, label='Analytical')
pyplot.xlim([0, 2 * numpy.pi])
pyplot.ylim([0, 10])
pyplot.grid()
pyplot.legend();
```



First figure above shows the initial condition to solve the Burger's equation analytically. Since the boundary condition was u(0)=u(2π), it is indicated in the figure as the end y value of the plot is same as initial y value, ie. at t=0. While in second figure, the two plots represent the solution solved anatytically (solid blue) and computationally (solid yellow) using the initial condition. In both the plots of second figure, it is observed that the plots moved with the x-axis while the amplitude of the graph is also smaller compared to first figure. This shows the presence of convection (moving in direction of x-axis) and diffusion (decrease in the amplitude), which would indicate solving both the convective and diffusive term present in the Burger's equation.

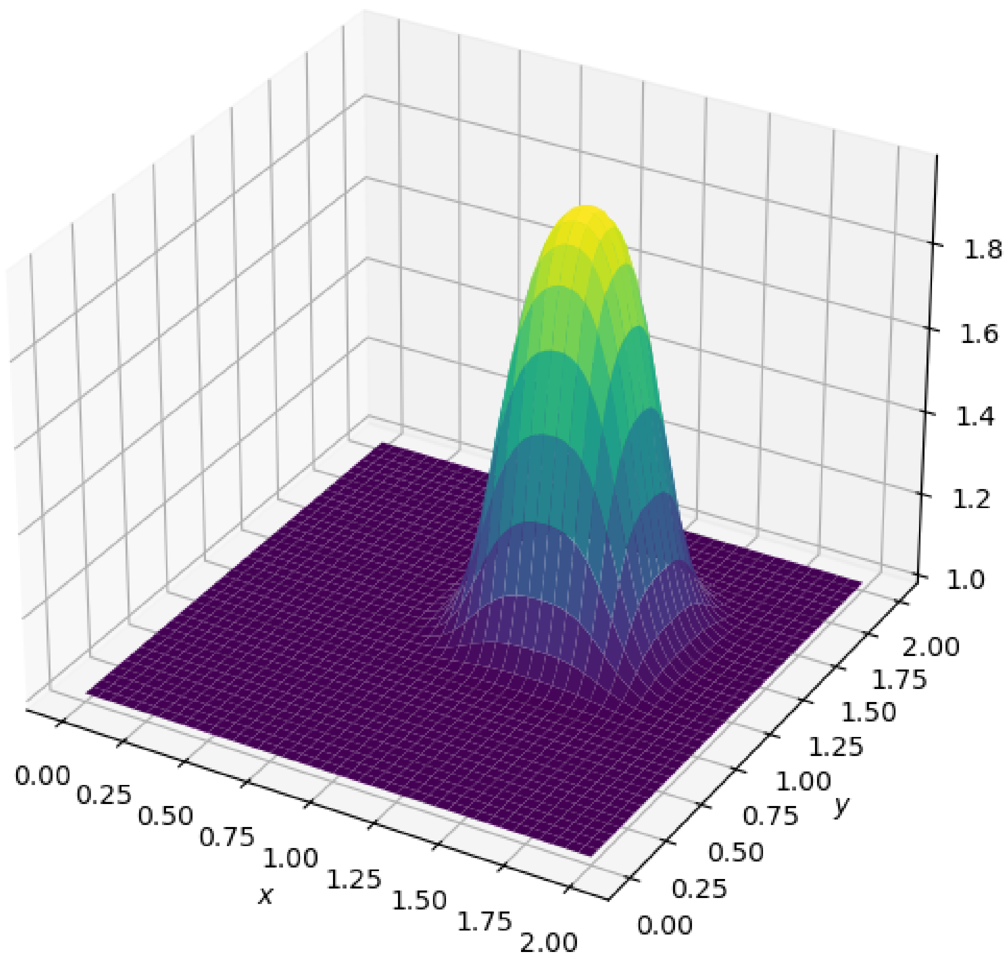
4.5 Step 5: 2-D Linear Convection

[73]:
```
nx = 81
ny = 81
nt = 100
c = 1
dx = 2 / (nx - 1)
dy = 2 / (ny - 1)
sigma = .2
dt = sigma * dx
x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)

u = numpy.ones((ny, nx)) ##create a 1xn vector of 1's
un = numpy.ones((ny, nx)) ##

###Assign initial conditions

##set hat function I.C. : u(.5<=x<=1 && .5<=y<=1 ) is 2
u[int(.5 / dy):int(1 / dy + 1),int(.5 / dx):int(1 / dx + 1)] = 2

###Plot Initial Condition
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = pyplot.axes(projection='3d')
X, Y = numpy.meshgrid(x, y)
surf = ax.plot_surface(X, Y, u[:], cmap=cm.viridis);
```



[74]:
```
u = numpy.ones((ny, nx))
u[int(.5 / dy):int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2

u = numpy.ones((ny, nx))
u[int(.5 / dy):int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2

for n in range(nt + 1): ##loop across number of time steps
    un = u.copy()
    u[1:, 1:] = (un[1:, 1:] - (c * dt / dx * (un[1:, 1:] - un[1:, :-1])) -
    (c * dt / dy * (un[1:, 1:] - un[:-1, 1:])))
    u[0, :] = 1
    u[-1, :] = 1
    u[:, 0] = 1
    u[:, -1] = 1
```

```
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = pyplot.subplot(projection='3d')
surf2 = ax.plot_surface(X, Y, u[:], cmap=cm.viridis)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$');
```



The 3d plot of the initial condition and the plot of the solved 2-D linear convection is shown in above figures. The first plot shows the initial condition for the two dimensional convection. It is a cuboid shaped graph which can be thought of a 3d plot of the hat function presented in 1-D linear convection. The second plot presents the convection occurring in 2-D after timestep of 100 units. Similar to the initial condition plot, the plot of the convection also can be thought of 3d plot of linear 1-D convection since the convection occurs in both x and y direction.

5 Conclusion

In conclusion, the first report on Computational Fluid Dynamics (CFD) has provided a comprehensive overview of fundamental concepts and numerical methods. Through the exploration of linear and nonlinear convection, diffusion, and Burgers' equation, a solid understanding of one-dimensional fluid flow phenomena has been established. Additionally, the extension to two-dimensional problems, including linear convection and diffusion, has expanded our insights into more complex fluid dynamics scenarios. These initial steps lay a strong foundation for delving deeper into CFD, promising further insights into the behavior of fluids and their applications in various engineering disciplines.