Project III: Implementation of Chorin's projection method to solve 2-dimensional Navier-Stokes equation

Assumptions:

1. Boussinesq approximation

Summary: The goal of the project is to solve the incompressible navier stokes equation to simulate a phenomenon similar to motion of a fluid in a cooking pot. The mechanism used to formulate the flow is similar to that of a small scale weather generation code. The equations that are being solved for the purpose are:

$$\partial_t u + u \cdot \nabla u + \frac{1}{\rho} \nabla p = \nu \Delta u + F \quad (1)$$

$$\nabla u = 0 \quad (2)$$

the boundary conditions are: $u(t, 0, z) = u(t, 1, z)$ and $u(t, y, 0) = 0 = w(t, y, 1)$ and F = **β**Tez. For part 3.2 F = 0.

*u, w are vectors representing horizontal and vertical velocity respectively.

CICD PIPELINES(DILAY)

I built a very basic pipeline to use a basic test code in our Project. In the server the code and the pipelines are preserved. It manages the pipeline execution and knows what needs to be done, basically. Connected to the server, there are gitlab runners and they execute the pipeline. Gitlab offers free runners and the system is very easy compared to other options.

The pipeline is written in code and it is hosted in our repository. The configuration is written in YAML format and the name for the file has to be .gitlab-ci.yml so that the gitlab can detect the code and execute it. I wrote a job for the test and called it first-one. The script line is where we need to write the command to be executed. To execute the test file which is dilaytest.py, I used pytest command. Since the job is executed by a runner and a Gitlab runners use docker containers. And these containers run based on Docker images. Depending on our image, we would have different tools available inside the container. In gitlab, runners use a ruby image to start the container. But our test was a python file and it needs an available pip and similar tools inside the docker container. That is why i overwrote the image and changed it to a python image. The version wasn't important but with new changes in the image we would run into problems in the future so i specified a tag.

```
before_script:
  - pip install -r requirements.txt
build first-one:
    image: python:3.9-slim-buster #for the container to have python and pip tools
    script:
      - echo "Hello, I am trying to test."
      - pytest dilaytest.py
```

I get the required elements before the script command runs with pip install –r requirements.txt. pytest works my dilaytest.py file. In dilaytest.py, I defined the variables and check if the variables are all instances.

```
from input_Navier import Navier_stokes_variables

def test_for_CICD():
    NX = 20
    NY = 20
    DOMAIN_SIZE_X = 0
    N_ITERATIONS = 0
    N_PRESSURE_POISSON_ITERATIONS = 0
    TIME_STEP_LENGTH = 0
    STABILITY_SAFETY_FACTOR = 0
    KINEMATIC_VISCOSITY = 0
    DENSITY = 0
    y = Navier_stokes_variables.input_variables(NX, NY, DOMAIN_SIZE_X, N_ITERATIONS, N_PRESSURE_POISSON_ITERATIONS, TIME_STEP_LENGTH, STABILITY_SA

    all(isinstance(x, int) for x in y)
```

the test file and the job logs can be seen in the figures.

```
48 You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade pip' command.
49 $ echo "Hello, I am trying to test."
50 Hello, I am trying to test.
51 $ pytest dilaytest.py
52 ============================= test session starts ==============================
53 platform linux -- Python 3.9.16, pytest-7.2.1, pluggy-1.0.0
54 rootdir: /builds/css-2022-project-3/incompressible-navier-stokes-equations
55 collected 1 item
56 dilaytest.py .                                                          [100%]
57 ============================== 1 passed in 0.08s ===============================
59 Cleaning up project directory and file based variables
61 Job succeeded
```

LITTLE GIT GUIDE(DILAY)

The basic commands on how to commit and push will be given in this guide. First, we clone our repository with:

‘git clone <repository link>’

If this is not our first time in this repository, we can go to the file with:

‘cd <filename>’

Then we check which branch we are:

'git status'

If we are in main, we need to change our branch to a feature branch so that we don't code directly into main. We create a new branch with:

'git checkout –b <branchname>'

And we can check if we are on the right branch with git status again. After this we g oto the file that our repository's copy lies in and add our code files. Then we go to command window and write:

'git status' again, so that we can see the files that are waiting to be committed. After this, we can add our files to our commit with:

'git add .'

This command will add all the changes and untracked files.

'git commit –m <our commit message>'

This command will make our commit. And to push to the repository you can use:

'git push origin <branchname>'

Gitlab will create a merge request link and by following it you can create a merge request.

We used Git for the maintainment of the Project. It is the most known and used control system in our day. Git is flexible in making changes on the Project and it has different workflows which we could use effectively. (1)

In choosing the git workflow, I used these three questions;

1. Is our team compact enough for this workflow?
2. Will it be easy to solve mistakes and errors made by team members in this flow?
3. Does the workflow bring an unnecessary new burden to my team members? (1)

My team was made out of 5 members and the workflow needed to be simple enough so they wouldn't have any problems with merging and navigating through the code. There are 5 different workflow types that are useful.

1. Basic Workflow
2. Feature Branch Workflow
3. Git Flow
4. Gitlab Flow
5. Forking Workflow (2)

On basic workflow (aka Centralized Workflow), there is one central repository. Each developer clones the repo and works on their own code, make commit and then put the code in the central repository for other developers. One problem with this is, there is one branch which is the master (main) branch. We would be facing a lot of merge conflicts so we decided it wasn't a good idea. (1) (2)

In Feature Branch Workflow, we have different feature branches that we can code in and we don't have to commit to the master (main) branch right away. (3) Additionally, before the branch is merged to the master branch, it needs to be verified by another senior in the Project. (2) This allows the code to be much more clear before getting in the master branch. It is not a suitable workflow for projects which need different kinds of one product but our Project was simple enough for this workflow. And it was easy to understand as well. (1)

In Git Flow workflow, the difference from Feature Branch Flow is, the members need to create a branch from the develop branch. They can't create a direct branch from the master branch, which allows the main code in master branch to be very clean. (2) But it would make our git history very crowded with a lot of merge commits and we didn't need a very complicated workflow. (4)

Gitlab Flow is an extended version of Git Flow and it has different types of branches. We stage on the master branch while the high level branch is production branch. (4) Other than these branches, there are feature branches and release branches as well. This would complicate the Project work and put more burden on the members. (2)

The Forking Workflow gives the team members two different git repositories. One of them is a local, private repository and the other one is is the public server-side one. (2) In contrast to cloning a repository, forking creates a break in the link between the code. (5) Since other team members can't see what the team member who forked a copy of the repository is doing with it, it seemed unnecessary to go with this kind of workflow.

After looking at different types of workflows, we chose the Feature Branch Workflow. It suited our Project, it was easy enough for team members to manage the merging and committing and it didn't put a new burden to their shoulders.

PART 3.2 INTRODUCTION (Linear advection)(MOSLEM)

We use different numerical methods for different processes. That is why, when PDE's and ODE's have various subjects that express different physics, we use operator splitting to solving the linear advection equation:

$$\frac{\partial \vec{u}}{\partial t} + \vec{a}.\nabla\vec{u}=0$$

With considering that:

$$\vec{u} = v\vec{j} + w\vec{k} \qquad \vec{a} = V\vec{j} + W\vec{k}$$

Then we have:

$$\vec{j}: \quad \frac{\partial v}{\partial t} + V * \frac{\partial v}{\partial y} + W * \frac{\partial v}{\partial z}=0$$

$$\vec{k}: \quad \frac{\partial w}{\partial t} + V * \frac{\partial w}{\partial y} + W * \frac{\partial w}{\partial z}=0$$

Because both of the equation are same, here we explain on one of them (y direction). For operator splitting technique, by having two operators in two different directions (y, z) we can divide the advection equation (y direction) to two simpler equations as bellow:

$$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 \qquad \mathcal{L}_1 = V\frac{\partial}{\partial y} \qquad \mathcal{L}_2 = W\frac{\partial}{\partial z}$$

Calculating the velocity by sweep in y direction in first, half of each step time and then calculate the new velocity by sweep in z direction in next half step time. As it is shown in fig. 1.

$$\frac{v_{j,k}^{n+1/2} - v_{j,k}^{n}}{\Delta t} = -V * \frac{v_{j,k}^{n} - v_{j-1,k}^{n}}{\Delta y}$$

$$\frac{v_{j,k}^{n+1} - v_{j,k}^{n+1/2}}{\Delta t} = -W * \frac{v_{j,k}^{n+1/2} - v_{j,k-1}^{n+1/2}}{\Delta z}$$

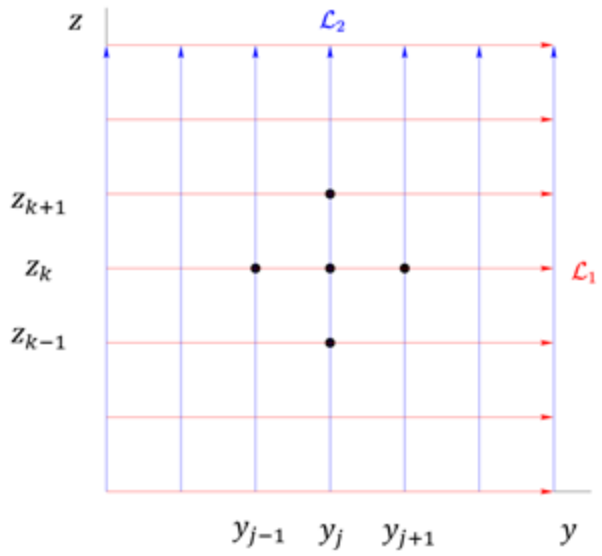


Fig. 1) Operator splitting schematic explanation

Operator splitting basically divides the equation to two different parts over a time step and calculates each part seperately. Afterwards it combines the two results to find a solution for the original equation. By splitting the equation, the computation of the solution gets faster and easier and thus, the splitting method works efficiently. (3)

For test case, the equation is solved analyticaly by using variable seperation method and solution and the baodany conditions are as bellow:

$$w(t,y,z) = e^{(-t+\frac{y}{2*V}+\frac{z}{2W})}$$

$$w(t,0,z) = \exp(-t + z/(2 * W))$$

$$w(t,y,0) = \exp(-t + y/(2 * V))$$

$$w(t,Ly,z) = \exp(-t + Ly/(2 * V) + z/(2 * W)))$$

$$w(t,y,Lz) = \exp(-t + y/(2 * V) + Lz/(2 * W)))$$

After comparing the numerical solution and the analytical solution we can calculate the error according the bellow function.

$$Error = \frac{1}{Ny * Nz}\sqrt{\sum (u_{i,j}^{exact} - u_{i,j}^{num})^2}$$

After ploting the calculated error and the number of meshes in full logaritmic scale it is clear that the error follows a first orthe decrease aganst number of meshes. Fig. 2.

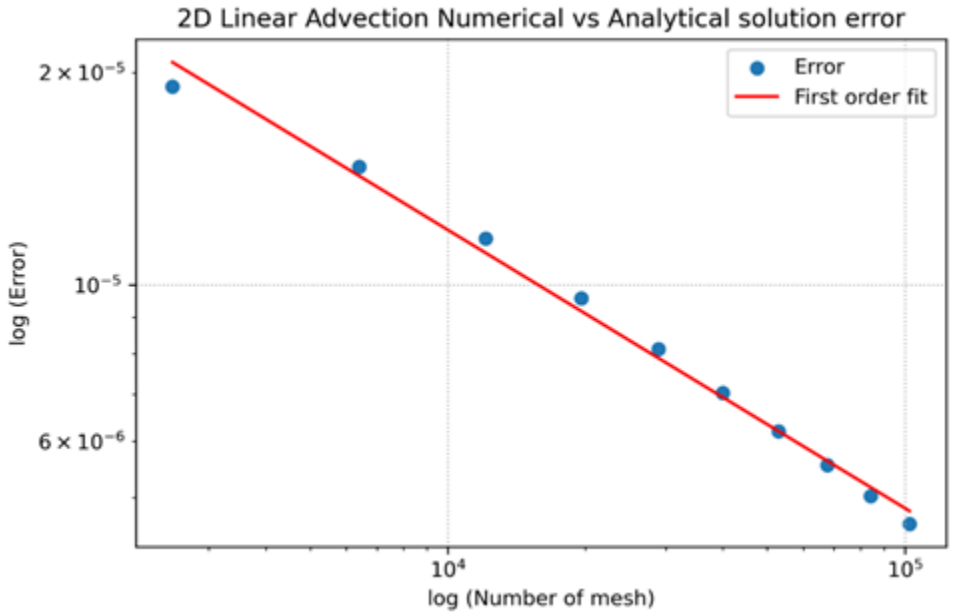


Fig. 2) Error of the numerical solution

Familiarize yourself with the concept of Chorin's projection method. Explain the necessity for a projection method. (DEEP)

Chorin's projection method is an operator splitting method which allows us to separate the differential equation into two parts calculated over a time step; the solutions from the individual parts are combined to obtain the solution to the original differential equation.

According to Chorin 1967(6), a large amount of computation power is required to solve the equations (1) - (2) due to the presence of pressure term, thus an efficient numerical method is necessary to reduce the necessary computational power. The chorin's method predominantly splits the pressure and diffusion term, and then utilizes the divergence condition in (2) to update the solution. The following steps describe the process in details:

1. First we solve (1) for an intermediate time step without the pressure term. The discritized form of (1) without the pressure term will look like, with F = 0,

$$\frac{u_{intermediate} - u_t}{\Delta t} = \left(u_{intermediate}\frac{du}{dx} + v_{intermediate}\frac{du}{dy}\right) + \nu\left(u_{intermediate}\frac{d^2u}{dx^2} + v_{intermediate}\frac{d^2u}{dy^2}\right) \quad (3)$$

Then following two functions are used t calculate horizontal and vertical velocity separately.

```
def advection_velocity_prediction_horizontal(u, v, d_u__d_x, d_u__d_y, laplace__u,  KINEMATIC_VISCOSITY, TIME_STEP_LENGTH ):

    u_half = (u + TIME_STEP_LENGTH *
            (
              -
              (
                u * d_u__d_x
                +
                v * d_u__d_y

              )
              +
             KINEMATIC_VISCOSITY * laplace__u
          )
       )

    return u_half

def advection_velocity_prediction_vertical(v, u, d_v__d_x, d_v__d_y, laplace__v,  KINEMATIC_VISCOSITY, TIME_STEP_LENGTH ):

    v_half = (v + TIME_STEP_LENGTH *
            (
              -
              (
                u * d_v__d_x
                +
                v * d_v__d_y

              )
              +
             KINEMATIC_VISCOSITY * laplace__v
          )
       )

    return v_half
```

2. The intermediate velocity obtained from step.1 can be considered as a predictor step. It is to be mentioned that effect of pressure term is not incorporated in it. Thus we need a corrector step, which is given by utlizing the eq.2.

3. We can say that the difference between the u_{t+1} and $u_{intermediate}$ corrected by utilizing $(\frac{1}{\rho})\nabla p$. If we take divergence on both side then by eq (2) we get,

$$\nabla\left(\frac{u_{t+1} - u_{intermediate}}{\Delta t}\right) = = -\left(\nabla\left(\nabla\left(\frac{1}{\rho}\right)p\right)\right) \quad (4)$$

Write a Poisson solver in two spatial dimensions.(DEEP)

Assuming the number of nodes are equal in both direction, i.e., △x=Δy, then the discretization of the rhs of equation 4 becomes,

$$(-4 * p_{i,j} + p_{i+1.j} + p_{i-1,j} + p_{i,j-1} + p_{i,j+1})/(\Delta x * \Delta x) = 0 \qquad (5)$$

The poisson solved is implemented in the following steps:

1. In the first step equation 3 is solved to obtain an intermediate velocity, the snippet of code below takes in horizontal velocity, vertical velocity, derivative of horizontal velocity w.r.t x, derivative of vertical velocity w.r.t y, laplacian of horizontal velocity and timestep to compute velocity at a point t < t+1.

```
class homogenous_advection:

  def advection_velocity_prediction(u, v, d_u__d_x, d_u__d_y, laplace__u,  KINEMATIC_VISCOSITY, TIME_STEP_LENGTH ):


    u_half = (u + TIME_STEP_LENGTH *
          (
            -
            (
              u * d_u__d_x
              +
              v * d_u__d_y

            )
            +
           KINEMATIC_VISCOSITY * laplace__u
        )
      )

    return u_half
```

   Same code can be used to calculate velocity_horizontal_intermediate and velocity_vertical_intermediate by changing the input operators.

2. In the next step we need to solve for the right hand side of the eq.4, also known as the pressure poisson equation. The code snippet below solves for the p_{t+1}.

```
class pressure_poisson:

  def pressure_solver(p, DX, rhs, N_PRESSURE_POISSON_ITERATIONS):


    for _ in range(N_PRESSURE_POISSON_ITERATIONS):

      p_next = np.zeros_like(p)
      p_next[1:-1, 1:-1] = 1/4 * (
        +
        p[1:-1, 0:-2]
        +
        p[0:-2, 1:-1]
        +
        p[1:-1, 2:  ]
```

```
            +
            p[2:  , 1:-1]
            -
            DX**2
            *
            rhs[1:-1, 1:-1]
        )


        # Pressure Boundary Conditions: Homogeneous Neumann Boundary
        # Conditions everywhere except for the top, where it is a
        # homogeneous Dirichlet BC

        p_prev = p_next

    return p_next

#-------------------------------------------------------------------------------------------#
rhs = (DENSITY / TIME_STEP_LENGTH *(d_u_tent__d_x + d_v_tent__d_y))
```

3. In this step we basically have both sides of the eq.4 and we can calculate or correct the predicted velocity (intermediate). Gradients of pressure can be calculated from step 2

```
class advection_velocity_correction:

  def advection_velocity(u_half, d_p__d_x, DENSITY, TIME_STEP_LENGTH):

    u_next = (
      u_half
      -
      TIME_STEP_LENGTH / DENSITY
      *
      d_p__d_x
    )

    return u_next
```

Do the implementation of the solver subsequently.(DEEP)

In the previous steps we have calculated homogenous advection, then solved pressure poisson equation, using which we then corrected the velocity we found from homogenous advection. The correction keeps the solution incompressible.

To implement the class_homogenous_advection, class_pressure_poisson and class_advection_velocity_correction, we need to build two more classes, i.e., class_mesh_grid and class_discritization_schemes:

The mesh grid class takes the number of elements and domain sizes as input and returns two 2d arrays X, Y, which are used to plot the mesh. It also returns element length in x and y directions.

```
class mesh_grid():

   def __init__():
      pass
#-------------------------------------------------#
   @abstractmethod
   def mesh(NX, NY, DOMAIN_SIZE_X, DOMAIN_SIZE_Y):
#-------------------------------------------------#
      DX = 2/(NX -1) #element length in x direction
      DY = 2/(NY -1) #element length in y direction
      element_length = DX
#-------------------------------------------------#
      x = np.linspace(0, DOMAIN_SIZE_X, NX) #range in xdirection
      y = np.linspace(0, DOMAIN_SIZE_Y, NY) #range in ydirection
      X, Y = np.meshgrid(x,y) #X, Y are 2d arrays containing same range again and again
#-------------------------------------------------#
#-------------------------------------------------#
      plot_mesh.mesh_plot(X,Y)

      return [X, Y, DX, DY]
```

The other class contains the discritization schemes that calculates a discritized value matrix by applying the coded equation pointwise to the original matrix.

```
class discretization_schemes():


#-------------------------------------------------#
   def central_difference_x(f, DX):
         diff = np.zeros_like(f)
         diff[1:-1, 1:-1] = (
            f[1:-1, 2:  ]
            -
            f[1:-1, 0:-2]
         ) / (
            2 * DX
         )
         return diff




#-------------------------------------------------#
#-------------------------------------------------#
   def central_difference_y(f, DY):
      diff = np.zeros_like(f)
      diff[1:-1, 1:-1] = (
            f[2:  , 1:-1]
            -
            f[0:-2, 1:-1]
         ) / (
            2 * DY
         )

      return diff
#-------------------------------------------------#
#-------------------------------------------------#
   def upwind_x(f, DX):

      diff = np.zeros_like(f, dtype = np.longdouble)
      diff[1:-1, 1:-1] = (

            f[1:-1, 1: -1 ]
            -
            f[1:-1,  : -2]
         ) / (
            DX
         )
```

```
        return diff
#-----------------------------------------------#
    def upwind_y(f, DY):

        diff = np.zeros_like(f, dtype = np.longdouble)
        diff[1:-1, 1:-1] = (

                f[1:-1, 1: -1 ]
                -
                f[: -2, 1:-1]
            ) / (
                DY
            )

        return diff
#-----------------------------------------------#

#-----------------------------------------------#
    def laplace(f, DX):
        diff = np.zeros_like(f)
        diff[1:-1, 1:-1] = (
                f[1:-1, 0:-2]
                +
                f[0:-2, 1:-1]
                -
                4
                *
                f[1:-1, 1:-1]
                +
                f[1:-1, 2:  ]
                +
                f[2:  , 1:-1]
            ) / (
                DX**2
            )
        return diff
#-----------------------------------------------#
```

After chorins operator splitting the code uses class_homogenous_advection to advect velocity and temperature. Then after solving for the pressure in next step using class_pressure_poisson, we correct the advected velocity using class_advection_velocity correction in Physics file.

class_homogenous_advection can be used to simulate homogenous (neu = 0) and nonhomogenous (neu != 0) both by changing the KINEMATIC_VISCOSITY or neu value passed to the function.

Implement Chorin's projection method [1] for the Navier Stokes equations (2) – (3) with β = 0 using the numerical methods built so far. Implement initial conditions to test your code, check the consistency order and visualize the data. (DEEP)

The code snippet below explains the initial condition and boundary condition for lid_driven cavity. The solution arrays were intialized using zero value.

```
#-----------------------------------------------#
#for lid driven cavity
class boundary_update():
    def __init__(self):
        pass
```

```
    def velocity_boundary_x(u_tent):
        u_tent[0, :] = 0.0
        u_tent[:, 0] = 0.0
        u_tent[:, -1] = 0.0
        u_tent[-1, :] = 10

        return u_tent
    #----------------------------------------------------#
    def velocity_boundary_y(v_tent):
        v_tent[0, :] =  0
        v_tent[:, 0] = 0.0
        v_tent[:, -1] = 0.0
        v_tent[-1, :] = 0

        return v_tent
    #----------------------------------------------------#
    def pressure_boundary(p_next):
        p_next[:, -1] = p_next[:, -2]
        p_next[0,  :] = p_next[1,  :]
        p_next[:,  0] = p_next[:,  1]
        p_next[-1, :] = 0.0

        return p_next
    #----------------------------------------------------#
class initial_condition():
    def __init__(self):
        pass

    def matrix_initialization(NX, NY, zero_initialization, intial_value):

        if zero_initialization == True:

            m_prev = np.zeros([NX,NY], dtype = np.longdouble)


        else:
            m_prev = np.full([NX,NY], intial_value,dtype = np.longdouble)

        m_tent = np.zeros([NX,NY], dtype = np.longdouble)
        m_next = np.zeros([NX,NY], dtype = np.longdouble)


        return [m_prev, m_tent, m_next]
#----------------------------------------------------#
```

The following snippet of code solves the intermediate or tent velocity and then solves the velocity for nex time step.

```
#----------------------------------------------------#

 u_tent =homogenous_advection.advection_velocity_prediction_horizontal(u_prev, v_prev, d_u_prev__d_x,  d_u_prev__d_y, laplace__u_prev,
KINEMATIC_VISCOSITY, TIME_STEP_LENGTH)
        v_tent =homogenous_advection.advection_velocity_prediction_vertical(v_prev, u_prev, d_v_prev__d_x,  d_v_prev__d_y, laplace__v_prev,
KINEMATIC_VISCOSITY, TIME_STEP_LENGTH)

#----------------------------------------------------#
        # Velocity Boundary Conditions: Homogeneous Dirichlet BC everywhere
        # except for the horizontal velocity at the top, which is prescribed

        u_tent = boundary_update.velocity_boundary_x(u_tent)
        v_tent = boundary_update.velocity_boundary_y(v_tent)
#----------------------------------------------------#
#----------------------------------------------------#

u_next = advection_velocity_correction.advection_velocity(u_tent, d_p_next__d_x, DENSITY, TIME_STEP_LENGTH  )
        v_next = advection_velocity_correction.advection_velocity(v_tent, d_p_next__d_y, DENSITY, TIME_STEP_LENGTH  )    + bouancy  *
TIME_STEP_LENGTH#'Add temperature part'
#----------------------------------------------------#
        # Velocity Boundary Conditions: Homogeneous Dirichlet BC everywhere
        # except for the horizontal velocity at the top, which is prescribed

        u_next = boundary_update.velocity_boundary_x(u_next)
        v_next = boundary_update.velocity_boundary_y(v_next)
#----------------------------------------------------#
#----------------------------------------------------#
```

The results are in the result section.

Ray-leigh benard convection:(DEEP AND GAMAL)

Convection is the process of moving thermal energy in a fluid as a result of a temperature difference , with natural convection defined as fluid movement due to changing densities in the fluid, or a buoyancy force. A layer of fluid is heated from below and a temperature difference is established. The fluid at the bottom becomes less dense than the fluid at the top, which gives rises to a top-heavy arrangement that is potentially unstable, due to this instability, the fluid will tend to redistribute itself to remedy the weakness in its arrangement, resulting in circular movement of the fluid (Rayleigh-B´enard), the temperature starts fluctuating in space and time once the convection has established in the system .

Now, the F term in equation 1 is not zero anymore, F = **β**T(ez), **β** = thermal expansion coefficient. We solve the temperature part using,

$\partial_t T + \vec{u} \cdot \nabla T = \Delta T$ Heat equation

```
#------------------------------------------------#
T_next = homogenous_advection.advection_temperature(T, u_prev, v_prev, d_T__d_x, d_T__d_y, laplace_T, 1, TIME_STEP_LENGTH )
#------------------------------------------------#
class homogenous_advection:

  def advection_temperature(T, u, v, d_T__d_x, d_T__d_y, laplace_T,KINEMATIC_VISCOSITY, TIME_STEP_LENGTH):

        T_next = (T + TIME_STEP_LENGTH *

         (

           -

           (

             u * d_T__d_x

             +

             v * d_T__d_y

           )

           +

           KINEMATIC_VISCOSITY * laplace_T

         )

        )

    return T_next
#------------------------------------------------#
```

we add the contribution of the temperature to the vertical velocity only because of the ez term. ez term is an unit vector pointing towards vertical direction. Buoyancy = F * TIME_STEP_LENGTH

```
#-------------------------------------------------#

u_next = advection_velocity_correction.advection_velocity(u_tent, d_p_next__d_x, DENSITY, TIME_STEP_LENGTH  )

 v_next = advection_velocity_correction.advection_velocity(v_tent, d_p_next__d_y, DENSITY, TIME_STEP_LENGTH  )    + bouancy  *
TIME_STEP_LENGTH#'Add temperature part'

#-------------------------------------------------#
```

we have to find a resendable scheme in order to git the next step (time) temperature which is very important to get also the next velocity and can also plot the classic Rayleigh-Benard convection .

That was based on the pervious scheme developed on before so we go face center scheme , now we are looking for the next step Temperature using the information of the current temperature and current velocity .

```
def advection_temperature(T, u, v, d_T__d_x, d_T__d_y, laplace_T,KINEMATIC_VISCOSITY, TIME_STEP_LENGTH):

T_next = dask.delayed((T + TIME_STEP_LENGTH * (- (u * d_T__d_x + v *d_T__d_y)+ KINEMATIC_VISCOSITY * laplace_T)))
```

so from this function we get the next temperature on the time scaler.

The velocity boundary conditions as will be logical thinking will be zero as every border of our system, and from the previous scheme the velocity will be update based on this scheme for velocity next,

```
def velocity_boundary_x(u_next):

u_next[:, -1] = u_next[:, -2]

u_next[0, :] = u_next[1, :]

u_next[:, 0] = u_next[:, 1]

u_next[-1, :] = u_next[-2, :]

def velocity_boundary_y(v_next):

v_next[:, -1] = v_next[:, -2]

v_next[0, :] = v_next[1, :]

v_next[:, 0] = v_next[:, 1]

v_next[-1, :] = v_next[-2, :]#
```

The temperature boundary condition is just added as a perturbation and we are not using the actual temperature value to maintain the order of magnitude when compared to other terms,

```
T_next[0,16:26] = 1.2 # for 41 nodes =NX=NY

T_next[:,0] = 0

T_next[:,-1] = 0
```

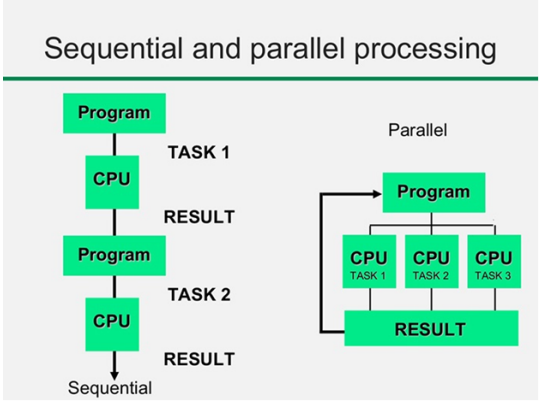
```
T_next[-1, :] = T_next[-2, :]
```

The results are in the result section.

Parallelization(SAEID)

Parallelization is the act of designing a computer program or system to process data in parallel. Normally, computer programs compute data serially (sequentially): they solve one problem, and then the next, then the next. If a computer program or system is parallelized, it breaks a problem down into smaller pieces to be independently solved simultaneously by discrete computing resources. When optimized for this type of computation, parallelized programs can arrive at a solution much faster than programs executing processes in serial.

Parallelization as a computing technique has been used for many years, especially in the field of supercomputing. Each new generation of processors approaches the physical limitations of microelectronics, a major engineering concern in CPU design. Because individual chips are approaching their fastest possible speeds, parallel processing becomes an important area where to improve computing performance. The majority of modern desktop computers and laptops have multiple cores on their CPU that help parallel processing in the operating system.



Concurrent.futures(SAEID)

The concurrent.futures module provides a high-level interface for asynchronously executing callable.

The asynchronous execution can be performed with threads, using ThreadPoolExecutor, or separate processes, using ProcessPoolExecutor. Both implement the same interface, which is defined by the abstract Executor class.

ProcessPoolExecutor:(SAEID)

The ProcessPoolExecutor class is an Executor subclass that uses a pool of processes to execute

calls asynchronously. ProcessPoolExecutor uses the multiprocessing module, which allows it to side-step the Global Interpreter Lock but also means that only packable objects can be executed and returned.

The __main__ module must be importable by worker sub-processes. This means that ProcessPoolExecutor will not work in the interactive interpreter.

Calling Executor or Future methods from a callable submitted to a ProcessPoolExecutor will result in a deadlock.

An Executor subclass executes calls asynchronously using a pool of at most max_workers processes. If max_workers is None or not given, it will default to the number of processors on the machine. If max_workers is less than or equal to 0, then a ValueError will be raised.

We used this library to splitting the grid system so the whole simulation will perform on splatted grid parts which makes the code run much faster.

Dask(SAEID)

Dask is a flexible library for parallel computing in Python.

Dask is composed of two parts:

1. “Dynamic task scheduling” optimized for computation. This is similar to “Airflow, Luigi, Celery, or Make”, but optimized for interactive

computational workloads.

2. “Big Data" collections” like parallel arrays, data frames, and lists that extend common interfaces like “NumPy, Pandas, or Python iterators” to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

Dask emphasizes the following virtues:

1. **Familiar**: Provides parallelized NumPy array and Pandas Data Frame objects.
2. **Flexible**: Provides a task scheduling interface for more custom workloads and integration with other projects.
3. **Native:** Enables distributed computing in pure Python with access to the PyData stack.
4. **Fast**: Operates with low overhead, low latency, and minimal serialization necessary for fast numerical algorithms

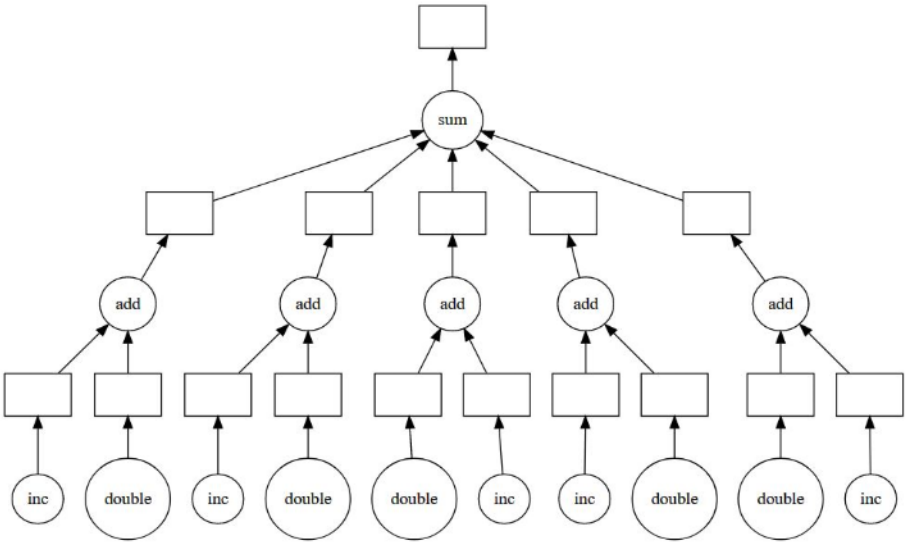
5. **Scales up**: Runs resiliently on clusters with 1000s of cores

6. **Scales down**: Trivial to set up and run on a laptop in a single process

7. **Responsive**: Designed with interactive computing in mind, it provides rapid feedback and diagnostics to aid humans.

Dask Delayed:(SAEID)

Sometimes problems don't fit into one of the collections like dask.array or dask.dataframe. In these cases, users can parallelize custom algorithms using the simpler dask.delayed interface. This allows you to create graphs directly with a light annotation of normal python code. In this project we used dask.delayed to parallelizing the functions in the simulation core.



Why we used these libraries?

1. They create the least errors

2. They are compatible with most the machines

3. Designed to use for in scientific programming and computing

4. They are Fast and reliable

Why we did not use GPU?

Because using GPU will limit the machines you can run the code on them; for example, Parallel code designed to run on Nvidia GPUs will raise errors on AMD GPUs.

But last, there might be much more efficient ways to parallelize this code like using a “parallelize algorithm” which needs a high level of algorithm design ability.

Instruction to use the code:(DEEP)

For running Lid driven cavity and Rayleigh benard convection solve lid_driven_cavity.py and rayleigh_benard.py.

1. Open simulation.py for setting up input parameters and discritization schemes:
   a) call_inputvars: number of nodes, number of iterations,
      pressure_poisson_iterations, TIME_STEP_LENGTH,
      KINEMATIC_VISCOSITY, DENSITY

   b) call_mesh_grid: Prints grid, and returns element length

   c) call_discritization_schemes: shift between central_difference and upwind

2. #Parameters: beta = 0.0, temperature dependence is off
   beta -> [0,1]

3. Solution loop:

   a) #Intial conditions: u, v, P matrices are intialized with zeros. using the
      function matrix_intialization. By defining
      zero_intialization = False, and passing a value
      to intial_value = value, we can intialize matrices
      with non_zero matrices.

   b) #Selecting up discretization schemes: calls the call_discritization_schemes

   c) Boundary_update: Contains three functions: velocity_boundary_x(u_tent),
      velocity_boundary_y(v_tent), pressure_boundary(p_next),
      temperature_boundary(T_next).

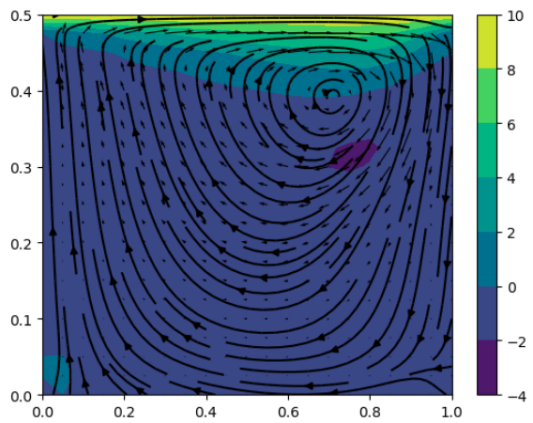
   d) #modify timestep based on CFL number: If the CFL number is greater than 1
      in x or y direction then timestep is recalculated.

4. Test:

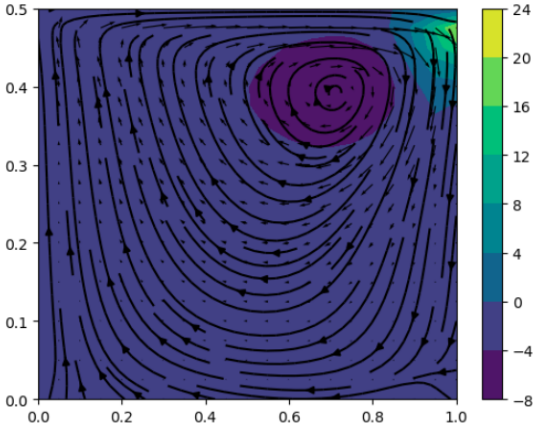
   a) time_step_length > 0.001 (Check under def test_input(input_return_list: list))
   b) cfl_x < 1 and cfl_y <1 (check test_CFL_number_calculation)

Results:

lid_driven_cavity:

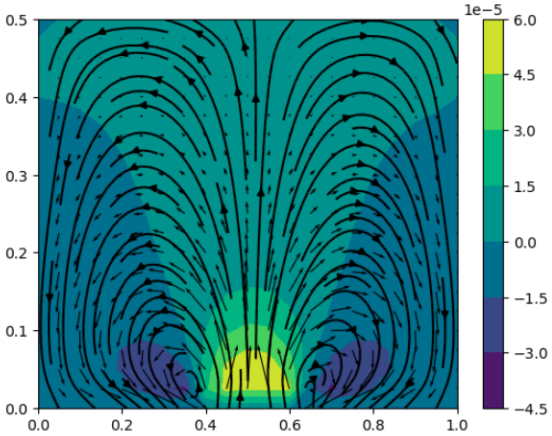


lid driven cavity horizontal velocity contour with (u,v) streamline plots (DEEP)

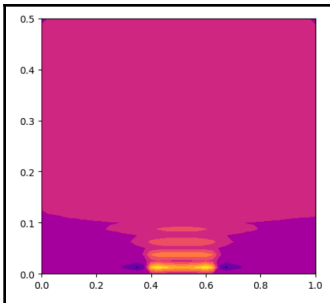


lid driven cavity horizontal pressurecontour with (u,v) streamline plots(DEEP)

Rayleigh benard:

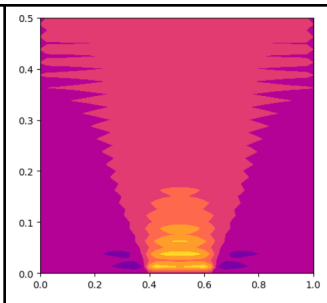


Vertical velocity plot in Rayleigh benard convection(DEEP)



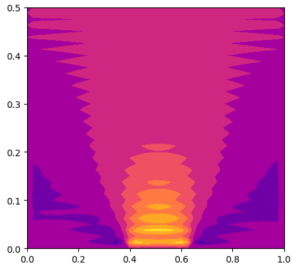
iterations: 500

Temperature plot Rayleigh benar (DEEP)



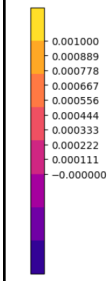
iterations: 20,000

Temperature plot Rayleigh benard(DEEP)



iterations: 40,000

Temperature plot Rayleigh benard(DEEP)



Learning:

Deep:

1. Solving poisson equation using discritization and numerical methods.
2. Understood the implementation of predictor - corrector method while learning chorins operator.
3. Understood that while solving the order of magnitude of source term should be maintained and the source term initial and boundary conditions should be provided accordingly.
4. The solution values are stored at nodes, thus we understood that it is tricky to imply some specific boundar conditions and a change of order of discretization schemes can cause boundary layers near wall.

Saeid:

1. What parallelization is and how it works.
2. Using DASK library for parrallelizing
3. Using futures3 library for parrallelizing
4. Working whit git lab, specially CI/CD and pipeline
5. Solving equations using numerical methods and code them

Dilay:

1. GIT AND GITLAB

   Git is a tool for source code management. It makes the communication and file share easier. It watches over the development and the commits the author makes is recorded by git. With this feature we can easily use different versions of the Project. Addditionally it gives us the feature of branches. Branches keep the code changes clean from the main Project and they can be used in many different ways. With them we can create a workflow and use it fort he development of the Project to be easier.

   GitLab is an open source code repository and collaborative software development platform for large DevOps and DevSecOps projects. It is free and it offers a location for online code storage and capabilities for issue tracking and CI/CD.

   Git workflow is a guide on how to use git to produce useful code in a stabile way. Without a workflow, the team would end up in a chaotic situation with the commits all complicated in the main branch. That is why, we needed a git workflow and I have come up with the feature branch model for it.

2. CONTINUOUS INTEGRATION AND PYTEST

   CICD works to test, build and release code changes to the deployment environment. Continuous integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project. It allows developers to frequently merge code changes into a central repository where builds and tests then run.

Automated tools are used to assert the new code's correctness before integration.

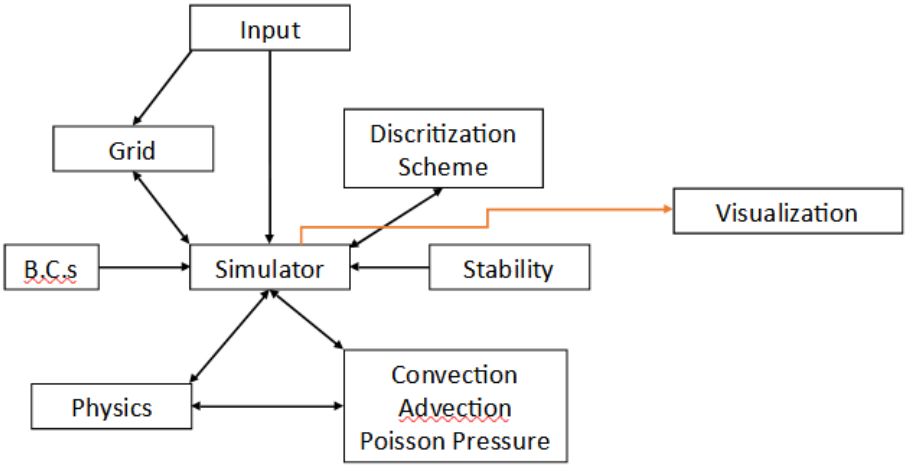
3. Why did we use feature branch flow?

   The workflows like fork, git and gitlab workflow were too complicated and burdening for our Project. We didn't need to go to this extend to keep our code clean. Basic workflow was too exposing for our main branch and the commits would get mixed up at some point.

4. Why was CI important for us?

   Running tests to assest the code that is in the new commit is a very good way to keep the code clean consistently. The pytest is excellent at test discovery. That is why we used these methods.

Diagram:



Deep	Moslem	Saeid	Gamal	Dilay
Part 3.3 **Part 3.4**	**Part 3.2**	**Part 3.5**	**Part 3.4**	**Part 3.1**
File_name: **class name** **function name** File: lid-driven cavity File: rayleigh bernard File: simulation **input_Navier**: class Navier_stokes_variables Method: input_variables **grid**: class mesh_grid Method: mesh class plot_mesh Method: mesh_plot **boundary_conditions**: class boundary_update(): Method: velocity_boundary_x velocity_boundary_y pressure_boundary temperature_boundary **intial_conditions**: class initial_condition Method: matrix_initialization **difference_equation**: class discretization_schemes() Method: central_difference_x	**File_name:** **class name** **function name** **Analytical:** Class: Analytical_sol Methods: analytical_solution **main_2advection_operators_splitting** Class: TwoD_linear_adv Methods: twoD_linear_advection **Error_linear_advection** Class: Error Methods: error_func **anim** Class: Anim Methods: my_animation	**File_name:** **class name** **function name** **Grid:** Class: mesh_grid executor.map(mesh) **Physics:** Class: homogenous_advection advection_temperature advection_velocity_prediction_horizontal advection_velocity_prediction_vertical **error_linear_advection:** eror error_func fitted_function **test:** class test_stability **Requirements.txt**	**physics**: class homogenous_advection: Method: advection_temperature	**File_name:** **class name** **function name** File: .gitlab-ci.yml File: dilaytest.py File: requirements.txt **.gitlab-ci.yml:** Build first-one **dilaytest.py:** test_for_CICD **Requirements.txt**

central_difference_y upwind_x upwind_y laplace **physics**: class homogenous_advection Method: advection_temperature advection_velocity_predictio n_horizontal advection_velocity_predictio n_vertical class pressure_poisson pressure_solver **test:** class test_stability Method: test_initial_timestep_value test_CFL_number_calculatio n test_pressure_poisson_conv ergence **simulation**: class setup_parameters Method: call_inputvars call_mesh_grid call_discretization_schemes **visualization**: class Visual Method: visualize_vector_plot visualize_contour_plot visualize_error_plot animation				

Reference:

1. Comparing Git Workflows: What You should Know. ***Atlassian Bitbucket Tutorials.*** **[Online] https://www.atlassian.com/git/tutorials/comparing-workflows#:~:text=A%20Git%20workflow%20is%20a,in%20how%20users%20manage%20changes..**

2. 5 Different Git Workflows. ***Medium.*** **[Online] 21 07 2021. https://medium.com/javarevisited/5-different-git-workflows-50f75d8783a7.**

3. Git Feature Branch Workflow. ***Atlassian Bitbucket Tutorials.*** **[Online] https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow.**

4. Introduction to GitLab Flow . ***Gitlab.*** **[Online] https://docs.gitlab.com/ee/topics/gitlab_flow.html.**

5. Git fork vs. clone: What's the difference? ***TheServerSide.*** **[Online] https://www.theserverside.com/answer/Git-fork-vs-clone-Whats-the-difference.**

6. Chorin, Alexandre Joel. "A Numerical Method for Solving Incompressible Viscous Flow Problems." ***Journal of Computational Physics*****, vol. 2, no. 1, Aug. 1967, pp. 12–26, 10.1016/0021-9991(67)90037-x. Accessed 31 Oct. 2021.**