```python
# -*- coding: utf-8 -*-
"""
Created on Sat Aug 11 15:15:23 2018

@author: Edgar
"""
"""
Full AC Code. Comprises counter-clockwise and clockwise rotation, both inviscid
and viscous cases, four different airfoils and the Cheng's modification. The
flow curvature effect is available, although it is not guaranteed for viscous
airfoils. Includes data for DU06W200.
"""
"""MODULES AND ROUTINES FROM SCIPY AND NUMPY"""
from scipy.interpolate import interp1d
from scipy.interpolate import interp2d
from scipy.integrate import simps
from scipy.integrate import quad
import numpy as np
from math import pi
from math import sin
from math import cos
from math import atan
from math import sqrt
import time

"""GLOBAL GEOMETRIC AND OPERATIONAL CONSTANTS """
N = int(36)   #.........................................number of control points
DT = float(2.0*pi/N)   #..............................angular slice in radians
F = float(1.01)   #................................coordinates offset factor
TOL = float(0.01)   #.......................................relative tolerance
BETA = float(0.25)   #.........................relaxation factor for iterations
NU = float(1.4657e-5)   #...........................air's kinematic viscosity

"""USER'S INPUT FOR EITHER INVISCID OR VISCOUS CASE"""
vis = bool(input("INVISCID [0], VISCOUS [1]:..........."))
rot = int(input("COUNTER-CLOCKWISE [1], CLOCKWISE [0]:."))
flo = int(input("FLOW CURVATURE YES [1], NO [0]:......."))
if rot == 1: SGN = -1
if rot == 0: SGN = +1
if vis == False:
    lam = float(input("TIP-SPEED RATIO:....................."))
    sol = float(input("TURBINE'S SOLIDITY:.................."))
    bla = int(input("NUMBER OF BLADES:....................."))
    reg = 0.00
else:
    lam = float(input("TIP-SPEED RATIO:....................."))
    bla = int(input("NUMBER OF BLADES:....................."))
    cor = float(input("BLADE'S CHORD:......................."))
    rad = float(input("TURBINE'S RADIUS:...................."))
    ome = float(input("REVOLUTIONS PER MINUTE:.............."))
    wng = int(input("0012[1] 0015[2] 0018[3] DU06W200[4]..."))
    sol = (bla*cor)/(2.*rad)   #............................turbine's solidity
    reg = ome*rad*cor*pi/(30.*NU)   #...................global reynolds number

"""COMPUTATIONS BEGIN"""
start_time = time.clock()
```

```python
"""READ LIFT AND DRAG FROM FILES ACCORDING TO CROSS SECTION"""
if vis == True:
    if wng != 4:
        aa = np.loadtxt('naca-aa.csv', unpack=True)
        re = np.loadtxt('naca-re.csv', unpack=True)
    else:
        aa = np.loadtxt('du06w200aa.csv', unpack=True)
        re = np.loadtxt('du06w200re.csv', unpack=True)
    if wng == 1:
        clfilename = 'naca0012cl.csv'
        cdfilename = 'naca0012cd.csv'
    if wng == 2:
        clfilename = 'naca0015cl.csv'
        cdfilename = 'naca0015cd.csv'
    if wng == 3:
        clfilename = 'naca0018cl.csv'
        cdfilename = 'naca0018cd.csv'
    if wng == 4:
        clfilename = 'du06w200cl.csv'
        cdfilename = 'du06w200cd.csv'

    """CREATE LOOK-UP TABLES AND INTERPOLATOR FUNCTIONS"""
    lift = np.loadtxt(clfilename, delimiter=',')
    drag = np.loadtxt(cdfilename, delimiter=',')
    fcl = interp2d(re, aa, lift, kind='cubic')   #.......smooth function for cl
    fcd = interp2d(re, aa, drag, kind='cubic')   #.......smooth function for cd

    """VECTORIZE INTERPOLATOR FUNCTIONS FCL AND FCD"""
    veccl = np.vectorize(fcl)
    veccd = np.vectorize(fcd)

def vecc_inviscid(rey, ang):
    """RETURNS A TUPLE OF CL AND CD NDARRAYS"""
    return 2.*pi*1.11*np.sin(ang), np.zeros((N,))

def vecc_viscous(rey, ang):
    """RETURNS A TUPLE OF CL AND CD NDARRAYS"""
    ang = np.degrees(ang)
    return veccl(rey, ang), veccd(rey, ang)

"""CHOOSE LIFT AND DRAG POINTER-TO-FUNCTION ACCORDING TO CASE"""
if vis == False:
    fc = vecc_inviscid   #.......................pointer to function look-alike
if vis == True:
    fc = vecc_viscous   #........................pointer to function look-alike

"""CALCULATED PARAMETERS"""
ctr = 2.0*sol/float(bla)   #.............................chord-to-radius ratio
dme = ctr/2.0   #...............distance from aerodynamic center to eval. point
lag = atan(dme/1.0)   #..lag between azimuthal position and evaluation position

"""COORDINATES AND AZIMUTH ANGLE NDARRAY"""
t = np.fromfunction(lambda j:(1./2.+j)*DT, (N,), dtype = float)
t_fine = np.linspace(t[0], t[N-1], 10*N)
x = -F*np.sin(t)   #............................non-dimensional 'x' coordinate
y = +F*np.cos(t)   #............................non-dimensional 'y' coordinate
```

```python
"""INFLUENCE COEFFICIENTS"""
cx = np.zeros((N,N), dtype = float)
cy = np.zeros((N,N), dtype = float)
def return_row(_x, _y):
    """RETURNS ROW OF THE INFLUENCE COEFFICIENT MATRIX"""
    def cx_integrate_sector(_t):
        def fx(phi):
            return (-(_x + sin(phi))*sin(phi) + (_y - cos(phi))*cos(phi))\
            /((_x + sin(phi))**2 + (_y - cos(phi))**2)
        return quad(fx, _t - DT/2., _t + DT/2.)
    def cy_integrate_sector(_t):
        def fy(phi):
            return (-(_x + sin(phi))*cos(phi) - (_y - cos(phi))*sin(phi))\
            /((_x + sin(phi))**2 + (_y - cos(phi))**2)
        return quad(fy, _t - DT/2., _t + DT/2.)
    vecintx = np.vectorize(cx_integrate_sector)
    vecinty = np.vectorize(cy_integrate_sector)
    rowcx, erorx = vecintx(t)
    rowcy, erory = vecinty(t)
    return rowcx/(-2.*pi), rowcy/(-2.*pi)
"""FILL ROWS OF THE INFLUENCE COEFFICIENTS"""
for j in range(N):
    cx[j, :], cy[j, :] = return_row(x[j], y[j])

"""WAKE MATRICES FOR NORMAL AND TANGENTIAL LOADS"""
wkn, wkt = np.zeros((N,N), dtype = float), np.zeros((N,N), dtype = float)
rd = range(N/2, N)   #.......................right and downward index diagonal
ld = [N-j-1 for j in rd]   #..................left and downward index diagonal
wkn[rd, ld] = -1.0   #...................................simple ac wake terms
wkn[rd, rd] = 1.0   #....................................simple ac wake terms
for j in range(N/2+1, N-1):
    cheng_terms = -y[j]/sqrt(1.0-y[j]**2)
    wkt[j, N-j-1] = cheng_terms
    wkt[j, j] = cheng_terms

"""INITIALIZE PERTURBATION VELOCITES TO ZERO"""
wx, wy = np.zeros((N,), dtype = float), np.zeros((N,), dtype = float)

def get_velocity_lag(x_i, X, Y):
    """INTERPOLATES PERTURBATION VELOCITY DUE TO LAG"""
    for i in range(N-1):
        if x_i < X[0]:
            return Y[0]
        if x_i >= X[i] and x_i <= X[i+1]:
            break
    loc = (x_i - X[i])/(X[i+1] - X[i])
    y_i = Y[i] + loc*(Y[i+1] - Y[i])
    return y_i

aoa34 = np.zeros((N,))
def calculate_aoa34():
    """RETURNS ANGLE OF ATTACK NDARRAY AT EVALUATION 3/4 CHORD POINT"""
    for k in range(N):
        wx34 = get_velocity_lag(t[k]+SGN*lag, t, wx)
        wy34 = get_velocity_lag(t[k]+SGN*lag, t, wy)
        vx34 = 1. + wx34
        vy34 = wy34
```

```python
        vn34 = vx34*sin(t[k]) - vy34*cos(t[k]) + lam*sin(lag)
        vt34 = -SGN*vx34*cos(t[k]) - SGN*vy34*sin(t[k]) + lam*cos(lag)
        aoa34[k] = atan(vn34/vt34)

def correct_velocities(qn_, qt_):
    """PERFORMS LINEAR CORRECTION"""
    k3 = 0.0892
    k2 = 0.0544
    k1 = 0.2511
    k0 = -0.0017
    f = qn_*np.sin(t) + qt_*np.cos(t)   #.....................required function
    f_smooth = interp1d(t, f, kind='cubic')   #.............cubic interpolation
    f_vec = np.vectorize(f_smooth)   #...................ready for integration
    cth_ = simps(f_vec(t_fine), t_fine)   #..................thrust coefficient
    a_ = k3*cth_**3 + k2*cth_**2 + k1*cth_ + k0   #............induction factor
    ka_ = 1./(1. - a_)   #....................................correction factor
    return a_, cth_, ka_

def get_power(qt_):
    """RETURNS THE POWER COEFFICIENT FOR EITHER CCW OR CW ROTATION"""
    qt_smooth = interp1d(t, qt_, kind='cubic')   #..........smooth the qt curve
    qt_vec = np.vectorize(qt_smooth)
    val = simps(qt_vec(t_fine), t_fine)
    return SGN*lam*val

def get_ccw_vt(vx_, vy_):
    """TANGENTIAL VELOCITY NDARRAY COUNTER-CLOCKWISE ROTATION"""
    return vx_*np.cos(t) + vy_*np.sin(t) + lam

def get_cw_vt(vx_, vy_):
    """TANGENTIAL VELOCITY NDARRAY CLOCKWISE DIRECTION"""
    return -vx_*np.cos(t) - vy_*np.sin(t) + lam

"""CHOOSE TANGENTIAL VELOCITY FUNCTION ACCORDING TO DIRECTION OF ROTATION"""
if rot == 1: get_vt = get_ccw_vt
if rot == 0: get_vt = get_cw_vt

def get_inviscid_rl(vr_):
    """LOCAL REYNOLDS NUMBER NDARRAY INVISCID CASE"""
    return np.zeros((N,), dtype = float)
def get_viscous_rl(vr_):
    """LOCAL REYNOLDS NUMBER NDARRAY VISCOUS CASE"""
    return reg*vr_/(lam*1e6)

"""CHOOSE LOCAL REYNOLDS NUMBER FUNCTION ACCORDING TO CASE"""
if vis == False: get_rl = get_inviscid_rl
if vis == True: get_rl = get_viscous_rl

"""CONVERGENCE LOOP"""
loops = 0
while True:
    calculate_aoa34()   #..............................fill aoa34 from function
    vx = 1.0 + wx   #....................................vectorized nondim vx
    vy = wy   #..........................................vectorized nondim vy
    vn = vx*np.sin(t) - vy*np.cos(t)   #.................vectorized nondim vn
    vt = get_vt(vx, vy)   #................................vectorized nondim vt
    vr = np.sqrt(vn**2 + vt**2)  #........................#vectorized nondim vr
```

```
    aoa = np.arctan(vn/vt)   #....................................vectorized aoa
    rel = get_rl(vr)   #..............................vectorized local reynolds
    if flo == 1:
        cl, cd = fc(rel, aoa34)   #.........................vectorized cl and cd
    elif flo == 0:
        cl, cd = fc(rel, aoa)
    cn = cl*np.cos(aoa) + cd*np.sin(aoa)   #......................vectorized cn
    ct = cl*np.sin(aoa) - cd*np.cos(aoa)   #......................vectorized ct
    qn = (sol/(2.*pi))*(vr**2)*cn   #.............................vectorized qn
    qt = SGN*(sol/(2.*pi))*(vr**2)*ct   #.........................vectorized qt
    a, cth, ka = correct_velocities(qn, qt)
    """NEW PERTURBATION VELOCITIES"""
    wx_new = (cx + wkn).dot(qn) + (cy + wkt).dot(qt)   #..matrix multiplication
    wy_new = cy.dot(qn) - cx.dot(qt)   #..................matrix multiplication
    wx_new = wx_new*ka   #....................................correction for wx
    wy_new = wy_new*ka   #....................................correction for wy
    if wx_new[N/4] > 0.:   #..............................prevent dual solution
        wx_new = -1*wx_new
        wy_new = -1*wy_new
    """CHECK FOR CONVERGENCE"""
    if np.allclose(wx, wx_new, rtol=TOL) and np.allclose(wy, wy_new, rtol=TOL):
        break
    else:
        wx = BETA*wx_new + (1. - BETA)*wx   #...........under-relaxation factor
        wy = BETA*wy_new + (1. - BETA)*wy   #...........under-relaxation factor
        loops += 1

"""PRINT TO CONSOLE"""
print '\n'
print "Number of loops:     %d" % loops
print "Induction factor:    %4.3f" % a
print "Thrust coefficient: %4.3f" % cth
print "Correction factor:   %4.3f" % ka
print "Power coefficient:   %4.3f" % get_power(qt)
print "Global Reynolds:     %4.3f" % reg
"""EXECUTION TIME"""
print "Execution time:      %4.3f seconds." % (time.clock() - start_time)

"""MULTIPLE SUBPLOTS"""
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
t_axis = np.degrees(t)
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(8,8))
labels = [(r'$w_x$',r'$w_y$'), (r'$Q_n$',r'$Q_t$'), (r'$AOA$',r'$v_{rel}$')]
subs = [(wx, wy), (qn, qt), (np.degrees(aoa), vr)]
colors = ['b', 'g', 'r']
steps = [[0.1, 0.05], [0.05, 0.01], [5.0, 0.5]]
for i in range(3):
    for j in range(2):
        ax = axes[i, j]
        ax.plot(t_axis, subs[i][j], c=colors[i], marker='.')
        ax.set_title(labels[i][j])
        ax.set_xlabel(r'$\theta$')
        ax.set_ylabel(labels[i][j])
        ax.grid()
        ax.set_xticks([k for k in np.linspace(0, 360, 9)])
        ax.yaxis.set_major_locator(ticker.MultipleLocator(steps[i][j]))
```

```
fig.tight_layout()
plt.show()

"""EXPORT TO CSV FILES"""
np.savetxt("cx.csv", cx, delimiter=",", fmt='%6.4f')
np.savetxt("cy.csv", cy, delimiter=",", fmt='%6.4f')
np.savetxt("wkn.csv", wkn, delimiter=",", fmt='%6.4f')
np.savetxt("wkt.csv", wkt, delimiter=",", fmt='%6.4f')
```