**Liquid Jet Impingement**

# Model Description

Figure 1 shows the numerical grid used for the discretization of vectorial and scalar quantities. The choice of grid was staggered in nature because the flow in our case in driven by pressure. As a result, the scalar quantities, i.e., pressure is moved to the center of the cell, whereas, the vectorial quantities, i.e., velocities (*u* and *v*) are moved to the cell faces. Such arrangement ensures that the pressure oscillations resulting in checkerboarding pattern is avoided. Furthermore, the shifting of the quantities demands an additional padding of layer for implementing the boundary conditions. There an extra layer of half-cells spans the entire length of the grid on all sides. Figure 2 shows the boundary conditions imposed on the grid.

Chart, diagram, box and whisker chart

Description automatically generated

Figure 1. Staggered grid arrangement for discretization of vectorial (u- and v- velocities) and scalar (pressure and temperature) quantities

Diagram

Description automatically generated

Figure 2. Boundary conditions implement on the computational grid

# Mathematical Formulation

Consider the Navier–Stokes equation:



The following simplifications are considered:

1. Incompressibility,
2. Newtonian fluid
3. Gravity as the external force is assumed



The variables in the above equations are:



The following are the dimensionless definitions:







Assuming all the variables are centered at zero, which further implies that all the reference values are zero, and for simplicity’s sake, all the position and velocity component are scaled by the same amount (*u*s and *L*s). The new definition for dimensionless quantities is:







Substituting the above definitions into the Navier–Stokes equation, we obtain:



Dividing the equation by the coefficient of the highest derivative, which is , we obtain:



Furthermore, considering the definition for time scale as , and followed by simplification, we obtain:



The final form of dimensionless Navier–Stokes equation for momentum is:



where, Eu = Euler number , Fr = Froude number , and Re = Reynolds number

# Algorithm

1. Initialization
   1. Initialize the *u*-velocities uniformly with ones (including wall boundaries)
   2. Initialize the *v*-velocities uniformly with zeros
   3. Initialize the *p* (pressure) uniformly with zeros
2. Update the *u*-velocities (including boundary conditions)

*u* ← *u* + *dt* ⋅ (− ∂*p*/∂*x* + *μ* ∇²*u* − ∂*u*²/∂*x* − v ∂*u*/∂*y*)

# Update the *v*-velocity (including boundary conditions)

*v* ← *v* + *dt* ⋅ (− ∂*p*/∂*y* + *μ* ∇²*v* − *u* ∂*v*/∂*x* − ∂*v*²/∂*y*)

# Compute the divergence of the tentative velocity components

# *d* = ∂*u*/∂*x* + ∂*v*/∂*y*

# Solve the Poisson equation for the pressure correction *q*

∇²*q* = *d* / *dt* for *q*

# Update the pressure

*p* ← *p* + *q*

# Update the velocities to be incompressible

*u* ← *u* − *dt* ⋅ ∂*q*/∂*x*

*v* ← *v* − *dt* ⋅ ∂*q*/∂*y*

# Repeat time loop until steady-state is reached

# For visualization purpose, the velocities are mapped to the original vertex-centered grid

# Results

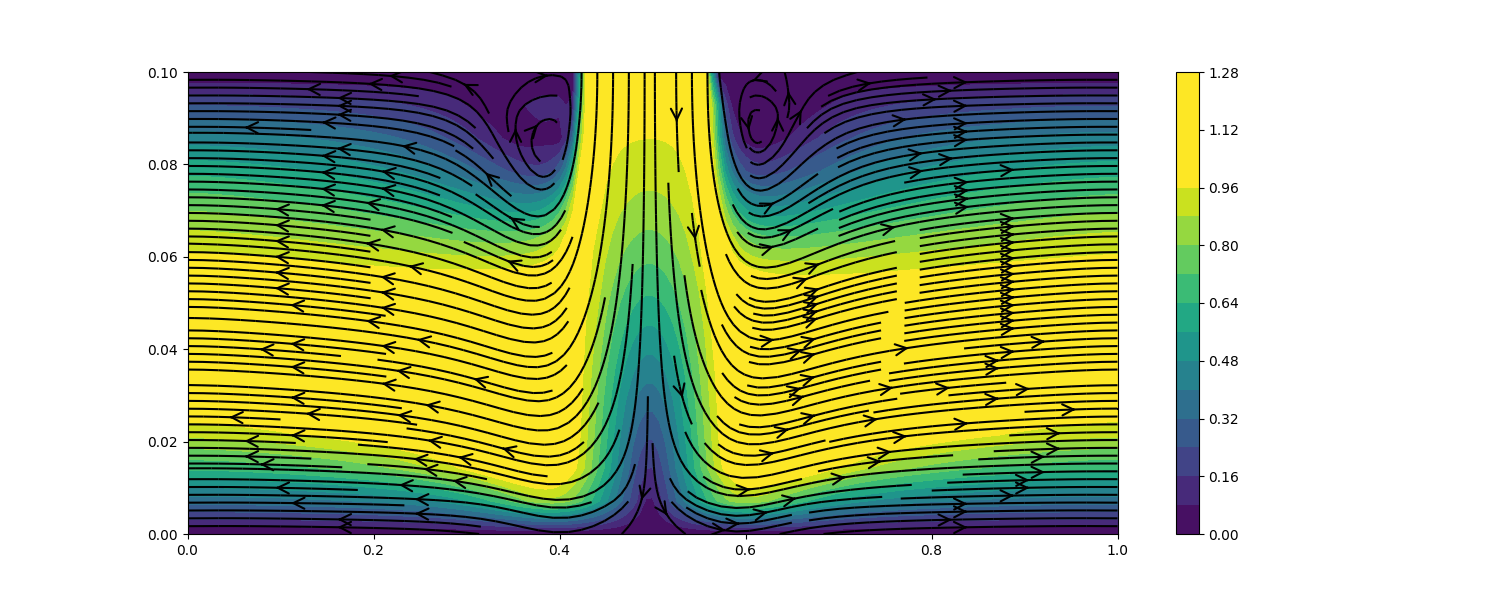


Figure 3. Contours of magnitude of cell-centered velocities and streamlines obtained for the flow

# Python Code

import numpy as np

import matplotlib.pyplot as plt

import cmasher as cmr

from tqdm import tqdm

N\_POINTS\_Y = 15

ASPECT\_RATIO = 10

KINEMATIC\_VISCOSITY = 0.01

TIME\_STEP\_LENGTH = 0.001

N\_TIME\_STEPS = 10000

PLOT\_EVERY = 50

N\_PRESSURE\_POISSON\_ITERATIONS = 50

def main():

    cell\_length = 1.0 / (N\_POINTS\_Y - 1)

    n\_points\_x = (N\_POINTS\_Y - 1) \* ASPECT\_RATIO + 1

    x\_range = np.linspace(0.0, 1.0 \* ASPECT\_RATIO, n\_points\_x)

    y\_range = np.linspace(0.0, 1.0, N\_POINTS\_Y)

    coordinates\_x, coordinates\_y = np.meshgrid(x\_range, y\_range)

    # Initial condition

    velocity\_x\_prev = np.zeros((N\_POINTS\_Y + 1, n\_points\_x))

    velocity\_y\_prev = np.zeros((N\_POINTS\_Y, n\_points\_x+1))

    pressure\_prev = np.zeros((N\_POINTS\_Y+1, n\_points\_x+1))

    # Pre-Allocate arrays

    velocity\_x\_tent = np.zeros\_like(velocity\_x\_prev)

    velocity\_x\_next = np.zeros\_like(velocity\_x\_prev)

    velocity\_y\_tent = np.zeros\_like(velocity\_y\_prev)

    velocity\_y\_next = np.zeros\_like(velocity\_y\_prev)

    plt.figure(figsize=(1.5\*ASPECT\_RATIO, 6))

    for iter in tqdm(range(N\_TIME\_STEPS)):

        # UPDATE INTERIOR OF U-VELOCITY

        diffusion\_x = KINEMATIC\_VISCOSITY \* (

            (

                +

                velocity\_x\_prev[1:-1, 2:  ]

                +

                velocity\_x\_prev[2:  , 1:-1]

                +

                velocity\_x\_prev[1:-1,  :-2]

                +

                velocity\_x\_prev[ :-2, 1:-1]

                - 4 \*

                velocity\_x\_prev[1:-1, 1:-1]

            ) / (

                cell\_length\*\*2

            )

        )

        convection\_x = (

            (

                velocity\_x\_prev[1:-1, 2:  ]\*\*2

                -

                velocity\_x\_prev[1:-1,  :-2]\*\*2

            ) / (

                2 \* cell\_length

            )

            +

            (

                velocity\_y\_prev[1:  , 1:-2]

                +

                velocity\_y\_prev[1:  , 2:-1]

                +

                velocity\_y\_prev[ :-1, 1:-2]

                +

                velocity\_y\_prev[ :-1, 2:-1]

            ) / 4

            \*

            (

                velocity\_x\_prev[2:  , 1:-1]

                -

                velocity\_x\_prev[ :-2, 1:-1]

            ) / (

                2 \* cell\_length

            )

        )

        pressure\_gradient\_x = (

            (

                pressure\_prev[1:-1, 2:-1]

                -

                pressure\_prev[1:-1, 1:-2]

            ) / (

                cell\_length

            )

        )

        velocity\_x\_tent[1:-1, 1:-1] = (

            velocity\_x\_prev[1:-1, 1:-1]

            +

            TIME\_STEP\_LENGTH

            \*

            (

                -

                pressure\_gradient\_x

                +

                diffusion\_x

                -

                convection\_x

            )

        )

        # Apply boundary conditions

        velocity\_x\_tent[1:-1, 0] = velocity\_x\_tent[1:-1, 1] # Left-side boundary condition -> Outflow boundary condition

        velocity\_x\_tent[1:-1, -1] = velocity\_x\_tent[1:-1, -2] # Right-side boundary condition -> Outflow boundary condition

        velocity\_x\_tent[0, :] = - velocity\_x\_tent[1, :] # Bottom edge boundary condition -> No-slip boundary condition

        velocity\_x\_tent[-1, :] = - velocity\_x\_tent[-2, :] # Top edge boundary condition -> No-slip boundary condition

        # UPDATE INTERIOR OF V VELOCITY

        diffusion\_y = KINEMATIC\_VISCOSITY \* (

            (

                +

                velocity\_y\_prev[1:-1, 2:  ]

                +

                velocity\_y\_prev[2:  , 1:-1]

                +

                velocity\_y\_prev[1:-1,  :-2]

                +

                velocity\_y\_prev[ :-2, 1:-1]

                -

                4 \* velocity\_y\_prev[1:-1, 1:-1]

            ) / (

                cell\_length\*\*2

            )

        )

        convection\_y = (

            (

                velocity\_x\_prev[2:-1, 1:  ]

                +

                velocity\_x\_prev[2:-1,  :-1]

                +

                velocity\_x\_prev[1:-2, 1:  ]

                +

                velocity\_x\_prev[1:-2,  :-1]

            ) / 4

            \*

            (

                velocity\_y\_prev[1:-1, 2:  ]

                -

                velocity\_y\_prev[1:-1,  :-2]

            ) / (

                2 \* cell\_length

            )

            +

            (

                velocity\_y\_prev[2:  , 1:-1]\*\*2

                -

                velocity\_y\_prev[ :-2, 1:-1]\*\*2

            ) / (

                2 \* cell\_length

            )

        )

        pressure\_gradient\_y = (

            (

                pressure\_prev[2:-1, 1:-1]

                -

                pressure\_prev[1:-2, 1:-1]

            ) / (

                cell\_length

            )

        )

        velocity\_y\_tent[1:-1, 1:-1] = (

            velocity\_y\_prev[1:-1, 1:-1]

            +

            TIME\_STEP\_LENGTH

            \*

            (

                -

                pressure\_gradient\_y

                +

                diffusion\_y

                -

                convection\_y

            )

        )

        # Apply boundary condition

        velocity\_y\_tent[1:-1, 0] = velocity\_y\_tent[1:-1, 1] # Left-side boundary condition -> Neumann boundary condition

        velocity\_y\_tent[1:-1, -1] = velocity\_y\_tent[1:-1, -2] # Right-side boundary condition -> Neumann boundary condition

        velocity\_y\_tent[0, :] = 0.0 # Bottom edge boundary condition -> No-slip boundary condition

        velocity\_y\_tent[-1, 0:60] = 0.0 # Top-edge boundary condition

        velocity\_y\_tent[-1, 60:80] = -1.0 # Top-edge boundary condition

        velocity\_y\_tent[-1, 80:] = 0.0 # Top-edge boundary condition

        # Compute the divergence as it will be the rhs of the pressure poisson

        # problem

        divergence = (

            (

                velocity\_x\_tent[1:-1, 1:  ]

                -

                velocity\_x\_tent[1:-1,  :-1]

            ) / (

                cell\_length

            )

            +

            (

                velocity\_y\_tent[1:  , 1:-1]

                -

                velocity\_y\_tent[ :-1, 1:-1]

            ) / (

                cell\_length

            )

        )

        pressure\_poisson\_rhs = divergence / TIME\_STEP\_LENGTH

        # Solve the pressure correction poisson problem

        pressure\_correction\_prev = np.zeros\_like(pressure\_prev)

        for \_ in range(N\_PRESSURE\_POISSON\_ITERATIONS):

            pressure\_correction\_next = np.zeros\_like(pressure\_correction\_prev)

            pressure\_correction\_next[1:-1, 1:-1] = 1/4 \* (

                +

                pressure\_correction\_prev[1:-1, 2:  ]

                +

                pressure\_correction\_prev[2:  , 1:-1]

                +

                pressure\_correction\_prev[1:-1,  :-2]

                +

                pressure\_correction\_prev[ :-2, 1:-1]

                -

                cell\_length\*\*2

                \*

                pressure\_poisson\_rhs

            )

            # Apply pressure BC: Homogeneous Neumann everywhere except for the left and right where is a homogeneous Dirichlet

            pressure\_correction\_next[1:-1, 0] = - pressure\_correction\_next[1:-1, 1] # Left-side boundary condition

            pressure\_correction\_next[1:-1, -1] = - pressure\_correction\_next[1:-1, -2] # Right-side boundary condition

            pressure\_correction\_next[0, :] = pressure\_correction\_next[1, :] # Bottom-edge boundary condition

            pressure\_correction\_next[-1, :] = pressure\_correction\_next[-2, :] # Top-edge boundary condition

            # Advance in smoothing

            pressure\_correction\_prev = pressure\_correction\_next

        # Update the pressure

        pressure\_next = pressure\_prev + pressure\_correction\_next

        # Correct the velocities to be incompressible

        pressure\_correction\_gradient\_x = (

            (

                pressure\_correction\_next[1:-1, 2:-1]

                -

                pressure\_correction\_next[1:-1, 1:-2]

            ) / (

                cell\_length

            )

        )

        velocity\_x\_next[1:-1, 1:-1] = (

            velocity\_x\_tent[1:-1, 1:-1]

            -

            TIME\_STEP\_LENGTH

            \*

            pressure\_correction\_gradient\_x

        )

        pressure\_correction\_gradient\_y = (

            (

                pressure\_correction\_next[2:-1, 1:-1]

                -

                pressure\_correction\_next[1:-2, 1:-1]

            ) / (

                cell\_length

            )

        )

        velocity\_y\_next[1:-1, 1:-1] = (

            velocity\_y\_tent[1:-1, 1:-1]

            -

            TIME\_STEP\_LENGTH

            \*

            pressure\_correction\_gradient\_y

        )

        # Again enforce boundary condition

        velocity\_x\_next[1:-1, 0] = velocity\_x\_next[1:-1, 1] # Left-side boundary condition -> Outflow boundary condition

        velocity\_x\_next[1:-1, -1] = velocity\_x\_next[1:-1, -2] # Right-side boundary condition -> Outflow boundary condition

        velocity\_x\_next[0, :] = - velocity\_x\_next[1, :] # Bottom edge boundary condition -> No-slip boundary condition

        velocity\_x\_next[-1, :] = - velocity\_x\_next[-2, :] # Top edge boundary condition -> No-slip boundary condition

        velocity\_y\_next[1:-1, 0] = velocity\_y\_next[1:-1, 1] # Left-side boundary condition -> Neumann boundary condition

        velocity\_y\_next[1:-1, -1] = velocity\_y\_next[1:-1, -2] # Right-side boundary condition -> Neumann boundary condition

        velocity\_y\_next[0, :] = 0.0 # Bottom edge boundary condition -> No-slip boundary condition

        velocity\_y\_next[-1, 0:60] = 0.0 # Top-edge boundary condition

        velocity\_y\_next[-1, 60:80] = -1.0 # Top-edge boundary condition

        velocity\_y\_next[-1, 80:] = 0.0 # Top-edge boundary condition

        # Advance in time

        velocity\_x\_prev = velocity\_x\_next

        velocity\_y\_prev = velocity\_y\_next

        pressure\_prev = pressure\_next

        inflow\_mass\_rate\_next = np.sum(velocity\_x\_next[1:-1, 0])

        outflow\_mass\_rate\_next = np.sum(velocity\_x\_next[1:-1, -1])

        print(f"Inflow: {inflow\_mass\_rate\_next}")

        print(f"Outflow: {outflow\_mass\_rate\_next}")

        print()

        # Visualization

        if iter % PLOT\_EVERY == 0:

            velocity\_x\_vertex\_centered = (

                (

                    velocity\_x\_next[1:  , :]

                    +

                    velocity\_x\_next[ :-1, :]

                ) / 2

            )

            velocity\_y\_vertex\_centered = (

                (

                    velocity\_y\_next[:, 1:  ]

                    +

                    velocity\_y\_next[:,  :-1]

                ) / 2

            )

            plt.contourf(

                coordinates\_x,

                coordinates\_y,

                np.sqrt(velocity\_x\_vertex\_centered\*\*2 + velocity\_y\_vertex\_centered\*\*2),

                levels=20,

            )

            plt.colorbar()

            #plt.quiver(

                #coordinates\_x[:, ::6],

                #coordinates\_y[:, ::6],

                #velocity\_x\_vertex\_centered[:, ::6],

                #velocity\_y\_vertex\_centered[:, ::6],

                #alpha=0.4,

            #)

            plt.streamplot(coordinates\_x, coordinates\_y, velocity\_x\_vertex\_centered, velocity\_y\_vertex\_centered, arrowsize=2, arrowstyle='->', density=2, color="black")

            plt.draw()

            plt.pause(0.05)

            plt.clf()

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    main()