In [2]:

```python
import pandas as pd
import os
from enum import Enum
import numpy as np
import scipy
from numpy import sqrt, sin, cos, tan, pi
from scipy.integrate import odeint
from scipy.optimize import fsolve
from scipy.interpolate import InterpolatedUnivariateSpline
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
from matplotlib import ticker, cm
from matplotlib import ticker
from scipy.optimize import minimize
from scipy.optimize import Bounds
import import_ipynb
from cfdPostProcessing import postProcess, rakeProcess;
from teslaModelValidation import pathLine;
%matplotlib

'''Water'''
density = 997
dynamicViscosity = 0.0008891
kinematicViscosity = 8.917*10**-7
TotalMassFlowRate = 1 # 0.5, 2

'''Can be Set'''
voluteThickness = 0.005
discThickness = 0.0008
discSpacing = 0.0002
wallSpace = 0.001
wallDisplacement = 0.003

'''Base Case'''
nDisc = 5
chosenScaleDownFactor = 0.164/0.073
rotorOuter = 0.073
rotorInner = 0.3*rotorOuter
revPerMinute = [2000]

'''formatting for plots'''
formatter = ticker.ScalarFormatter(useMathText=True)
formatter.set_scientific(True)
formatter.set_powerlimits((-1,1))
class flowParameters():
    def __init__(self, innerRadius, outerRadius, discSpacing, discThickness, numberSpacing,
                 voluteThickness, voluteWallSpace, upperClearance, wallDisplacement,
                 totMassFlowRate, density, RPM, k_Width_h0 = 0.798, profileN = 2):
        self.innerRadius = innerRadius
        self.outerRadius = outerRadius
        self.discSpacing = discSpacing
        self.discThickness = discThickness
        self.voluteWallSpace = voluteWallSpace
        self.upperClearance = upperClearance
        self.numberSpacing = numberSpacing
        self.massFlowRate = totMassFlowRate
        self.density = density
        self.Fpo = (profileN + 1)/3

        self.voluteSpace = numberSpacing*discSpacing + (numberSpacing-1)*discThickness
        self.totalVoluteSpace = 2*discThickness + self.voluteSpace + 2*self.voluteWallSpace +
2*wallDisplacement
        self.h0 = k_Width_h0*self.totalVoluteSpace
        self.r0 = self.outerRadius + self.upperClearance + voluteThickness +
self.totalVoluteSpace/2 + \
            cos(np.arcsin((self.totalVoluteSpace - self.h0)/self.h0))*self.h0

        #formula
        self.inletAngle = flowParameters.derivedAngle(self.voluteSpace, self.h0, self.r0)
        self.vRadial, self.vTheta = flowParameters.velocityInlet(self)

        self.omega = RPM*2*pi/60
        self.DH = 2*self.discSpacing
        self.massFlowRatePD = self.massFlowRate/self.numberSpacing
        self.volumeFlowRatePD = self.massFlowRatePD/density

        self.tipVelocity = self.omega*self.outerRadius

        self.relativeTipTangential = (self.vTheta - self.tipVelocity)/self.tipVelocity
        self.relativeTipRadial = self.vRadial/self.tipVelocity

        self.innerOuterRatio = self.innerRadius/self.outerRadius
        self.reynoldM = self.massFlowRatePD/(pi*self.outerRadius*dynamicViscosity)
        self.reynoldMS = self.reynoldM * self.DH / self.outerRadius

    def derivedAngle(vSpace, vIRadius, vRadius):
        degree = np.arctan(2*vSpace*vRadius/(vIRadius**2))
        return 0.5*pi - degree

    def velocityInlet(self):
        effectiveArea = 2*pi*(self.outerRadius+self.upperClearance)*
(self.numberSpacing*self.discSpacing)
        vRadial = self.massFlowRate/(effectiveArea*self.density)
        vTheta = vRadial/tan(self.inletAngle)
        vRadialDisc = vRadial*self.outerRadius/(self.outerRadius-self.upperClearance)
        return -vRadialDisc, vTheta

def bothODE(y,x,instance):
    y0,y1 = y

    nTerm = 3*instance.Fpo - 1 # article definition
    Vr0 = instance.vRadial/instance.tipVelocity

    firstSolution = -(2*nTerm + 1)/(nTerm + 1) + (8*(2*nTerm + 1)*x/instance.reynoldMS - 1/x)*y0
    secondSolution = (4*(nTerm + 1)/(2*nTerm + 1))*(1/x**3)*(Vr0**2 + (y0*x)**2) +\
                     4*y0 + 2*x + 32*(nTerm + 1)*(Vr0**2)/(x*instance.reynoldMS)
    return [firstSolution, secondSolution]

def rotorEff(firstAnswer, rs, instance): # ignore for now
    return (1 - (firstAnswer[-1] + instance.innerOuterRatio)*instance.innerOuterRatio\
            /(firstAnswer[0] + 1))

def power(firstAnswer, rs, instance):
    firstAnswerFlip = np.squeeze(np.flip(firstAnswer))
    rsFlip = np.flip(rs)

    constantTerm = (instance.outerRadius**3)*(2*pi/instance.discSpacing)*\
            (6*dynamicViscosity*instance.tipVelocity)*instance.Fpo
    integrateTerm = firstAnswerFlip*np.power(rsFlip,2)
    return 2*instance.omega*instance.numberSpacing*\
        constantTerm*scipy.integrate.simps(integrateTerm, x = rsFlip)

def efficiencyIdeal(solution, rs, instance):
    innerOuterRatio = instance.innerRadius/ instance.outerRadius
    innerDiscSpeed = innerOuterRatio*instance.tipVelocity
    inletKE = instance.vRadial**2 + instance.vTheta**2
    pressureDrop = abs(solution[-1, 1])*((density*(instance.tipVelocity)**2)/2)

    outletVr = instance.vRadial/innerOuterRatio
    outletVt = (solution[-1, 0]*instance.tipVelocity) + instance.omega*instance.innerRadius
    outletKE = outletVr**2 + outletVt**2
    energyInput = (0.5*(inletKE-outletKE) + pressureDrop/density)

    energyOutput = (instance.tipVelocity**2)*(solution[0]+1) - \
            innerDiscSpeed*(solution[-1]*instance.tipVelocity + innerDiscSpeed)
    return (energyOutput*100/energyInput)[0] # in percentage

def solutionGenerator(rotorInner, rotorOuter, discSpacing, discThickness, numberSpacing,
                     voluteThickness, wallSpace, upperClearance, wallDisplacement,
                     TotalMassFlowRate, density, RPM, profileN = 2, rsPoint = 100):
    KJ = flowParameters(rotorInner, rotorOuter, discSpacing, discThickness, numberSpacing,
                voluteThickness, wallSpace, upperClearance, wallDisplacement,
                TotalMassFlowRate, density, RPM, profileN = profileN)
    firstODEinitial,secondODEinitial = KJ.relativeTipTangential, 0
    rs = np.linspace(1, KJ.innerOuterRatio, rsPoint)
    sol = odeint(bothODE, [firstODEinitial,secondODEinitial], rs, args=(KJ,))
    return KJ, sol, rs

'''Extras'''
def profilePlot(axPlot, instance, numberZPoints): # profile plot in between discs
    zPoints = np.linspace(-instance.discSpacing/2, instance.discSpacing/2, numberZPoints)
    nVal = 3*instance.Fpo - 1
    xVal = ((nVal+1)/nVal)*(np.full((numberZPoints,),1) -
np.power(2*zPoints/instance.discSpacing,nVal))
    axPlot.set_ylabel("Z position relative to DSC", color="white")
    axPlot.set_xlabel("Profile Magnitude (Dimensionless)", color="white")
    axPlot.plot(xVal, zPoints)

'''finding the best rpm for highest power output (not working)'''
def costJ(x, instance):
    optRPM = x[0]
    instance.omega = optRPM*2*pi/60

    firstODEinitial, secondODEinitial = instance.relativeTipTangential, 0
    rs = np.linspace(1, instance.innerOuterRatio, 100)
    sol = odeint(bothODE, [firstODEinitial,secondODEinitial], rs, args=(instance,))
    return (1000-power(sol[:,0], rs, instance))**2



def shaftLosses(instance): # negligible
    wR2 = instance.omega*instance.outerRadius**2
    reDisc = wR2/kinematicViscosity
    tipGap = (instance.totalVoluteSpace)/2 + instance.upperClearance
    endGap = instance.voluteWallSpace
    condition = 470.5*instance.outerRadius/endGap
    if reDisc < condition:
        exponent = 1 # laminar gap
    else:
        exponent = 0.25 # turbulent gap
    cFactorGap = ((instance.outerRadius/(reDisc*endGap))**exponent)*\
        (2*pi if exponent == 1 else 0.00622)*(1/(instance.numberSpacing + 1))
    cFactorTip = (instance.discThickness/tipGap)*(4*pi*kinematicViscosity/wR2)
    torqueLoss = (0.5*instance.discSpacing/instance.outerRadius)*(cFactorGap + cFactorTip)
    return torqueLoss*instance.omega*(instance.numberSpacing+1)

def shearPoints(solution, instance):
    nVal = 3*instance.Fpo - 1
    factor = 2*dynamicViscosity*(nVal+1)*instance.tipVelocity/KJ.discSpacing
    return factor*solution[:,0]

def torqueCalculator(solution, rs, instance):
    totalPower = power(solution[:,0], rs, instance)
    torquePerDisc = totalPower/(instance.omega*2*(instance.numberSpacing))
    return torquePerDisc

def findRPM(rpmGuess, \
            torqueToHit, rotorInner, rotorOuter, discSpacing, discThickness, \
            numberSpacing,voluteThickness, wallSpace, upperClearance, wallDisplacement, \
            TotalMassFlowRate, density, profileN = 2, rsPoint = 100):
    KJ, solKJ, rsKJ = solutionGenerator(rotorInner, rotorOuter, discSpacing, discThickness,
numberSpacing,
                     voluteThickness, wallSpace, upperClearance, wallDisplacement,
                     TotalMassFlowRate, density, rpmGuess, profileN = profileN, rsPoint =
rsPoint)
    torqueOutput = 2*KJ.numberSpacing*torqueCalculator(solKJ, rsKJ, KJ)
    return (torqueOutput - torqueToHit)**2
```

```
Using matplotlib backend: Qt5Agg
```

In [3]:

```python
'''Torque RPM'''
massFlowRateRange = np.arange(0.5,2.5,0.5)
nProfileList = [2,4,6,8]
rpmRange = np.arange(250,4250,50)
style = ['-', '--', '-.', ':']

massStoreDict = {}
for k in range(len(massFlowRateRange)):
    storeDict = {}
    for i in range(len(nProfileList)):
        storeRPMTorque = np.zeros((len(rpmRange), 2))
        for j in range(len(rpmRange)):
            KJ, solKJ, rsKJ = solutionGenerator(rotorInner, rotorOuter, discSpacing,
discThickness, nDisc-1,
                     voluteThickness, wallSpace, 0, wallDisplacement,
                     massFlowRateRange[k], density, rpmRange[j], profileN = nProfileList[i])
            torqueOutput = 2*KJ.numberSpacing*torqueCalculator(solKJ, rsKJ, KJ)
            storeRPMTorque[j, 0], storeRPMTorque[j, 1] = rpmRange[j], torqueOutput
        storeDict[nProfileList[i]] = storeRPMTorque
    massStoreDict[massFlowRateRange[k]] = storeDict

plt.rcParams["figure.figsize"]=12, 12
fig, axs = plt.subplots(2,2)
rowIndex, colIndex = 0, 0
for i in massStoreDict:
    if colIndex > 1:
        rowIndex += 1
        colIndex = 0
    axs[rowIndex, colIndex].set_title(f'$\dot m$ = {i}kg/s', fontsize=15)
    for j in massStoreDict[i]:
        axs[rowIndex, colIndex].plot(massStoreDict[i][j][:,0], massStoreDict[i][j][:,1], \
                         style[list(massStoreDict[i].keys()).index(j)], color="black",
label=f"n = {j}")
    colIndex += 1

plt.subplots_adjust(wspace = .2)
fig.add_subplot(111, frame_on=False)
fig.suptitle("Torque vs RPM", fontsize=30)

plt.tick_params(labelcolor="none", bottom=False, left=False)
plt.xlabel("RPM", fontsize=25, labelpad=25)
plt.ylabel("Torque (Nm)", fontsize=25,  labelpad=25)

fig.tight_layout()

handles, labels = axs[0,0].get_legend_handles_labels()
fig.legend(handles, labels, loc='upper right', fontsize=25)
```

Out[3]:

```
<matplotlib.legend.Legend at 0x172d3e8e348>
```

In [18]:

```python
'''Torque @ 0 RPM - Jerryl'''
massFlowRateRange = np.arange(0.5,4.6,0.1)
nProfileList = [2,4,6,8]
style = ['-', '--', '-.', ':']

massStoreDict = {}
for i in range(len(nProfileList)):
    storeMFRTorque = np.zeros((len(massFlowRateRange), 2))
    for j in range(len(massFlowRateRange)):
        KJ, solKJ, rsKJ = solutionGenerator(rotorInner, rotorOuter, discSpacing, discThickness,
nDisc-1,
                 voluteThickness, wallSpace, 0, wallDisplacement,
                 massFlowRateRange[j], density, 1, profileN = nProfileList[i])
        torqueOutput = 2*KJ.numberSpacing*torqueCalculator(solKJ, rsKJ, KJ)
        storeMFRTorque[j, 0], storeMFRTorque[j, 1] = massFlowRateRange[j], torqueOutput
    massStoreDict[nProfileList[i]] = storeMFRTorque

fig, ax = plt.subplots()
for i in massStoreDict:
    ax.plot(massStoreDict[i][:,0], massStoreDict[i][:,1],
style[list(massStoreDict.keys()).index(i)]\
            , color="black", label=f"n = {i}")
ax.set_title("Torque vs Mass Flow Rate", fontsize=40, pad=10)
ax.set_xlabel("Mass Flow Rate (kg/s)", fontsize=25, labelpad=25)
ax.set_ylabel("Maximum Torque (Nm)", fontsize=25, labelpad=25)

ax.tick_params(axis='x', labelsize=20)
ax.tick_params(axis='y', labelsize=20)

ax.legend(loc='upper left', fontsize=25)
```

Out[18]:

```
<matplotlib.legend.Legend at 0x172cd163d08>
```

In [2]:

```python
testTorque = 0.5
data = (testTorque, rotorInner, rotorOuter, discSpacing, discThickness, nDisc-1,\
        voluteThickness, wallSpace, 0, wallDisplacement, TotalMassFlowRate, density)

startingRPM = 50
setRPM = fsolve(findRPM, startingRPM, args=data)[0]


KJ, solKJ, rsKJ = solutionGenerator(rotorInner, rotorOuter, discSpacing, discThickness, nDisc-1,
                     voluteThickness, wallSpace, 0, wallDisplacement,
                     TotalMassFlowRate, density, setRPM)
torqueOutput = 2*KJ.numberSpacing*torqueCalculator(solKJ, rsKJ, KJ)
print(torqueOutput)
```

```
0.5000000000000613
```

In [8]:

```python
'''Power Countour'''
for n in range(1):

    maxRotorOuter = 0.164
    kFactorRange = np.linspace(1,5,25)

    nDiscRange = np.arange(1,17,1)
    discSpacingRange = np.arange(0.0002,0.001,0.0001)
    torqueRange = np.linspace(0.1, 1.3, 35)

    bSpacing = discSpacingRange[0] # choosing smallest possible disc spacing
    hiOutput = 0

    powerStorageStore = []
    bestLine = []
    for j in range(len(nDiscRange)):
        powerStorage = np.zeros([len(torqueRange),len(kFactorRange)])
        X,Y = np.meshgrid(torqueRange,kFactorRange)

        discNumberSpacing = nDiscRange[j]
        for l in range(len(kFactorRange)):
            highTorque, highPower = 0, 0
            for k in range(len(torqueRange)):
                currentTorque = torqueRange[k]
                maxRotorOuterCase = maxRotorOuter/kFactorRange[l]
                guessRPM = 50

                data = (currentTorque, 0.3*maxRotorOuterCase, maxRotorOuterCase, bSpacing, \
                        discThickness, discNumberSpacing, voluteThickness, wallSpace, 0, \
                        wallDisplacement, TotalMassFlowRate, density, 2*(n+1))
                setRPM = fsolve(findRPM, guessRPM, args=data)[0]
                setOmega = setRPM*2*pi/60

                powerStorage[k,l] = setOmega*currentTorque

                '''Base Case'''
                if powerStorage[k,l]>highPower and discNumberSpacing==4:
                    highTorque, highPower = currentTorque, powerStorage[k,l]
                if powerStorage[k,l] > hiOutput:
                    hiOutput = powerStorage[k,l]
            if highPower == 0 or highTorque == torqueRange[-1]:
                pass
            else:
                bestLine.append([highTorque, kFactorRange[l]])
        powerStorageStore.append(powerStorage)
        print(f"n_disc = {discNumberSpacing+1} done")


    X,Y = np.meshgrid(torqueRange,kFactorRange)
    powerStorageStore = np.array(powerStorageStore)
    bestLine = np.array(bestLine)

    '''Plots'''

    plt.rcParams["figure.figsize"]=12, 12
    fig, axs = plt.subplots(4,4)
    # levels = np.arange(0,int(hiOutput)+1,1)
    levels = np.arange(0,301,5)
    countX, countY = 0,0
    for i in range(len(powerStorageStore)):
        if(countY==4):
            countX+=1
            countY=0
        cs = axs[countX,countY].contourf(X,Y,powerStorageStore[i].transpose(),levels=levels,\
                                         extend='both',cmap="OrRd")
        axs[countX,countY].set_title(f'n = {i+2}',color='white',fontsize=15)
        axs[countX,countY].tick_params(axis='x', colors='white',labelsize=12)
        axs[countX,countY].tick_params(axis='y', colors='white',labelsize=12)
        if countX==0 and countY==3:
            axs[countX,countY].plot(bestLine[:,0], bestLine[:,1], "-.", color="black")
            axs[countX,countY].plot([torqueRange[0],torqueRange[-1]],
                                    [chosenScaleDownFactor,chosenScaleDownFactor], ":",
color="black")
            axs[countX,countY].grid(color='white')
        countY+=1

    plt.subplots_adjust(wspace = .2)
    fig.add_subplot(111, frame_on=False)
    fig.suptitle("Power Output Contour", color="white", fontsize=30, x=0.45)

    plt.tick_params(labelcolor="none", bottom=False, left=False)
    plt.xlabel("Torque (Nm)", fontsize=25, color='white', position=(0.4, 0.5), labelpad=15)
    plt.ylabel("Scale Down Factor", fontsize=25, color='white', position=(0.4, 0.5),
labelpad=15)

    fig.tight_layout()
    cbar = plt.colorbar(cs,ax=axs)
    cbar.ax.set_ylabel('Power (W)', fontsize=25)
    cbar.ax.yaxis.label.set_color('white')
    cbar.ax.tick_params(axis='y', colors='white')

    ticklabs = cbar.ax.get_yticklabels()
    cbar.ax.set_yticklabels(ticklabs, fontsize=15)

    fig.patch.set_facecolor('#373E4B')
```

```
n_disc = 2 done
n_disc = 3 done
n_disc = 4 done
n_disc = 5 done
n_disc = 6 done
n_disc = 7 done
n_disc = 8 done
n_disc = 9 done
n_disc = 10 done
n_disc = 11 done
n_disc = 12 done
n_disc = 13 done
n_disc = 14 done
n_disc = 15 done
n_disc = 16 done
n_disc = 17 done
```

In [ ]: