廣西大學

小作品

基于 Python 的流体力学仿真程序

学　　院　机械工程学院

专　　业　工程科学专业

班　　级　工程科学创培 201 班

学　　号　2001190137

姓　　名　于若涛

指导教师　李会芬

二〇二一年四月

摘 要

在这篇文章中，我借助 Python 计算机语言，建立了一个对流体力学过程中流体动态进行模拟的程序，共有三个子程序：对流体力学有关物理量进行创建的 head_file.py、基于 head_file.py 中创建的类与对象，按照使用者给出的初始值，进行模拟的主程序 main_program.py 和将结果整理成 gif 文件，在时域实现可视化的 Animation.py。

关键字：流体力学， 计算流体力学， 纳维斯托克斯方程， 数值计算

Abstract

In this essay, the author established a program that could operate simulations of some dynamic processes in fluid mechanics using Python. The program is composed of three subprograms. head_file.py establishes the classes and objects of physical quantities around the fluid mechanics. Based on those classes and objects, main_program.py initiates the main process of simulation. Finally, Animation.py summarizes the obtained results and visualizes them by composing them into a gif file.

Keywords: Fluid Mechanics, Computational Fluid Mechanics, Navier-Stokes Equation, Numerical Calculation

目录

1. 设计目的

1.1 课程介绍

流体力学是物理学中研究流体的运动、动力及其规律的一个分支。流体力学在生产、生活相关的各领域中有着广泛的应用，如：机械、电气、航空航天等。流体力学是研究液体和气体的行为，特别是它们产生的力的学科。许多科学研究领域都对流体力学感兴趣，例如：气象学家试图预测绕地球旋转的流体大气的运动，以便预测天气；物理学家研究极高温气体在磁场中的流动，以寻求一种可接受的方法来利用核聚变反应的能量。工程师对流体力学感兴趣，是因为流体产生的力可以为实际目的进行服务。著名的例子有喷气推进、机翼设计、风力涡轮机和液压制动器、机械心脏瓣膜的设计等。

流体力学可分为流体静力学、流体动力学。流体静力学主要研究静止流体，而流体动力学主要研究力对流体运动状态的影响。它是连续介质力学的一个分支，是不直接使用原子构成的信息来对物质进行模型的建立的学科，也就是说，它从宏观而非微观的角度来建模。流体力学是一个活跃的研究领域，其中许多问题在数学上极其复杂，无法求出解析解，只能借助计算机用数值方法解决，一门称为计算流体力学（CFD）的现代学科致力于解决此类问题。

1.2 发展历史

流体力学的研究最早可以追溯到古希腊时期：阿基米德对流体静力学和浮力进行研究，发现了阿基米德原理，因而他被认为是第一个主要研究流体力学的学者。流体力学的快速发展始于列奥纳多·达·芬奇（观察和实验）、埃万杰利斯塔·托里切利（发明气压计）、艾萨克·牛顿（研究粘度）和布莱斯·帕斯卡（研究流体静力学，阐述了帕斯卡定律）、丹尼尔·伯努利在其作品《流体力学》中引入了与流体力学相关的数学描述（1739 年）。

此后，许多数学家对无粘流进行了进一步的分析，包括 Jean Léonard Marie Poiseuille 和 Gotthilf Hagen 在内的众多工程师对粘性流进行了探索。进一步的数学证明由 Claude-Louis Navier 和 George Gabriel Stokes 在 Navier - Stokes 方程中提供，边界层理论被提出（Ludwig

Prandtl, Theodore von Kármán），而 Osborne Reynolds, Andrey Kolmogorov 和 Geoffrey Ingram Taylor 等科学家提出了对流体粘度和湍流的理解。

1.3 课程要求

广西大学本学期开设的，由李会芬老师主讲的《工程流体力学》课程，主要介绍流体力学的发展与演变、流体的相关物理性质和基本规律，并学习如何解决土木和环境工程师感兴趣的各种问题。除了了解流体力学的工作知识，这门课程还将为学习更高层次的流体力学课程做准备。

为了贯彻理论与实践结合的思想，本课程除授课外，还有实验和以仿真为主题的大作业。

2. 设计过程

流体的流动可以在许多自然现象中观察到：从舒缓的瀑布到恼人的咖啡溅在你的电脑键盘上。从悬崖上潺潺而下的水流，宁静而又充满活力，肯定能唤起一种奇妙的感觉。但这些现象也使得我们思考：我们能理解这些情况下的流体流动吗？我们能预测流体在特定条件下是如何运动的吗？我们能防止咖啡溅出来吗？

要回答这些问题，一种方法是在实验室里用实际的流体进行实验，并使用各种成像仪器研究流体的性质，这是实验性的方法。另一种方法是编写一组能够描述流体流动方程，应用一套简化的假设和条件，执行一些数学变换，并推导控制方程，进而预测动态流的能力，这属于分析方法。

然而，随着计算能力的提高，出现了第三种方法来回答这些问题——数值方法。虽然描述流体流动的方程组在任何条件下都是不可解析的，但如果你有一台足够强大的计算机，它们的输出是可以计算出来的。用这种方法在计算机上研究流体流动的动力学通常被称为计算流体力学(CFD)。

2.1 流体力学相关知识

考虑一个在空间中具有固定体积的 2D 盒子，这就是我们所说的控制体：

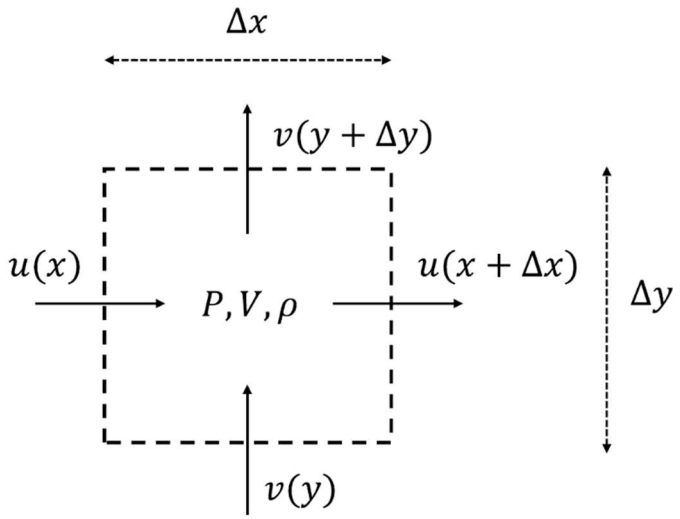


图 2-1

首先，我们将质量守恒原理应用于控制体积内的流体。对于不可压缩流体（大多数液体），这意味着任何进入盒子的液体都必须从盒子里出来。这在流体力学中称为连续性方程。

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

其次，我们将把动量守恒原理应用于控制体积。与前一种情况相比，这稍微抽象和复杂一些。通过简化，我们可以得到不可压缩流体的 Navier-Stokes 方程。

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}=-\frac{1}{\rho}\frac{\partial p}{\partial x}+\nu\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)$$

$$\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}=-\frac{1}{\rho}\frac{\partial p}{\partial y}+\nu\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)$$

如果我们能够在应用必要的边界条件的同时解出这些偏微分方程（PDE），我们将获得作为时间函数的瞬时速度和压力，从而使我们能够预测流体将如何流动。然而，如果不应用简化的假设，就没有解析的方法来解这些方程。因此，我们求助于数值技算方法来求解这些方程。

求解偏微分方程存在多种不同的数值方法，每一种方法都有自己的一套注意事项。最简单的方法是有限差分法，其中一个低阶泰勒级数逼近是用来将偏微分方程转换为一组代数方程。下面给出一个例子，说明如何将一阶和二阶导数转换为它们的有限差分近似。

$$\frac{\partial u}{\partial x} \approx \frac{u(x+\Delta x)-u(x-\Delta x)}{2\Delta x} = \frac{\Delta u}{\Delta x}$$
$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+\Delta x)-2u(x)+u(x-\Delta x)}{\Delta x^2} = \frac{\Delta^2 u}{\Delta x^2}$$

虽然这不是在所有情况下建模流体流动的最佳方法，但我们将继续使用它，因为它简化了建模的其他方面。对于更严格的数值处理，我们可能需要使用有限体积或有限单元法。

下面我们介绍一种能够同时求解连续性方程和 Navier-Stokes 方程的预测-校正方法：

在不考虑压强的情况下，由初速度计算速度：

$$u^{\star}(t) = u(t) + \Delta t\left[-u(t)\frac{\Delta u(t)}{\Delta x} - v(t)\frac{\Delta u(t)}{\Delta y} + \nu\left(\frac{\Delta^2 u(t)}{\Delta x^2} + \frac{\Delta^2 u(t)}{\Delta y^2}\right)\right]$$

通过上述速度，迭代求解泊松压力方程：

$$\frac{\Delta^2 p(t+\Delta t)}{\Delta x^2} + \frac{\Delta^2 p(t+\Delta t)}{\Delta y^2} = -\frac{\rho}{\Delta t}\left(\frac{\Delta u^{\star}(t)}{\Delta x} + \frac{\Delta u^{\star}(t)}{\Delta y}\right)$$

利用步进法进行速度预测：

$$u(t+\Delta t) = u^{\star}(t) + \Delta t\left(-\frac{1}{\rho}\frac{\Delta p}{\Delta x}\right)$$

下面我们借助 Python，在上述数学基础之上，对流体的二维流动进行编程实现。

2.2 代码实现解析

2.2.1 head_file.py

```
import os

import numpy as np
```

os 模块提供了多数操作系统的功能接口函数。当 os 模块被导入后，它会自适应于不同的操作系统平台，根据不同的平台进行相应的操作。在 python 编程时，经常和文件、目录打交道，这时就离不开 os 模块；NumPy（Numerical Python）是 Python 的一种开源的数值计算扩展。这种工具可用来存储和处理大型矩阵，比 Python 自身的嵌套列表（nested list structure）结构要高效的多（该结构也可以用来表示矩阵），支持大量的维度数组与矩阵运算，此外也针对数组运算提供大量的数学函数库。我们在此导入 os 与 numpy 库。

```
# CLASS CONFIGURATION 配置类
```

```
class Boundary:
    def __init__(self, boundary_type, boundary_value):
        self.DefineBoundary(boundary_type, boundary_value)

    def DefineBoundary(self, boundary_type, boundary_value):
        self.type = boundary_type
        self.value = boundary_value
# 设置边界条件，便于 PDE 的求解
# 种类有两种：dirichlet 与 neumann
```

下面，考虑到流体力学研究对象的种类具有较高的区分度，我们通过配置不同的“类”对其进行表示。首先，从边界条件开始。通过应用特定的边界条件来求解偏微分方程，这些边界条件能够预测流体在边界处的行为。例如，流经管道的流体将具有流体速度为零的壁，以及具有特定流速的入口和出口。

在数学上，边界条件可以用两种形式表示——狄利克雷和诺伊曼边界。前者指定因变量在边界处的值，而后者指定因变量在边界处的导数的值。因此，我们创建了一个有两个属性——类型和值的 Boundary 类。

```
class Space:
    def __init__(self):
        pass

    def CreateMesh(self, rowpts, colpts):
        self.rowpts = rowpts
        self.colpts = colpts
        self.u = np.zeros((self.rowpts + 2, self.colpts + 2))
        self.v = np.zeros((self.rowpts + 2, self.colpts + 2))
        self.p = np.zeros((self.rowpts + 2, self.colpts + 2))
        self.p_c = np.zeros((self.rowpts, self.colpts))
        self.u_c = np.zeros((self.rowpts, self.colpts))
        self.v_c = np.zeros((self.rowpts, self.colpts))
        self.SetSourceTerm()

    def SetDeltas(self, breadth, length):
        self.dx = length / (self.colpts - 1)
        self.dy = breadth / (self.rowpts - 1)

    def SetInitialU(self, U):
        self.u = U * self.u

    def SetInitialV(self, V):
        self.v = V * self.v
```

```
    def SetInitialP(self, P):
        self.p = P * self.p

    def SetSourceTerm(self, S_x=0, S_y=0):
        self.S_x = S_x
        self.S_y = S_y
# 定义边界围成的区域
```

下面，我们利用交错网格方法，将被边界包围的区域（比如一根管道的内部）用二维网格或网格表示，利用因变量在网格中盒子的中心（用于压力）或盒子的表面（用于速度）进行计算。为了表示网格，我们创建一个名为 Space 的类；CreateMesh 方法为因变量创建一个给定大小的矩阵；setdelta 方法根据域的指定长度和宽度计算微分长度的值。

```
class Fluid:
    def __init__(self, rho, mu):
        self.SetFluidProperties(rho, mu)

    def SetFluidProperties(self, rho, mu):
        self.rho = rho
        self.mu = mu


# 定义区域内部的流体，其特征由密度和粘度表示
```

最后，我们创建 Fluid 类，通过流体密度（rho）和粘度(mu)来表征流体的性质。

```
# 设置边界条件
# U,V 为两轴，下面设置速度的边界条件

def SetUBoundary(space, left, right, top, bottom):
    if left.type == "D":
        space.u[:, 0] = left.value
    elif left.type == "N":
        space.u[:, 0] = -left.value * space.dx + space.u[:, 1]

    if right.type == "D":
        space.u[:, -1] = right.value
    elif right.type == "N":
        space.u[:, -1] = right.value * space.dx + space.u[:, -2]

    if top.type == "D":
        space.u[-1, :] = 2 * top.value - space.u[-2, :]
```

```
    elif top.type == "N":
        space.u[-1, :] = -top.value * space.dy + space.u[-2, :]

    if bottom.type == "D":
        space.u[0, :] = 2 * bottom.value - space.u[1, :]
    elif bottom.type == "N":
        space.u[0, :] = bottom.value * space.dy + space.u[1, :]


def SetVBoundary(space, left, right, top, bottom):
    if left.type == "D":
        space.v[:, 0] = 2 * left.value - space.v[:, 1]
    elif left.type == "N":
        space.v[:, 0] = -left.value * space.dx + space.v[:, 1]

    if right.type == "D":
        space.v[:, -1] = 2 * right.value - space.v[:, -2]
    elif right.type == "N":
        space.v[:, -1] = right.value * space.dx + space.v[:, -2]

    if top.type == "D":
        space.v[-1, :] = top.value
    elif top.type == "N":
        space.v[-1, :] = -top.value * space.dy + space.v[-2, :]

    if bottom.type == "D":
        space.v[0, :] = bottom.value
    elif bottom.type == "N":
        space.v[0, :] = bottom.value * space.dy + space.v[1, :]

# 压力边界条件
def SetPBoundary(space, left, right, top, bottom):
    if left.type == "D":
        space.p[:, 0] = left.value
    elif left.type == "N":
        space.p[:, 0] = -left.value * space.dx + space.p[:, 1]

    if right.type == "D":
        space.p[1, -1] = right.value
    elif right.type == "N":
        space.p[:, -1] = right.value * space.dx + space.p[:, -2]

    if top.type == "D":
        space.p[-1, :] = top.value
```

```
    elif top.type == "N":
        space.p[-1, :] = -top.value * space.dy + space.p[-2, :]

    if bottom.type == "D":
        space.p[0, :] = bottom.value
    elif bottom.type == "N":
        space.p[0, :] = bottom.value * space.dy + space.p[1, :]
```

按照 2.1 节中介绍的方法，我们首先编写实现 2D 域左右、顶部和底部边界水平速度(u)、垂直速度(v)和压力(p)的边界条件的函数。该函数将接受 Space 和 Boundary 类的对象，并根据这些对象的属性设置边界条件。例如，如果一个类型为 Dirichlet 且值为 0 的 Boundary 对象作为左边界对象传递，该函数将在左边界上设置该条件。

```
# 设置函数
def SetTimeStep(CFL, space, fluid):
    with np.errstate(divide='ignore'):
        dt = CFL / np.sum([np.amax(space.u) / space.dx, np.amax(space.v) /
space.dy])
    # Escape condition if dt is infinity due to zero velocity initially
    if np.isinf(dt):
        dt = CFL * (space.dx + space.dy)
    space.dt = dt


# 时间间隔函数
```

在我们编写有限差分函数之前，我们需要确定一个时间步长来使模拟向前进行。为了保证有限差分方法的收敛性，采用了基于 Courant-Friedrichs-Lewy （CFL）准则的时间步长的上界，并将其作为时间步长的上界，利用 SetTimeStep 函数进行仿真。遵循 CFL 准则可以保证时间步长传播的信息不超过两个网格单元之间的距离。

```
def GetStarredVelocities(space, fluid):
    rows = int(space.rowpts)
    cols = int(space.colpts)
    u = space.u.astype(float, copy=False)
    v = space.v.astype(float, copy=False)
    dx = float(space.dx)
    dy = float(space.dy)
    dt = float(space.dt)
    S_x = float(space.S_x)
    S_y = float(space.S_y)
    rho = float(fluid.rho)
    mu = float(fluid.mu)
```

```
    u_star = u.copy()
    v_star = v.copy()

    u1_y = (u[2:, 1:cols + 1] - u[0:rows, 1:cols + 1]) / (2 * dy)
    u1_x = (u[1:rows + 1, 2:] - u[1:rows + 1, 0:cols]) / (2 * dx)
    u2_y = (u[2:, 1:cols + 1] - 2 * u[1:rows + 1, 1:cols + 1] + u[0:rows,
1:cols + 1]) / (dy ** 2)
    u2_x = (u[1:rows + 1, 2:] - 2 * u[1:rows + 1, 1:cols + 1] + u[1:rows + 1,
0:cols]) / (dx ** 2)
    v_face = (v[1:rows + 1, 1:cols + 1] + v[1:rows + 1, 0:cols] + v[2:, 1:cols
+ 1] + v[2:, 0:cols]) / 4
    u_star[1:rows + 1, 1:cols + 1] = u[1:rows + 1, 1:cols + 1] - dt * (
          u[1:rows + 1, 1:cols + 1] * u1_x + v_face * u1_y) + (dt * (mu / rho)
* (u2_x + u2_y)) + (dt * S_x)

    v1_y = (v[2:, 1:cols + 1] - v[0:rows, 1:cols + 1]) / (2 * dy)
    v1_x = (v[1:rows + 1, 2:] - v[1:rows + 1, 0:cols]) / (2 * dx)
    v2_y = (v[2:, 1:cols + 1] - 2 * v[1:rows + 1, 1:cols + 1] + v[0:rows,
1:cols + 1]) / (dy ** 2)
    v2_x = (v[1:rows + 1, 2:] - 2 * v[1:rows + 1, 1:cols + 1] + v[1:rows + 1,
0:cols]) / (dx ** 2)
    u_face = (u[1:rows + 1, 1:cols + 1] + u[1:rows + 1, 2:] + u[0:rows, 1:cols
+ 1] + u[0:rows, 2:]) / 4
    v_star[1:rows + 1, 1:cols + 1] = v[1:rows + 1, 1:cols + 1] - dt * (
          u_face * v1_x + v[1:rows + 1, 1:cols + 1] * v1_y) + (dt * (mu / rho)
* (v2_x + v2_y)) + (dt * S_y)

    space.u_star = u_star.copy()
    space.v_star = v_star.copy()


# 速度


def SolvePressurePoisson(space, fluid, left, right, top, bottom):
    # Save object attributes as local variable with explicit typing for
improved readability
    rows = int(space.rowpts)
    cols = int(space.colpts)
    u_star = space.u_star.astype(float, copy=False)
    v_star = space.v_star.astype(float, copy=False)
    p = space.p.astype(float, copy=False)
    dx = float(space.dx)
```

```
    dy = float(space.dy)
    dt = float(space.dt)
    rho = float(fluid.rho)
    factor = 1 / (2 / dx ** 2 + 2 / dy ** 2)

    error = 1
    tol = 1e-3

    ustar1_x = (u_star[1:rows + 1, 2:] - u_star[1:rows + 1, 0:cols]) / (2 * dx)
    vstar1_y = (v_star[2:, 1:cols + 1] - v_star[0:rows, 1:cols + 1]) / (2 * dy)

    i = 0
    while error > tol:
        i += 1
        p_old = p.astype(float, copy=True)
        p2_xy = (p_old[2:, 1:cols + 1] + p_old[0:rows, 1:cols + 1]) / dy ** 2 +
(
                p_old[1:rows + 1, 2:] + p_old[1:rows + 1, 0:cols]) / dx ** 2
        p[1:rows + 1, 1:cols + 1] = p2_xy * factor - (rho * factor / dt) *
(ustar1_x + vstar1_y)
        error = np.amax(abs(p - p_old))
        # Apply Boundary Conditions
        SetPBoundary(space, left, right, top, bottom)

        if i > 500:
            tol *= 10


# 泊松压力方程


def SolveMomentumEquation(space, fluid):
    # Save object attributes as local variable with explicit typing for
improved readability
    rows = int(space.rowpts)
    cols = int(space.colpts)
    u_star = space.u_star.astype(float)
    v_star = space.v_star.astype(float)
    p = space.p.astype(float, copy=False)
    dx = float(space.dx)
    dy = float(space.dy)
    dt = float(space.dt)
    rho = float(fluid.rho)
    u = space.u.astype(float, copy=False)
```

```
    v = space.v.astype(float, copy=False)

    p1_x = (p[1:rows + 1, 2:] - p[1:rows + 1, 0:cols]) / (2 * dx)
    u[1:rows + 1, 1:cols + 1] = u_star[1:rows + 1, 1:cols + 1] - (dt / rho) *
p1_x

    p1_y = (p[2:, 1:cols + 1] - p[0:rows, 1:cols + 1]) / (2 * dy)
    v[1:rows + 1, 1:cols + 1] = v_star[1:rows + 1, 1:cols + 1] - (dt / rho) *
p1_y


# 动量方程
```

确定了时间步长后，我们现在就准备实施有限差分步骤。为了同时求解连续性方程和 Navier-Stokes 方程，我们使用了 2.1 中提到的预测-校正方法。

```
    def SetCentrePUV(space):
    space.p_c = space.p[1:-1, 1:-1]
    space.u_c = space.u[1:-1, 1:-1]
    space.v_c = space.v[1:-1, 1:-1]


# 保存边界内的速度与压力


def MakeResultDirectory(wipe=False):
    cwdir = os.getcwd()
    dir_path = os.path.join(cwdir, "Result")
    if not os.path.isdir(dir_path):
        os.makedirs(dir_path, exist_ok=True)
    else:
        if wipe:
            os.chdir(dir_path)
            filelist = os.listdir()
            for file in filelist:
                os.remove(file)

    os.chdir(cwdir)


# 将结果保存到 Result 里


def WriteToFile(space, iteration, interval):
    if iteration % interval == 0:
```

```
        dir_path = os.path.join(os.getcwd(), "Result")
        filename = "PUV{0}.txt".format(iteration)
        path = os.path.join(dir_path, filename)
        with open(path, "w") as f:
            for i in range(space.rowpts):
                for j in range(space.colpts):
                    f.write("{}\t{}\t{}\n".format(space.p_c[i, j], space.u_c[i,
j], space.v_c[i, j]))
# 在迭代间隔中将变量值进行保存
```

此段代码的作用为存储仿真中的数据和结果。

2.2.2 main_program.py

```
import sys
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import matplotlib.rcsetup

from head_file import *

matplotlib.rcParams["font.sans-serif"] = ["KaiTi"]
plt.rcParams['axes.unicode_minus'] = False
```

Matplotlib 是一个 Python 的 2D 绘图库，它以各种硬拷贝格式和跨平台的交互式环境生成出版质量级别的图形，我们后面会利用它对数据进行直观展示。

```
# 用户输入
# 定义空间、时间参数
length = 40
breadth = 40
colpts = 257
rowpts = 257
time = 20
# length 为 x 方向计算区域的长度
# breadth 为 y 方向计算区域的长度
# colpts 为 x 方向格点数（奇数）
# rowpts 为 y 方向格点数（奇数）
# time 为模拟时间



CFL_number = 0.8  # 结果收敛时，请勿改变
file_flag = 1  # 为 1 时将结果输入文档
interval = 10  # 每一个 interval 的迭代数对结果进行记录
plot_flag = 1  # 为 1 时能够将图画出
```

```
# 定义物理参数
rho = 1
mu = 0.01

# 定义动量参数
u_in = 2
v_wall = 0
p_out = 0
# u_in 为盖子处速度
# v_wall 为壁处速度
# p_out 为边界处计示压强

# 建立空间 object
cavity = Space()
cavity.CreateMesh(rowpts, colpts)
cavity.SetDeltas(breadth, length)
water = Fluid(rho, mu)

# 建立边界 object
# 速度
flow = Boundary("D", u_in)
noslip = Boundary("D", v_wall)
zeroflux = Boundary("N", 0)
# 压强
pressureatm = Boundary("D", p_out)
# 用户输入结束
```

利用 head_file.py 中建立的类、对象，对空间、边界及流体参数进行设置。

```
# 初始化
t = 0
i = 0

print("########          模拟开始          ########")
print("################################################")
print("# 模拟时间: {0:.2f}".format(time))
print("# 格点: {0} x {1}".format(colpts, rowpts))
print("# Re/u: {0:.2f}\tRe/v:{1:.2f}".format(rho * length / mu, rho * breadth
/ mu))
print("# 输出保存: {0}".format(bool(file_flag)))
MakeResultDirectory(wipe=True)

while (t < time):
    sys.stdout.write("\rSimulation time left: {0:.2f}\n".format(time - t))
    sys.stdout.flush()
```

```
    CFL = CFL_number
    SetTimeStep(CFL, cavity, water)
    timestep = cavity.dt

    SetUBoundary(cavity, noslip, noslip, flow, noslip)
    SetVBoundary(cavity, noslip, noslip, noslip, noslip)
    SetPBoundary(cavity, zeroflux, zeroflux, pressureatm, zeroflux)
    GetStarredVelocities(cavity, water)

    SolvePressurePoisson(cavity, water, zeroflux, zeroflux, pressureatm,
zeroflux)
    SolveMomentumEquation(cavity, water)

    SetCentrePUV(cavity)
    if (file_flag == 1):
        WriteToFile(cavity, i, interval)

    t += timestep
    i += 1
# 运行结束
```

程序运行，按照 2.1 中介绍的方法进行计算。

```
# 结果可视化设置
x = np.linspace(0, length, colpts)
y = np.linspace(0, breadth, rowpts)
[X, Y] = np.meshgrid(x, y)

u = cavity.u
v = cavity.v
p = cavity.p
u_c = cavity.u_c
v_c = cavity.v_c
p_c = cavity.p_c

y_g = [0, 0.0547, 0.0625, 0.0703, 0.1016, 0.1719, 0.2813, 0.4531, 0.5, 0.6172,
0.7344, 0.8516, 0.9531, 0.9609, 0.9688,
       0.9766]
u_g = [0, -0.08186, -0.09266, -0.10338, -0.14612, -0.24299, -0.32726, -
0.17119, -0.11477, 0.02135, 0.16256, 0.29093,
       0.55892, 0.61756, 0.68439, 0.75837]

x_g = [0, 0.0625, 0.0703, 0.0781, 0.0983, 0.1563, 0.2266, 0.2344, 0.5, 0.8047,
0.8594, 0.9063, 0.9453, 0.9531, 0.9609,
       0.9688]
```

```
v_g = [0, 0.1836, 0.19713, 0.20920, 0.22965, 0.28124, 0.30203, 0.30174,
0.05186, -0.38598, -0.44993, -0.23827, -0.22847,
      -0.19254, -0.15663, -0.12146]

y_g = [breadth * y_g[i] for i in range(len(y_g))]
x_g = [length * x_g[i] for i in range(len(x_g))]

if (plot_flag == 1):
    plt.figure(figsize=(20, 20))
    plt.contourf(X, Y, p_c, cmap=cm.viridis)
    plt.colorbar()
    plt.quiver(X, Y, u_c, v_c)
    plt.title("Velocity and Pressure Plot")

    plt.figure(figsize=(20, 20))
    plt.plot(y, u_c[:, int(np.ceil(colpts / 2))], "darkblue")
    plt.plot(y_g, u_g, "rx")
    plt.xlabel("Vertical distance along center")
    plt.ylabel("Horizontal velocity")
    plt.title("plot 1")

    plt.figure(figsize=(20, 20))
    plt.plot(x, v_c[int(np.ceil(rowpts / 2)), :], "darkblue")
    plt.plot(x_g, v_g, "rx")
    plt.xlabel("Horizontal distance along center")
    plt.ylabel("Vertical velocity")
    plt.title("plot 2")
    plt.show()
```

对得到的仿真数据进行处理并调用 Matplotlib 包中的函数进行可视化。

2.2.3 Animation.py

```
from __future__ import division
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

from main_program import *
# 定义读取模拟中数据的函数
def read_datafile(iteration):
    filename = "PUV{0}.txt".format(iteration)
    filepath = os.path.join(dir_path, filename)
    arr = np.loadtxt(filepath, delimiter="\t")
    rows, cols = arr.shape
```

```
    p_p = np.zeros((rowpts, colpts))
    u_p = np.zeros((rowpts, colpts))
    v_p = np.zeros((rowpts, colpts))
    p_arr = arr[:, 0]
    u_arr = arr[:, 1]
    v_arr = arr[:, 2]

    p_p = p_arr.reshape((rowpts, colpts))
    u_p = u_arr.reshape((rowpts, colpts))
    v_p = v_arr.reshape((rowpts, colpts))

    return p_p, u_p, v_p


# 结果文件
cwdir = os.getcwd()
dir_path = os.path.join(cwdir, "Result")
os.chdir(dir_path)

# 遍历 directory 中的文件
filenames = []
iterations = []
for root, dirs, files in os.walk(dir_path):
    for datafile in files:
        if "PUV" in datafile:
            filenames.append(datafile)
            no_ext_file = datafile.replace(".txt", "").strip()
            iter_no = int(no_ext_file.split("V")[-1])
            iterations.append(iter_no)

# 辨别最后的迭代与迭代数
initial_iter = np.amin(iterations)
final_iter = np.amax(iterations)
inter = (final_iter - initial_iter) / (len(iterations) - 1)
number_of_frames = len(iterations)  # int(final_iter/inter)+1
sorted_iterations = np.sort(iterations)

# 创建数组和网格
x = np.linspace(0, length, colpts)
y = np.linspace(0, breadth, rowpts)
[X, Y] = np.meshgrid(x, y)

# 定义 streamplot 的编号
index_cut_x = int(colpts / 10)
```

```
index_cut_y = int(rowpts / 10)

# 产生空白图片
fig = plt.figure(figsize=(16, 8))
ax = plt.axes(xlim=(0, length), ylim=(0, breadth))

# 初始等高线图
p_p, u_p, v_p = read_datafile(0)
ax.set_xlim([0, length])
ax.set_ylim([0, breadth])
ax.set_xlabel("$x$", fontsize=12)
ax.set_ylabel("$y$", fontsize=12)
ax.set_title("Frame No: 0")
cont = ax.contourf(X, Y, p_p)
stream = ax.streamplot(X[::index_cut_y, ::index_cut_x],
Y[::index_cut_y, ::index_cut_x],
                       u_p[::index_cut_y, ::index_cut_x],
v_p[::index_cut_y, ::index_cut_x], color="k")
fig.colorbar(cont)
fig.tight_layout()


def animate(i):
    sys.stdout.write("\rframe remain: {0:03d}".format(len(sorted_iterations) -
i))
    sys.stdout.flush()
    iteration = sorted_iterations[i]
    p_p, u_p, v_p = read_datafile(iteration)
    ax.clear()
    ax.set_xlim([0, length])
    ax.set_ylim([0, breadth])
    ax.set_xlabel("$x$", fontsize=12)
    ax.set_ylabel("$y$", fontsize=12)
    ax.set_title("Frame No: {0}".format(i))
    cont = ax.contourf(X, Y, p_p)
    stream = ax.streamplot(X[::index_cut_y, ::index_cut_x],
Y[::index_cut_y, ::index_cut_x],
                           u_p[::index_cut_y, ::index_cut_x],
v_p[::index_cut_y, ::index_cut_x], color="k")
    return cont, stream


print("########          animation          ########")
print("##############################################")
```

```
anim = animation.FuncAnimation(fig, animate, frames=number_of_frames,
interval=50, blit=False)
movie_path = os.path.join(dir_path, "Animation.gif")
anim.save(r"{0}".format(movie_path), writer=PillowWriter())
print("\n 动画在 Result 中保存为 Animation.gif")
```

对 main_program.py 中得到的数据进行可视化，并将其转化为 gif 格式。

3. 设计成果

3.1 功能展示

对一定范围内初始条件的不可压缩流体进行二维的仿真，部分仿真结果如下：

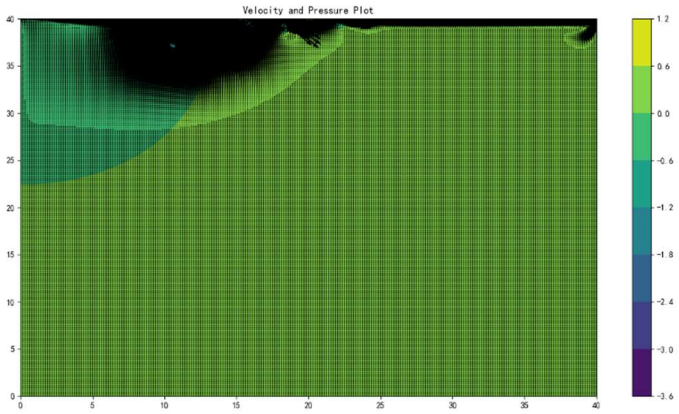


图 3-1

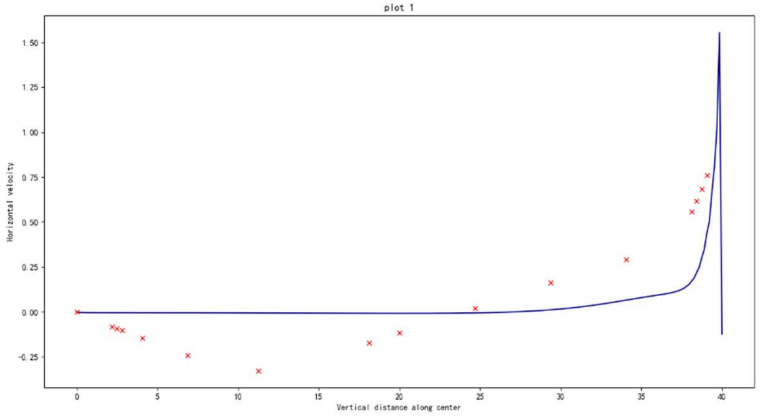
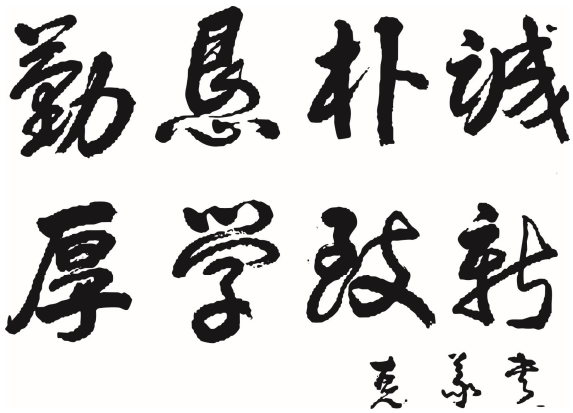


图 3-2


勤恳朴诚
厚学改新