```
import visualization as vis
from scipy.stats import linregress
from get_global import *
from init import *
import operators as op
import operator_verf as opf
import matplotlib.pyplot as plt
from matplotlib import cm
from cgs import *
from matplotlib.animation import FuncAnimation
from scipy.sparse.linalg import cg

# Choose function with known analytic solution for the diffusion equation
functions = {
        "u_xyt"   : lambda x, y, t, nu, a: (2*np.pi*nu)*((np.sin(np.pi*x)*np.exp(-np.pi**2*
nu*t))/\
                (a + np.cos(np.pi*x)*np.exp(-np.pi**2*nu*t ))) + y*0,
        "v_xyt"   : lambda x, y, t: 0*x + 0*y + 0*t

        }

save=True
u_xyt = functions["u_xyt"]
v_xyt = functions["v_xyt"]

dxdy = []
L2 = []
Linf = []
acc = 0
qBC_nm1 = {}
qBC = {}

dt = .004
T = 1
Nt = int(T/dt)
t = np.linspace(0, Nt*dt, Nt)
alpha = .5 # Crank-Nicholson
nu = 0.05
a = 2

grid = zip(dx, dy, nx, ny, q_size)
for dxi, dyi, nxi, nyi, q_sizei in grid:
    time = []
    Xu_data = []
    Tu_data = []
    U_data  = []

    # ---------- Initialize Simulation Domain ------------------

    [ui, vi, pi] = init(nxi, nyi, pinned=False)

    # U Positions
    xu = dxi*(1. + np.arange(0, nxi-1))
    yu = dyi*(0.5 + np.arange(0, nyi))
    Xu, Yu = np.meshgrid(xu, yu)

    # V Positions
    xv = dxi*(0.5 + np.arange(0, nxi))
    yv = dyi*(1.0 + np.arange(0, nyi-1))
    Xv, Yv = np.meshgrid(xv, yv)

    # IC U, V @(x,y,t=0)
    t0 = 0
    U = np.reshape(u_xyt(Xu, Yu, t0, nu, a), (1, nyi*(nxi-1)))
    V = np.reshape(v_xyt(Xv, Yv, t0), (1, nxi*(nyi-1)))

    q_nm1 = np.concatenate( (U, V), axis = 1)
    q_nm1 = q_nm1[0]
```

```
    # ---------- Set Boundary Conditions -----------------------

    # Top Wall BC
    qBC_nm1["uT"] = u_xyt(xu,Ly, dt*t0, nu, a)
    qBC_nm1["vT"] = v_xyt(xv,Ly, dt*t0)
    # Bottom Wall BC
    qBC_nm1["uB"] = u_xyt(xu,0, dt*t0, nu, a)
    qBC_nm1["vB"] = v_xyt(xv,0, dt*t0)
    # Left Wall BC
    qBC_nm1["uL"] = u_xyt(0,yu, dt*t0, nu, a)
    qBC_nm1["vL"] = v_xyt(0,yv, dt*t0)
    # Right Wall BC
    qBC_nm1["uR"] = u_xyt(Lx,yu, dt*t0, nu, a)
    qBC_nm1["vR"] = v_xyt(Lx,yv, dt*t0)


    # ---------- SOLVE FOR u(x,y,tn) WHERE n = 1 ------------
    # ---------- Set Boundary Conditions for n+1 ------------

    U = np.reshape(u_xyt(Xu, Yu, dt*(t0+1), nu, a), (1, nyi*(nxi-1)))
    V = np.reshape(v_xyt(Xv, Yv, dt*(t0+1)), (1, nxi*(nyi-1)))

    q_n = np.concatenate( (U, V), axis = 1)
    q_n = q_n[0]

    # Top Wall BC
    qBC["uT"] = u_xyt(xu,Ly, dt*(t0+1), nu, a)
    qBC["vT"] = v_xyt(xv,Ly, dt*(t0+1))
    # Bottom Wall BC
    qBC["uB"] = u_xyt(xu,0, dt*(t0+1), nu, a)
    qBC["vB"] = v_xyt(xv,0, dt*(t0+1))
    # Left Wall BC
    qBC["uL"] = u_xyt(0,yu, dt*(t0+1), nu, a)
    qBC["vL"] = v_xyt(0,yv, dt*(t0+1))
    # Right Wall BC
    qBC["uR"] = u_xyt(Lx,yu, dt*(t0+1), nu, a)
    qBC["vR"] = v_xyt(Lx,yv, dt*(t0+1))

    bcL_n = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)

    # ---------- Plot Initial U ------------------
    plotInit = False
    if plotInit:
        fig = plt.figure()
        ax = plt.axes(projection='3d')

        q_u = np.reshape(q_n[0:nyi*(nxi-1)], (Xu.shape))

        surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                cmap=cm.viridis, linewidth=0, antialiased=True)
        ax.set_zlim(0, 1.5)
        ax.set_xlabel('$xu$')
        ax.set_ylabel('$yu$')
        ax.view_init(30, 45)
        plt.show()

    # ---------- Plot Laplacian (it works) ------------------
    plotLap = False
    if plotLap:
        fig = plt.figure()
        ax = plt.axes(projection='3d')

        Lq_n = op.laplace(q_n, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)
        Lq_n = Lq_n[0:nyi*(nxi-1)]
        LqBC = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)
        LqBC = LqBC[0:nyi*(nxi-1)]
        Lq = Lq_n + LqBC
```

```
        Lq_n_ex = np.reshape(Lu_xyt(Xu, Yu, t0), (1, nyi*(nxi-1)))
        Lq_n_ex = Lq_n_ex[0]

        error = LA.norm(Lq - Lq_n_ex, np.inf)
        for j in range(0,nyi):
            print('***** Row %d *****' % (j) )
            Lq_n_row = Lq[(nxi-1)*j:(nxi-1)*(j+1)]
            Lq_n_ex_row = Lq_n_ex[(nxi-1)*j:(nxi-1)*(j+1)]
            error = LA.norm(Lq_n_row - Lq_n_ex_row, np.inf)
            print('Error norm for j = %d is %.3e' % (j, error))

        Lq = np.reshape(Lq[0:nyi*(nxi-1)], (Xu.shape))
        Lq_n_ex = np.reshape(Lq_n_ex[0:nyi*(nxi-1)], (Xu.shape))

        surf = ax.plot_surface(Xu, Yu, Lq, rstride=1, cstride=1,\
                cmap=cm.viridis, linewidth=0, antialiased=True)
        ax.set_title('Error Norm of Laplacian: ' + str(error))
        ax.set_xlabel('$xu$')
        ax.set_ylabel('$yu$')
        ax.view_init(30, 45)
        plt.show()

    dt = dxi*.1
    T = 1
    Nt = int(T/dt)

    # ---------- Begin Time-Stepping ---
    for tn in range(1, Nt):

        # ---------- Set Boundary Conditions for n+1 ------------

        # Top Wall BC
        qBC["uT"] = u_xyt(xu,Ly, dt*(tn+1), nu, a)
        qBC["vT"] = v_xyt(xv,Ly, dt*(1+tn))
        # Bottom Wall BC
        qBC["uB"] = u_xyt(xu,0, dt*(tn+1), nu, a)
        qBC["vB"] = v_xyt(xv,0, dt*(1+tn))
        # Left Wall BC
        qBC["uL"] = u_xyt(0,yu, dt*(tn+1), nu, a)
        qBC["vL"] = v_xyt(0,yv, dt*(1+tn))
        # Right Wall BC
        qBC["uR"] = u_xyt(Lx,yu, dt*(tn+1), nu, a)
        qBC["vR"] = v_xyt(Lx,yv, dt*(1+tn))

        bcL_np1 = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)

        # ---------- Set RHS of Ax = b for Diffusion Eq.  --------
        bc = np.multiply(0.5*dt*nu, np.add(bcL_n, bcL_np1))
        Sq_n = op.S(q_n, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt, pinned=Fal
se)

        Aq_nm1 = op.adv(q_nm1, qBC_nm1, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=Fal
se)
        Aq_n = op.adv(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)
        adv = np.multiply(0.5*dt, np.subtract(np.multiply(3, Aq_n), Aq_nm1))

        b = Sq_n + bc + adv

        # Solve without CGS
        cgs = True
        if not cgs:
            R = op.R(np.ones(q_n.shape), ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu
, dt, pinned=False)
            A = np.diag(R[-1])
            q_np1 = LA.solve(A, b)
        else:
            [q_np1, Rq_np1] = Atimes(np.zeros(q_n.shape), b, 3, ui, vi, pi, dxi, dyi, nxi,
nyi, q_sizei, q_sizei, alpha, nu, dt, pinned=False)
```

```
        qu_np1 = q_np1[0:nyi*(nxi-1)]
        q_np1_ex = np.concatenate(\
                    (np.reshape(u_xyt(Xu, Yu, (1+tn)*dt, nu, a), (1, nyi*(nxi-1))),\
                    np.reshape(v_xyt(Xv, Yv, (1+tn)*dt), (1, nxi*(nyi-1)))),\
                    axis = 1)
        qu_np1_ex = q_np1_ex[0][0:nyi*(nxi-1)]

        error = LA.norm(qu_np1 - qu_np1_ex, np.inf)
        if (tn % 2) == 0:
            print('Time = %f' % ((tn+1)*dt))
            print('Error b/w qu_np1 and qu_np1_ex: ' + str(error))

        # ---------- Plot U^n+1 ------------------

        plotInit = False
        if plotInit:
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            q_u = np.reshape(q_n[0:nyi*(nxi-1)], (Xu.shape))
            surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            ax.set_zlim(0, 1.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()

        # ---------- Save X-Data at y = 0.5 ------------------
        plotXTime = True
        if plotXTime:
            q_u = np.reshape(q_n[0:nyi*(nxi-1)], (Xu.shape))
            U_data.append(q_u[5])
            time.append(tn*dt)
            #plt.plot(xu, q_u[5])

        q_nm1 = q_n
        qBC_nm1 = qBC
        q_n = q_np1
        bcL_n = bcL_np1

        # ---------- Plot Uex^n+1 ------------------
        plotExact = False
        if plotExact:
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            #q_u_exact = np.reshape(q_n_exact[0:(nyi*(nxi-1))], (Xu.shape))
            b_ex = op.R(q_n_exact, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt,
pinned=False)
            q_u_exact = np.reshape(b_ex[0:nyi*(nxi-1)], Xu.shape)
            #plt.contourf(Xu, Yu, q_u_exact)
            #ax.contour3D(Xu, Yu, q_u_exact, 50)
            surf = ax.plot_surface(Xu, Yu, q_u_exact, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            ax.set_zlim(0, 1.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()

        plotCurrent = False
        if plotCurrent:
            # Current Simulation
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            q_u = np.reshape(q_n[0:(nyi*(nxi-1))], (Xu.shape))
            #plt.contourf(Xu, Yu, q_u_exact)
```

```
            #ax.contour3D(Xu, Yu, q_u_exact, 50)
            surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            ax.set_zlim(0, 0.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()


    Linf.append(LA.norm(qu_np1 - qu_np1_ex, np.inf))
    dxdy.append(dxi)
err = Linf
lin = linregress(np.log10(dxdy), np.log10(err))
acc = lin.slope
vis.plotL2vsGridSize(lin, dxdy, err, 'Burgers_Eq', 'Burgers Eq.', save=save)

    ##print(U_data)
    #Xu_data, Tu_data = np.meshgrid(xu, time)
    #U_data = np.array(U_data)
    #fig = plt.figure()
    #ax = plt.axes(projection='3d')
    ##plt.contourf(Xu, Yu, q_u_exact)
    ##ax.contour3D(Xu, Yu, q_u_exact, 50)
    #surf = ax.plot_surface(Xu_data, Tu_data, U_data, rstride=1, cstride=1,\
    #        cmap=cm.viridis, linewidth=0, antialiased=True)
    #ax.set_zlim(0, 0.2)
    #ax.set_xlabel('$x$')
    #ax.set_ylabel('$time$')
    ##ax.view_init(30, 45)
    #plt.show()
```

```
"""
Created on May 2 2021
@author: Shehan M. Parmar
Conjugate Gradient solver for pressure poisson and
momentum equations.
"""
from numba import jit
import numpy as np
from sklearn.datasets import make_spd_matrix
import numpy.linalg as LA
from numpy.random import rand
from numpy.random import seed
import matplotlib.pyplot as plt
import sys
import operators as op

#np.set_printoptions(threshold=sys.maxsize)
def Atimes(x, b, eqn, u, v, p, dx, dy, nx, ny, q_size, g_size, alpha, nu, dt, pinned=False,
 **kwargs):

    i = 1
    imax = 5000
    eps = 1e-6

    if eqn == 0:
        if "A" not in [*kwargs]:
            raise("Must specify matrix variable 'A'.")
        A = kwargs["A"]
        Ax = np.dot(A, x)
    elif eqn == 1:
        Ax = op.R(x, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=True) # Momentu
m Eq.
    elif eqn == 2:
        GP_np1 = op.grad(x, u, v, p, dx, dy, nx, ny, q_size)
        RinvGP_np1 = op.Rinv(GP_np1, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned
=True)
        DRinvGP_np1 = op.div(RinvGP_np1, u, v, p, dx, dy, nx, ny, g_size, pinned=True)
        Ax = np.multiply(-1., DRinvGP_np1)
        # Pressure Poisson Eq
    elif eqn == 3: # Diffusion Eq.
        Ax = op.R(x, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=False)
    r = np.subtract(b, Ax)
    d = r
    del_new = np.dot(r.T, r)
    del0 = del_new

    del_new_vals = []
    del_new_vals.append(del_new)
    while (i < imax) and (del_new > eps**2*del0):

        if (i % 500) == 0:
            print('Iteration No: %d' % (i))
            print('del_new = %.3e' % (del_new))

        if eqn == 0:
            q = np.dot(A, d)
        elif eqn == 1:
            Ad = op.R(d, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=False)
            q = Ad
        elif eqn == 2:
            GP_np1 = op.grad(d, u, v, p, dx, dy, nx, ny, q_size)
            RinvGP_np1 = op.Rinv(GP_np1, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pi
nned=True)
            DRinvGP_np1 = op.div(RinvGP_np1, u, v, p, dx, dy, nx, ny, g_size, pinned=True)
            Ad = np.multiply(-1., DRinvGP_np1)
            #checkAx(Ad)
            q = Ad
        elif eqn == 3: # Diffusion Eq.
```

```
            Ad = op.R(d, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=False)
            q = Ad

        alpha_cg = np.divide( del_new , np.dot(d.T, q) )
        x = np.add(x , np.multiply(alpha_cg,d))

        if (i % 50) == 0:
            if eqn == 0:
                r = np.subtract(b, np.dot(A, x))
            elif eqn == 1:
                Ax = op.R(x, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=False)
                r = np.subtract(b, Ax)
            elif eqn == 2:
                GP_np1 = op.grad(x, u, v, p, dx, dy, nx, ny, q_size)
                RinvGP_np1 = op.Rinv(GP_np1, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt
, pinned=True)
                DRinvGP_np1 = op.div(RinvGP_np1, u, v, p, dx, dy, nx, ny, g_size, pinned=Tr
ue)
                Ax = np.multiply(-1., DRinvGP_np1)
                #checkAx(Ax)
                r = np.subtract(b, Ax)
            elif eqn == 3: # Diffusion Eq.
                Ax = op.R(x, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=False)
                r = np.subtract(b, Ax)
        else:
            r = np.subtract(r , np.multiply(alpha_cg,q))
        del_old = del_new
        del_new = np.dot(r.T, r)
        del_new_vals.append(del_new)
        beta = del_new / del_old

        d = np.add(r , beta*d)
        i += 1

    if eqn == 0:
        Ax = np.dot(A, x)
    elif eqn == 1: # Momentum Eq.:
        Ax = op.R(x, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=False)
    elif eqn == 2:
        GP_np1 = op.grad(x, u, v, p, dx, dy, nx, ny, q_size)
        RinvGP_np1 = op.Rinv(GP_np1, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned
=True)
        DRinvGP_np1 = op.div(RinvGP_np1, u, v, p, dx, dy, nx, ny, g_size, pinned=True)
        #checkAx(Ax)
        Ax = np.multiply(-1., DRinvGP_np1)
    elif eqn == 3: # Diffusion Eq.
        Ax = op.R(x, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=False)

    if 'convIter' in kwargs:
        return [i, Ax]
    else:
        #plt.scatter(list(range(0,len(del_new_vals))), del_new_vals, marker='o')
        #plt.show()
        #print('CGS cnverged in %d iterations.' % (i))
        return [x, Ax]

def testMatrix(ndim, seed = None):

    A = make_spd_matrix(ndim, random_state=seed)

    eigVals = LA.eigvals(A)
    posDef = (eigVals > 0).all()
    symmetric = LA.norm(A.T - A, np.inf) < 1e-6

    if not posDef or not symmetric:
        raise("Matrix is not Positive Definite.")

    return A
```

```
def checkAx(Ax):
    A = np.diag(Ax)
    eigVals = LA.eigvals(A)
    posDef = (eigVals > 0).all()

    if not posDef:
        print(eigVals)
        raise("Matrix is not Positive Definite.")

# Test CGS
#ndim_val = [10, 10**2, 10**3, int(10**(3.5))]
#for ndim in ndim_val:
#    A_test = testMatrix(ndim, seed = None)
#    print('%.1e' % (np.size(A_test)))
#    #b = np.rand(ndim, 1)
#    b = np.ones((ndim, 1))
#
#    soln = np.dot(LA.inv(A_test), b)
#    x_guess = np.zeros( (ndim, 1))
#    [i, Ax] = Atimes( x_guess, b, eqn = 0, A = A_test, convIter = None)
#
#    print('Error Norm:')
#    print(LA.norm(Ax-b, np.inf))
#    print('Converged in %d iterations.' % (i))
#
```

```
"""
Created on May 2 2021
@author: Shehan M. Parmar
Conjugate Gradient solver for pressure poisson and
momentum equations.
"""
import numpy as np
from sklearn.datasets import make_spd_matrix
import numpy.linalg as LA
from numpy.random import rand
from numpy.random import seed
import matplotlib.pyplot as plt
import sys
import operators as op
#def unpackInputs(**kwargs):
#    for key in kwarg:

#np.set_printoptions(threshold=sys.maxsize)
def Atimes(x, b, eqn, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=False, **kwarg
s):

    #print('Input b (RHS)')
    #print(b)
    if eqn == 0:
        if "A" not in [*kwargs]:
            raise("Must specify matrix variable 'A'.")
        A = kwargs["A"]
        Ax = np.dot(A, x)
    elif eqn == 1:
        # Ax = ...
        pass # Momentum Eq.
    elif eqn == 2:
        # Ax = div(Rinv(grad(x))
        pass # Pressure Poisson Eq
    elif eqn == 3: # Diffusion Eq.
        # Check Laplace Operator
        A_m = np.zeros([q_size, q_size])
        for i in range(0, q_size):
            z = np.zeros([q_size])
            z[i] = 1
            A_m[:,i] = op.laplace(z, u, v, p, dx, dy, nx, ny, q_size, pinned=False)
            R_m = np.subtract(z, np.multiply(1*alpha*nu*dt, A_m))
        eigvals = np.linalg.eigvals(R_m)
        #print('laplace eigvals =', eigvals)
        #print('\nCondition No. = %.3e.\n' % (max(eigvals)/min(eigvals)))
        if not (np.linalg.eigvals(R_m) > 0).all():
            print(R_m)
            raise("R operator is not positive def.")
            #raise("Laplace operator is not positive def.")


        #Lq = op.laplace(np.ones(x.shape), u, v, p, dx, dy, nx, ny, q_size, pinned=False)
        #
        #A2 = np.diag(Lq)
        #eigVals2 = LA.eigvals(A2)
        #posDef2 = (eigVals2 > 0).all()
        #print('\nBEFORE 1ST ITER: Laplace Operator Pos Def = %s\n' % (str(posDef2)))


        #R = np.subtract(1, np.multiply(Lq, alpha*nu*dt))

        #A = np.diag(R)
        #eigVals = LA.eigvals(A)
        #print('\nCondition No. = %.3e.\n' % (max(eigvals)/min(eigvals)))
        #posDef = (eigVals > 0).all()
        #symmetric = LA.norm(A.T - A, np.inf) < 1e-6
        #if not posDef or not symmetric:
        #    print(R)
```

```
        #    raise("Matrix is not Positive Definite.")

        [Lq, a, I, Ax] = op.R(x, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=Fal
se)

        #print('x')
        #print(x)
        #print('Lq')
        #print(Lq)
        #print('a')
        #print(a)
        #print('Ax')
        #print(Ax)
    r = np.subtract(b, Ax)
    d = r
    #print('Initial d = r = b - Ax = b - op.R(x)')
    #print(d)
    del_new = np.dot(r.T, r)
    del0 = del_new
    ##print('************Initial Variables')
    ##print('b')
    ##print(b)
    ##print('Ax')
    ##print(Ax)
    ##print('r')
    ##print(r)
    # Initial Solver Conditions
    i = 1
    imax = 100
    eps = 1e-6

    #print('del_new = r.T*r = %f' % (del_new))
    #print('eps**2*del0 = %.4e' % (eps**2*del0))
    del_new_vals = []
    del_new_vals.append(del_new)
    while (i < imax) and (del_new > eps**2*del0):
        #print('************Iteration %d ************\n' % (i) )

        if eqn == 0:
            q = np.dot(A, d)
        elif eqn == 1:
            # Ad = ...
            q = Ad
        elif eqn == 2:
            # Ad = div(rinv(grad(d)))
            q = Ad
        elif eqn == 3: # Diffusion Eq.
            #Lq = op.laplace(d, u, v, p, dx, dy, nx, ny, q_size, pinned=False)
            #R = np.subtract(d, np.multiply(Lq, alpha*nu*dt))
            #A = np.diag(R)
            #eigVals = LA.eigvals(A)
            #print('\nCondition No. = %.3e.\n' % (max(eigvals)/min(eigvals)))
            #posDef = (eigVals > 0).all()
            #symmetric = LA.norm(A.T - A, np.inf) < 1e-6
            #if not posDef or not symmetric:
            #    print(d)
            #    print('posDef = %s' %(str(posDef)))
            #    print('symmetric = %s' %(str(symmetric)))

            #    A2 = np.diag(Lq)
            #    eigVals2 = LA.eigvals(A2)
            #    posDef2 = (eigVals2 > 0).all()
            #    print('Laplace Operator Pos Def = %s' % (str(posDef2)))
                #raise("Matrix is not Positive Definite.")
            [Lq, a, I, Ad] = op.R(d, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned
=False)
            #Ad = R
            #
```

```
            #print('1) Determine Ad:\n')
            #print('R = I - a*dt*nu*L')
            #print('Lq')
            #print(Lq)
            #print('a')
            #print(a)
            #print('I -a*Lq')
            #print(1 - a*Lq)
            #print('alpha = %.3f, nu = %.3f, dt = %.3f, alpha*nu*dt = %.3e' % (alpha, nu, d
t, a))
            #print('d')
            #print(d)
            #print('q updated with aboce d')
            q = Ad
            #print('q = Ad = op.R(d)')
            #print(q)

        #print('2) Update r:\n')
        alpha_cg = np.divide( del_new , np.dot(d.T, q) )
        #print('alpha_cg = del_new / (d.T*q) = %4e' % (alpha_cg))
        x = np.add(x , np.multiply(alpha_cg,d))
        #print('x = x + alpha_cg*d = ')
        #print(x)

        if (i % 50) == 0:
            if eqn == 0:
                r = np.subtract(b, np.dot(A, x))
            elif eqn == 1:
                # Ax = ...
                r = np.subtract(b, Ax)
            elif eqn == 2:
                # Ax = div(rinv(grad(x)))
                r = np.subtract(b, Ax)
            elif eqn == 3: # Diffusion Eq.
                #Lq = op.laplace(x, u, v, p, dx, dy, nx, ny, q_size, pinned=False)
                #R = np.subtract(x, np.multiply(Lq, alpha*nu*dt))
                #A = np.diag(R)
                #eigVals = LA.eigvals(A)
                #print('\nCondition No. = %.3e.\n' % (max(eigvals)/min(eigvals)))
                #posDef = (eigVals > 0).all()
                #symmetric = LA.norm(A.T - A, np.inf) < 1e-6
                #if not posDef or not symmetric:
                #    print(R)
                    #raise("Matrix is not Positive Definite.")
                [Lq, a, I, Ax] = op.R(x, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pi
nned=False)
                Ax = R
                r = np.subtract(b, Ax)
        else:
            r = np.subtract(r , np.multiply(alpha_cg,q))
        #print('r = r - alpha_cg*q')
        #print(r)
        #print('r.T*r')
        #print(r.T*r)
        #print('np.dot(r.T,r)')
        #print(np.dot(r.T,r))
        #print('3) Values for next iteration and conv check:')
        del_old = del_new
        del_new = np.dot(r.T, r)
        del_new_vals.append(del_new)
        beta = del_new / del_old

        #print('del_new = np.dot(r.T,r) @ (i = %d) = %.3e' % (i, del_new))
        #print('beta = del_new / del_old @ (i = %d) = %.3e' % (i, beta))
        d = np.add(r , beta*d)
        #print('d = r + beta*d')
        #print(d)
        i += 1
```

```
    if eqn == 0:
        Ax = np.dot(A, x)
    elif eqn == 1:
        # Ax = ...
        pass # Momentum Eq.
    elif eqn == 2:
        # Ax = div(Rinv(grad(x))
        pass # Pressure Poisson Eq.
    elif eqn == 3: # Diffusion Eq.
        #Lq = op.laplace(np.ones(x.shape), u, v, p, dx, dy, nx, ny, q_size, pinned=False)
        #R = np.subtract(1, np.multiply(Lq, alpha*nu*dt))
        #A = np.diag(R)
        #eigVals = LA.eigvals(A)
        #print('\nCondition No. = %.3e.\n' % (max(eigvals)/min(eigvals)))
        #posDef = (eigVals > 0).all()
        #symmetric = LA.norm(A.T - A, np.inf) < 1e-6
        #if not posDef or not symmetric:
        #    print(R)
            #raise("Matrix is not Positive Definite.")
        Ax = op.R(x, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=False)
        #Ax = R

    if 'convIter' in kwargs:
        return [i, Ax]
    else:
        #plt.scatter(list(range(0,len(del_new_vals))), del_new_vals, marker='o')
        #plt.show()
        #print('CGS cnverged in %d iterations.' % (i))
        return [x, Ax]

def testMatrix(ndim, seed = None):

    A = make_spd_matrix(ndim, random_state=seed)

    eigVals = LA.eigvals(A)
    posDef = (eigVals > 0).all()
    symmetric = LA.norm(A.T - A, np.inf) < 1e-6

    if not posDef or not symmetric:
        raise("Matrix is not Positive Definite.")

    return A

# Test CGS
#ndim_val = [10, 10**2, 10**3, int(10**(3.5))]
#for ndim in ndim_val:
#    A_test = testMatrix(ndim, seed = None)
#    print('%.1e' % (np.size(A_test)))
#    #b = np.rand(ndim, 1)
#    b = np.ones((ndim, 1))
#
#    soln = np.dot(LA.inv(A_test), b)
#    x_guess = np.zeros( (ndim, 1))
#    [i, Ax] = Atimes( x_guess, b, eqn = 0, A = A_test, convIter = None)
#
#    print('Error Norm:')
#    print(LA.norm(Ax-b, np.inf))
#    print('Converged in %d iterations.' % (i))
#
```

```
"""
Created on May 2 2021
@author: Shehan M. Parmar
Test the Crank-Nicholson Method for the
2D diffusion equation, ut = a (uxx + uyy)
"""

# Main.py local dependencies
from get_global import * # nx, ny, Lx, Ly, dx, dy, q_size, p_size are 'GLOBAL'
from init import *
import operators as op
import operator_verf as opf
import matplotlib.pyplot as plt
from matplotlib import cm
from cgs import *
from matplotlib.animation import FuncAnimation
from scipy.sparse.linalg import cg

outFile = 'output'+filename.split('inputs')[-1]


# Choose function with known analytic solution for the diffusion equation
functions = {
        "u_xyt"   : lambda x, y, t: np.exp(-2*np.pi**2*t)*np.sin(np.pi*x)*np.sin(np.pi*y),
        "v_xyt"   : lambda x, y, t: np.exp(-2*np.pi**2*t)*np.sin(np.pi*x)*np.sin(np.pi*y),
        "Lu_xyt"  : lambda x, y, t: -2*np.pi**2*np.exp(-2*np.pi**2*t)*np.sin(np.pi*x)*np.si
n(np.pi*y)
        }

u_xyt = functions["u_xyt"]
Lu_xyt = functions["Lu_xyt"]
v_xyt = functions["v_xyt"]

dxdy = []
L2 = []
Linf = []
acc = 0
qBC = {}

dt = .1
T = 100
Nt = int(round(T/float(dt)))
t = np.linspace(0, Nt*dt, Nt+1)
alpha = .5 # Crank-Nicholson
nu = 1

Nt = inttf = 10
grid = zip(dx, dy, nx, ny, q_size)
for dxi, dyi, nxi, nyi, q_sizei in grid:

    # ---------- Initialize Simulation Domain ------------------

    print('dxi = %.3e, dyi = %.3e, dx*dy = %.3e' % (dxi, dyi, dxi*dyi))
    print('dt < dxdx/nu -> %.3e < %.3e / %.5f -> %s' % (dt, dxi*dyi, nu, str(dt<(dxi*dyi/nu
)) ))
    [ui, vi, pi] = init(nxi, nyi, pinned=False)

    # U Positions
    xu = dxi*(1. + np.arange(0, nxi-1))
    yu = dyi*(0.5 + np.arange(0, nyi))
    Xu, Yu = np.meshgrid(xu, yu)

    # V Positions
    xv = dxi*(0.5 + np.arange(0, nxi))
    yv = dyi*(1.0 + np.arange(0, nyi-1))
    Xv, Yv = np.meshgrid(xv, yv)

    # IC t = 0
```

```
    t0 = 0
    U = np.reshape(u_xyt(Xu, Yu, t0), (1, nyi*(nxi-1)))
    V = np.reshape(v_xyt(Xv, Yv, t0), (1, nxi*(nyi-1)))

    q_n = np.concatenate( (U, V), axis = 1)
    q_n = q_n[0]

    # ---------- Plot Initial U ------------------
    plotInit = False
    if plotInit:
        fig = plt.figure()
        ax = plt.axes(projection='3d')
        q_u = np.reshape(q_n[0:nyi*(nxi-1)], (Xu.shape))
        surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                cmap=cm.viridis, linewidth=0, antialiased=True)
        ax.set_zlim(0, 1.5)
        ax.set_xlabel('$xu$')
        ax.set_ylabel('$yu$')
        ax.view_init(30, 45)
        plt.show()

    # ---------- Set Boundary Conditions -----------------------

    # Top Wall BC
    qBC["uT"] = u_xyt(xu,Ly, 0)
    qBC["vT"] = v_xyt(xv,Ly, 0)
    # Bottom Wall BC
    qBC["uB"] = u_xyt(xu,0, 0)
    qBC["vB"] = v_xyt(xv,0, 0)
    # Left Wall BC
    qBC["uL"] = u_xyt(0,yu, 0)
    qBC["vL"] = v_xyt(0,yv, 0)
    # Right Wall BC
    qBC["uR"] = u_xyt(Lx,yu, 0)
    qBC["vR"] = v_xyt(Lx,yv, 0)


    q_np1 = np.zeros(q_n.shape)
    q_np1 = q_np1[0]

    for tn in range(0, Nt):

        # ---------- Set RHS of Ax = b for Diffusion Eq.  --------

        bcL_n = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)

        # Top Wall BC
        qBC["uT"] = u_xyt(xu,Ly, (1+tn)*dt)
        qBC["vT"] = v_xyt(xv,Ly, (1+tn)*dt)
        # Bottom Wall BC
        qBC["uB"] = u_xyt(xu,0,  (1+tn)*dt)
        qBC["vB"] = v_xyt(xv,0,  (1+tn)*dt)
        # Left Wall BC
        qBC["uL"] = u_xyt(0,yu,  (1+tn)*dt)
        qBC["vL"] = v_xyt(0,yv,  (1+tn)*dt)
        # Right Wall BC
        qBC["uR"] = u_xyt(Lx,yu, (1+tn)*dt)
        qBC["vR"] = v_xyt(Lx,yv, (1+tn)*dt)
        if tn == 0:
            bcL_np1 = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=Fa
lse)
        else:
            bcL_np1 = op.bclap(q_np1, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=
False)


        # ---------- Plot Lq ------------------
```

```
        plotInit = False
        if plotInit:
            Lu = np.reshape(Lu_xyt(Xu, Yu, t0), (1, nyi*(nxi-1)))
            Lu = Lu[0]

            Lq_u = Lq[0:nyi*(nxi-1)] + bcl_n[0:nyi*(nxi-1)]
            print(LA.norm(Lu-Lq_u, np.inf))

            Lq_u = np.reshape(Lq_u, (Xu.shape))

            fig = plt.figure()
            ax = plt.axes(projection='3d')
            surf = ax.plot_surface(Xu, Yu, Lq_u, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            #ax.set_zlim(0, 1.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()

        #S = op.S(q_n, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt, pinned=False
)
        Lq = op.laplace(q_n, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)
        S = np.add(1, np.multiply(alpha*nu*dt, Lq))
        bcL_n = np.multiply(alpha*dt*nu, bcL_n)
        bcL_np1 = np.multiply(alpha*dt*nu, bcL_np1)

        b = S + bcL_n + bcL_np1
        # ---------- Compare exact "b" value, op.R(q_ex)  --------
        q_n_exact = np.concatenate(\
                    (np.reshape(u_xyt(Xu, Yu, (tn)*dt), (1, nyi*(nxi-1))),\
                    np.reshape(v_xyt(Xv, Yv, (tn)*dt), (1, nxi*(nyi-1)))),\
                    axis = 1)
        q_np1_exact = np.concatenate(\
                    (np.reshape(u_xyt(Xu, Yu, (1+tn)*dt), (1, nyi*(nxi-1))),\
                    np.reshape(v_xyt(Xv, Yv, (1+tn)*dt), (1, nxi*(nyi-1)))),\
                    axis = 1)
        q_n_exact = q_n_exact[0]
        q_np1_exact = q_np1_exact[0]
        RHS = op.S(q_n_exact, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt, pinne
d=False)\
            + bcL_n + bcL_np1
        LHS = op.R(q_np1_exact, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt, pin
ned=False)
        LHS = LHS[-1]
        print('RHS of Ax = b (op.S(q_n))')
        print(RHS)
        print('LHS of Ax = b (op.R(q_np1))')
        print(LHS)
        diff_sides = LHS-RHS
        print(RHS-LHS)
        print(LA.norm(RHS-LHS, np.inf))


        # ---------- Solve for the LHS of Ax = b for Diffusion Eq.  --------

        #[x, Ax] = Atimes(np.ones(q_n.shape), b, 3, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei
, alpha, nu, dt, pinned=False)

        #Lq = op.laplace(b, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)
        #R = np.add(1, np.multiply(Lq, -alpha*nu*dt))


        #R = op.R(np.ones(q_n.shape), ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, d
t, pinned=False)
        R = np.subtract(1, np.multiply(Lq, alpha*nu*dt))
        A = np.diag(R)
        eigVals = LA.eigvals(A)
```

```
        posDef = (eigVals > 0).all()
        symmetric = LA.norm(A.T - A, np.inf) < 1e-6
        if not posDef or not symmetric:
            print(R)
            raise("Matrix is not Positive Definite.")


        #c = cg(A, b, x0 = np.zeros(b.shape))
        #c = c[0]
        #c = LA.solve(A, b)
        print(c)
        [c, Ax] = Atimes(np.ones(q_n.shape), b, 3, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei,
 alpha, nu, dt, pinned=False)
        #c = np.multiply(-1,c)
        #b2 = np.dot(A, c)
        print('LA.solve norm')
        print(LA.norm(b-np.multiply(A,c), np.inf))

        # ---------- Plot U^n+1 ------------------

        plotInit = True
        if plotInit:
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            q_u = np.reshape(c[0:nyi*(nxi-1)], (Xu.shape))
            surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            #ax.set_zlim(0, 1.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()

        q_np1 = c

        q_n = q_np1
        #q_np1 = q_n

        q_n_exact = np.concatenate(\
                    (np.reshape(u_xyt(Xu, Yu, (1+tn)*dt), (1, nyi*(nxi-1))),\
                    np.reshape(v_xyt(Xv, Yv, (1+tn)*dt), (1, nxi*(nyi-1)))),\
                    axis = 1)

        q_n_exact = q_n_exact[0]
        print(q_n_exact)
        print('Time = %.3f' % ((tn+1)*dt))
        print('Error = %.3e' % (LA.norm(q_n_exact-q_n, np.inf)))

        # ---------- Plot Uex^n+1 ------------------
        plotExact = True
        if plotExact:
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            #q_u_exact = np.reshape(q_n_exact[0:(nyi*(nxi-1))], (Xu.shape))
            b_ex = op.R(q_n_exact, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt,
pinned=False)
            q_u_exact = np.reshape(b_ex[0:nyi*(nxi-1)], Xu.shape)
            #plt.contourf(Xu, Yu, q_u_exact)
            #ax.contour3D(Xu, Yu, q_u_exact, 50)
            surf = ax.plot_surface(Xu, Yu, q_u_exact, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            ax.set_zlim(0, 1.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()

        plotCurrent = False
```

```
        if plotCurrent:
            # Current Simulation
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            q_u = np.reshape(q_n[0:(nyi*(nxi-1))], (Xu.shape))
            #plt.contourf(Xu, Yu, q_u_exact)
            #ax.contour3D(Xu, Yu, q_u_exact, 50)
            surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            ax.set_zlim(0, 1.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()
```

```
from get_global import *
from init import *
from scipy.stats import linregress
import operators as op
import operator_verf as opf
import matplotlib.pyplot as plt
from matplotlib import cm
from cgs import *
from matplotlib.animation import FuncAnimation
from scipy.sparse.linalg import cg
import visualization as vis

# Choose function with known analytic solution for the diffusion equation
functions = {
        "u_xyt"   : lambda x, y, t: np.exp(-2*np.pi**2*t)*np.sin(np.pi*x)*np.sin(np.pi*y),
        "v_xyt"   : lambda x, y, t: np.exp(-2*np.pi**2*t)*np.sin(np.pi*x)*np.sin(np.pi*y),
        "Lu_xyt"  : lambda x, y, t: -2*np.pi**2*np.exp(-2*np.pi**2*t)*np.sin(np.pi*x)*np.si
n(np.pi*y)

        }


u_xyt = functions["u_xyt"]
v_xyt = functions["v_xyt"]
Lu_xyt = functions["Lu_xyt"]

dxdy = []
L2 = []
Linf = []
acc = 0
qBC = {}

save = True
dt = 1
T = 1
Nt = int(T/dt)
t = np.linspace(0, Nt*dt, Nt+1)
alpha = .5 # Crank-Nicholson
nu = 1

grid = zip(dx, dy, nx, ny, q_size)
for dxi, dyi, nxi, nyi, q_sizei in grid:

    # ---------- Initialize Simulation Domain ------------------

    [ui, vi, pi] = init(nxi, nyi, pinned=False)

    # U Positions
    xu = dxi*(1. + np.arange(0, nxi-1))
    yu = dyi*(0.5 + np.arange(0, nyi))
    Xu, Yu = np.meshgrid(xu, yu)

    # V Positions
    xv = dxi*(0.5 + np.arange(0, nxi))
    yv = dyi*(1.0 + np.arange(0, nyi-1))
    Xv, Yv = np.meshgrid(xv, yv)

    # IC U, V @(x,y,t=0)
    t0 = 0
    U = np.reshape(u_xyt(Xu, Yu, t0), (1, nyi*(nxi-1)))
    V = np.reshape(v_xyt(Xv, Yv, t0), (1, nxi*(nyi-1)))

    q_n = np.concatenate( (U, V), axis = 1)
    q_n = q_n[0]

    # ---------- Set Boundary Conditions -----------------------

    # Top Wall BC
```

```
    qBC["uT"] = u_xyt(xu,Ly, 0)
    qBC["vT"] = v_xyt(xv,Ly, 0)
    # Bottom Wall BC
    qBC["uB"] = u_xyt(xu,0, 0)
    qBC["vB"] = v_xyt(xv,0, 0)
    # Left Wall BC
    qBC["uL"] = u_xyt(0,yu, 0)
    qBC["vL"] = v_xyt(0,yv, 0)
    # Right Wall BC
    qBC["uR"] = u_xyt(Lx,yu, 0)
    qBC["vR"] = v_xyt(Lx,yv, 0)

    bcL_n = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)

    # ---------- Plot Initial U ------------------
    plotInit = False
    if plotInit:
        fig = plt.figure()
        ax = plt.axes(projection='3d')

        q_u = np.reshape(q_n[0:nyi*(nxi-1)], (Xu.shape))

        surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                cmap=cm.viridis, linewidth=0, antialiased=True)
        ax.set_zlim(0, 1.5)
        ax.set_xlabel('$xu$')
        ax.set_ylabel('$yu$')
        ax.view_init(30, 45)
        plt.show()

    # ---------- Plot Laplacian (it works) ------------------
    plotLap = False
    if plotLap:
        fig = plt.figure()
        ax = plt.axes(projection='3d')

        Lq_n = op.laplace(q_n, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)
        Lq_n = Lq_n[0:nyi*(nxi-1)]
        LqBC = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)
        LqBC = LqBC[0:nyi*(nxi-1)]
        Lq = Lq_n + LqBC
        Lq_n_ex = np.reshape(Lu_xyt(Xu, Yu, t0), (1, nyi*(nxi-1)))
        Lq_n_ex = Lq_n_ex[0]

        error = LA.norm(Lq - Lq_n_ex, np.inf)
        for j in range(0,nyi):
            print('***** Row %d *****' % (j) )
            Lq_n_row = Lq[(nxi-1)*j:(nxi-1)*(j+1)]
            Lq_n_ex_row = Lq_n_ex[(nxi-1)*j:(nxi-1)*(j+1)]
            error = LA.norm(Lq_n_row - Lq_n_ex_row, np.inf)
            print('Error norm for j = %d is %.3e' % (j, error))

        Lq = np.reshape(Lq[0:nyi*(nxi-1)], (Xu.shape))
        Lq_n_ex = np.reshape(Lq_n_ex[0:nyi*(nxi-1)], (Xu.shape))

        surf = ax.plot_surface(Xu, Yu, Lq, rstride=1, cstride=1,\
                cmap=cm.viridis, linewidth=0, antialiased=True)
        ax.set_title('Error Norm of Laplacian: ' + str(error))
        ax.set_xlabel('$xu$')
        ax.set_ylabel('$yu$')
        ax.view_init(30, 45)
        plt.show()


    # ---------- Begin Time-Stepping ---
    tn = 0
    dt = dxi #dt * 0.3
    Nt = int(T/dt)
```

```
    print(Nt)
    for tn in range(0, Nt):

        # ---------- Set Boundary Conditions for n+1 ------------

        # Top Wall BC
        qBC["uT"] = u_xyt(xu,Ly, dt*(1+tn))
        qBC["vT"] = v_xyt(xv,Ly, dt*(1+tn))
        # Bottom Wall BC
        qBC["uB"] = u_xyt(xu,0, dt*(1+tn))
        qBC["vB"] = v_xyt(xv,0, dt*(1+tn))
        # Left Wall BC
        qBC["uL"] = u_xyt(0,yu, dt*(1+tn))
        qBC["vL"] = v_xyt(0,yv, dt*(1+tn))
        # Right Wall BC
        qBC["uR"] = u_xyt(Lx,yu, dt*(1+tn))
        qBC["vR"] = v_xyt(Lx,yv, dt*(1+tn))

        bcL_np1 = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)

        # ---------- Set RHS of Ax = b for Diffusion Eq.  --------
        bc = np.multiply(0.5*dt*nu, np.add(bcL_n, bcL_np1))
        Sq_n = op.S(q_n, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt, pinned=Fal
se)
        b = Sq_n + bc

        # Solve without CGS
        cgs = True
        if not cgs:
            R = op.R(np.ones(q_n.shape), ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu
, dt, pinned=False)
            A = np.diag(R[-1])
            q_np1 = LA.solve(A, b)
        else:
            [q_np1, Rq_np1] = Atimes(np.zeros(q_n.shape), b, 3, ui, vi, pi, dxi, dyi, nxi,
nyi, q_sizei, q_sizei, alpha, nu, dt, pinned=False)


        qu_np1 = q_np1[0:nyi*(nxi-1)]
        q_np1_ex = np.concatenate(\
                    (np.reshape(u_xyt(Xu, Yu, (1+tn)*dt), (1, nyi*(nxi-1))),\
                    np.reshape(v_xyt(Xv, Yv, (1+tn)*dt), (1, nxi*(nyi-1)))),\
                    axis = 1)
        qu_np1_ex = q_np1_ex[0][0:nyi*(nxi-1)]

        error = LA.norm(qu_np1 - qu_np1_ex, np.inf)
        #print('Time = %f' % ((tn+1)*dt))
        #print('Error b/w qu_np1 and qu_np1_ex: ' + str(error))

        # ---------- Plot U^n+1 ------------------

        plotInit = False
        if plotInit:
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            q_u = np.reshape(q_n[0:nyi*(nxi-1)], (Xu.shape))
            surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            ax.set_zlim(0, 1.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()

        q_n = q_np1
        bcL_n = bcL_np1

        # ---------- Plot Uex^n+1 ------------------
```

```
        plotExact = False
        if plotExact:
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            #q_u_exact = np.reshape(q_n_exact[0:(nyi*(nxi-1))], (Xu.shape))
            b_ex = op.R(q_n_exact, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt,
pinned=False)
            q_u_exact = np.reshape(b_ex[0:nyi*(nxi-1)], Xu.shape)
            #plt.contourf(Xu, Yu, q_u_exact)
            #ax.contour3D(Xu, Yu, q_u_exact, 50)
            surf = ax.plot_surface(Xu, Yu, q_u_exact, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            ax.set_zlim(0, 1.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()

        plotCurrent = False
        if plotCurrent:
            # Current Simulation
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            q_u = np.reshape(q_n[0:(nyi*(nxi-1))], (Xu.shape))
            #plt.contourf(Xu, Yu, q_u_exact)
            #ax.contour3D(Xu, Yu, q_u_exact, 50)
            surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                    cmap=cm.viridis, linewidth=0, antialiased=True)
            ax.set_zlim(0, 1.5)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()
    Linf.append(LA.norm(qu_np1 - qu_np1_ex, np.inf))
    dxdy.append(dt)
err = Linf
lin = linregress(np.log10(dxdy), np.log10(err))
acc = lin.slope
vis.plotL2vsGridSize(lin, dxdy, err, 'Diffusion_Eq', 'Diff. Eq.', save=save)
```

```
"""
Created on May 2 2021
@author: Shehan M. Parmar
Global variables for NS solver.
"""
import numpy as np
filename = 'inputs.txt'
#filename = 'inputsLDCTest.txt'
filename = 'inputsDiffEqTest.txt'
#filename = 'inputsLapTest.txt'
#filename = 'inputsAdvTest.txt'
#filename = 'inputsDivTest.txt'
#filename = 'inputsGradTest.txt'
#filename = 'inputsGradTest_min.txt'
inpFilePath = './InputFiles/'
with open(inpFilePath + filename, 'r') as inp:

    inputs = {}

    for line in inp:
        key = line.split('=')[0].strip()
        attr = line.split('=')[1].strip()
        if ',' in attr: # applies only for nx, ny, or dt
            attr = attr.split(',')
            if ("nx" == key) or ("ny" == key):
                attr = np.array([int(entry) for entry in attr])
                inputs[key] = attr
            elif "dt" == key:
                attr = np.array([float(entry) for entry in attr])
                inputs[key] = attr
            continue

        inputs[key] = float(attr)

    if isinstance(inputs["nx"], str): # NOTE: will not occure, remove in future push
        nx = int(inputs["nx"])
        ny = int(inputs["ny"])
    elif isinstance(inputs["nx"], float): # occurs everytime unless nx is an array
        nx = int(inputs["nx"])
        ny = int(inputs["ny"])
    elif isinstance(inputs["nx"], np.ndarray):
        nx = inputs["nx"]
        ny = inputs["ny"]

    try:
        if len(nx) != len(ny):
            raise("Inputs in nx and ny MUST match.")
    except:
        pass # inputs in nx, ny are single integer values

    Lx = float(inputs["Lx"])
    Ly = float(inputs["Ly"])

    dx = Lx/(nx)
    dy = Ly/(ny)

    q_size = (nx-1)*ny + nx*(ny-1)
    #q_size = (nx-2)*(ny-1) + (nx-1)*(ny-2)

    p_size = nx*ny-1 # subtract one only for pinned pressure values
    #p_size = (nx-1)*(ny-1)-1 # subtract one only for pinned pressure values

    #dt = inputs["dt"]
```

```
"""
Created on May 2 2021
@author: Shehan M. Parmar
Initialize pointer arrays to ease coding of
velocity and pressure variables matrices.
"""
import numpy as np

def init(nx, ny, pinned = True):

    u = np.ndarray((nx-1, ny), dtype=object)
    v = np.ndarray((nx, ny-1), dtype=object)
    p = np.ndarray((nx, ny)  , dtype=object)

    # Create pointers for velocity, u, v
    ind = int(0)
    for j in range(0,ny):
        for i in range(0,nx-1):
            u[i,j] = int(ind)
            ind += 1
    for j in range(0,ny-1):
        for i in range(0,nx):
            v[i,j] = int(ind)
            ind += 1
    if ind != ((nx-1)*ny + nx*(ny-1)):
        raise IndexError('wrong velocity size')

    # create points for pressure, p
    ind = 0
    for j in range(0,ny):
        for i in range(0,nx):
            if (i==0) and (j==0):
                if pinned:
                    #p[i,j] = None
                    pass # skip pinned pressure
                else:
                    p[i,j] = int(ind)
                    ind += 1
            else:
                p[i,j] = int(ind)
                ind += 1

    if ind != (nx*ny-1):
        if pinned:
            raise IndexError('wrong pressure index (pinned)')
        elif not pinned and (ind != (nx*ny)):
            raise IndexError('wrong pressure index (not pinned)')

    return u, v, p
```

```
from get_global import *
from init import *
import operators as op
import operator_verf as opf
import matplotlib.pyplot as plt
from matplotlib import cm
from cgs import *
from matplotlib.animation import FuncAnimation
from scipy.sparse.linalg import cg
import visualization as vis
import csv
import pandas as pd
from ma import *

plotCurrent = False

dxdy = []
L2 = []
Linf = []
acc = 0
qBC_nm1 = {}
qBC = {}

dt = 5e-3
T = 10
Nt = int(T/dt)
print('Nt = %d' % (Nt))
t = np.linspace(0, Nt*dt, Nt)
alpha = .5 # Crank-Nicholson
Re = 100
nu = 1./Re
a = 2

grid = zip(dx, dy, nx, ny, q_size, p_size)
for dxi, dyi, nxi, nyi, q_sizei, g_sizei in grid:

    # ---------- Initialize Simulation Domain ------------------

    [ui, vi, pi] = init(nxi, nyi)

    # U Positions
    xu = dxi*(1. + np.arange(0, nxi-1))
    yu = dyi*(0.5 + np.arange(0, nyi))
    Xu, Yu = np.meshgrid(xu, yu)

    # V Positions
    xv = dxi*(0.5 + np.arange(0, nxi))
    yv = dyi*(1.0 + np.arange(0, nyi-1))
    Xv, Yv = np.meshgrid(xv, yv)

    # IC U, V @(x,y,t=0)
    q_nm1 = np.zeros(q_sizei)

    # ---------- Set Boundary Conditions -----------------------

    # Top Wall BC
    qBC_nm1["uT"] = np.ones(xu.shape)
    qBC_nm1["vT"] = xv*0
    # Bottom Wall BC
    qBC_nm1["uB"] = xu*0
    qBC_nm1["vB"] = xv*0
    # Left Wall BC
    qBC_nm1["uL"] = yu*0
    qBC_nm1["vL"] = yv*0
    # Right Wall BC
    qBC_nm1["uR"] = yu*0
    qBC_nm1["vR"] = yv*0
```

```
    # ---------- SOLVE FOR u(x,y,tn) WHERE n = 1 ------------
    # ---------- Set Boundary Conditions for n+1 ------------

    q_n = q_nm1

    # Top Wall BC
    qBC["uT"] = np.ones(xu.shape)
    qBC["vT"] = xv*0
    # Bottom Wall BC
    qBC["uB"] = xu*0
    qBC["vB"] = xv*0
    # Left Wall BC
    qBC["uL"] = yu*0
    qBC["vL"] = yv*0
    # Right Wall BC
    qBC["uR"] = yu*0
    qBC["vR"] = yv*0

    bcL_n = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei)

    # ---------- Plot Initial U ------------------
    plotInit = False
    if plotInit:
        fig = plt.figure()
        ax = plt.axes(projection='3d')

        q_u = np.reshape(q_n[0:nyi*(nxi-1)], (Xu.shape))

        surf = ax.plot_surface(Xu, Yu, q_u, rstride=1, cstride=1,\
                cmap=cm.viridis, linewidth=0, antialiased=True)
        ax.set_zlim(0, 1.5)
        ax.set_xlabel('$xu$')
        ax.set_ylabel('$yu$')
        ax.view_init(30, 45)
        plt.show()

    X = np.reshape(q_nm1[0:nyi*(nxi-1)], (Xu.shape))

    # ---------- Begin Time-Stepping ---
    for tn in range(1, Nt+1):

        # ---------- Set Boundary Conditions for n+1 ------------

        # Top Wall BC
        qBC["uT"] = np.ones(xu.shape)
        qBC["vT"] = xv*0
        # Bottom Wall BC
        qBC["uB"] = xu*0
        qBC["vB"] = xv*0
        # Left Wall BC
        qBC["uL"] = yu*0
        qBC["vL"] = yv*0
        # Right Wall BC
        qBC["uR"] = yu*0
        qBC["vR"] = yv*0

        bcL_np1 = op.bclap(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei)

        # ---------- Momentum Eq.  ----------------------------------------
        bcL = np.multiply(0.5*dt*nu, np.add(bcL_n, bcL_np1))
        Sq_n = op.S(q_n, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt)

        Aq_nm1 = op.adv(q_nm1, qBC_nm1, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei)
        Aq_n = op.adv(q_n, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei)
        adv = np.multiply(-0.5*dt, np.subtract(np.multiply(3, Aq_n), Aq_nm1))

        b = Sq_n + bcL + adv
```

```
        [q_F, Rq_np1] = Atimes(np.zeros(q_n.shape), b, 3, ui, vi, pi, dxi, dyi, nxi, nyi, q
_sizei, g_sizei, alpha, nu, dt, pinned=True)

        # ---------- Pressure Poisson Eq. ----------------------------------------
        Du_F = op.div(q_F, ui, vi, pi, dxi, dyi, nxi, nyi, g_sizei)\
             + op.bcdiv(qBC, ui, vi, pi, dxi, dyi, nxi, nyi, g_sizei)

        ppe_rhs = np.multiply(1./dt, Du_F)
        b2 = -ppe_rhs

        [P_np1, Ax_PPE] = Atimes(np.zeros(g_sizei), b2, 2, ui, vi, pi, dxi, dyi, nxi, nyi,
q_sizei, g_sizei, alpha, nu, dt)

        # ---------- Projection Step ----------------------------------------
        GP_np1 = op.grad(P_np1, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei)
        RinvGP_np1 = op.Rinv(GP_np1, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, alpha, nu, dt
, pinned=True)
        q_np1 = np.subtract(q_F, np.multiply(dt, RinvGP_np1))

        q_nm1 = q_n
        qBC_nm1 = qBC
        q_n = q_np1
        bcL_n = bcL_np1

        # ---------- Visualization & Save Data ------------------------------
        #vis.plotVelocity(q_n, qBC, xu, xv, yu, yv, nxi, nyi, dt*tn, Re, drawNow = True, qu
iverOn = True)

        #if (tn % 5) == 0:
        #    vis.plotVelocity(q_n, qBC, xu, xv, yu, yv, nxi, nyi, dt*tn, Re, drawNow = Fals
e, quiverOn = False)
        #    print('Time = %f' % ((tn+1)*dt))
        #    #plotCurrent = True

        U_data = np.reshape(q_n[0:nyi*(nxi-1)], (Xu.shape))
        X = np.concatenate((X,U_data))
        if (tn % 5) == 0:
            modal_analysis(X, Xu, Yu)

        # ---------- Save X-Data at y = 0.5 ------------------
        plotXTime = False
        if plotXTime:
            q_u = np.reshape(q_n[0:nyi*(nxi-1)], (Xu.shape))
            U_data.append(q_u[5])
            time.append(tn*dt)
            #plt.plot(xu, q_u[5])

        if plotCurrent:
            # Current Simulation
            levels = np.linspace(-0.3, 1, 1000)
            fig, ax = plt.subplots()
            q_u = np.reshape(q_n[0:(nyi*(nxi-1))], (Xu.shape))
            CS = ax.contourf(Xu, Yu, q_u, levels=levels, cmap=cm.viridis)
            fig.colorbar(CS)
            ax.set_xlabel('$X$')
            ax.set_ylabel('$Y$')
            plotCurrent = False
```

```python
"""
Create June 6th, 2021
@author: Shehan Parmar
Python routine for modal analysis of
lid-driven cavity.
"""
import numpy as np
import numpy.linalg as LA
import matplotlib.pyplot as plt
from matplotlib import cm
def modal_analysis(data, x, y):
    """
    data -- numpy stack of arrays (i.e. U(x,y,t1), U(x,y,t2), ... U(x,y,tn)
    """
    POD, sing_val, temp = LA.svd(data)
    LidDrivenRecon = np.matrix(POD[:, :20])*np.diag(sing_val[:20])*np.matrix(temp[:20, :])
    plt.imshow(LidDrivenRecon, cmap=cm.viridis)
    plt.show()
    #plt.contourf(Xu, Yu, POD)
    print(POD[:,0].shape)
    print(x.shape)
    print(y.shape)
```

```
"""
Created on May 2 2021
@author: Shehan M. Parmar
Main Navier-Stokes solver
"""

# Main.py local dependencies
from get_global import * # nx, ny, Lx, Ly, dx, dy, q_size, p_size are 'GLOBAL'
from init import *
import operators as op
import operator_verf as opf
from cgs import *

outFile = 'output'+filename.split('inputs')[-1]
#[u, v, p] = init(nx, ny)

# Test Gradient Operator is Second-Order Accurate
#[dxdy, err, acc] = opf.test_grad(dx, dy, nx, ny, Lx, Ly, q_size, outFile, save=True)

# Test Divergence Operator is Second-Order Accurate
#[dxdy, err, acc] = opf.test_div(dx, dy, nx, ny, Lx, Ly, p_size, outFile, save=True)

# Test Laplace Operator is Second-Order Accurate
#[dxdy, err, acc] = opf.test_laplace(dx, dy, nx, ny, Lx, Ly, q_size, outFile, save=True)

# Test Advective Operator is Second-Order Accurate
[dxdy, err, acc] = opf.test_adv(dx, dy, nx, ny, Lx, Ly, q_size, outFile, save=True)

# Test CGS Solver

#A = testMatrix()
#Ax = Atimes(x, b, 0, A)
```

```
"""
Created on May 12 2021
@author: S. M. Parmar
Verify discrete operators for Navier-Stokes solver
with known exact solutions.
"""
import numpy as np
from scipy.stats import linregress
from numpy import linalg as LA
from matplotlib import cm
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import mpltex

import operators as op
from init import *
import visualization as vis

def test_grad(dx, dy, nx, ny, Lx, Ly, q_size, outFile, plots=True, save=False):

    # Choose function with known analytic solution for gradient
    functions = {
            "f1"   : lambda x, y : np.sin(x*y),
            "dfx1" : lambda x, y : y*np.cos(x*y),
            "dfy1" : lambda x, y : x*np.cos(x*y),
            "f2"   : lambda x, y : x**2*y**2,
            "dfx2" : lambda x, y : 2*x*y**2,
            "dfy2" : lambda x, y : 2*y*x**2,
            "f3"   : lambda x, y : x*np.cos(y) + y,
            "dfx3" : lambda x, y : np.cos(y),
            "dfy3" : lambda x, y : -x*np.sin(y) + 1,
            "f4"   : lambda x, y : np.sin(x)*np.sin(y),
            "dfx4" : lambda x, y : np.sin(y)*np.cos(x),
            "dfy4" : lambda x, y : np.sin(x)*np.cos(y),
            "f5"   : lambda x, y : np.sin(y)+np.cos(x),
            "dfy5" : lambda x, y : np.cos(y),
            "dfx5" : lambda x, y : -np.sin(x)
            }

    f = functions["f4"]
    dfx = functions["dfx4"]
    dfy = functions["dfy4"]

    dxdy = []
    L2 = []
    Linf = []
    acc = 0

    grid = zip(dx, dy, nx, ny, q_size)
    for dxi, dyi, nxi, nyi, q_sizei in grid:

        [ui, vi, pi] = init(nxi, nyi, pinned=False)

        xu = dxi*(1. + np.arange(0, nxi-1))
        yu = dyi*(0.5 + np.arange(0, nyi))
        Xu, Yu = np.meshgrid(xu, yu)
        Zxu = dfx(Xu, Yu)
        grad_x_ex = np.reshape(Zxu, (1, nyi*(nxi-1)))

        xv = dxi*(0.5 + np.arange(0, nxi))
        yv = dyi*(1.0 + np.arange(0, nyi-1))
        Xv, Yv = np.meshgrid(xv, yv)
        Zyv = dfy(Xv, Yv)
        grad_y_ex = np.reshape(Zyv, (1, nxi*(nyi-1)))

        grad_ex = np.concatenate((grad_x_ex, grad_y_ex), axis=1)
        grad_ex = grad_ex[0]
```

```
        xp = dxi*(0.5+np.arange(0, nxi))
        yp = dyi*(0.5+np.arange(0, nyi))
        Xp, Yp = np.meshgrid(xp, yp)
        Zp = f(Xp,Yp)
        g_test = np.reshape(Zp, (1,nxi*nyi))
        g_test = g_test[0]

        # Alternative Approach that also works:
        grad_ex2 = np.zeros( nyi*(nxi-1) + nxi*(nyi-1) )
        for j in range(0,nyi):
            for i in range(0,nxi-1):
                grad_ex2[ui[i,j]] = dfx( (i+1.)*dxi , (j+0.5)*dyi  )
        for j in range(0,nyi-1):
            for i in range(0,nxi):
                grad_ex2[vi[i,j]] = dfy( (i+0.5)*dxi , (j+1.)*dyi  )

        g = np.zeros(nxi*nyi)
        for j in range(0,nyi):
            for i in range(0,nxi):
                g[pi[i,j]] = f( (i+0.5)*dxi, (j+0.5)*dyi )

        q = op.grad(g_test, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)

        diff = np.abs(q-grad_ex)

        dxdy.append(dxi)
        L2.append(  LA.norm(diff) / len(diff) )
        Linf.append(LA.norm(diff, ord=np.inf))

    err = Linf
    lin = linregress(np.log10(dxdy), np.log10(err))
    acc = lin.slope

    if  plots:
        vis.plotL2vsGridSize(lin, dxdy, err, outFile, 'Gradient', save=save)

    return dxdy, err, acc

def test_div(dx, dy, nx, ny, Lx, Ly, g_size, outFile, plots=True, save=False):

    # Choose function with known analytic solution for divergence
    functions = {
            "fx1"   : lambda x, y : -y + x*0,
            "fy1"   : lambda x, y : x*y,
            "divf1" : lambda x, y : x + y*0,
            "fx2"   : lambda x, y : np.sin(x)*np.cos(y),
            "fy2"   : lambda x, y : -np.cos(x)*np.sin(y),
            "fxy2"  : lambda x, y : np.cos(x)*np.cos(y) - np.cos(x)*np.cos(y),
            "divf2" : lambda x, y : x*0. + y*0.
            }

    fx = functions["fx2"]
    fy = functions["fy2"]
    fxy = functions["fxy2"]
    divf = functions["divf2"]


    dxdy = []
    L2 = []
    Linf = []
    acc = 0
    qBC = {}

    grid = zip(dx, dy, nx, ny, g_size)
    for dxi, dyi, nxi, nyi, g_sizei in grid:

        [ui, vi, pi] = init(nxi, nyi, pinned=False)
```

```
        xu = dxi*(1. + np.arange(0, nxi-1))
        yu = dyi*(0.5 + np.arange(0, nyi))
        Xu, Yu = np.meshgrid(xu, yu)
        Zxu = fx(Xu, Yu)
        q_test_x = np.reshape(Zxu, (1, nyi*(nxi-1)))

        xv = dxi*(0.5 + np.arange(0, nxi))
        yv = dyi*(1.0 + np.arange(0, nyi-1))
        Xv, Yv = np.meshgrid(xv, yv)
        Zyv = fy(Xv, Yv)
        q_test_y = np.reshape(Zyv, (1, nxi*(nyi-1)))

        q_test = np.concatenate((q_test_x, q_test_y), axis=1)
        q_test = q_test[0]

        xp = dxi*(0.5+np.arange(0, nxi))
        yp = dyi*(0.5+np.arange(0, nyi))
        Xp, Yp = np.meshgrid(xp, yp)
        Zp = divf(Xp,Yp)
        divf_ex = np.reshape( Zp, (1,nxi*nyi))
        divf_ex = divf_ex[0]

        # Top Wall BC
        qBC["uT"] = fx(xu,Ly)
        qBC["vT"] = fy(xv,Ly)
        # Bottom Wall BC
        qBC["uB"] = fx(xu,0)
        qBC["vB"] = fy(xv,0)
        # Left Wall BC
        qBC["uL"] = fx(0,yu)
        qBC["vL"] = fy(0,yv)
        # Right Wall BC
        qBC["uR"] = fx(Lx,yu)
        qBC["vR"] = fy(Lx,yv)

        gDiv = op.div(q_test, ui, vi, pi, dxi, dyi, nxi, nyi, g_sizei, pinned=False)
        gBC  =  op.bcdiv(qBC, ui, vi, pi, dxi, dyi, nxi, nyi, g_sizei, pinned=False)
        g = gDiv + gBC
        dxdy.append(dxi)
        L2.append( LA.norm(g-divf_ex) / len(g) )
        Linf.append(LA.norm(g-divf_ex, np.inf))

    err = Linf
    lin = linregress(np.log10(dxdy), np.log10(err))
    acc = lin.slope

    if plots:
        vis.plotL2vsGridSize(lin, dxdy, err, outFile, 'Divergence', save=save)

    return dxdy, err, acc

def test_laplace(dx, dy, nx, ny, Lx, Ly, q_size, outFile, plots=True, save=False):

    # Choose function with known analytic solution for divergence
    functions = {
            "fx1"   : lambda x, y : x**2 + np.sin(y),
            "fy1"   : lambda x, y : x**2 + np.sin(y),
            "Lfx1" : lambda x, y : 2. + x*0. - np.sin(y),
            "Lfy1" : lambda x, y : 2. + x*0. - np.sin(y),

            "fx2"   : lambda x, y : x**2 + y**2,
            "fy2"   : lambda x, y : x**2 + y**2,
            "Lfx2" : lambda x, y : 4. + x*0. + y*0,
            "Lfy2" : lambda x, y : 4. + x*0. + y*0,

            "fx3"   : lambda x, y : x**2 * y**2,
            "fy3"   : lambda x, y : x**2 * y**2,
            "Lfx3" : lambda x, y : 2. * (x**2 + y**2),
```

```
            "Lfy3" : lambda x, y : 2. * (x**2 + y**2),

            "fx4"   : lambda x, y : (np.sin(x)/np.sin(3*np.pi)) + (np.sinh(y)/np.sinh(np.pi
)),
            "fy4"   : lambda x, y : (np.sin(x)/np.sin(3*np.pi)) + (np.sinh(y)/np.sinh(np.pi
)),
            "Lfx4" : lambda x, y : x*0 + y*0,
            "Lfy4" : lambda x, y : x*0 + y*0,

            "fx5"   : lambda x, y : (np.exp(-0.5*np.pi*x) * np.sin(0.5*np.pi*y)),
            "fy5"   : lambda x, y : (np.exp(-0.5*np.pi*x) * np.sin(0.5*np.pi*y)),
            "Lfx5" : lambda x, y : x*0 + y*0,
            "Lfy5" : lambda x, y : x*0 + y*0,

            "fx6"   : lambda x, y : np.sin(np.pi*x)*np.sin(np.pi*y),
            "fy6"   : lambda x, y : np.sin(np.pi*x)*np.sin(np.pi*y),
            "Lfx6" : lambda x, y : -2*np.pi**2*np.sin(np.pi*x)*np.sin(np.pi*y),
            "Lfy6" : lambda x, y : -2*np.pi**2*np.sin(np.pi*x)*np.sin(np.pi*y)
            }

    fx = functions["fx6"]
    Lfx = functions["Lfx6"]

    fy = functions["fy6"]
    Lfy = functions["Lfy6"]

    dxdy = []
    L2 = []
    Linf = []
    acc = 0
    qBC = {}

    grid = zip(dx, dy, nx, ny, q_size)
    for dxi, dyi, nxi, nyi, q_sizei in grid:

        [ui, vi, pi] = init(nxi, nyi, pinned=False)

        xu = dxi*(1. + np.arange(0, nxi-1))
        yu = dyi*(0.5 + np.arange(0, nyi))
        Xu, Yu = np.meshgrid(xu, yu)
        Zxu = fx(Xu, Yu)
        Zxu_ex = Lfx(Xu, Yu)
        q_test_x = np.reshape(Zxu, (1, nyi*(nxi-1)))
        q_test_x_ex = np.reshape(Zxu_ex, (1, nyi*(nxi-1)))

        xv = dxi*(0.5 + np.arange(0, nxi))
        yv = dyi*(1.0 + np.arange(0, nyi-1))
        Xv, Yv = np.meshgrid(xv, yv)
        Zyv = fy(Xv, Yv)
        Zyv_ex = Lfy(Xv, Yv)
        q_test_y = np.reshape(Zyv, (1, nxi*(nyi-1)))
        q_test_y_ex = np.reshape(Zyv_ex, (1, nxi*(nyi-1)))

        q_test = np.concatenate((q_test_x, q_test_y), axis=1)
        q_test_ex = np.concatenate((q_test_x_ex, q_test_y_ex), axis=1)

        q_test = q_test[0]
        q_test_ex = q_test_ex[0]

        # Top Wall BC
        qBC["uT"] = fx(xu,Ly)
        qBC["vT"] = fy(xv,Ly)
        # Bottom Wall BC
        qBC["uB"] = fx(xu,0)
        qBC["vB"] = fy(xv,0)
        # Left Wall BC
        qBC["uL"] = fx(0,yu)
        qBC["vL"] = fy(0,yv)
```

```
        # Right Wall BC
        qBC["uR"] = fx(Lx,yu)
        qBC["vR"] = fy(Lx,yv)

        Lq = op.laplace(q_test, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)
        LqBC  =  op.bclap(q_test, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=Fals
e)
        q = Lq + LqBC

        checkL_T = False
        if checkL_T:
            A = np.diag(q)
            #LA.norm(A-A.T, np.inf) -> results in segmentation faults for larger cases

        # ------------------Plot U or V--------------------------
        plotting = False
        if plotting:
            qu = q[0:nyi*(nxi-1)]
            QU = np.reshape(qu, (Xu.shape))
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            surf = ax.plot_surface(Xu, Yu, QU, rstride=1, cstride=1,\
                    cmap=cm.plasma, linewidth=0, antialiased=True)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()

            qu_ex = q_test_ex[0:nyi*(nxi-1)]
            QU_ex = np.reshape(qu_ex, (Xu.shape))
            fig = plt.figure()
            ax = plt.axes(projection='3d')
            surf = ax.plot_surface(Xu, Yu, QU_ex, rstride=1, cstride=1,\
                    cmap=cm.magma, linewidth=0, antialiased=True)
            ax.set_xlabel('$xu$')
            ax.set_ylabel('$yu$')
            ax.view_init(30, 45)
            plt.show()



        diff = q-q_test_ex
        dxdy.append(dxi)
        L2.append( LA.norm(diff) / len(q) )
        Linf.append(LA.norm(diff, np.inf))
    err = Linf
    lin = linregress(np.log10(dxdy), np.log10(err))
    acc = lin.slope

    if plots:
        vis.plotL2vsGridSize(lin, dxdy, err, outFile, 'Laplace', save=save)

    return dxdy, err, acc

def test_adv(dx, dy, nx, ny, Lx, Ly, q_size, outFile, plots=True, save=False):

    # Choose function with known analytic solution for divergence
    functions = {
            "fu1"   : lambda x, y : np.sin(x)*np.sin(y),
            "fv1"   : lambda x, y : np.cos(x)*np.cos(y),
            "Nx1"   : lambda x, y :   np.cos(x)*np.sin(x)*np.cos(y)**2 + np.cos(x)*np.sin(x
)*np.sin(y)**2,
            "Ny1"   : lambda x, y : - np.cos(y)*np.sin(y)*np.cos(x)**2 - np.cos(y)*np.sin(y
)*np.sin(x)**2,
            "fu2"   : lambda x, y :   np.cos(x)*np.cos(y) + np.sin(x)*np.sin(y),
            "fv2"   : lambda x, y :   x*np.exp(-y/2),
            "Nx2"   : lambda x, y :   2*(np.cos(x)*np.cos(y) + np.sin(x)*np.sin(y))*(np.cos
(x)*np.sin(y) \
```

```
                                    - np.cos(y)*np.sin(x)) - (x*np.exp(-y/2)*(np.cos(x)*np.
cos(y) \
                                    + np.sin(x)*np.sin(y)))/2 - x*np.exp(-y/2)*(np.cos(x)*n
p.sin(y) \
                                    - np.cos(y)*np.sin(x)),
            "Ny2"   : lambda x, y : np.exp(-y/2)*(np.cos(x)*np.cos(y) + np.sin(x)*np.sin(y)
) \
                                    - x**2*np.exp(-y) + x*np.exp(-y/2)*(np.cos(x)*np.sin(y)
 \
                                    - np.cos(y)*np.sin(x))
            }

    fu = functions["fu1"]
    fv = functions["fv1"]
    Nx = functions["Nx1"]
    Ny = functions["Ny1"]

    dxdy = []
    L2 = []
    Linf = []
    acc = 0
    qBC = {}

    grid = zip(dx, dy, nx, ny, q_size)
    for dxi, dyi, nxi, nyi, q_sizei in grid:

        [ui, vi, pi] = init(nxi, nyi, pinned=False)

        xu = dxi*(1. + np.arange(0, nxi-1))
        yu = dyi*(0.5 + np.arange(0, nyi))
        Xu, Yu = np.meshgrid(xu, yu)
        Zxu = fu(Xu, Yu)
        Nx_ex = Nx(Xu, Yu)

        q_test_x = np.reshape(Zxu, (1, nyi*(nxi-1)))
        q_test_x_ex = np.reshape(Nx_ex, (1, nyi*(nxi-1)))

        xv = dxi*(0.5 + np.arange(0, nxi))
        yv = dyi*(1.0 + np.arange(0, nyi-1))
        Xv, Yv = np.meshgrid(xv, yv)
        Zyv = fv(Xv, Yv)
        Ny_ex = Ny(Xv, Yv)

        q_test_y = np.reshape(Zyv, (1, nxi*(nyi-1)))
        q_test_y_ex = np.reshape(Ny_ex, (1, nxi*(nyi-1)))

        q_test = np.concatenate((q_test_x, q_test_y), axis=1)
        q_test_ex = np.concatenate((q_test_x_ex, q_test_y_ex), axis=1)

        q_test = q_test[0]
        q_test_ex = q_test_ex[0]

        # Top Wall BC
        qBC["uT"] = fu(xu,Ly)
        qBC["vT"] = fv(xv,Ly)
        # Bottom Wall BC
        qBC["uB"] = fu(xu,0)
        qBC["vB"] = fv(xv,0)
        # Left Wall BC
        qBC["uL"] = fu(0,yu)
        qBC["vL"] = fv(0,yv)
        # Right Wall BC
        qBC["uR"] = fu(Lx,yu)
        qBC["vR"] = fv(Lx,yv) # added +0.5*dxi

        N = op.adv(q_test, qBC, ui, vi, pi, dxi, dyi, nxi, nyi, q_sizei, pinned=False)

        diff = N-q_test_ex
```

```
        dxdy.append(dyi)

        L2.append( LA.norm(diff) / len(N) )
        Linf.append(LA.norm(diff, np.inf))

    err = Linf
    lin = linregress(np.log10(dxdy), np.log10(err))
    acc = lin.slope

    if plots:
        vis.plotL2vsGridSize(lin, dxdy, err, outFile, 'Nonlinear Advection', save=save)

    return dxdy, err, acc
```

```
"""
Created on May 2 2021
@author: Shehan M. Parmar
Discrete operators for Navier-Stokes solver.
"""
import numpy as np
from numpy import linalg as LA
#from numba import jit

def grad(g, u, v, p, dx, dy, nx, ny, q_size, pinned = True): # Gradient Operator

    q = np.zeros(q_size)

    # Be careful with p(0,0) for the pinned pressure location

    # compute x-dir gradient, u
    for j in [0]:
        for i in [0]:
            if pinned:
                q[u[i,j]] = (g[p[i+1,j]]            )/dx       # - g[p[0,0]]/dx = 0
            else:
                q[u[i,j]] = (g[p[i+1,j]] - g[p[0,0]])/dx       #
        for i in range(1,nx-1):
            q[u[i,j]] = (g[p[i+1,j]] - g[p[i,j]])/dx       #
    for j in range(1,ny):
        for i in range(0,nx-1):
            q[u[i,j]] = (g[p[i+1,j]] - g[p[i,j]])/dx       #

    # compute y-dir gradient, v
    for j in [0]:
        for i in [0]:
            if pinned:
                q[v[i,j]] = (g[p[i,j+1]]            )/dy       # - g[p[0,0]]/dy = 0
            else:
                q[v[i,j]] = (g[p[i,j+1]] - g[p[0,0]])/dy       #

        for i in range(1,nx):
            q[v[i,j]] = (g[p[i,j+1]] - g[p[i,j]])/dy       #

    for j in range(1,ny-1):
        for i in range(0,nx):
            q[v[i,j]] = (g[p[i,j+1]] - g[p[i,j]])/dy       #

    return q

def div(q, u, v, p, dx, dy, nx, ny, p_size, pinned=True): # Divergence Operator

    if pinned:
        g = np.zeros(p_size)
    elif not pinned:
        g = np.zeros(p_size+1)

    # Bottom Row of Grid
    for j in [0]:
        for i in range(1,nx-1):
            g[p[i,j]] = ( q[u[i,j]] - q[u[i-1, j]])/dx \
                      + ( q[v[i,j]]                )/dy
                      #             - q[v[i,j-1]]  /dy
    # Bottom Right
    for j in [0]:
        for i in [nx-1]:
            g[p[i,j]] = (           - q[u[i-1,j]])/dx  \
                      + ( q[v[i,j]]              )/dy
                      #   q[u[i,j]]                /dx
                      #             - q[v[i,j-1]] /dy
    # Left Wall
    for j in range(1, ny-1):
        for i in [0]:
```

```
            g[p[i,j]] = ( q[u[i,j]]               )/dx \
                      + ( q[v[i,j]] - q[v[i,j-1]])/dy
                      #             - q[u[i-1,j]] /dx
    # Right Wall
    for j in range(1,ny-1):
        for i in [nx-1]:
            g[p[i,j]] = (           - q[u[i-1,j]])/dx \
                      + ( q[v[i,j]] - q[v[i,j-1]])/dy
                      #   q[u[i,j]]               /dx
    # Top Wall
    for j in [ny-1]:
        for i in range(1,nx-1):
            g[p[i,j]] = ( q[u[i,j]] - q[u[i-1,j]])/dx \
                      + (           - q[v[i,j-1]])/dy
                      #   q[v[i,j]]               /dy
    # Top Left Corner
    for j in [ny-1]:
        for i in [0]:
            g[p[i,j]] = ( q[u[i,j]]               )/dx \
                      + (           - q[v[i,j-1]])/dy
                      #             - q[u[i-1,j]] /dx
                      #   q[v[i,j]]               /dy
    # Top Right Corner
    for j in [ny-1]:
        for i in [nx-1]:
            g[p[i,j]] = (           - q[u[i-1,j]])/dx \
                      + (           - q[v[i,j-1]])/dy
                      #   q[u[i,j]]               /dx
                      #   q[v[i,j]]               /dy
    # Interior Points
    for j in range(1,ny-1):
        for i in range(1,nx-1):
            g[p[i,j]] = ( q[u[i,j]] - q[u[i-1,j]])/dx \
                      + ( q[v[i,j]] - q[v[i,j-1]])/dy
    return g

def bcdiv(qbc, u, v, p, dx, dy, nx, ny, p_size, pinned=True):
    """
    INPUTS:
    ------
    qbc - dictionary with 8 keys (u and v
    boundary conditions for each wall)
    """
    if pinned:
        bcD = np.zeros(p_size)
    elif not pinned:
        bcD = np.zeros(p_size+1)


    uB, uL, uR, uT = qbc["uB"], qbc["uL"], qbc["uR"], qbc["uT"]
    vB, vL, vR, vT = qbc["vB"], qbc["vL"], qbc["vR"], qbc["vT"]


    # Bottom
    for j in [0]:
        for i in range(1, nx-1):
            bcD[p[i,j]] = - vB[i]/dy
    # Bottom Right
    for j in [0]:
        for i in [nx-1]:
            bcD[p[i,j]] = uR[j]/dx - vB[i]/dy
    # Left Wall
    for j in range(1,ny-1):
        for i in [0]:
            bcD[p[i,j]] = - uL[j]/dx
    # Right Wall
    for j in range(1, ny-1):
        for i in [nx-1]:
```

```
            bcD[p[i,j]] = uR[j]/dx

    # Top Wall
    for j in [ny-1]:
        for i in range(1,nx-1):
            bcD[p[i,j]] = vT[i]/dy
    # Top Left Corner
    for j in [ny-1]:
        for i in [0]:
            bcD[p[i,j]] = -uL[j]/dx + vT[i]/dy
    # Top Right Corner
    for j in [ny-1]:
        for i in [nx-1]:
            bcD[p[i,j]] = uR[j]/dx + vT[i]/dy
    # Interior Points (Zeroed to match q dimensions
    #for j in range(1,ny-1):
    #    for i in range(1,nx-1):
    #        bcD[p[i,j]] = 0

    return bcD

def laplace(q, u, v, p, dx, dy, nx, ny, q_size, pinned=True):

    Lq = np.zeros(q_size)

    # NOTE: coeff. = 3 are for ghost cell terms (e.g. (2*uBC - 3*u[i,1] + u[i,2]) / dy^2
    # U-COMPONENT
    # Bottom Row
    for j in [0]:
        for i in [0]:
            Lq[u[i,j]] = ( q[u[i+1,j]] - 2*q[u[i,j]]                   ) / dx**2 \
                       + ( q[u[i,j+1]] - 2*q[u[i,j]]                   ) / dy**2
                       #                              + q[u[i-1,j]]   / dx**2
                       #                              + q[u[i,j-1]]   / dy**2
        for i in range(1,nx-2):
            Lq[u[i,j]] = ( q[u[i+1,j]] - 2*q[u[i,j]] + q[u[i-1,j]] ) / dx**2 \
                       + ( q[u[i,j+1]] - 2*q[u[i,j]]                   ) / dy**2
                       #                              + q[u[i,j-1]]   / dy**2
        for i in [nx-2]:
            Lq[u[i,j]] = (             - 2*q[u[i,j]] + q[u[i-1,j]] ) / dx**2 \
                       + ( q[u[i,j+1]] - 2*q[u[i,j]]                   ) / dy**2
                       #   q[u[i+1,j]]                                / dx**2
                       #                              + q[u[i,j-1]]   / dy**2
    # Top Row
    for j in [ny-1]:
        for i in [0]:
            Lq[u[i,j]] = ( q[u[i+1,j]] - 2*q[u[i,j]]                   ) / dx**2 \
                       + (             - 2*q[u[i,j]] + q[u[i,j-1]] ) / dy**2
                       #                              + q[u[i-1,j]]   / dx**2
                       #   q[u[i,j+1]]                                / dy**2
        for i in range(1,nx-2):
            Lq[u[i,j]] = ( q[u[i+1,j]] - 2*q[u[i,j]] + q[u[i-1,j]] ) / dx**2 \
                       + (             - 2*q[u[i,j]] + q[u[i,j-1]] ) / dy**2
                       #   q[u[i,j+1]]                                / dy**2
        for i in [nx-2]:
            Lq[u[i,j]] = (             - 2*q[u[i,j]] + q[u[i-1,j]] ) / dx**2 \
                       + (             - 2*q[u[i,j]] + q[u[i,j-1]] ) / dy**2
                       #   q[u[i+1,j]]                                / dx**2
                       #   q[u[i,j+1]]                                / dy**2

    # Interior Points
    for j in range(1,ny-1):
        for i in [0]:
            Lq[u[i,j]] = ( q[u[i+1,j]] - 2*q[u[i,j]]                   ) / dx**2 \
                       + ( q[u[i,j+1]] - 2*q[u[i,j]] + q[u[i,j-1]] ) / dy**2
                       #                              + q[u[i-1,j]]   / dx**2
        for i in range(1,nx-2):
            Lq[u[i,j]] = ( q[u[i+1,j]] - 2*q[u[i,j]] + q[u[i-1,j]] ) / dx**2 \
```

```
                     + ( q[u[i,j+1]] - 2*q[u[i,j]] + q[u[i,j-1]] ) / dy**2
        for i in [nx-2]:
            Lq[u[i,j]] = (              - 2*q[u[i,j]] + q[u[i-1,j]] ) / dx**2 \
                     + ( q[u[i,j+1]] - 2*q[u[i,j]] + q[u[i,j-1]] ) / dy**2
                     #   q[u[i+1,j]]                                / dx**2

    # V-COMPONENT

    # Bottom Row
    for j in [0]:
        for i in [0]:
            Lq[v[i,j]] = ( q[v[i+1,j]] - 2*q[v[i,j]]                ) / dx**2 \
                     + ( q[v[i,j+1]] - 2*q[v[i,j]]                ) / dy**2
                     #                              + q[v[i-1,j]]   / dx**2
                     #                              + q[v[i,j-1]]   / dy**2
        for i in range(1,nx-1):
            Lq[v[i,j]] = ( q[v[i+1,j]] - 2*q[v[i,j]] + q[v[i-1,j]] ) / dx**2 \
                     + ( q[v[i,j+1]] - 2*q[v[i,j]]                ) / dy**2
                     #                              + q[v[i,j-1]]   / dy**2
        for i in [nx-1]:
            Lq[v[i,j]] = (              - 2*q[v[i,j]] + q[v[i-1,j]] ) / dx**2 \
                     + ( q[v[i,j+1]] - 2*q[v[i,j]]                ) / dy**2
                     #   q[v[i+1,j]]                                / dx**2
                     #                              + q[v[i,j-1]]   / dy**2
    # Top Row
    for j in [ny-2]:
        for i in [0]:
            Lq[v[i,j]] = ( q[v[i+1,j]] - 2*q[v[i,j]]                ) / dx**2 \
                     + (              - 2*q[v[i,j]] + q[v[i,j-1]] ) / dy**2
                     #                              + q[v[i-1,j]]   / dx**2
                     #   q[v[i,j+1]]                                / dy**2
        for i in range(1,nx-1):
            Lq[v[i,j]] = ( q[v[i+1,j]] - 2*q[v[i,j]] + q[v[i-1,j]] ) / dx**2 \
                     + (              - 2*q[v[i,j]] + q[v[i,j-1]] ) / dy**2
                     #   q[v[i,j+1]]                                / dy**2
        for i in [nx-1]:
            Lq[v[i,j]] = (              - 2*q[v[i,j]] + q[v[i-1,j]] ) / dx**2 \
                     + (              - 2*q[v[i,j]] + q[v[i,j-1]] ) / dy**2
                     #   q[v[i+1,j]]                                / dx**2
                     #   q[v[i,j+1]]                                / dy**2
    # Interior Points
    for j in range(1,ny-2):
        for i in [0]:
            Lq[v[i,j]] = ( q[v[i+1,j]] - 2*q[v[i,j]]                ) / dx**2 \
                     + ( q[v[i,j+1]] - 2*q[v[i,j]] + q[v[i,j-1]] ) / dy**2
                     #                              + q[v[i-1,j]]   / dx**2
        for i in range(1,nx-1):
            Lq[v[i,j]] = ( q[v[i+1,j]] - 2*q[v[i,j]] + q[v[i-1,j]] ) / dx**2 \
                     + ( q[v[i,j+1]] - 2*q[v[i,j]] + q[v[i,j-1]] ) / dy**2
        for i in [nx-1]:
            Lq[v[i,j]] = (              - 2*q[v[i,j]] + q[v[i-1,j]] ) / dx**2 \
                     + ( q[v[i,j+1]] - 2*q[v[i,j]] + q[v[i,j-1]] ) / dy**2
                     #   q[v[i+1,j]]                                / dx**2


    return Lq

def bclap(q, qbc, u, v, p, dx, dy, nx, ny, q_size, pinned=True):

    bcL = np.zeros(q_size)

    uB, uL, uR, uT = qbc["uB"], qbc["uL"], qbc["uR"], qbc["uT"]
    vB, vL, vR, vT = qbc["vB"], qbc["vL"], qbc["vR"], qbc["vT"]

    # U-COMPONENT

    # Bottom Row
    for j in [0]:
```

```
        # BC + Ghost Cell
        for i in [0]:

            uB_ghost2 = (2*uB[i] - q[u[i,j]]) # 2-pt. stencil
            uB_ghost3 = (8*uB[i] - 6*q[u[i,j]] + q[u[i,j+1]]) / 3. # 3-pt. stencil
            uB_ghost4 = (16*uB[i] - 15*q[u[i,j]] + 5*q[u[i,j+1]] - q[u[i,j+2]]) / 5. # 4-pt
. stencil

            bcL[u[i,j]] = uL[j] / dx**2 +  uB_ghost4 / dy**2

        # Ghost Cell
        for i in range(1,nx-2):

            uB_ghost2 = (2*uB[i] - q[u[i,j]]) # 2-pt. stencil
            uB_ghost3 = (8*uB[i] - 6*q[u[i,j]] + q[u[i,j+1]]) / 3. # 3-pt. stencil
            uB_ghost4 = (16*uB[i] - 15*q[u[i,j]] + 5*q[u[i,j+1]] - q[u[i,j+2]]) / 5. # 4-pt
. stencil

            bcL[u[i,j]] = uB_ghost4 / dy**2

        # BC + Ghost Cell
        for i in [nx-2]:

            uB_ghost2 = (2*uB[i] - q[u[i,j]]) # 2-pt. stencil
            uB_ghost3 = (8*uB[i] - 6*q[u[i,j]] + q[u[i,j+1]]) / 3. # 3-pt. stencil
            uB_ghost4 = (16*uB[i] - 15*q[u[i,j]] + 5*q[u[i,j+1]] - q[u[i,j+2]]) / 5. # 4-pt
. stencil

            bcL[u[i,j]] = uR[j] / dx**2 + uB_ghost4 / dy**2

    # Top Row
    for j in [ny-1]:
        # BC + Ghost Cell
        for i in [0]:

            uT_ghost2 = (2*uT[i] - q[u[i,j]]) # 2-pt. stencil
            uT_ghost3 = (8*uT[i] - 6*q[u[i,j]] + q[u[i,j-1]]) / 3. # 3-pt. stencil
            uT_ghost4 = (16*uT[i] - 15*q[u[i,j]] + 5*q[u[i,j-1]] - q[u[i,j-2]]) / 5. # 4-pt
. stencil

            bcL[u[i,j]] = uL[j] / dx**2 + uT_ghost4 / dy**2
        # Ghost Cell
        for i in range(1,nx-2):

            uT_ghost2 = (2*uT[i] - q[u[i,j]]) # 2-pt. stencil
            uT_ghost3 = (8*uT[i] - 6*q[u[i,j]] + q[u[i,j-1]]) / 3. # 3-pt. stencil
            uT_ghost4 = (16*uT[i] - 15*q[u[i,j]] + 5*q[u[i,j-1]] - q[u[i,j-2]]) / 5. # 4-pt
. stencil

            bcL[u[i,j]] = uT_ghost4 / dy**2
        # BC + Ghost Cell
        for i in [nx-2]:

            uT_ghost2 = (2*uT[i] - q[u[i,j]]) # 2-pt. stencil
            uT_ghost3 = (8*uT[i] - 6*q[u[i,j]] + q[u[i,j-1]]) / 3. # 3-pt. stencil
            uT_ghost4 = (16*uT[i] - 15*q[u[i,j]] + 5*q[u[i,j-1]] - q[u[i,j-2]]) / 5. # 4-pt
. stencil

            bcL[u[i,j]] = uR[j] / dx**2 + uT_ghost4 / dy**2

    # Interior Nodes (DONE)
    for j in range(1,ny-1):
        # BC
        for i in [0]:
            bcL[u[i,j]] = uL[j] / dx**2;
        for i in range(1,nx-2):
            bcL[u[i,j]] = 0
        # BC
```

```
        for i in [nx-2]:
            bcL[u[i,j]] = uR[j] / dx**2;

    # V-COMPONENT

    # Bottom Row
    for j in [0]:
        # BC + Ghost Cell
        for i in [0]:

            vL_ghost2 = (2*vL[j] - q[v[i,j]]) # 2-pt. stencil
            vL_ghost3 = (8*vL[j] - 6*q[v[i,j]] + q[v[i+1,j]]) / 3. # 3-pt. stencil
            vL_ghost4 = (16*vL[j] - 15*q[v[i,j]] + 5*q[v[i+1,j]] - q[v[i+2,j]]) / 5. # 4-pt
. stencil

            bcL[v[i,j]] = vL_ghost4 / dx**2 + vB[i] / dy**2;
        # BC
        for i in range(1,nx-1):
            bcL[v[i,j]] = vB[i] / dy**2;
        # BC + Ghost Cell
        for i in [nx-1]:

            vR_ghost2 = (2*vR[j] - q[v[i,j]]) # 2-pt. stencil
            vR_ghost3 = (8*vR[j] - 6*q[v[i,j]] + q[v[i-1,j]]) / 3. # 3-pt. stencil
            vR_ghost4 = (16*vR[j] - 15*q[v[i,j]] + 5*q[v[i-1,j]] - q[v[i-2,j]]) / 5. # 4-pt
. stencil

            bcL[v[i,j]] = vR_ghost4 / dx**2 + vB[i] / dy**2;

    # Top Row
    for j in [ny-2]:
        # BC + Ghost Cell
        for i in [0]:

            vL_ghost2 = (2*vL[j] - q[v[i,j]]) # 2-pt. stencil
            vL_ghost3 = (8*vL[j] - 6*q[v[i,j]] + q[v[i+1,j]]) / 3. # 3-pt. stencil
            vL_ghost4 = (16*vL[j] - 15*q[v[i,j]] + 5*q[v[i+1,j]] - q[v[i+2,j]]) / 5. # 4-pt
. stencil

            bcL[v[i,j]] = vL_ghost4  / dx**2 + vT[i] / dy**2;
        # BC
        for i in range(1,nx-1):
            bcL[v[i,j]] = vT[i] / dy**2
        # BC + Ghost Cell
        for i in [nx-1]:

            vR_ghost2 = (2*vR[j] - q[v[i,j]]) # 2-pt. stencil
            vR_ghost3 = (8*vR[j] - 6*q[v[i,j]] + q[v[i-1,j]]) / 3. # 3-pt. stencil
            vR_ghost4 = (16*vR[j] - 15*q[v[i,j]] + 5*q[v[i-1,j]] - q[v[i-2,j]]) / 5. # 4-pt
. stencil

            bcL[v[i,j]] = vR_ghost4  / dx**2 + vT[i] / dy**2;

    # Interior Nodes
    for j in range(1,ny-2):
        # Ghost Cell
        for i in [0]:

            vL_ghost2 = (2*vL[j] - q[v[i,j]]) # 2-pt. stencil
            vL_ghost3 = (8*vL[j] - 6*q[v[i,j]] + q[v[i+1,j]]) / 3. # 3-pt. stencil
            vL_ghost4 = (16*vL[j] - 15*q[v[i,j]] + 5*q[v[i+1,j]] - q[v[i+2,j]]) / 5. # 4-pt
. stencil

            bcL[v[i,j]] =  vL_ghost4 / dx**2;

        for i in range(1,nx-1):
            bcL[v[i,j]] =  0
        # Ghost Cell
```

```
        for i in [nx-1]:

            vR_ghost2 = (2*vR[j] - q[v[i,j]]) # 2-pt. stencil
            vR_ghost3 = (8*vR[j] - 6*q[v[i,j]] + q[v[i-1,j]]) / 3. # 3-pt. stencil
            vR_ghost4 = (16*vR[j] - 15*q[v[i,j]] + 5*q[v[i-1,j]] - q[v[i-2,j]]) / 5. # 4-pt
. stencil

            bcL[v[i,j]] =  vR_ghost4 / dx**2;

    return bcL

def adv(q, qbc, u, v, p, dx, dy, nx, ny, q_size, pinned=True):

    advq = np.zeros(q_size)

    uB, uL, uR, uT = qbc["uB"], qbc["uL"], qbc["uR"], qbc["uT"]
    vB, vL, vR, vT = qbc["vB"], qbc["vL"], qbc["vR"], qbc["vT"]

    # Nx(i,j) -> u
    # Interpolation Operations, _uy_vx (cell vertices) and _ux_ux (cell centers)
    # Difference Operations, del_x, del_y
    for j in range(0, ny):
        for i in range(0, nx-1): # Interior

            if i == 0: # Left Wall
                _ux_ux_ = -(0.5*(uL[j]       + q[u[i,j]]))**2  \
                        +  (0.5*(q[u[i,j]]   + q[u[i+1,j]]))**2
            elif i == nx-2: # Right Wall
                _ux_ux_ = -(0.5*(q[u[i-1,j]] + q[u[i,j]]))**2  \
                        +  (0.5*(q[u[i,j]]   + uR[j]))**2
            else: # Interior
                _ux_ux_ = -(0.5*(q[u[i-1,j]] + q[u[i,j]]))**2  \
                        +  (0.5*(q[u[i,j]]   + q[u[i+1,j]]))**2

            if j == 0: # Bottom Wall

                uB_ghost2 = 2*uB[i] - q[u[i,j]] # 2-pt stencil
                uB_ghost3 = (8*uB[i] - 6*q[u[i,j]] + q[u[i,j+1]]) / 3. # 3-pt stencil
                uB_ghost4 = (16*uB[i] - 15*q[u[i,j]] + 5*q[u[i,j+1]] - q[u[i,j+2]]) / 5. #
4-pt stencil

                _vx_uy_ = -0.5*(vB[i] + vB[i+1])             * 0.5*(uB_ghost4   + q[u[i,j]]
) \
                        +  0.5*(q[v[i,j]] + q[v[i+1,j]])     * 0.5*(q[u[i,j]]   + q[u[i,j+1
]])

            elif j == ny-1: # Top Wall

                uT_ghost2 = 2*uT[i] - q[u[i,j]] # 2-pt stencil
                uT_ghost3 = (8*uT[i] - 6*q[u[i,j]] + q[u[i,j-1]]) / 3. # 3-pt stencil
                uT_ghost4 = (16*uT[i] - 15*q[u[i,j]] + 5*q[u[i,j-1]] - q[u[i,j-2]]) / 5. #
4-pt stencil

                _vx_uy_ = -0.5*(q[v[i,j-1]] + q[v[i+1,j-1]]) * 0.5*(q[u[i,j-1]] + q[u[i,j]]
) \
                        +  0.5*(vT[i] + vT[i+1])             * 0.5*(q[u[i,j]]   + uT_ghost4
)

            else: # Interior
                _vx_uy_ = -0.5*(q[v[i,j-1]] + q[v[i+1,j-1]]) * 0.5*(q[u[i,j-1]] + q[u[i,j]]
) \
                        +  0.5*(q[v[i,j]]   + q[v[i+1,j]])   * 0.5*(q[u[i,j]]   + q[u[i,j+1
]])

            del_y_vx_uy = _vx_uy_ / dy
            del_x_ux_ux = _ux_ux_ / dx

            advq[u[i,j]] = del_x_ux_ux + del_y_vx_uy
```

```
    # Ny(i,j) -> v
    # Interpolation Operations, _uy_vx (cell vertices) and _vy_vy (cell centers)
    for j in range(0, ny-1):
        for i in range(0, nx):

            if i == 0: # Left Wall

                vL_ghost2 = 2*vL[j] - q[v[i,j]] # 2-pt stencil
                vL_ghost3 = (8*vL[j] - 6*q[v[i,j]] + q[v[i+1,j]]) / 3. # 3-pt stencil
                vL_ghost4 = (16*vL[j] - 15*q[v[i,j]] + 5*q[v[i+1,j]] - q[v[i+2,j]]) / 5. #
4-pt stencil

                _uy_vx_ = -0.5*(uL[j]       + uL[j+1])        * 0.5*(vL_ghost4 + q[v[i,j]])
\
                        +  0.5*(q[u[i,j]]   + q[u[i,j+1]])    * 0.5*(q[v[i,j]]   + q[v[i+1,j
]])

            elif i == nx-1: # Right Wall

                vR_ghost2 = 2*vR[j] - q[v[i,j]] # 2-pt stencil
                vR_ghost3 = (8*vR[j] - 6*q[v[i,j]] + q[v[i-1,j]]) / 3. # 3-pt stencil
                vR_ghost4 = (16*vR[j] - 15*q[v[i,j]] + 5*q[v[i-1,j]] - q[v[i-2,j]]) / 5. #
4-pt stencil

                _uy_vx_ = -0.5*(q[u[i-1,j]] + q[u[i-1,j+1]]) * 0.5*(q[v[i-1,j]] + q[v[i,j]]
) \
                        +  0.5*(uR[j] + uR[j+1])             * 0.5*(q[v[i,j]]   + vR_ghost4
)

            else:
                _uy_vx_ = -0.5*(q[u[i-1,j]] + q[u[i-1,j+1]]) * 0.5*(q[v[i-1,j]] + q[v[i,j]]
) \
                        +  0.5*(q[u[i,j]]   + q[u[i,j+1]])   * 0.5*(q[v[i,j]]   + q[v[i+1,j
]])

            if j == 0: # Bottom Wall
                _vy_vy_ = -(0.5*(vB[i]       + q[v[i,j]]))**2  \
                        +  (0.5*(q[v[i,j]]   + q[v[i,j+1]]))**2
            elif j == ny-2: # Top Wall
                _vy_vy_ = -(0.5*(q[v[i,j-1]] + q[v[i,j]]))**2  \
                        +  (0.5*(q[v[i,j]]   + vT[i]))**2
            else: # Interior
                _vy_vy_ = -(0.5*(q[v[i,j-1]] + q[v[i,j]]))**2  \
                        +  (0.5*(q[v[i,j]]   + q[v[i,j+1]]))**2

            del_x_uy_vx = _uy_vx_ / dx
            del_y_vy_vy = _vy_vy_ / dy

            advq[v[i,j]] = del_x_uy_vx + del_y_vy_vy

    return advq

def S(q, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=True):

    Lq = laplace(q, u, v, p, dx, dy, nx, ny, q_size, pinned=False)
    a = alpha*nu*dt
    I = np.ones(Lq.shape)
    Sq = np.add(q, np.multiply(a, Lq))

    return Sq

def R(q, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=True):

    Lq = laplace(q, u, v, p, dx, dy, nx, ny, q_size, pinned=False)
    a = alpha*nu*dt
    I = np.ones(Lq.shape)
```

```
    Rq = np.subtract(q, np.multiply(a, Lq))

    return Rq

def Rinv(q, u, v, p, dx, dy, nx, ny, q_size, alpha, nu, dt, pinned=True):

    Lq = laplace(q, u, v, p, dx, dy, nx, ny, q_size, pinned=False)
    Lq2 = laplace(Lq, u, v, p, dx, dy, nx, ny, q_size, pinned=False)
    a = alpha*nu*dt
    a2 = a**2
    I = np.ones(Lq.shape)

    # Taylor Series Expansion
    term1 = np.multiply(I, q)
    term2 = np.multiply(a, Lq)
    term3 = np.multiply(a2, Lq2)
    Rinvq = np.add(np.add(term1, term2), term3)

    return Rinvq
```

```
"""
Created on May 27 2021
@author S. M. Parmar
Various visualization routines for
verification and flow visualization.
"""
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib.ticker as ticker
import numpy as np
import pandas as pd

def plotL2vsGridSize(linReg, dxdy, error, outFile, oprtr, save=False):
    """
    INPUTS:
    ------
    linReg - linear regression data from linregress function
    dxdy   - array of spatial grid sizes (x-axis)
    error  - array of error values (y-axis)
    oprtr  - string value name of the operator being tested (for title)
    outFile- name of output file for figure
    """
    figFilePath = "./Figures/"

    plt.rc('text', usetex=True)
    plt.rc('font', family='serif')
    plt.rc('xtick',labelsize=16)
    plt.rc('ytick',labelsize=16)
    plt.rc('grid', c='0.5', ls='-', alpha=0.5, lw=0.5)

    fig = plt.figure(figsize=(8,6))

    ax = fig.add_subplot(1,1,1)
    #ax.set_xlabel(r'$\Delta$ $t$', fontsize=16)
    ax.set_xlabel(r'$\Delta$ $x$, $\Delta$ $y$', fontsize=16)
    ax.set_ylabel(r'$L^{\infty}$ Norm, $||x||_{\infty}$', fontsize=16)
    #ax.set_title(r"Temporal Convergence", fontsize=20)
    ax.set_title(r"Spatial Convergence of " + oprtr + " Operator", fontsize=20)
    ax.annotate(r"Log-Log Slope = $%.2f$" % (linReg.slope),
            xy=(0.75, 0.05),
            xycoords="axes fraction",
            size=16,
            ha='center',
            va='center',
            bbox=dict(boxstyle="round4", fc="aqua", ec="k", alpha=0.7))

    plt.loglog(dxdy, error, 'bo', mfc="none", markersize=8, label=oprtr + ' Operator Tests'
)
    plt.loglog(dxdy, 10**(linReg.slope*np.log10(dxdy)+linReg.intercept), '-r', label='Fitte
d Line',linewidth=2)
    plt.legend(prop={'size':14})
    plt.grid(True, which="both")
    if save:
        plt.savefig(figFilePath + outFile.split('.')[0])
    plt.show()

    return

def plotVelocity(q, qBC, xu, xv, yu, yv, nx, ny, time, Re, drawNow, strmOn = True, quiverOn
 = False, save=True):


    figFilePath = "./Figures/"
    subDir = "Re" + str(Re) + "/"


    u = q[0:ny*(nx-1)]
    v = q[ny*(nx-1):]
```

```
    if (len(u) != ny*(nx-1)) or (len(v) != nx*(ny-1)):
        raise("Velocity Components have inccorect length")

    U = np.reshape(u, (ny, nx-1))
    V = np.reshape(v, (ny-1, nx))

    U_vert = 0.5*(U[0:-1,:] + U[1:,:])
    #U_vert = U_vert.T
    V_vert = 0.5*(V[:,0:-1] + V[:,1:])
    #V_vert = V_vert.T
    X, Y = np.meshgrid(xu, yv)

    # Geometric Center (x = 0.5, y = 0.5)
    u_ce = U[:, int((nx-1)/2)]
    v_ce = V[int((ny-1)/2), :]

    # Read in Ghia Data for Validation
    df = pd.read_csv('Ghia1982_uData.csv', dtype='float')
    uGhia = df.to_dict(orient='list')
    df = pd.read_csv('Ghia1982_vData.csv', dtype='float')
    vGhia = df.to_dict(orient='list')

    u_ce_Ghia = uGhia[str(Re)]
    y_ce_Ghia = uGhia['y']

    v_ce_Ghia = vGhia[str(Re)]
    x_ce_Ghia = vGhia['x']

    plt.rc('text', usetex=True)
    plt.rc('font', family='serif')
    plt.rc('xtick',labelsize=16)
    plt.rc('ytick',labelsize=16)

    # ---------- Velocity Profiles 1D ------------------------------

    fig1 = plt.figure(figsize=(8,6))
    ax1 = fig1.add_subplot(1,1,1)
    plt.scatter(yu, u_ce, marker='o', c='b', label='Parmar 2021 (Re = '+str(Re) + ')')
    plt.scatter(y_ce_Ghia, u_ce_Ghia, marker='s', c='r', label='Ghia 1982 (Re = '+str(Re) +
 ')')
    ax1.set_xlabel(r'$y$ position @ $x = 0.5$', fontsize=16)
    ax1.set_ylabel(r'$u$ velocity', fontsize=16)
    plt.legend(prop={"size":14})
    ax1.set_title(r"$u$ Velocity Profile along $x = 0.5$ at t = {:.3f}".format(time), fonts
ize=20)
    plt.savefig(figFilePath + subDir \
            + "t_{:.3f}_".format(time).replace('.','p') \
            + "Re_" + str(Re) \
            + "dx_{:.3f}".format(xu[1]-xu[0]).replace('.','p')\
            + '_uVALIDATION')

    fig2 = plt.figure(figsize=(8,6))
    ax2 = fig2.add_subplot(1,1,1)
    plt.scatter(xv, v_ce, marker='o', c='b', label='Parmar 2021 (Re = '+str(Re) + ')')
    plt.scatter(x_ce_Ghia, v_ce_Ghia, marker='s', c='r', label='Ghia 1982 (Re = '+str(Re) +
 ')')
    ax2.set_title(r"$v$ Velocity Profile along $y = 0.5$ at t = {:.3f}".format(time), fonts
ize=20)
    ax2.set_xlabel(r'$x$ position @ $y = 0.5$', fontsize=16)
    ax2.set_ylabel(r'$v$ velocity', fontsize=16)
    plt.legend(prop={"size":14})
    plt.savefig(figFilePath + subDir \
            + "t_{:.3f}_".format(time).replace('.','p') \
            + "Re_" + str(Re) \
            + "dx_{:.3f}".format(xu[1]-xu[0]).replace('.','p')\
            + '_vVALIDATION')

    # ---------- Velocity Profiles 2D ------------------------------
```

```
    fig3 = plt.figure(figsize=(8,6))
    ax3 = fig3.add_subplot(1,1,1)
    ax3.set_xlim([0, 1])
    ax3.set_ylim([0, 1])
    ax3.set_xlabel(r'$X$', fontsize=16)
    ax3.set_ylabel(r'$Y$', fontsize=16)
    ax3.set_title(r"Velocity Profile at t = {:.3f}".format(time), fontsize=20)

    levels = np.linspace(0,1,1000)
    cntrf = ax3.contourf(X, Y, np.sqrt(U_vert**2 + V_vert**2), levels=levels, cmap=cm.virid
is)
    cbar = plt.colorbar(cntrf, format='%.2f')
    cbar.set_label('Velocity Magnitude', fontsize=14)
    cbar.ax.tick_params(labelsize=14)

    if quiverOn:
        quiv = plt.quiver(X, Y, U_vert, V_vert, color='white')

    # ---------- Streamplots 2D ------------------------------
    if strmOn:
        strm = plt.streamplot(X, Y, U_vert, V_vert, color='white', linewidth=.5)
    if save:
        plt.savefig(figFilePath + subDir \
                + "t_{:.3f}_".format(time).replace('.','p') \
                + "Re_" + str(Re) \
                + "dx_{:.3f}".format(xu[1]-xu[0]).replace('.','p'))
    if drawNow:
        plt.show()
    vorticity = True
    if vorticity:
        w = (V[:,1:] - V[:,0:-1])/(xu[1]-xu[0]) - (U[1:, :] - U[0:-1, :])/(yv[1]-yv[0])
        fig_vorti = plt.figure()
        vort = plt.contour(X, Y, w)
        plt.show()
```