Project_2\Project2BackendCalcsV2.py

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
'''
AEEM4063 Project 2 Backend Calculations

Computes compressor and turbine states, angles, and velocities for the multi-stage compressor and turbine design

Authors: Matt Boller, Pierce Elliott

University of Cincinnati FS23

'''

# Begin the code
class TurboMachineryComputationV2:

    def __init__(self):
        self.BPR = 6.0 # Bypass ratio
        self.mdot = 280.0 # Total mass flow, kg/s


        # Sea level static conditions
        self.pa = 1.01325 # Ambient pressure, bar
        self.Ta = 288.15 # Ambient temperature, K
        self.a = 340.3 # Speed of sound, m/s
        self.rho_a = 1.2250 # Ambient density, kg/m^3
        self.R = 287 # Gas constant, J/(kg*K)
        # Compressor parameters
        self.cpr = 15 # Compressor pressure ratio
        self.n_inf_c = 0.9 # Lower bound of polytropic efficiency
        self.pi_f = 1.4 # Fan pressure ratio
        self.n_inf_f = 0.92 # Fan polytropic efficiency
        self.haller = 0.65 # Lower bound on de Haller criteria
        self.TMach_r = 1.2 # Upper bound on tip Mach number
        self.y_c = 1.4 # Gamma in the cold section
        self.cp_c = 1.005 # Specific heat cold section, kJ/(kg*K)
        self.rr_rt_o = 0.5 # Root to tip ratio at entry to the compressor
        self.lam = 0.98 # Loading coefficient for the first stage
        # Turbine parameters
        self.T_04 = 2275 # Turbine inlet temperature
        self.n_inf_t = 0.92 # Turbine polytropic efficiency
        self.phi = 0.78 # Lower bound on flow coefficient
        self.psi = 3.3 # Upper bound on loading coefficient
        self.y_h = 1.333 # Gamma in the hot section
        self.cp_h = 1.148 # Specific heat hot section, kJ/(kg*K)
        # Basic design parameters - constant tip radius assumption
        self.U_t1 = 420 # Inlet Tip speed, m/s

        self.C_a = 180 # Inlet axial velocity, m/s

        # Other parameters
        self.n_m = 0.99 # assumed mechanical efficiency of 99%
        self.delP_b = 0.00 # assumed burner pressure loss of 0%
        self.n_b = 1.0 # assumed burner eff. of 100%

        # Calcs for inlet conditions to the compressor
        # First, calculate fan outlet (assume intake has no loss)
        self.P_02 = self.pi_f*self.pa
        self.T_02 = self.Ta + self.Ta*((self.pi_f)**((self.y_c-1)/(self.y_c*self.n_inf_f)) - 1)

        # Static temperature, pressure, and density at the inlet
        T2 = self.T_02 - self.C_a**2/(2*self.cp_c*1000)
        self.p2 = self.P_02*(T2/self.T_02)**(self.y_c/(self.y_c-1))
        self.rho2 = self.p2*1e5/(self.R*T2)

        # Tip radius and rotation rate
        self.rt = np.sqrt((self.mdot/(self.BPR+1))/(np.pi*self.rho2*self.C_a*(1-self.rr_rt_o**2)))
        self.N = self.U_t1/(2*np.pi*self.rt)

        # Root and mean radius at the inlet
        self.rr = self.rt*self.rr_rt_o
        self.rm = (self.rr+self.rt)/2

        # Velocity and Mach number relative to the inlet
        V1_t = np.sqrt(self.U_t1**2+self.C_a**2)
```

```
        self.M1_t = V1_t/(np.sqrt(self.y_c*self.R*T2))

        # Outlet calculations for the compressor
        self.P_03 = self.P_02*self.cpr
        self.T_03 = self.T_02*(self.cpr)**((self.y_c-1)/(self.y_c*self.n_inf_c))
        # Assume the exit velocity is axial and the same as the initial inlet velocity
        T3 = self.T_03 - self.C_a**2/(2*self.cp_c*1000)
        self.p3 = self.P_03*(T3/self.T_03)**(self.y_c/(self.y_c-1))
        self.rho3 = self.p3*1e5/(self.R*T3)

        # Calculates the fuel flow
        FAR = (self.cp_h*(self.T_04) - self.cp_c*(self.T_03))/(43100 - self.cp_h*self.T_03)
        m_fuel = FAR*(self.mdot / (self.BPR + 1))
        self.mdoth = (self.mdot / (self.BPR + 1)) + m_fuel # Hot Section Flow, kg/s

        # Calculates inlet pressure to the turbine using burner efficiency
        self.P_04 = self.P_03*self.n_b

        A3 = self.mdot/(self.BPR+1)/(self.rho3*self.C_a)
        h3 = A3/(2*np.pi*self.rm)
        self.rt_3 = self.rm + (h3/2)
        self.rr_3 = self.rm - (h3/2)

        return

    def update(self):
        return self.N, self.rt, self.rm, self.rr, self.M1_t, self.rt_3, self.rr_3, self.P_02, self.T_02, self.P_03, self.T_03

    def fullcompressor(self):
        # Temperature rise through the compressor
        self.delt_Ts = self.T_03-self.T_02
        # Assuming axial velocity remains constant throughout each stage
        # Calculating mean blade speed
        self.U_m = 2*np.pi*self.rm*self.N
        # Calculating the estimated temperature rise per stage
        beta1 = np.arctan(self.U_m/self.C_a)
        V1 = self.C_a/np.cos(beta1)
        # Using de Haller criteria to find V2
        V2 = V1*self.haller
        beta2 = np.arccos(self.C_a/V2)
        # Estimated delta T0 per stage
        T0s_est = self.lam*self.U_m*self.C_a*(np.tan(beta1)-np.tan(beta2))/(self.cp_c*1e3)
        # Estimated number of stages
        stage_est = np.ceil(self.delt_Ts/T0s_est)

        alpha1 = 0.305 # radians, inlet blade angle

        ################ Stage 1 ################
        # First stage calculation
        delta_T0 = 30.0 ## Desired temperature rise per stage for the first stage
        delta_C_w = self.cp_c*1e3*delta_T0/(self.lam*self.U_m)
        self.lam = self.workdone(1) ## Update work done factor
        # Whirl velocities
        C_w1 = self.C_a*np.tan(alpha1)
        C_w2 = delta_C_w + C_w1
        # Relative angles
        beta1 = np.arctan((self.U_m - C_w1)/self.C_a)
        beta2 = np.arctan((self.U_m - C_w2)/self.C_a)
        # Outlet angle of rotor
        alpha2 = np.arctan(C_w2/self.C_a)
        diffusion = np.cos(beta1)/np.cos(beta2)
        # Organizing data for tables
        data = np.concatenate((np.rad2deg(np.array([alpha1,alpha2,beta1,beta2])), np.array([0.0,diffusion,0.0])))
        # Outlet stagnation pressure calculation of the first stage
        p031 = self.P_02*(1 + (self.n_inf_c*delta_T0)/self.T_02)**(self.y_c/(self.y_c-1))
        # Inlet tip Mach number
        T2 = self.T_02 - self.C_a**2/(2*self.cp_c*1000)
        # M11t = self.C_a/np.cos(beta1)/np.sqrt(self.y_c*self.R*T2)
        # Outlet tip Mach number
        C21 = self.C_a/np.cos(alpha2)
        T021 = self.T_02 + delta_T0
        T21 = T021 - C21**2/(2*self.cp_c*1e3)
        # M21t = self.C_a/np.cos(beta2)/np.sqrt(self.y_c*self.R*T21)
        # Approximate Degree of Reaction at the mean radius
        React = 1 - (C_w1+C_w2)/(2*self.U_m)
        # Radius at the inlet to the rotor
        T11 = self.T_02 - self.C_a**2/(2*self.cp_c*1e3)
        p11 = self.P_02*(T11/self.T_02)**(self.y_c/(self.y_c-1))
        rho11 = p11*1e5/(T11*self.R)
        A11 = self.mdot/(self.BPR+1)/(rho11*self.C_a)
```

```
        h11 = A11/(2*np.pi*self.rm)
        rt11 = self.rm + (h11/2)
        rr11 = self.rm - (h11/2)
        rr_rt1 = rr11/rt11
        # Radius in between rotor and stator
        p21 = p031*(T21/T021)**(self.y_c/(self.y_c-1))
        rho21 = p21*1e5/(T21*self.R)
        A21 = self.mdot/(self.BPR+1)/(rho21*self.C_a)
        h21 = A21/(2*np.pi*self.rm)
        rt21 = self.rm + (h21/2)
        rr21 = self.rm - (h21/2)
        rr_rt = rr21/rt21

        sizingtable = pd.DataFrame(np.round([np.array([rr_rt1,rr_rt,0.0, h11,h21,0.0])],5),columns=['r_t 1','r_t 2','r_t 3','h1','h2','h3'])

        meantable = pd.DataFrame(np.round([np.concatenate((data,np.array([T021,T021-self.T_02]),np.array([p031/self.P_02]), np.array([p031]),np.array([0.0]),np.array([0.0,C_w1,C_w2]),np.array([React]),np.array([self.lam])))],3), columns=['alpha1','alpha2','beta1','beta2','alpha3','V2/V1','C3/C2','T02','deltaT','P03/P01','P03','M1t','M2t','Cw1','Cw2','Reaction','Work Done'])
        # meantable['C3/C2'][0] = 2.0

        ################ Stage 2 ################
        delta_T0 = 41.0 ## Desired temperature rise for the second stage
        React = 0.68  ## Degree of reaction for the second stage
        self.lam = self.workdone(2) ## Update work done factor
        # Calculate relative blade angles by solving system of eqs
        B1 = delta_T0*self.cp_c*1e3/(self.lam*self.U_m*self.C_a)
        B2 = React*2*self.U_m/self.C_a
        beta2 = np.arctan((B2-B1)/2)
        beta1 = np.arctan(B1+np.tan(beta2))
        # Calculate absolute blade angles
        alpha1 = np.arctan(self.U_m/self.C_a - np.tan(beta1))
        alpha2 = np.arctan(self.U_m/self.C_a - np.tan(beta2))
        # Calculate whirl velocities
        C_w1 = self.C_a*np.tan(alpha1)
        C_w2 = self.C_a*np.tan(alpha2)
        # Calculation diffusion for de Haller criteria
        diffusion = np.cos(beta1)/np.cos(beta2)
        # Calculate outlet pressure
        p032 = p031*(1 + (self.n_inf_c*delta_T0)/T021)**(self.y_c/(self.y_c-1))
        # Calculate outlet temperature
        T022 = T021 + delta_T0
        # Calculate inlet Mach number
        C21 = self.C_a/np.cos(alpha1)
        T21 = T021 - C21**2/(2*self.cp_c*1e3)
        # M12t = self.C_a/np.cos(beta1)/np.sqrt(self.y_c*self.R*T21)
        # Calculate outlet Mach number
        C22 = self.C_a/np.cos(alpha2)
        T22 = T022 - C22**2/(2*self.cp_c*1e3)
        # M22t = self.C_a/np.cos(beta2)/np.sqrt(self.y_c*self.R*T22)
        # Data organization
        data = np.round(np.concatenate((np.rad2deg(np.array([alpha1,alpha2,beta1,beta2])), np.array([0.0,diffusion, 0.0,T022, T022-T021,p032/p031, p032, 0.0, 0.0,C_w1,C_w2, React, self.lam]))),3)
        # Radius at the inlet to the rotor
        T12 = T021 - C21**2/(2*self.cp_c*1e3)
        p12 = p031*(T12/T021)**(self.y_c/(self.y_c-1))
        rho12 = p12*1e5/(T12*self.R)
        A12 = self.mdot/(self.BPR+1)/(rho12*self.C_a)
        h12 = A12/(2*np.pi*self.rm)
        rt12 = self.rm + (h12/2)
        rr12 = self.rm - (h12/2)
        rr_rt1 = rr12/rt12
        # Radius in between rotor and stator
        p22 = p032*(T22/T022)**(self.y_c/(self.y_c-1))
        rho22 = p22*1e5/(T21*self.R)
        A22 = self.mdot/(self.BPR+1)/(rho22*self.C_a)
        h22 = A22/(2*np.pi*self.rm)
        rt22 = self.rm + (h22/2)
        rr22 = self.rm - (h22/2)
        rr_rt = rr22/rt22

        s_data = np.round(np.array([rr_rt1,rr_rt,0.0, h12,h22,0.0]),5)

        # Update table
        meantable.loc[len(meantable)] = data
        sizingtable.loc[len(sizingtable)] = s_data

        ################ Stage 3 ################
        delta_T0 = 43 ## Desired temperature rise for the second stage
        React = 0.67 ## Degree of reaction for the second stage
        self.lam = self.workdone(3) ## Update work done factor
```

```
        # Calculate relative blade angles by solving system of eqs
        B1 = delta_T0*self.cp_c*1e3/(self.lam*self.U_m*self.C_a)
        B2 = React*2*self.U_m/self.C_a
        beta2 = np.arctan((B2-B1)/2)
        beta1 = np.arctan(B1+np.tan(beta2))
        # Calculate absolute blade angles
        alpha1 = np.arctan(self.U_m/self.C_a - np.tan(beta1))
        alpha2 = np.arctan(self.U_m/self.C_a - np.tan(beta2))
        # Calculate whirl velocities
        C_w1 = self.C_a*np.tan(alpha1)
        C_w2 = self.C_a*np.tan(alpha2)
        # Calculation diffusion for de Haller criteria
        diffusion = np.cos(beta1)/np.cos(beta2)
        # Calculate outlet pressure
        p033 = p032*(1 + (self.n_inf_c*delta_T0)/T022)**(self.y_c/(self.y_c-1))
        # Calculate outlet temperature
        T023 = T022 + delta_T0
        # Calculate inlet Mach number
        C22 = self.C_a/np.cos(alpha1)
        T22 = T022 - C22**2/(2*self.cp_c*1e3)
        # M13t = self.C_a/np.cos(beta1)/np.sqrt(self.y_c*self.R*T22)
        # Calculate outlet Mach number
        C23 = self.C_a/np.cos(alpha2)
        T23 = T023 - C23**2/(2*self.cp_c*1e3)
        # M23t = self.C_a/np.cos(beta2)/np.sqrt(self.y_c*self.R*T23)
        # Data organization
        data = np.round(np.concatenate((np.rad2deg(np.array([alpha1,alpha2,beta1,beta2])), np.array([0.0,diffusion, 0.0,T023, T023-
T022,p033/p032, p033, 0.0, 0.0,C_w1,C_w2, React, self.lam]))),3)
        # Radius at the inlet to the rotor
        p13 = p032*(T22/T022)**(self.y_c/(self.y_c-1))
        rho13 = p13*1e5/(T22*self.R)
        A13 = self.mdot/(self.BPR+1)/(rho13*self.C_a)
        h13 = A13/(2*np.pi*self.rm)
        rt13 = self.rm + (h13/2)
        rr13 = self.rm - (h13/2)
        rr_rt1 = rr13/rt13
        # Radius in between rotor and stator
        p23 = p033*(T23/T023)**(self.y_c/(self.y_c-1))
        rho23 = p23*1e5/(T23*self.R)
        A23 = self.mdot/(self.BPR+1)/(rho23*self.C_a)
        h23 = A23/(2*np.pi*self.rm)
        rt23 = self.rm + (h23/2)
        rr23 = self.rm - (h23/2)
        rr_rt = rr23/rt23

        s_data = np.round(np.array([rr_rt1,rr_rt,0.0,h13,h23,0.0]),5)

        # Update table
        meantable.loc[len(meantable)] = data
        sizingtable.loc[len(sizingtable)] = s_data

        test_p03 = p033
        test_T02 = T023

        test3_p03 = test_p03

        final_P03 = self.cpr*self.P_02
        ns = 4

        ################ Intermediate Stages ################
        while test3_p03 < final_P03:
            self.lam = self.workdone(ns) ## Update work done factor
            ns += 1

            delta_T0 = 100.0
            React = 0.62
            data, s_data, test2_p03, test2_T02, diffusion = self.compressorstage(delta_T0, React, test_p03, test_T02)
            if ns == 5:
                React = 0.645
                while diffusion < self.haller+0.008:
                    delta_T0 -= 0.001
                    data, s_data, test2_p03, test2_T02, diffusion = self.compressorstage(delta_T0, React, test_p03, test_T02)
            elif ns == 6:
                React = 0.625
                while diffusion < self.haller+0.01:
                    delta_T0 -= 0.001
                    data, s_data, test2_p03, test2_T02, diffusion = self.compressorstage(delta_T0, React, test_p03, test_T02)
            else:
                while diffusion < self.haller+0.01:
                    delta_T0 -= 0.001
                    data, s_data, test2_p03, test2_T02, diffusion = self.compressorstage(delta_T0, React, test_p03, test_T02)
```

```
315
316             test_p03 = test2_p03
317             test_T02 = test2_T02
318             # Update table
319             meantable.loc[len(meantable)] = data
320             sizingtable.loc[len(sizingtable)] = s_data
321
322             # Calculate next estimated max
323             delta_T0 = 100.0
324             data, s_data, test2_p03, test2_T02, diffusion = self.compressorstage(delta_T0, React, test_p03, test_T02)
325             while diffusion < self.haller+0.01:
326                 delta_T0 -= 0.01
327                 data, s_data, test2_p03, test2_T02, diffusion = self.compressorstage(delta_T0, React, test_p03, test_T02)
328
329             test3_p03 = test2_p03
330
331         p01 = test_p03
332         T01 = test_T02
333
334         React = 0.65
335         ################ Last Stage ################
336         delta_T0 = ((15.0*self.P_02/p01)**((self.y_c-1)/self.y_c)-1)*T01/self.n_inf_c # Desired temperature rise for the second stage
337         React = 0.5 ## Degree of reaction for the second stage
338         self.lam = self.workdone(ns+1) ## Update work done factor
339         # Calculate relative blade angles by solving system of eqs
340         B1 = delta_T0*self.cp_c*1e3/(self.lam*self.U_m*self.C_a)
341         B2 = React*2*self.U_m/self.C_a
342         beta2 = np.arctan((B2-B1)/2)
343         beta1 = np.arctan(B1+np.tan(beta2))
344         # Calculate absolute blade angles
345         alpha1 = np.arctan(self.U_m/self.C_a - np.tan(beta1))
346         alpha2 = np.arctan(self.U_m/self.C_a - np.tan(beta2))
347         # Calculate whirl velocities
348         C_w1 = self.C_a*np.tan(alpha1)
349         C_w2 = self.C_a*np.tan(alpha2)
350         # Calculation diffusion for de Haller criteria
351         diffusion = np.cos(beta1)/np.cos(beta2)
352         # Calculate outlet pressure
353         p03 = p01*(1 + (self.n_inf_c*delta_T0)/T01)**(self.y_c/(self.y_c-1))
354         # Calculate outlet temperature
355         T02 = T01 + delta_T0
356         # Calculate inlet Mach number
357         C2 = self.C_a/np.cos(alpha1)
358         T1 = T01 - C2**2/(2*self.cp_c*1e3)
359         # M1t = self.C_a/np.cos(beta1)/np.sqrt(self.y_c*self.R*T1)
360         # Calculate outlet Mach number
361         C23 = self.C_a/np.cos(alpha2)
362         T2 = T02 - C23**2/(2*self.cp_c*1e3)
363         # M2t = self.C_a/np.cos(beta2)/np.sqrt(self.y_c*self.R*T2)
364         # Data organization
365         data = np.round(np.concatenate((np.rad2deg(np.array([alpha1,alpha2,beta1,beta2])), np.array([0.0,diffusion, 0.0,T02, T02-T01,
    p03/p01, p03, 0.0, 0.0,C_w1,C_w2, React, self.lam]))),3)
366          # Radius at the inlet to the rotor
367         C1 = self.C_a/np.cos(alpha1)
368         T1 = T01 - C1**2/(2*self.cp_c*1e3)
369         p1 = p01*(T1/T01)**(self.y_c/(self.y_c-1))
370         rho1 = p1*1e5/(T1*self.R)
371         A1 = self.mdot/(self.BPR+1)/(rho1*self.C_a)
372         h1 = A1/(2*np.pi*self.rm)
373         rt1 = self.rm + (h1/2)
374         rr1 = self.rm - (h1/2)
375         rr_rt1 = rr1/rt1
376         # Radius in between rotor and stator
377         p2 = p03*(T2/T02)**(self.y_c/(self.y_c-1))
378         rho2 = p2*1e5/(T2*self.R)
379         A2 = self.mdot/(self.BPR+1)/(rho2*self.C_a)
380         h2 = A2/(2*np.pi*self.rm)
381         rt2 = self.rm + (h2/2)
382         rr2 = self.rm - (h2/2)
383         rr_rt = rr2/rt2
384         # Radius outlet of the stator
385         T3 = T02 - self.C_a**2/(2*self.cp_c*1e3)
386         p3 = p03*(T3/T02)**(self.y_c/(self.y_c-1))
387         rho3 = p3*1e5/(T3*self.R)
388         A3 = self.mdot/(self.BPR+1)/(rho3*self.C_a)
389         h3 = A3/(2*np.pi*self.rm)
390         rt3 = self.rm + (h3/2)
391         rr3 = self.rm - (h3/2)
392         rr_rt3 = rr3/rt3
393
394         s_data = np.round(np.array([rr_rt1,rr_rt,rr_rt3,h1,h2,h3]),5)
```

```
395
396         # Update table
397         meantable.loc[len(meantable)] = data
398         sizingtable.loc[len(sizingtable)] = s_data
399
400         # Iterate through table to organize stator outlet angles
401         for i in range(0,len(meantable)-1):
402             meantable['alpha3'][i] = meantable['alpha1'][i+1]
403             sizingtable['r_t 3'][i] = sizingtable['r_t 1'][i+1]
404             sizingtable['h3'][i] = sizingtable['h1'][i+1]
405
406         # Iterate through table to calculate de Haller values for the stators
407         for i in range(0,len(meantable)):
408             alpha2 = np.deg2rad(meantable['alpha2'][i])
409             alpha3 = np.deg2rad(meantable['alpha3'][i])
410             s_diffusion = np.cos(alpha2)/np.cos(alpha3)
411             meantable['C3/C2'][i] = np.round(s_diffusion,3)
412
413         tiproot_table, whirl_table, vel_table, meantable, dif_table, velBlade_table = self.comp_root_tip(meantable=meantable,
    sizingtable=sizingtable,rm=self.rm)
414
415         sizingtable.index = np.arange(1, len(sizingtable)+1)
416         meantable.index = np.arange(1, len(meantable)+1)
417         tiproot_table.index = np.arange(1, len(tiproot_table)+1)
418         whirl_table.index = np.arange(1, len(whirl_table)+1)
419         vel_table.index = np.arange(1, len(vel_table)+1)
420         dif_table.index = np.arange(1, len(dif_table)+1)
421         velBlade_table.index = np.arange(1, len(velBlade_table)+1)
422         # meantable.index.name = 'Stage'
423         # meantable.reset_index().to_string(index=False)
424         return meantable, sizingtable, tiproot_table, whirl_table, vel_table, dif_table, velBlade_table
425
426
427
428
429     def fullturbine(self):
430
431         # Specifies the ranges for iteration
432         # psi_range1 = np.arange(2.0, 3.3, 0.05)
433         psi_1 = 2.75
434         phi_range1 = np.arange(0.85, 1.2, 0.05)
435         lambda_range1 = np.arange(0.4,0.95,0.05)
436
437         phi_range2 = np.arange(0.85, 1.2, 0.05)
438         lambda_range2 = np.arange(0.4,0.95,0.05)
439
440         dCw3 = 1000.0 # Sets a max for whirl to compare to
441         dh = 1000.0 #sets a max height
442         desiredParams = np.array([0.0,0.0,0.0,0.0,0.0,0.0,0.0]) # [psi1, phi1, lambda1, psi2, phi2, lambda2, Cw3]
443
444         ################ Preliminary Sizing ################
445         # Sets the rotational speed and mean blade speed
446         N = self.N
447         lamdaN = 0.05 # assumed standard value
448         Um = 450 # m/s
449         maxMach = 1.2 # sets the max Mach at the tip
450
451         # Calculates the total temperature drop based on a work balance from the compressor (using assummed mech. eff.)
452         dT0_turb = (self.cp_c*(self.T_03-self.T_02))/(self.cp_h*self.n_m)
453         desired_dT0s = dT0_turb/2 # splits the temp drop over the 2 stgs based on trial/error
454
455         ################ Stage 1 ################
456         # Uses stage estimation based on constant drop (initial guess) over the stages
457         T0s_est = 1000 # K
458         # Assumptions for first stage
459         alpha1 = 0.0
460         alpha3 = 0.0
461         P_011 = self.P_04
462         T_011 = self.T_04
463
464
465         for j in range(len(phi_range1)):
466             for k in range(len(lambda_range1)):
467                 psi_temp1 = psi_1
468                 phi_temp1 = phi_range1[j]
469                 lambda_temp1 = lambda_range1[k]
470
471                 # Calculates the temp drop for the first stage based on the psi
472                 T0s_rev1 = (psi_temp1*(Um**2))/(2*self.cp_h*1000)
473
```

```
                gasParamsStg1, measurementsStg1, axialV1, otherV1 = self.turbinestage(alpha1, alpha3, Um, T_011, P_011, T0s_rev1, psi_temp1, phi_temp1, lambda_temp1)

                [h11, h21, h31, M31t] = [measurementsStg1[3], measurementsStg1[4], measurementsStg1[5], gasParamsStg1[8]]
                # checks if the geometry is satisfied before continuing
                if (h31 > h21 and h21 > h11 and M31t < maxMach):
                    ################ Stage 2 ################
                    # Starts second stage calculations
                    # Calculates the remaining temp drop
                    T0_remaining = dT0_turb - T0s_rev1
                    psi_turb2 = (2*self.cp_h*1e3*T0_remaining)/(Um**2)
                    if psi_turb2 > 3.3:
                        continue # Continues if temp drop too coeff. out of range
                    else:
                        # Iterates through another set of phi and lambda values to optimize second stage based on first
                        for m in range(len(phi_range2)):
                            for n in range(len(lambda_range2)):
                                phi_temp2 = phi_range2[m]
                                lambda_temp2 = lambda_range2[n]

                                # Calculates the new gas states
                                alpha1 = np.deg2rad(gasParamsStg1[2]) # new alpha1 is the previous stage alpha3
                                alpha3 = 0.0
                                T_012 = self.T_04 - T0s_rev1
                                P_012 = self.P_04*gasParamsStg1[6]
                                gasParamsStg2, measurementsStg2, axialV2, otherV2 = self.turbinestage(alpha1, alpha3, Um, T_012, P_012, T0_remaining, psi_turb2, phi_temp2, lambda_temp2)

                                [h12, h22, h32, M32t, Cw3_2] = [measurementsStg2[3], measurementsStg2[4], measurementsStg2[5], gasParamsStg2[8], axialV2[4]]
                                # checks that the geometry is satisfied
                                if (h12 > h31 and np.round(h22,3) > np.round(h12,3) and M32t < maxMach):
                                    # if lowest whirl, adds it to the optimal params
                                    if Cw3_2 < dCw3:
                                        desiredParams[0] = psi_temp1
                                        desiredParams[1] = phi_temp1
                                        desiredParams[2] = lambda_temp1
                                        desiredParams[3] = psi_turb2
                                        desiredParams[4] = phi_temp2
                                        desiredParams[5] = lambda_temp2
                                        desiredParams[6] = Cw3_2
                                        dCw3 = Cw3_2
                                        print('Turbine Parameters Found: {}'.format(desiredParams))
                                    else: continue
                                else: continue
                else: continue


        # Uses the optimized parameters to calculate the stages
        # [psi1, phi1, lambda1, psi2, phi2, lambda2, Cw3, dh]

        # Assumptions for first stage
        alpha1 = 0.0
        alpha3 = 0.0
        P_011 = self.P_04
        T_011 = self.T_04

        T0s_rev1 = (psi_1*(Um**2))/(2*self.cp_h*1000)

        gasParamsStg1, measurementsStg1, axialV1, otherV1 = self.turbinestage(alpha1, alpha3, Um, T_011, P_011, T0s_rev1, desiredParams[0], desiredParams[1], desiredParams[2])

        rootVals1, tipVals1, rtMeasure1 = self.turb_root_tip(gasParamsStg1, measurementsStg1, Um, axialV1)

        # Stage 2
        T0_remaining = dT0_turb - T0s_rev1

        # Calculates the new gas states
        alpha1 = np.deg2rad(gasParamsStg1[2]) # new alpha1 is the previous stage alpha3
        alpha3 = 0.0
        T_012 = self.T_04 - T0s_rev1
        P_012 = self.P_04*gasParamsStg1[6]

        gasParamsStg2, measurementsStg2, axialV2, otherV2 = self.turbinestage(alpha1, alpha3, Um, T_012, P_012, T0_remaining, desiredParams[3], desiredParams[4], desiredParams[5])

        rootVals2, tipVals2, rtMeasure2 = self.turb_root_tip(gasParamsStg2, measurementsStg2, Um, axialV2)



        # Adds the data to Pandas DFs
        gasParamData = np.round(np.array([gasParamsStg1,gasParamsStg2]),2)
```

```
        measurementsData = np.round(np.array([measurementsStg1,measurementsStg2]),2)
        rootData = np.round(np.array([rootVals1, rootVals2]),2)
        tipData = np.round(np.array([tipVals1, tipVals2]),2)
        rtMeasurements = np.round(np.array([rtMeasure1[0],rtMeasure1[1], rtMeasure1[2], rtMeasure2[3], rtMeasure2[4]]),2) #[rootInlet, tipInlet, rm, 2ndRoot(outlet), 2ndTip(outlet)]
        vData = np.round(np.array([np.concatenate((axialV1, otherV1)), np.concatenate((axialV2,otherV2))]),2)
        # vData2 = np.round(np.array([otherV1, otherV2]),2)

        vDF = pd.DataFrame(vData, index=[1,2],columns=['Ca1','Cw1','Ca2','Cw2','Cw3','C2','Ca3','C3', 'V2', 'V3'])
        # vDF2 = pd.DataFrame(vData2, index=[1,2],columns=['C2','Ca3','C3', 'V2', 'V3'])

        # Creates the DF for the gasAnglesDF
        # [alpha2r,alpha2m,alpha2t,beta2r,beta2m,beta2t,alpha3r,alpha3m,alpha3t,beta3r,beta3m,beta3t]
        gasAngles1 = np.round(np.array([tipVals1[2],gasParamsStg1[1],rootVals1[2],tipVals1[4],gasParamsStg1[3],rootVals1[4],tipVals1[3],gasParamsStg1[2],rootVals1[3],tipVals1[5],gasParamsStg1[4],rootVals1[5]]),2)
        gasAngles2 = np.round(np.array([tipVals2[2],gasParamsStg2[1],rootVals2[2],tipVals2[4],gasParamsStg2[3],rootVals2[4],tipVals2[3],gasParamsStg2[2],rootVals2[3],tipVals2[5],gasParamsStg2[4],rootVals2[5]]),2)
        gasAnglesDF= pd.DataFrame(np.array([gasAngles1, gasAngles2]), index=[1,2],columns=['alpha1_r','alpha1_m','alpha1_t','beta1_r','beta1_m','beta1_t','alpha2_r','alpha2_m','alpha2_t','beta2_r','beta2_m','beta2_t'])


        gasParamDF = pd.DataFrame(gasParamData, index=[1,2], columns=['α1','α2','α3','β2','β3','ΔT0s','P02/P01','Cw3','M3t','Φ','ψ','Λ','MV2r'])
        measurementsDF = pd.DataFrame(measurementsData, index=[1,2], columns=['r_t 1','r_t 2','r_t 3','h1','h2','h3','rm'])
        rootDF = pd.DataFrame(rootData, index=[1,2], columns=['Ur2', 'Ur3', 'alpha2r', 'alpha3r', 'beta2r', 'beta3r', 'Cw1r', 'V2r', 'C2r', 'Cw2r', 'V3r', 'C3r', 'Cw3r', 'phiRoot', 'psiRoot', 'lambdaRoot'])
        tipDF = pd.DataFrame(tipData, index=[1,2], columns=['Ut2', 'Ut3', 'alpha2t', 'alpha3t', 'beta2t', 'beta3t', 'Cw1t', 'V2t', 'C2t', 'Cw2t', 'V3t', 'C3t', 'Cw3t', 'phiTip', 'psiTip', 'lambdaTip'])

        return gasParamDF, measurementsDF, rootDF, tipDF, rtMeasurements, Um, desiredParams, gasAnglesDF, vDF




    def compressorstage(self, delta_T0, React, p01, T01):
        # Calculate relative blade angles by solving system of eqs
        B1 = delta_T0*self.cp_c*1e3/(self.lam*self.U_m*self.C_a)
        B2 = React*2*self.U_m/self.C_a
        beta2 = np.arctan((B2-B1)/2)
        beta1 = np.arctan(B1+np.tan(beta2))
        # Calculate absolute blade angles
        alpha1 = np.arctan(self.U_m/self.C_a - np.tan(beta1))
        alpha2 = np.arctan(self.U_m/self.C_a - np.tan(beta2))
        # Calculate whirl velocities
        C_w1 = self.C_a*np.tan(alpha1)
        C_w2 = self.C_a*np.tan(alpha2)
        # Calculation diffusion for de Haller criteria
        diffusion = np.cos(beta1)/np.cos(beta2)
        # Calculate outlet pressure
        p03 = p01*(1 + (self.n_inf_c*delta_T0)/T01)**(self.y_c/(self.y_c-1))
        # Calculate outlet temperature
        T02 = T01 + delta_T0
        # Calculate inlet Mach number
        C2 = self.C_a/np.cos(alpha1)
        T1 = T01 - C2**2/(2*self.cp_c*1e3)
        # M1t = self.C_a/np.cos(beta1)/np.sqrt(self.y_c*self.R*T1)
        # Calculate outlet Mach number
        C23 = self.C_a/np.cos(alpha2)
        T2 = T02 - C23**2/(2*self.cp_c*1e3)
        # M2t = self.C_a/np.cos(beta2)/np.sqrt(self.y_c*self.R*T2)
        # Data organization
        data = np.round(np.concatenate((np.rad2deg(np.array([alpha1,alpha2,beta1,beta2])), np.array([0.0,diffusion, 0.0,T02, T02-T01, p03/p01, p03, 0.0, 0.0,C_w1,C_w2, React, self.lam]))),3)
        # Radius at the inlet to the rotor
        C1 = self.C_a/np.cos(alpha1)
        T1 = T01 - C1**2/(2*self.cp_c*1e3)
        p1 = p01*(T1/T01)**(self.y_c/(self.y_c-1))
        rho1 = p1*1e5/(T1*self.R)
        A1 = self.mdot/(self.BPR+1)/(rho1*self.C_a)
        h1 = A1/(2*np.pi*self.rm)
        rt1 = self.rm + (h1/2)
        rr1 = self.rm - (h1/2)
        rr_rt1 = rr1/rt1
        # Radius in between rotor and stator
        p2 = p03*(T2/T02)**(self.y_c/(self.y_c-1))
        rho2 = p2*1e5/(T2*self.R)
        A2 = self.mdot/(self.BPR+1)/(rho2*self.C_a)
        h2 = A2/(2*np.pi*self.rm)
        rt2 = self.rm + (h2/2)
        rr2 = self.rm - (h2/2)
        rr_rt = rr2/rt2

```

```
        s_data = np.round(np.array([rr_rt1,rr_rt,0.0,h1,h2,0.0]),5)

        return data, s_data, p03, T02, diffusion

    def turbinestage(self, alpha1, alpha3, Um, T_01, P_01, T0s_rev, psi_turb, phi, desired_lambda):
        '''
            Inputs: alpha1 = stage inlet angle
                    alpha3 = stage exit angle
                    Um = mean blade speed
                    T01 = Inlet stag. temp
                    P01 = Inlet stag. pressure
                    T0s_rev = revised stage temp drop
                    psi_turb = turbine temp drop coeff.
                    phi = flow coefficient (>=0.78)
                    desired_lambda = desired deg. of reaction (~0.5)
            Outputs:
                    gasParams = [alpha1,alpha2,alpha3,beta2,beta3,T0s_rev,Pr,Cw3,M3t,phi,psi_turb,Lambda]
                    measurements = [rtRat1,rtRat2,rtRat3,h1,h2,h3,rm]
                    axialVelocities = [Ca1,Cw1,Ca2,Cw2,Cw3]
        '''

        lambdaN = 0.05 # nozzle loss coefficient based on experience

        # Calculates the B3 and deg. of reaction
        beta3 = np.arctan(np.tan(alpha3) + (1/phi))
        Lambda = (2*phi*np.tan(beta3)- (psi_turb/2))/2
        # iterates to find a suitable degree of reaction
        if np.round(Lambda, 3) < desired_lambda:
            while np.round(Lambda, 3) < desired_lambda:
                alpha3 += np.deg2rad(0.01)
                beta3 = np.arctan(np.tan(alpha3) + (1/phi))
                Lambda = (2*phi*np.tan(beta3)- (psi_turb/2))/2

        # Calculates B2 and a2
        beta2 = np.arctan((1/(2*phi))*(psi_turb/2 - 2*Lambda))
        alpha2 = np.arctan(np.tan(beta2) + (1/phi))

        # Calculates Ca2 and C2
        Ca2 = Um*phi
        C2 = Ca2 / np.cos(alpha2)

        # Calculates the isentropic T2', T2 and p2
        T2 = T_01 - (C2**2/(2*self.cp_h*1e3))
        T2prime = T2- lambdaN*(C2**2/(2*self.cp_h*1e3))
        stagStaticRat = (T_01/T2prime)**(self.y_h/(self.y_h-1))
        CritPrat = 1.853
        if stagStaticRat > CritPrat:
            print('ask the child if he or she is choking')

        P2 = P_01/stagStaticRat

        # Calculate the rho2 and A2
        rho2 = (P2*100)/(0.287*T2)
        A2 = self.mdoth/(rho2*Ca2)

        # Calculates the Ca1, using Ca2 = Ca3 and C1 = C3
        Ca3 = Ca2
        C3 = Ca3/np.cos(alpha3)
        C1 = C3
        Ca1 = C1*np.cos(alpha1)

        # Calculates rho1 and A1
        T1 = T_01 - (C1**2/(2*self.cp_h*1e3))
        P1 = P_01*(T1/T_01)**(self.y_h/(self.y_h-1))
        rho1 = (P1*100)/(0.287*T1)
        A1 = self.mdoth/(rho1*Ca1)

        # Calculates the outlet (station 3) conditions
        T_03 = T_01 - T0s_rev
        T3 = T_03 - (C3**2)/(2*self.cp_h*1e3)

        # Calculates the P03 from the isentropic eff. and P3 from isentropic
        P_03 = P_01*(1 - (T0s_rev/(self.n_inf_t*T_01)))**(self.y_h/(self.y_h-1))
        P3 = P_03*(T3/T_03)**(self.y_h/(self.y_h-1))

        # Calculates the rho3, A3
        rho3 = (P3*100)/(0.287*T3)
        A3 = self.mdoth/(rho3*Ca3)


```

```
        # Mean radius calculation
        rm = Um/(2*np.pi*self.N)

        # Calculates the h1-h3
        h1 = (self.N/Um)*A1
        h2 = (self.N/Um)*A2
        h3 = (self.N/Um)*A3

        # Calculates the rt/rr
        rtRat1 = (rm + (h1/2))/(rm - (h1/2))
        rtRat2 = (rm + (h2/2))/(rm - (h2/2))
        rtRat3 = (rm + (h3/2))/(rm - (h3/2))

        # Calculates the V2 and V3
        V2 = Ca2/np.cos(beta2)
        V3 = ((Um+(C3*np.cos(alpha3)))**2 + Ca3**2)**0.5

        # Calculates the Cw1, Cw2 and Cw3
        Cw1 = C1*np.sin(alpha1)
        Cw2 = Um + V2*np.sin(beta2)
        Cw3 = C3*np.sin(alpha3)

        # Calculates the M3t to check for shocks
        alpha3tip = np.arctan((rm/(rm + h3/2))*np.tan(alpha3))
        C3t = Ca3/np.cos(alpha3tip)
        Ut = self.N*2*np.pi*(rm + h3/2)
        V3t = ((Ut+C3t)**2 + Ca3**2)**0.5
        T3t = T_03 - (C3t**2)/(2*self.cp_h*1e3)
        M3t = V3t/(self.y_h*self.R*T3t)**0.5

        #Calculates the MV2r = V2r/sqrt(y*R*T2r)
        beta2root = np.arctan(((rm/(rm - h2/2))*np.tan(alpha2)) - (((rm - h2/2)/rm)*(Um/Ca2)))
        alpha2root = np.arctan(((rm/(rm - h2/2))*np.tan(alpha2)))
        V2r = Ca2*(1/np.cos(beta2root))
        C2r = Ca2*(1/np.cos(alpha2root))
        T2r = T_01 - (C2r**2)/(2*self.cp_h*1e3)
        MV2r = V2r/(self.y_h*self.R*T2r)**0.5

        # Organizes return statements
        Pr = P_03/P_01

        gasParams = np.array([np.rad2deg(alpha1),np.rad2deg(alpha2),np.rad2deg(alpha3),np.rad2deg(beta2),np.rad2deg(beta3),T0s_rev,Pr,
Cw3,M3t,phi,psi_turb,Lambda,MV2r])
        measurements = np.array([rtRat1,rtRat2,rtRat3,h1,h2,h3,rm])

        return gasParams, measurements, np.array([Ca1,Cw1,Ca2,Cw2,Cw3]), np.array([C2,Ca3,C3, V2, V3])

    def turb_root_tip(self, meanParams, meanMeasurements, Um, axialVelocities):
        # Pulls the mean values
        [alpha1m, alpha2m, alpha3m, beta2m, beta3m, phiMean, psiMean] = np.deg2rad([meanParams[0],meanParams[1],meanParams[2],
meanParams[3],meanParams[4],meanParams[9],meanParams[10]])
        [h1, h2, h3, rm] = [meanMeasurements[3], meanMeasurements[4], meanMeasurements[5], meanMeasurements[6]]
        [Ca1,Cw1,Ca2,Cw2,Cw3] = [axialVelocities[0],axialVelocities[1], axialVelocities[2], axialVelocities[3], axialVelocities[4]]
        Ca3 = Ca2
        # Calculate radii
        rt1 = rm + (h1/2); rr1 = rm - (h1/2)
        rt2 = rm + (h2/2); rr2 = rm - (h2/2)
        rt3 = rm + (h3/2); rr3 = rm - (h3/2)
        # Calculate blade speed
        Ut2 = rt2/rm * Um; Ur2 = rr2/rm * Um
        Ut3 = rt3/rm * Um; Ur3 = rr3/rm * Um
        # Calculates whirl velocities
        # Cw2t = rt1/rm * Cw2; Cw2r = rr1/rm * Cw2
        # Cw3t = rt2/rm * Cw3; Cw3r = rr2/rm * Cw3

        # Calculates the angles
        ######## Root ########
        alpha2r = np.arctan((rm/rr2)*np.tan(alpha2m))
        alpha3r = np.arctan((rm/rr3)*np.tan(alpha3m))
        beta2r = np.arctan(np.tan(alpha2r) - Ur2/Ca2)
        beta3r = np.arctan(np.tan(alpha3r) + Ur3/Ca3)

        ######## Tip ########
        alpha2t = np.arctan((rm/rt2)*np.tan(alpha2m))
        alpha3t = np.arctan((rm/rt3)*np.tan(alpha3m))
        beta2t = np.arctan(np.tan(alpha2t) - Ut2/Ca2)
        beta3t = np.arctan(np.tan(alpha3t) + Ut3/Ca3)

        # Calculates the velocities
        ######## Root ########
        V2r = Ca2/np.cos(beta2r)
```

```
        Cw2r = Ur2 + V2r*np.sin(beta2r)
        C2r = (Cw2r**2 + Ca2**2)**0.5
        C3r = Ca3/np.cos(alpha3r)
        Cw3r = C3r*np.sin(alpha3r)
        V3r = ((Cw3r+Ur3)**2 + Ca3**2)**0.5
        Cw1r = (rm/rr1)*Cw1

        ######## Tip ########
        V2t = Ca2/np.cos(beta2t)
        Cw2t = Ut2 + V2t*np.sin(beta2t)
        C2t = (Cw2t**2 + Ca2**2)**0.5
        C3t = Ca3/np.cos(alpha3t)
        Cw3t = C3t*np.sin(alpha3t)
        V3t = ((Cw3t+Ut3)**2 + Ca3**2)**0.5
        Cw1t = (rm/rt1)*Cw1

        # Calculates the phi, psi, and the lambda for the tip and the root
        phiTip = np.max(np.array([Ca2/Ut2, Ca3/Ut3]))
        phiRoot = np.max(np.array([Ca2/Ur2, Ca3/Ur3]))
        psiTip = 2*phiTip*(np.tan(beta2t)+ np.tan(beta3t))
        psiRoot = 2*phiRoot*(np.tan(beta2r)+ np.tan(beta3r))
        lambdaTip = (phiTip/2)*(np.tan(beta3t) - np.tan(beta2t))
        lambdaRoot = (phiRoot/2)*(np.tan(beta3r) - np.tan(beta2r))

        # Organizes return
        rootValues = np.array([Ur2, Ur3, np.rad2deg(alpha2r), np.rad2deg(alpha3r), np.rad2deg(beta2r), np.rad2deg(beta3r), Cw1r, V2r, C2r, Cw2r, V3r, C3r, Cw3r, phiRoot, psiRoot, lambdaRoot])
        tipValues = np.array([Ut2, Ut3, np.rad2deg(alpha2t), np.rad2deg(alpha3t), np.rad2deg(beta2t), np.rad2deg(beta3t), Cw1t, V2t, C2t, Cw2t, V3t, C3t, Cw3t, phiTip, psiTip, lambdaTip])
        rtMeasurements = np.array([rr1,rt1, rm, rr3, rt3])

        return rootValues, tipValues, rtMeasurements

    def comp_root_tip(self, meantable,sizingtable,rm):
        for i in range(0,len(meantable)):
            # Pull blade heights
            h1 = sizingtable['h1'][i]
            h2 = sizingtable['h2'][i]
            h3 = sizingtable['h3'][i]
            # Pull mean blade angles
            alpha1m = meantable['alpha1'][i]
            alpha2m = meantable['alpha2'][i]
            beta1m = meantable['beta1'][i]
            beta2m = meantable['beta2'][i]
            # Calculate radii
            rt1 = rm + (h1/2); rr1 = rm - (h1/2)
            rt2 = rm + (h2/2); rr2 = rm - (h2/2)
            rt3 = rm + (h3/2); rr3 = rm - (h3/2)
            # Calculate blade speed
            Ut1 = rt1/rm * self.U_m; Ur1 = rr1/rm * self.U_m
            Ut2 = rt2/rm * self.U_m; Ur2 = rr2/rm * self.U_m
            # Calculate whirl velocities
            Cw1 = meantable['Cw1'][i]; Cw2 = meantable['Cw2'][i]
            Cw1t = rm/rt1 * Cw1; Cw1r = rm/rr1 * Cw1
            Cw2t = rm/rt2 * Cw2; Cw2r = rm/rr2 * Cw2
            alpha1t = np.arctan(Cw1t/self.C_a); alpha1r = np.arctan(Cw1r/self.C_a)
            beta1t = np.arctan((Ut1-Cw1t)/self.C_a); beta1r = np.arctan((Ur1-Cw1r)/self.C_a)
            alpha2t = np.arctan(Cw2t/self.C_a); alpha2r = np.arctan(Cw2r/self.C_a)
            beta2t = np.arctan((Ut2-Cw2t)/self.C_a); beta2r = np.arctan((Ur2-Cw2r)/self.C_a)
            # Calculate absolute velocities
            C1m = self.C_a/np.cos(np.deg2rad(alpha1m)); C2m = self.C_a/np.cos(np.deg2rad(alpha2m))
            C1t = self.C_a/np.cos(alpha1t); C2t = self.C_a/np.cos(alpha2t)
            C1r = self.C_a/np.cos(alpha1r); C2r = self.C_a/np.cos(alpha2r)
            # Calculate relative velocities
            V1m = self.C_a/np.cos(np.deg2rad(beta1m)); V2m = self.C_a/np.cos(np.deg2rad(beta2m))
            V1t = self.C_a/np.cos(beta1t); V2t = self.C_a/np.cos(beta2t)
            V1r = self.C_a/np.cos(beta1r); V2r = self.C_a/np.cos(beta2r)
            # Get temperatures
            if i == 0:
                T01 = self.T_02
                T02 = meantable['T02'][i]
            else:
                T01 = meantable['T02'][i-1]
                T02 = meantable['T02'][i]
            # Calculate Mach number
            T1 = T01 - C1t**2/(2*self.cp_c*1e3)
            T2 = T02 - C2t**2/(2*self.cp_c*1e3)
            M1t = V1t/np.sqrt(self.y_c*self.R*T1)
            M2t = V2t/np.sqrt(self.y_c*self.R*T2)
            # Calculate blade velocities
            U1t = (np.tan(alpha1t)+np.tan(beta1t))*self.C_a
```

```
            U1m = (np.tan(np.deg2rad(alpha1m))+np.tan(np.deg2rad(beta1m)))*self.C_a
            U1r = (np.tan(alpha1r)+np.tan(beta1r))*self.C_a
            U2t = (np.tan(alpha2t)+np.tan(beta2t))*self.C_a
            U2m = (np.tan(np.deg2rad(alpha2m))+np.tan(np.deg2rad(beta2m)))*self.C_a
            U2r = (np.tan(alpha2r)+np.tan(beta2r))*self.C_a
            # Update tables
            if i == 0:
                tiproot_table = pd.DataFrame(np.round(np.rad2deg([np.array([alpha1t,np.deg2rad(alpha1m),alpha1r,beta1t,
np.deg2rad(beta1m),beta1r,alpha2t,np.deg2rad(alpha2m),alpha2r,beta2t,np.deg2rad(beta2m),beta2r,0.0,0.0,0.0])]),2),columns=['alpha1_t','
alpha1_m','alpha1_r','beta1_t','beta1_m','beta1_r','alpha2_t','alpha2_m','alpha2_r','beta2_t','beta2_m','beta2_r','alpha3_t','alpha3_m'
,'alpha3_r'])
                whirl_table = pd.DataFrame(np.round([np.array([Cw1t,Cw1,Cw1r,Cw2t,Cw2,Cw2r,0.0,0.0,0.0])],2),columns=['Cw1_t','Cw1_m','
Cw1_r','Cw2_t','Cw2_m','Cw2_r','Cw3_t','Cw3_m','Cw3_r'])
                vel_table = pd.DataFrame(np.round([np.array([C1t,C1m,C1r,C2t,C2m,C2r,0.0,0.0,0.0,V1t,V1m,V1r,V2t,V2m,V2r])],1),columns=
['C1_t','C1_m','C1_r','C2_t','C2_m','C2_r','C3_t','C3_m','C3_r','V1_t','V1_m','V1_r','V2_t','V2_m','V2_r'])
                dif_table = pd.DataFrame(np.round([np.array([V2t/V1t,V2m/V1m,V2r/V1r,0.0,0.0,0.0])],3),columns=['V2/V1_t','V2/V1_m','
V2/V1_r','C3/C2_t','C3/C2_m','C3/C2_r'])
                velBlade_table = pd.DataFrame(np.round([np.array([U1t,U1m,U1r,U2t,U2m,U2r])],3),columns=['U1_t','U1_m','U1_r','U2_t','
U2_m','U2_r'])
            else:
                data = np.round(np.rad2deg(np.array([alpha1t,np.deg2rad(alpha1m),alpha1r,beta1t,np.deg2rad(beta1m),beta1r,alpha2t,
np.deg2rad(alpha2m),alpha2r,beta2t,np.deg2rad(beta2m),beta2r,0.0,0.0,0.0])),2)
                data2 = np.round(np.array([Cw1t,Cw1,Cw1r,Cw2t,Cw2,Cw2r,0.0,0.0,0.0]),2)
                data3 = np.round(np.array([C1t,C1m,C1r,C2t,C2m,C2r,0.0,0.0,0.0,V1t,V1m,V1r,V2t,V2m,V2r]),1)
                data4 = np.round(np.array([V2t/V1t,V2m/V1m,V2r/V1r,0.0,0.0,0.0]),3)
                tiproot_table.loc[len(tiproot_table)] = data
                whirl_table.loc[len(whirl_table)] = data2
                vel_table.loc[len(vel_table)] = data3
                dif_table.loc[len(dif_table)] = data4
                velBlade_table.loc[len(velBlade_table)] = np.round(np.array([U1t,U1m,U1r,U2t,U2m,U2r]))

            meantable['M1t'][i] = np.round(M1t,3)
            meantable['M2t'][i] = np.round(M2t,3)

        for i in range(0,len(tiproot_table)-1):
            tiproot_table['alpha3_t'][i] = tiproot_table['alpha1_t'][i+1]
            tiproot_table['alpha3_m'][i] = tiproot_table['alpha1_m'][i+1]
            tiproot_table['alpha3_r'][i] = tiproot_table['alpha1_r'][i+1]

        for i in range(0,len(tiproot_table)):
            whirl_table['Cw3_t'][i] = np.round(np.tan(np.deg2rad(tiproot_table['alpha3_t'][i]))*self.C_a,2)
            whirl_table['Cw3_m'][i] = np.round(np.tan(np.deg2rad(tiproot_table['alpha3_m'][i]))*self.C_a,2)
            whirl_table['Cw3_r'][i] = np.round(np.tan(np.deg2rad(tiproot_table['alpha3_r'][i]))*self.C_a,2)
            vel_table['C3_t'][i] = np.round(self.C_a/np.cos(np.deg2rad(tiproot_table['alpha3_t'][i])),1)
            vel_table['C3_m'][i] = np.round(self.C_a/np.cos(np.deg2rad(tiproot_table['alpha3_m'][i])),1)
            vel_table['C3_r'][i] = np.round(self.C_a/np.cos(np.deg2rad(tiproot_table['alpha3_r'][i])),1)
            dif_table['C3/C2_t'][i] = np.round(vel_table['C3_t'][i]/vel_table['C2_t'][i],3)
            dif_table['C3/C2_m'][i] = np.round(vel_table['C3_m'][i]/vel_table['C2_m'][i],3)
            dif_table['C3/C2_r'][i] = np.round(vel_table['C3_r'][i]/vel_table['C2_r'][i],3)

        return tiproot_table, whirl_table, vel_table, meantable, dif_table,velBlade_table

    def workdone(self,stage):
        # Loading factor calculation curve fit
        a=0.847936849639078; b=0.15697659830655492; c=-0.2412700053204237
        return a + b*np.exp(c*stage)

    def comp_plotter(self,tiproot_table,stage_array,compressor=True,nplotsbefore=0):
        nplotsbefore = int(nplotsbefore)
        alpha_1 = np.zeros([3,len(stage_array)])
        alpha_2 = np.zeros([3,len(stage_array)])
        beta_1 = np.zeros([3,len(stage_array)])
        beta_2 = np.zeros([3,len(stage_array)])
        def func(x,a,b,c):
            return a*x**2 + b*x + c

        for i in range(len(stage_array)):
            alpha_1[0,i] = tiproot_table['alpha1_r'][stage_array[i]]
            alpha_1[1,i] = tiproot_table['alpha1_m'][stage_array[i]]
            alpha_1[2,i] = tiproot_table['alpha1_t'][stage_array[i]]

            alpha_2[0,i] = tiproot_table['alpha2_r'][stage_array[i]]
            alpha_2[1,i] = tiproot_table['alpha2_m'][stage_array[i]]
            alpha_2[2,i] = tiproot_table['alpha2_t'][stage_array[i]]

            beta_1[0,i] = tiproot_table['beta1_r'][stage_array[i]]
            beta_1[1,i] = tiproot_table['beta1_m'][stage_array[i]]
            beta_1[2,i] = tiproot_table['beta1_t'][stage_array[i]]

            beta_2[0,i] = tiproot_table['beta2_r'][stage_array[i]]
            beta_2[1,i] = tiproot_table['beta2_m'][stage_array[i]]
            beta_2[2,i] = tiproot_table['beta2_t'][stage_array[i]]
```

```
            a1check = a2check = b1check = b2check = 0

            if np.round(alpha_1[0,i],3) != 0.0 or np.round(alpha_1[1,i],3) != 0.0 or np.round(alpha_1[2,i],3) != 0.0:
                fit_a1, opt = curve_fit(func,np.array([0,1,2]),alpha_1[:,i])
            else:
                a1check = 2
            if np.round(alpha_2[0,i],3) != 0.0 and np.round(alpha_2[1,i],3) != 0.0 and np.round(alpha_2[2,i],3) != 0.0:
                fit_a2, opt = curve_fit(func,np.array([0,1,2]),alpha_2[:,i])
            else:
                a2check = 2
            if np.round(beta_1[0,i],3) != 0.0 and np.round(beta_1[1,i],3) != 0.0 and np.round(beta_1[2,i],3) != 0.0:
                fit_b1, opt = curve_fit(func,np.array([0,1,2]),beta_1[:,i])
            else:
                b1check = 2
            if np.round(beta_2[0,i],3) != 0.0 and np.round(beta_2[1,i],3) != 0.0 and np.round(beta_2[2,i],3) != 0.0:
                fit_b2, opt = curve_fit(func,np.array([0,1,2]),beta_2[:,i])
            else:
                b2check = 2

            # print(fit_a1)

            plt.figure(i+nplotsbefore)
            if compressor:
                plt.title('Compressor Stage '+str(int(stage_array[i])))
                # plt.hold('on')
                # plt.plot(alpha_1[:,i],'b')
                if a1check != 2:
                    plt.plot(np.arange(0,2,0.001),func(np.arange(0,2,0.001), fit_a1[0], fit_a1[1], fit_a1[2]),'b')
                # plt.plot(alpha_2[:,i],'g')
                if a2check != 2:
                    plt.plot(np.arange(0,2,0.001),func(np.arange(0,2,0.001), fit_a2[0], fit_a2[1], fit_a2[2]),'b--')
                if b1check != 2:
                    plt.plot(np.arange(0,2,0.001),func(np.arange(0,2,0.001), fit_b1[0], fit_b1[1], fit_b1[2]),'r')
                if b2check != 2:
                    plt.plot(np.arange(0,2,0.001),func(np.arange(0,2,0.001), fit_b2[0], fit_b2[1], fit_b2[2]),'r--')
                ys = plt.gca().get_ylim()
                plt.plot(np.array([1.0,1.0]),np.array([ys[0]-50,ys[1]]),'k')
                plt.gca().set_ylim(ys)
                plt.xticks([0, 1, 2],['Root','Mean','Tip'])
                plt.legend(['$\\alpha_1$','$\\alpha_2$','$\\beta_1$','$\\beta_2$'],bbox_to_anchor=(1.05, 1), loc='upper left',
borderaxespad=0.)
            else:
                plt.title('Turbine Stage '+str(int(stage_array[i])))
                    # plt.plot(alpha_1[:,i],'b')
                if a1check != 2:
                    plt.plot(np.arange(0,2,0.001),func(np.arange(0,2,0.001), fit_a1[0], fit_a1[1], fit_a1[2]),'g',label='$\\alpha_2$')
                # plt.plot(alpha_2[:,i],'g')
                if a2check != 2:
                    plt.plot(np.arange(0,2,0.001),func(np.arange(0,2,0.001), fit_a2[0], fit_a2[1], fit_a2[2]),'g--',label='$\\alpha_3$'
)
                if b1check != 2:
                    plt.plot(np.arange(0,2,0.001),func(np.arange(0,2,0.001), fit_b1[0], fit_b1[1], fit_b1[2]),'m',label='$\\beta_2$')
                if b2check != 2:
                    plt.plot(np.arange(0,2,0.001),func(np.arange(0,2,0.001), fit_b2[0], fit_b2[1], fit_b2[2]),'m--',label='$\\beta_3$')
                ys = plt.gca().get_ylim()
                plt.plot(np.array([1.0,1.0]),np.array([ys[0]-50,ys[1]]),'k')
                plt.gca().set_ylim(ys)
                plt.xticks([0, 1, 2],['Root','Mean','Tip'])
                plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)









# backend2 = TurboMachineryComputationV2()
# optimalParams = backend2.fullturbine()
# print('\nTurbine $\Delta$ T: {}'.format(dT0_turb))
# print('\nStage $\Delta$ T: {} K'.format(T0s))
# print('\nTip Mach Numbers: {}'.format(M))
# print('No. Turb Stages = {}'.format(stages))
# stuff = TurboMachineryComputation()
# morestuff = stuff.fullcompressor()
```