ma_simu_cavity_classical

October 29, 2021

1 Simulateur de fluide en 2D dans une cavité

Basé sur **Numerical Simulation in Fluid Dynamics** édité par SIAM

$\rho \frac{D\vec{v}}{Dt} = -\vec{\nabla} p + \rho \vec{g} + \mu \nabla^2 \vec{v}$ On simule ici les équations de Navier-Stokes dans une cavité en 2D.

Les équations sont, pour la vitesse horizontale u et la vitesse verticale v:

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

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

(conservation de la quantité de mouvement)

$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$ (conservation de la masse)

[1]:
```
import numpy as np
import numpy.linalg as alg
import time
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import cm
```

[20]:
```
nu = 1e-5 # m^2/s
rho = 1000 # kg / m^3
g = 9.81
gx = 0
gy = -g # m/s^2
```

1.0.1 Initialisation des constantes

[21]:
```
tf = 1 # l'instant final max
n = 500 # nombre d'étapes de temps max de la simulation
dt = tf/n # quantité amenée à changer

t = 0

a = .1 # largeur en mètres de la boîte selon Ox
b = .1 # hauteur en mètres de la boîte selon Oy
```

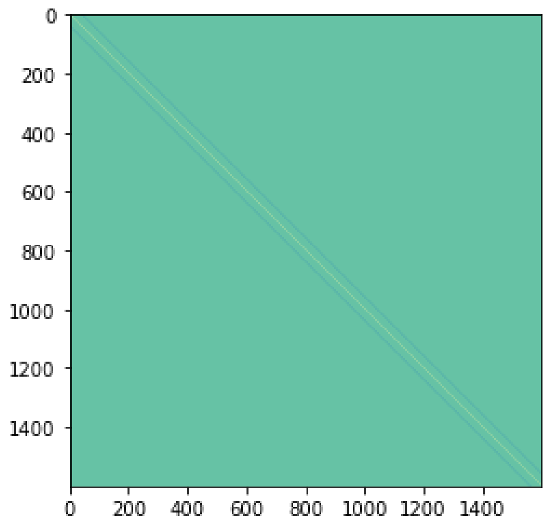
```
imax = 40 # nombre de cases en espace selon Ox
jmax = 40 # nombre de cases en espace selon Oy
s = imax*jmax

dx = a/imax
dy = b/jmax # les indices 0 et jmax+1 sont réservés aux frontières extérieures␣
 ↪à la boîte

print("taille de la matrice du système linéaire",s*s)
```

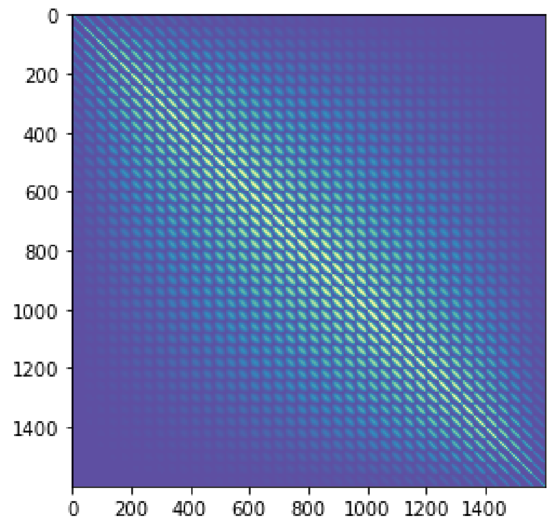
```
taille de la matrice du système linéaire 2560000
```

1.0.2 Initialisation de la matrice du système linéaire pour le laplacien

[22]:
```
# matrice du laplacien pour trouver la pression

def carre2vec(i,j): # entrées indicées à partir de 1
    return imax*(j-1)+i-1 # sortie indicée à partir de 0 (pour python)

S = np.zeros(shape=(s,s),dtype=float)

for i in range(1,imax+1):     # jusqu'à imax
    for j in range(1,jmax+1): # jusqu'à jmax
        k = carre2vec(i,j)
        S[k][k] += (-2/dx**2-2/dy**2)
        if j<jmax:
            S[k][carre2vec(i,j+1)] += 1/dy**2
        if j>1:
            S[k][carre2vec(i,j-1)] += 1/dy**2
        if i<imax:
            S[k][carre2vec(i+1,j)] += 1/dx**2
        if i>1:
            S[k][carre2vec(i-1,j)] += 1/dx**2

Si = alg.inv(S) # ne pas le faire si S est trop grande ! typiquement à partir␣
 ↪de imax ~jmax ~60 on arrête
```

[23]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 6))
imf = ax[0].imshow(S, cmap='Spectral')
ime = ax[1].imshow(Si, cmap='Spectral')# on regarde la matrice S
```





1.0.3 Initialisation des tableaux de stockage

[24]:
```
# tout est indicé de 0 à imax+1 ou de 0 à jmax+1 INCLUS
# tableau de stockage du temps
T = np.zeros(shape=(n+1,),dtype=float)

# tableaux de stockage des résultats
U = np.zeros(shape=(n+2,imax+2,jmax+2),dtype=np.float) # pour le temps variant␣
 ↪de 0 à n+1 inclus
V = np.zeros(shape=(n+2,imax+2,jmax+2),dtype=np.float)
P = np.zeros(shape=(n+2,imax+2,jmax+2),dtype=np.float)



# tableaux pour la variation temporelle
F = np.zeros(shape=(n+2,imax+2,jmax+2),dtype=np.float)
G = np.zeros(shape=(n+2,imax+2,jmax+2),dtype=np.float)

# tableaux pour la contribution implicite de la pression
dpdx = np.zeros(shape=(n+2,imax+2,jmax+2),dtype=np.float)
dpdy = np.zeros(shape=(n+2,imax+2,jmax+2),dtype=np.float)



DeltaP = np.zeros(shape=(n+2,imax+2,jmax+2)) # mêmes valeurs que le précédent␣
 ↪mais en matrice
```

[25]:
```
Gx = np.zeros(shape=(imax+2,jmax+2),dtype=np.float)
Gy = gy*np.zeros(shape=(imax+2,jmax+2),dtype=np.float)

def set1G(tt):
```

```python
    omega = 2*np.pi*1
    cx = a/2
    cy = b/2

    for i in range(imax+2):
        for j in range(jmax+2):
            x = i*dx - cx
            y = j*dy - cy
            fa = np.exp(-200*x**2-500*y**2)
            r = np.sqrt((x)**2+(y)**2)
            Gx[i][j] = .08*np.cos(.36+50*x+omega*tt)*fa*np.exp(-.5*tt)
            Gy[i][j] = .08*np.sin(.23+50*y+omega*tt)*fa*np.exp(-.5*tt)



def set2G(tt):
    omega = 2*np.pi*6
    cx = a/2
    cy = b/2
    for i in range(imax+2):
        for j in range(jmax+2):
            xx = i*dx - cx
            yy = j*dy - cy
            Gy[i][j] = yy*np.sin(omega*tt)
```

[26]:

```python
###### (3.19a)
def dUe2_dx(e,i,j):
    t1 = 1/dx/4 * ((U[e][i][j]+U[e][i+1][j])**2-(U[e][i-1][j]+U[e][i][j])**2)
    t2 = gamma/dx/4 * (abs(U[e][i][j] + U[e][i+1][j]) * (U[e][i][j] - U[e][i+1][j]) - abs(U[e][i-1][j]+U[e][i][j]) * (U[e][i-1][j]-U[e][i][j]))
    return t1 + t2

def dUV_dy(e,i,j):
    t1 = 1/dy/4 * (V[e][i][j]+V[e][i+1][j])*(U[e][i][j]+U[e][i][j+1]) - (V[e][i][j-1]+V[e][i+1][j-1])*(U[e][i][j-1]+U[e][i][j])
    t2 = gamma/dy/4*(abs(V[e][i][j]+V[e][i+1][j])*(U[e][i][j]-U[e][i][j+1]) - abs(V[e][i][j-1]+V[e][i+1][j-1])*(U[e][i][j-1]-U[e][i][j]))
    return t1 + t2

def d2U_dx2(e,i,j):
    return (U[e][i+1][j] - 2*U[e][i][j] + U[e][i-1][j])/dx/dx

def d2U_dy2(e,i,j):
    return (U[e][i][j+1] - 2*U[e][i][j] + U[e][i][j-1])/dy/dy
```

```
def dU_dx(e,i,j):
    return (U[e][i][j] - U[e][i-1][j])/dx

def dV_dy(e,i,j):
    return (V[e][i][j] - V[e][i][j-1])/dy

def dp_dx(e,i,j):
    return (P[e][i+1][j] - P[e][i][j])/dx


###### (3.19b)

def dUV_dx(e,i,j):
    t1 = 1/dx/4 * (U[e][i][j]+U[e][i][j+1])*(V[e][i][j]+V[e][i+1][j]) -␣
↪(U[e][i-1][j]+U[e][i-1][j+1])*(V[e][i-1][j]+V[e][i][j])
    t2 = gamma/dx/4*(abs(U[e][i][j]+U[e][i][j+1])*(V[e][i][j]-V[e][i+1][j]) -␣
↪abs(U[e][i-1][j]+U[e][i-1][j+1])*(V[e][i-1][j]-V[e][i][j]))
    return t1 + t2



def dVe2_dy(e,i,j):
    t1 = 1/dy/4 * ((V[e][i][j]+V[e][i][j+1])**2-(V[e][i][j-1]+V[e][i][j])**2)
    t2 = gamma/dy/4 * (abs(V[e][i][j] + V[e][i][j+1]) * (V[e][i][j] -␣
↪V[e][i][j+1]) - abs(V[e][i][j-1]+V[e][i][j]) * (V[e][i][j-1]-V[e][i][j]))
    return t1 + t2




def d2V_dx2(e,i,j):
    return (V[e][i+1][j] - 2*V[e][i][j] + V[e][i-1][j])/dx/dx

def d2V_dy2(e,i,j):
    return (V[e][i][j+1] - 2*V[e][i][j] + V[e][i][j-1])/dy/dy



def dp_dy(e,i,j):
    return (P[e][i][j+1] - P[e][i][j])/dy
###
def div(e,i,j):
    return (U[e][i][j] - U[e][i-1][j])/dx + (V[e][i][j] - V[e][i][j-1])/dy
```

1.0.4 Début de la simulation

[27]:
```
def computeFG(e):
    F[e,:,:] = U[e,:,:] # on ajoute d'abord la partie en u_i,j
    G[e,:,:] = V[e,:,:] # on ajoute d'abord la partie en v_i,j
    set1G(t) # on paramètre l'accélération de l'instant qu'on regarde
```

```
    for i in range(1,imax+1):
        for j in range(1,jmax+1): # tout ici à l'étape e
            F[e][i][j] += dt * (nu*(d2U_dx2(e,i,j) + d2U_dy2(e,i,j)) -␣
↪dUe2_dx(e,i,j) - dUV_dy(e,i,j) + Gx[i][j])
            G[e][i][j] += dt * (nu*(d2V_dx2(e,i,j) + d2V_dy2(e,i,j)) -␣
↪dVe2_dy(e,i,j) - dUV_dx(e,i,j) + Gy[i][j])

    # (3.42) du livre
    for j in range(1,jmax+1):
        F[e][0][j] = U[e][0][j]
        F[e][imax][j] = U[e][imax][j]

    for i in range(1,imax+1):
        G[e][i][0] = V[e][i][0]
        G[e][i][jmax] = V[e][i][jmax]


def computeUpdatePressure(e):
    RHSpression = np.zeros(shape=(s,),dtype=float)

    for i in range(1,imax+1): # sur l'intérieur
        for j in range(1,jmax+1):
            k = carre2vec(i,j)
            # les valeurs du laplacien à l'étape précédente servent de calcul à␣
↪la pression courante
            # c'est un schéma implicite pour la pression
            qtt = 1/dt/dx*(F[e-1][i][j]-F[e-1][i-1][j]) + 1/dt/
↪dy*(G[e-1][i][j]-G[e-1][i][j-1])
            RHSpression[k] += qtt
            DeltaP[e][i][j] += qtt # on calcule le laplacien de la pression à␣
↪l'étape

    #calcul de la pression à l'étape avec le laplacien
    #calcul de la pression à l'étape avec le laplacien

    VALpression = np.matmul(Si,RHSpression) # valpression vecteur comme␣
↪RHSpression # A DECOMMENTER


    # màj de la pression à l'étape au centre A DECOMMENTER
    for j in range(1,jmax+1):
        P[e,1:imax+1,j] = VALpression[(j-1)*imax:j*imax] # enfin on remplit p

    # màj de la pression à l'étape sur les bords
    P[e,0,1:jmax+1] = P[e,1,1:jmax+1]              # bord gauche
    P[e,imax+1,1:jmax+1] = P[e,imax,1:jmax+1]      # bord droit
```

```
    P[e,1:imax+1,0] = P[e,1:imax+1,1]              # bord en bas
    P[e,1:imax+1,jmax+1] = P[e,1:imax+1,jmax]      # bord en haut

    P[e][0][0] = 1/2*(P[e][0][1] + P[e][1][0])
    P[e][imax+1][0] = 1/2*(P[e][imax][0] + P[e][imax+1][1])
    P[e][0][jmax+1] = 1/2*(P[e][0][jmax] + P[e][1][jmax+1])
    P[e][imax+1][jmax+1] = 1/2*(P[e][imax][jmax+1] + P[e][imax+1][jmax])

    # màj des dérivées premières spatiales de la pression
    for i in range(1,imax+1):
        for j in range(1,jmax+1):
            dpdx[e][i][j] = dp_dx(e,i,j)
            dpdy[e][i][j] = dp_dy(e,i,j)

def computeUpdateUV(e):
    ### (3.31)
    U[e,1:imax+1,1:jmax+1] = F[e-1,1:imax+1,1:jmax+1] - dt*dpdx[e,1:imax+1,1:
 ↪jmax+1]
    V[e,1:imax+1,1:jmax+1] = G[e-1,1:imax+1,1:jmax+1] - dt*dpdy[e,1:imax+1,1:
 ↪jmax+1]

    # FREE-SLIP condition
    # assigne les valeurs des frontières à celles adjacentes pour les vitesses␣
 ↪tangentes
    #         et  valeurs nulles                            pour les vitesses␣
 ↪orthogonales
    U[e,0,   1:jmax+1] = 0
    U[e,imax,1:jmax+1] = 0
    V[e,1:imax+1,0] = 0
    V[e,1:imax+1,jmax+1] = 0

    V[e,0,1:jmax+1] = V[e,1,1:jmax+1]
    V[e,imax+1,1:jmax+1] = V[e,imax,1:jmax+1]

    U[e,1:imax+1,0] = U[e,1:imax+1,1]
    U[e,1:imax+1,jmax+1] = U[e,1:imax+1,jmax]
```

[28]:
```
t = 0
gamma = 0

for e in range(1,n):
    gamma = max(dt/dx*np.max(np.abs(U[e-1])), dt/dy*np.max(np.abs(V[e-1])))

    dt = .3*np.min([.05,dx/np.max(np.abs(U[e-1])),dy/np.max(np.abs(V[e-1]))]) #␣
 ↪1/2/nu/(1/dx**2+1/dy**2)
```

```
    computeFG(e-1) # calculer F et G à l'instant précédent

    computeUpdatePressure(e) # regarde FG de e-1
                             # calcule la pression et ses dérivées à l'instant e
    computeUpdateUV(e)

    t += dt
    T[e] = t
    if e%10==0:
        print("e=",e, "| t=",t, "| dt=",dt, "|  =",gamma,"\n")
print("Fin, durée: tmax=",t)
```

```
<ipython-input-28-6f11f335f367>:7: RuntimeWarning: divide by zero encountered in
double_scalars
  dt = .3*np.min([.05,dx/np.max(np.abs(U[e-1])),dy/np.max(np.abs(V[e-1]))]) #
1/2/nu/(1/dx**2+1/dy**2)
```

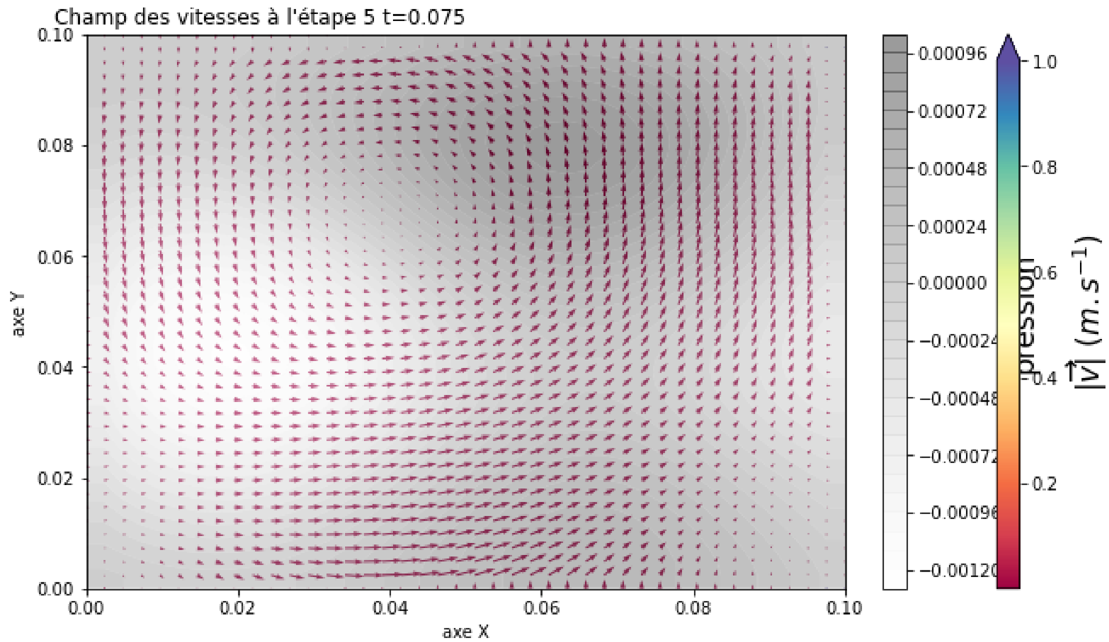
```
e= 10 | t= 0.15000000000000002 | dt= 0.015 |  = 0.03633433972766546

e= 20 | t= 0.30000000000000016 | dt= 0.015 |  = 0.047640016209197375

e= 30 | t= 0.4500000000000003 | dt= 0.015 |  = 0.052151727006822274

e= 40 | t= 0.6000000000000004 | dt= 0.015 |  = 0.052779630270269884

e= 50 | t= 0.7500000000000006 | dt= 0.015 |  = 0.036448859534990746

e= 60 | t= 0.9000000000000007 | dt= 0.015 |  = 0.032244901918502385

e= 70 | t= 1.0500000000000005 | dt= 0.015 |  = 0.02921631732090912

e= 80 | t= 1.1999999999999995 | dt= 0.015 |  = 0.029000285769116008

e= 90 | t= 1.3499999999999985 | dt= 0.015 |  = 0.02859644202037867

e= 100 | t= 1.4999999999999976 | dt= 0.015 |  = 0.02933805728609355

e= 110 | t= 1.6499999999999966 | dt= 0.015 |  = 0.029498452089282964

e= 120 | t= 1.7999999999999956 | dt= 0.015 |  = 0.030069095071489048

e= 130 | t= 1.9499999999999946 | dt= 0.015 |  = 0.027536140019581976

e= 140 | t= 2.099999999999995 | dt= 0.015 |  = 0.020354168340153255

e= 150 | t= 2.2499999999999964 | dt= 0.015 |  = 0.01846594171339911

e= 160 | t= 2.3999999999999977 | dt= 0.015 |  = 0.019507434107369326
```

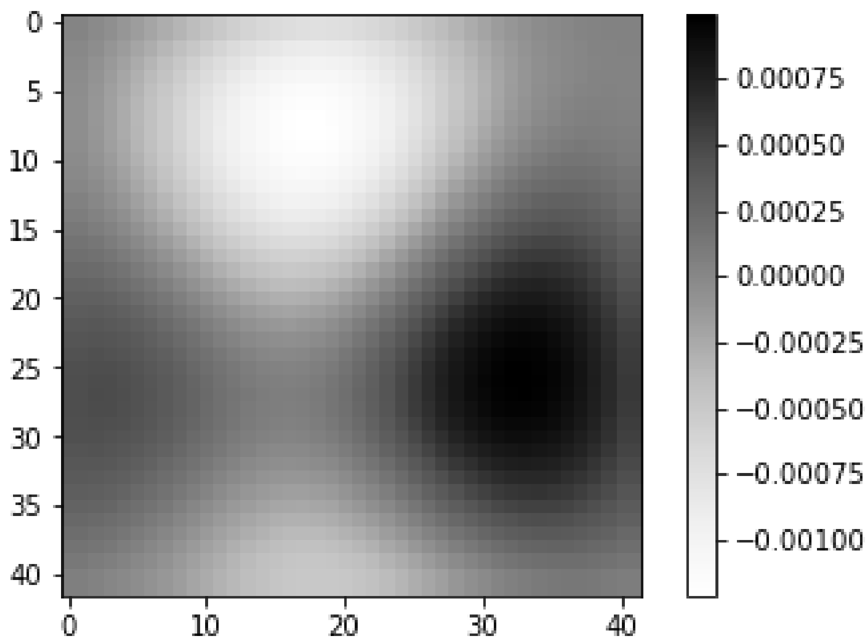
```
e= 170 | t= 2.549999999999999 | dt= 0.015 |  = 0.022287858267005946

e= 180 | t= 2.7 | dt= 0.015 |  = 0.025008103757454107

e= 190 | t= 2.8500000000000014 | dt= 0.015 |  = 0.025331060500585673

e= 200 | t= 3.0000000000000027 | dt= 0.015 |  = 0.021661476509443077

e= 210 | t= 3.150000000000004 | dt= 0.015 |  = 0.014651979376142824

e= 220 | t= 3.300000000000005 | dt= 0.015 |  = 0.015336921074385878

e= 230 | t= 3.4500000000000064 | dt= 0.015 |  = 0.0178806713553201

e= 240 | t= 3.6000000000000076 | dt= 0.015 |  = 0.021119014002799464

e= 250 | t= 3.750000000000009 | dt= 0.015 |  = 0.022658462547438342

e= 260 | t= 3.90000000000001 | dt= 0.015 |  = 0.021716749869243435

e= 270 | t= 4.05000000000001 | dt= 0.015 |  = 0.018300300246375086

e= 280 | t= 4.200000000000006 | dt= 0.015 |  = 0.015601505124135356

e= 290 | t= 4.350000000000003 | dt= 0.015 |  = 0.016625024705422008

e= 300 | t= 4.5 | dt= 0.015 |  = 0.018804978776382233

e= 310 | t= 4.649999999999997 | dt= 0.015 |  = 0.02057832360251984

e= 320 | t= 4.799999999999994 | dt= 0.015 |  = 0.020888640595089707

e= 330 | t= 4.94999999999999 | dt= 0.015 |  = 0.01953164819540954

e= 340 | t= 5.099999999999987 | dt= 0.015 |  = 0.017448725316753178

e= 350 | t= 5.249999999999984 | dt= 0.015 |  = 0.016949130518842854

e= 360 | t= 5.399999999999981 | dt= 0.015 |  = 0.017959902754296403

e= 370 | t= 5.549999999999978 | dt= 0.015 |  = 0.019366252152080793

e= 380 | t= 5.699999999999974 | dt= 0.015 |  = 0.020149964953835146

e= 390 | t= 5.849999999999971 | dt= 0.015 |  = 0.019886229805694024

e= 400 | t= 5.999999999999968 | dt= 0.015 |  = 0.018824354869814583
```

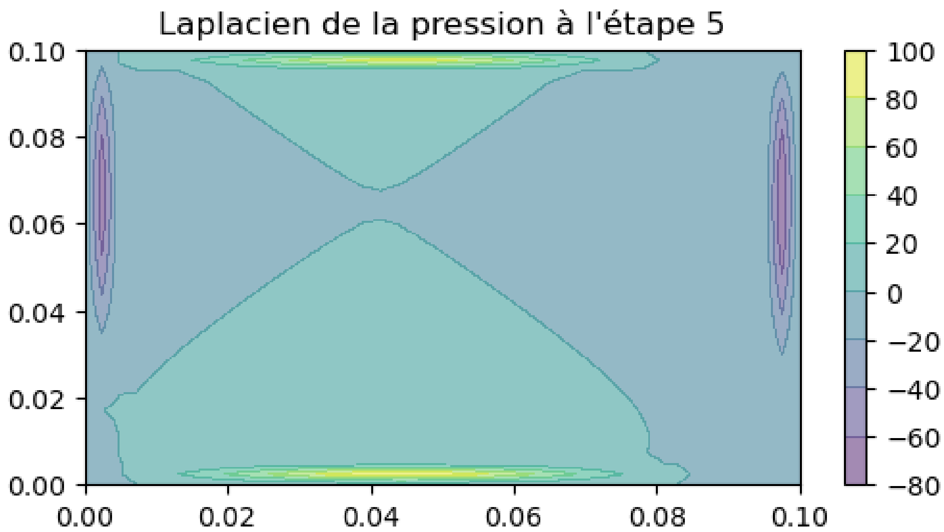
```
e= 410 | t= 6.149999999999965 | dt= 0.015 |  = 0.018016180432824856

e= 420 | t= 6.299999999999962 | dt= 0.015 |  = 0.01814999203248656

e= 430 | t= 6.449999999999958 | dt= 0.015 |  = 0.01897422137470629

e= 440 | t= 6.599999999999955 | dt= 0.015 |  = 0.019803601385664962

e= 450 | t= 6.749999999999952 | dt= 0.015 |  = 0.02009191364947834

e= 460 | t= 6.899999999999949 | dt= 0.015 |  = 0.019736743051077775

e= 470 | t= 7.049999999999946 | dt= 0.015 |  = 0.019172247849443852

e= 480 | t= 7.1999999999999424 | dt= 0.015 |  = 0.018947688023127206

e= 490 | t= 7.349999999999939 | dt= 0.015 |  = 0.01930519645680133

Fin, durée: tmax= 7.484999999999936
```

[31]:
```python
et = 5


x = np.linspace(0, a, imax+2)
y = np.linspace(0, b, jmax+2)
X, Y = np.meshgrid(x, y)
plotter(et,False)
##########################
fig = plt.figure()
c = plt.imshow(P[et],cmap='Greys')
plt.colorbar(c)
##########################
fig = plt.figure(figsize=(6,3), dpi=100)
# plotting the pressure field as a contour
plt.contourf(X,Y, DeltaP[et,:,:], 10, alpha=0.5, cmap=cm.viridis) # inverser
plt.title('Laplacien de la pression à l\'étape '+str(et))
plt.colorbar()
print(np.max(np.abs(U[et])), np.max(np.abs(V[et])) )
```

```
0.004229389491752586 0.003462202219595781
```


Champ des vitesses à l'étape 5 t=0.075
axe X
axe Y
pression
$|\vec{V}|\ (m.s^{-1})$
0.00096
0.00072
0.00048
0.00024
0.00000
−0.00024
−0.00048
−0.00072
−0.00096
−0.00120
1.0
0.8
0.6
0.4
0.2


0.00075
0.00050
0.00025
0.00000
−0.00025
−0.00050
−0.00075
−0.00100



[44]:
```
div_total = np.zeros(shape=(n+2,imax+2,jmax+2),dtype=np.float)
for et in range(n):
    for i in range(1,imax+1):
        for j in range(1,jmax+1):
            div_total[et][i][j] = div(et,i,j)
np.max(np.abs(div_total))
```

[44]: 154.53247868847788

1.0.5 Ici pour exporter les images

[30]:
```
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
from matplotlib import cm



def plotter(et,cl):
    x = np.linspace(0, a, imax+2)
    y = np.linspace(0, b, jmax+2)
    X, Y = np.meshgrid(x, y)
    M = np.hypot(U[et,:,:], V[et,:,:])
    #M = np.log(M)
    M[M == 0] = 1

    fig, ax = plt.subplots(1, 1, figsize=[8,6]) # 11, 7
    plt.xlabel('axe X')
    plt.ylabel('axe Y')
    ax.set_title('Champ des vitesses à l\'étape '+str(et)+'␣
↪t='+str(round(T[et],3)), horizontalalignment='right')
```

```
    axinsR = inset_axes(ax, width="3%", height="100%", loc='lower left',␣
↪bbox_to_anchor=(1.05, 0, 1, 1), bbox_transform=ax.transAxes, borderpad=0)
    axinsQ = inset_axes(ax, width="3%", height="100%", loc='lower left',␣
↪bbox_to_anchor=(1.2, 0, 1, 1), bbox_transform=ax.transAxes, borderpad=0)


    # plotting velocity field
    Velocity = ax.quiver(X, Y, U[et,:,:], V[et,:,:], M, pivot='tail', cmap=cm.
↪Spectral)
    barVelocity = fig.colorbar(Velocity, extend='max', orientation='vertical',␣
↪aspect=10, cax=axinsQ)
    barVelocity.set_label("$\|\overrightarrow{v}\|$ ($m.s^{-1}$)", fontsize=17)

    # plotting pressure field

    Pressure = ax.contourf(X,Y, np.transpose(P[et,:,:]), 30, alpha=0.4,␣
↪cmap='Greys', antialiased=True) # inverser
    barPressure = fig.colorbar(Pressure, extend='max', cax=axinsR, aspect=20)
    barPressure.set_label("pression", fontsize=17)

    # print(np.max(np.abs(U[et])), np.max(np.abs(V[et])) )

    if cl:
        plt.savefig('e'+str(et)+'.jpg', dpi=200, bbox_inches='tight',␣
↪facecolor='w', edgecolor='w',orientation='portrait')
        plt.close(fig)
```

[107]:
```
for et in range(1,100):
    ploter(et,True)
    if et%5==0:
        print('ok étape '+str(et))
```

```
ok étape 5
ok étape 10
ok étape 15
ok étape 20
ok étape 25
ok étape 30
ok étape 35
ok étape 40
ok étape 45
ok étape 50
ok étape 55
ok étape 60
ok étape 65
```

```
ok étape 70
ok étape 75
ok étape 80
ok étape 85
ok étape 90
ok étape 95
ok étape 100
ok étape 105
ok étape 110
ok étape 115
ok étape 120
ok étape 125
ok étape 130
ok étape 135
ok étape 140
ok étape 145
ok étape 150
ok étape 155
ok étape 160
ok étape 165
ok étape 170
ok étape 175
ok étape 180
ok étape 185
ok étape 190
ok étape 195
```

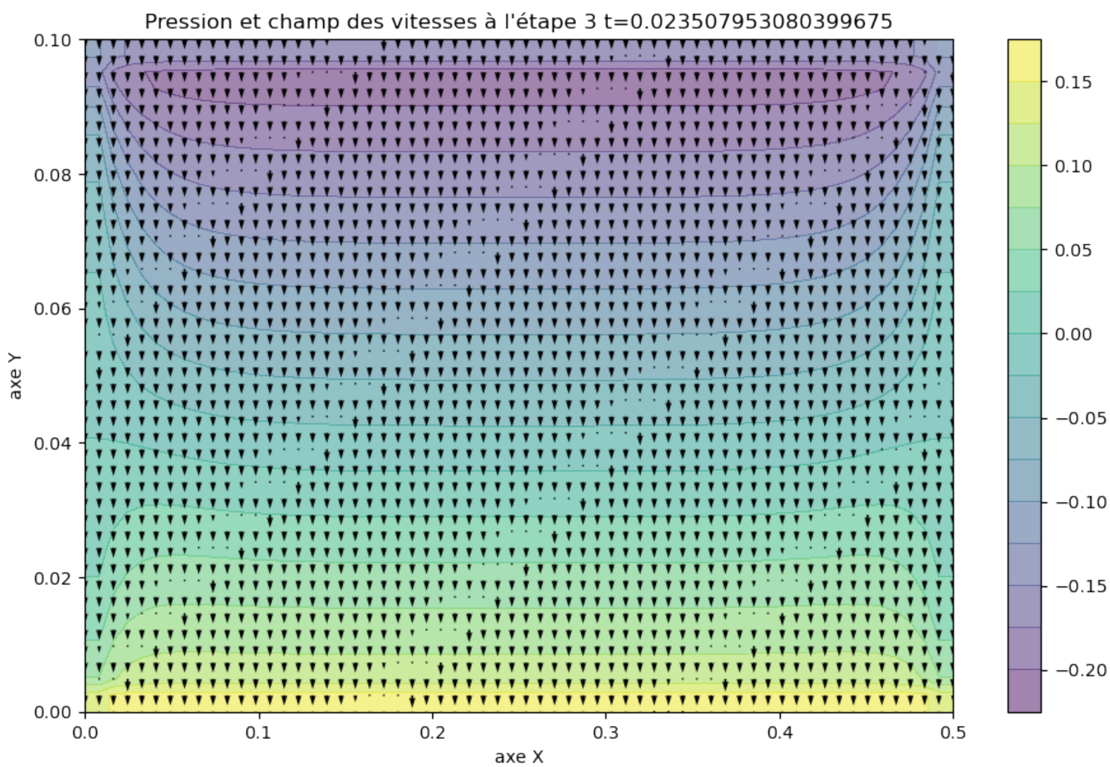
Pour créer une vidéo utiliser la commande `ffmpeg -start_number 1 -i %d.jpg -vcodec mpeg4 test.mp4 -b:v 5000k`

OU `ffmpeg -start_number 1 -i %d.jpg -framerate 25 -c:v libx264 -crf 0 -vf "pad=ceil(iw/2)*2:ceil(ih/2)*2" output.mp4` en changeant le paramètre `crf` pour la qualité (plus c'est proche de 0, moins ça compresse)

[163]:
```
et = 3

M = np.hypot(U[et,:,:], V[et,:,:])
M[M == 0] = 1

fig = plt.figure(figsize=(11,7), dpi=100)
plt.xlabel('axe X')
plt.ylabel('axe Y')
plt.title('Pression et champ des vitesses à l\'étape '+str(et)+' t='+str(T[et]))

# plotting the pressure field as a contour
plt.contourf(X,Y, np.rot90(P[et,:,:],k=3), 15, alpha=0.5, cmap=cm.viridis)  #␣
↪inverser
plt.colorbar()
```

```
# plotting the pressure field outlines
# plt.contour(X, Y, P[et,:,:], cmap=cm.viridis, alpha = 0)




# plotting velocity field
plt.quiver(X, Y, U[et,:,:], V[et,:,:])

# lw = 30*M / M.max()
# W = plt.streamplot(X, Y, U[et,:,:], V[et,:,:], density=3, color='black',␣
↪linewidth=lw)

np.max(np.abs(V[et]))
```

[163]: 0.2865896710255286



1.0.6 Théorie de Kolmogorov et dissipation

La base de la théorie de Kolmogorov énonce que une vision picturale de la turbulence qui place la dissipation en chaleur à la fin d'une cascadede transferts d'énergie. Ainsi ce qui sera dissipé par la turbulence est entièrement déterminé par les premières étapes qui sont totalement indépendantes de la viscosité. Avec cette idée de cascade, le tauxmoyen de dissipation $\langle \epsilon \rangle$ (puissance dissipée par unité de masse) se détermine par le transfertde l'énergie cinétique U^2 des plus gros tourbillons. Ce transfert doit se faire sur la durée de vie du tourbillon, soit sur un temps $\tau = \frac{L}{U}$, on doit donc avoir $\langle \epsilon \rangle \propto \frac{U^3}{L}$. Bien que la viscosité soit responsable de la dissipation, la puissance dissipée dans un écoulement turbulent ne dépend pas de la viscosité !

L'hypothèse de Kolmogorov est la suivante: Pour un nombre de Reynolds $Re \gg 1$, la statistique des mouvements turbulents estuniquement déterminée à partir de ν et $\langle \epsilon \rangle$.On peut alors construire dimensionnellement à l'aide de ces deux grandeurs, une échelle dite de Kolmogorov,η ayant pour vitesse caractéristique u_η et temps de caractéristique τ_η, telle que:

$\eta = (\frac{\nu^3}{\langle \epsilon \rangle})^{1/4}$

$u_\eta = (\frac{\nu}{\langle \epsilon \rangle})^{1/4}$

$\tau_\eta = \sqrt{\frac{\nu}{\langle \epsilon \rangle}}$

À cette échelle $Re = 1$.

L'énergie dissipée à l'échelle l est $\langle \epsilon \rangle = \frac{U_l^3}{l}$ si bien que

le schéma classique de la turbulence développée est interprétée dans la cascade de Richardson. Chaque structure d'échelle transfert son énergie cinétique u^2 pendant unedurée $\tau_l = \frac{l}{u_l}$, soit : $\langle \epsilon \rangle \propto \frac{u_l^2}{\tau_l}$. Une fois transférée l'énergie n'est plus disponible à l'échelle l maisstockée de façon incohérente à des échelles plus petites. En ce sens, l'énergie perdue pour l'échelle l correspond à une dissipation pour cette échelle. Arrivée à l'échelle de Kolmogorov η, l'énergie ci-nétique de ces plus petites structures de la cascade est dissipée sous forme de chaleur par diffusion visqueuse sur un temps caractéristique $\frac{\eta^2}{\nu}$.

Une des problématiques de la turbulence réside dans le coût prohibitif de la simulation numérique. En effet l'équation de Navier-Stokes doit être capable de reproduire tout type d'écoulement quelque-soit le nombre de Reynolds. On pourrait donc pour ainsi dire tout calculer et tout prévoir. Cependant,pour que ce calcul soit fidèle à la réalité, le schéma numérique devra résoudre toutes les échelles,jusqu'à l'échelle de dissipation de Kolmogorov. D'après la partie précédente, un écoulement de taille L^3 devra comporter au moins $\left(\frac{L}{\eta}\right)^3 \sim (Re)^{9/4}$ points de maillage. De même, le rapport des temps ca-ractéristiques entre la grande échelle (de tailleL) et l'échelle de Kolmogorov est $\frac{\tau_l}{\tau_\eta} \sim (Re)^{1/2}$. Ainsi larésolution de cet écoulement pendant un temps caractéristique de la grande échelle (ce qui est large-ment insuffisant pour effectuer des valeurs moyennes) nécessite de résoudre les équations de NavierStokesRe114fois. Ainsi plus le nombre de Reynolds est grand et bien plus la simulation coûtera entemps de calcul. Par exemple, il est inconcevable aujourd'hui de simuler les écoulements autour d'unevoiture ou d'un avion (qui présentent un nombre de Reynolds supérieur à10^6).

1.0.7 Calcul de l'odg du nombre de points nécessaire dans une direction d'espace pour que la simulation

[16]:
```
odgacc = 1e-2 # m/s^2
odgt = 1 # s
U = odgacc*odgt
L = .1 # m
nu = 5e-5
r = (U*L/nu)**(3/4) # on calcule ici le rapport L/eta
r
```

[16]: 9.457416090031758

[20]:
```
eta = L/r
eta # donc idéalement pour retraduire les tourbillons à cette échelle il␣
 ↪faudrait discrétiser ça en 10 fois
```

[20]: 0.010573712634405642

1.0.8 Further resources

https://www.thermal-engineering.org/fr/quest-ce-que-lequation-de-navier-stokes-definition/

https://hal-ensta-paris.archives-ouvertes.fr/cel-01228137/file/coursdeturbulence.pdf

https://github.com/barbagroup/CFDPython et https://piazza.com/bu/spring2013/me702/resources et https://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/

https://www.montana.edu/mowkes/research/source-codes/GuideToCFD.pdf (a guide to writing your first CFD solver)

Voir aussi "Real time fluid dynamics for games" https://www.youtube.com/watch?v=alhpH6ECFvQ

et les liens: - Code: https://thecodingtrain.com/CodingChal…

Links discussed in this video: - GitHub Thread: https://github.com/CodingTrain/Rainbo… - Real-Time Fluid Dynamics for Games by Jos Stam: http://www.dgp.toronto.edu/people/sta… - MSAFluid: https://www.memo.tv/msafluid/ - Lily Pad: https://github.com/weymouth/lily-pad - Fluid Simulation for Dummies by Mike Ash: https://mikeash.com/pyblog/fluid-simu… - Why Laminar Flow is AWESOME - Smarter Every Day 208: https://youtu.be/y7Hyc3MRKno - What DO we know about turbulence? by 3Blue1Brown: https://youtu.be/_UoTTq651dE - Perlin Noise: https://youtu.be/Qf4dIN99e2w

en dessous de cettte vidéo https://www.youtube.com/watch?v=alhpH6ECFvQ

ainsi que cet article qui cite le livre que j'ai utilisé https://www5.in.tum.de/pub/bartels_idp_14.pdf et cette vidéo https://www.youtube.com/watch?v=qsYE1wMEMPA qui parle d'uns simu en compressible

[ ]:
```
set1G(0)
x = np.linspace(0, a, imax+2)
y = np.linspace(0, b, jmax+2)
```

```python
X, Y = np.meshgrid(x, y)
fig = plt.figure(figsize=(5,4), dpi=100)
plt.quiver(X, Y, Gx, Gy)
print(np.abs(Gx).max(), np.max(np.abs(Gy)))
```

[ ]:

```python

```