# AE616 GAS DYNAMICS
# GROUP-8
# HW-2

**Q1) Fanno flow: Consider example 8.3 of Yahya's textbook that was discussed in class; it had a specified supersonic inlet, sonic outlet and pre-shock Mach number, and you were asked to find the lengths of the duct upstream and downstream of the shock. Now consider the related, and more practical, problem, where you are given the length of the duct and the inlet (supersonic) Mach number, and told that the outlet is sonic. Write a code to find the location of the normal shock in the duct, if it exists. Show that it reproduces the solution of the example problem. Also exercise your code by making up some other problems. Do you find situations where your code fails to produce any answer? Why?**

```
# length of our duct is l
# let fannosonic_length(M1,y,d,f) = L1
# if l < L1 ===> then we can never get sonic flow. as NS will only increase the length needed to reach sonic condition.
# if l = L1 ===> then we get sonic flow at exit. No NS needed
# if l > L1 ===> then we can get sonic flow, at exit through the aid of a NS.

# We can check using analytical expression that the length to achieve M=1 will only increase with the strength of NS.
# but this has a limit as the strongest NS we can get is NS @ inlet. any location other that inlet will have lower M so weaker shock.
# M1_prime = Mach_number_after_normal_shock(M1,y)
# fannosonic_length(M1_prime,y,d,f) = L2

# If l > L2
# then also no solution possible for normal shock as we cannot delay M=1 for longer length than this 'through a Normal Shock.'
# We might get M=1 but not 'through a Normal Shock.'
# so l range for getting a NS is (L1 , L2)
```

**CODE:**

```
import numpy as np
#----------------------------------------------------------------------------------------------------#
```

```
def fannosonic_length(m,y,d,f):
    # direct formula for fannosonic length l*
    # equation 3.107 from anderson
    z = m*m
    z = (1-z)/(y*z)   +   ( (y+1)/(2*y) ) * ( np.log(   ((y+1)*z) / (2 + (y-1)*z)   ) )
    z = (z*d) / (4*f)
    return z

#----------------------------------------------------------------------------------------------------#

def Mach_number_after_normal_shock(m,y):
    # direct formula for mach number after normal shock
    # equation 3.51 from anderson
    z = ( 1 + 0.5*(y-1)*m*m ) / ( y*m*m - 0.5*(y-1) )
    z = np.sqrt(z)
    return z

#----------------------------------------------------------------------------------------------------#
#                                            Inputs for the question

f = 0.003            # Fanning friction factor
d = 0.3              # Duct diameter
T1 = 400             # Inlet static temperature
P1 = 100000          # Inlet static pressure
y = 1.3              # Specific heat ratio
R = 287              # Gas constant
cp = y*R/(y-1)       # Specific heat at constant pressure

#----------------------------------------------------------------------------------------------------#

print("Specify the inlet conditions")
print("                            ")

M1 = float(input("Enter the Mach number M1: "))
if M1 <= 1:
    print("Supersonic flow is not specified. Hence Normal Shock is not possible.")
```

```
    print("Program Closed")
    exit()

l = float(input("Enter the duct length (in m): "))
if l <= 0:
    print("length cannot be negative or zero. Program Closed")
    exit()
elif l < fannosonic_length(M1,y,d,f):
    print("The length specfied is less than the minimum length necessary to reach Sonic state. Hence no solution is possible")
    exit()
elif l > fannosonic_length(  Mach_number_after_normal_shock(M1,y) , y , d ,f ):
    print("The length specfied is greater than the maximum possible length to reach Sonic state with the aid of a Normal Shock.")
    print("Solution may be possible but no solution with a Normal Shock is possible")
    print("Program Closed")
    exit()

#---------------------------------------------------------------------------------------------------#
# Mach Number at which NS occurs = M2
# correction = calculated length - l
# correction_factor = (calculated length - l) / (l)
# We will find new mach number at which NS occurs by using correction_factor.
# M2 = M2*(1 - correction_factor) + 1*(correction_factor)
# Slow convergence but STABLE
# The same is now implemented below

# initial guess, average of two
M2 = (M1 + 1)/2
l_star = fannosonic_length(M1,y,d,f)

# Iterate
# 1000 iter are usually enough. No need for any conditions
n = 0
while n < 1000:
    M2_prime = Mach_number_after_normal_shock(M2,y)
```

```
    duct_length = l_star - fannosonic_length(M2,y,d,f) + fannosonic_length(M2_prime,y,d,f)
    correction_factor = (duct_length- l ) / ( l )
    M2 = M2*(1-correction_factor) + 1*correction_factor
    n = n+1
    #print(M2)
    #print(l_star - fannosonic_length(M2,y,d,f))

#----------------------------------------------------------------------------------------------------#
mach_number = M2
location = (l_star - fannosonic_length(M2,y,d,f))
print("The Normal Shock occurs at ",location," meters from the inlet and the local mach number at that location is
", mach_number)
```

**OUTPUT:**

```
Specify the inlet conditions

Enter the Mach number M1: 2
Enter the duct length (in m): 10.8
The Normal Shock occurs at  5.350026369402681  meters from the inlet and the local mach number at that location is
1.4691532699582524
```

```
Q2) Interaction of shocks of opposite families: Write a code (and run it) to solve Example
6.5 of Oosthuizen's textbook (which has a mistake towards the end of the solution). Find and solve
similar examples from other textbooks/references to validate your code.
```

```
#-----------------------\
#                        \-------------------------------------
#                         \  2   /
#                          \    /
#       Flow                \  /                 Flow
#    ----------->            \/                 ----->
#        M1                  /\                  M4
#        T1                 /  \                 T4
#        P1                /    \                P4
#                         /  3   \
#                        /---------------------------------------
# ----------------------/
```

**CODE:**

```
import numpy as np
from scipy.interpolate import interp1d

#----------------------------------------------------------------------------------------------------#

def Theta(b,m,y):
    sign = b/np.abs(b)
    b = np.abs(b)
    # Directly available from θ-β-M relation
    t = ( ( m * np.sin(b) )**2 - 1 ) / ( m*m*( y + np.cos(2*b) ) + 2)
    t = 2*t/np.tan(b)
    t = np.arctan(t)
    return sign*np.abs(t)

#----------------------------------------------------------------------------------------------------#
```

```python
def Beta(t,M,y):
    # Plot the grid for θ-β-M curve
    beta_grid = np.linspace( np.arcsin(1/M) , 0.25*np.pi , 1000 )
    theta_grid = Theta(beta_grid,M,y)
    # Interpolate the curve to find β for a given θ
    beta_for_theta = interp1d(theta_grid, beta_grid)
    if t == 0:
        sign = 1
    else:
        sign = t/np.abs(t)
    t = np.abs(t)
    return sign*beta_for_theta(t)

#----------------------------------------------------------------------------------------------------#

def Mach_number_after_normal_shock(m,y):
    # equation 3.51 from anderson
    m2 = ( 1 + 0.5*(y-1)*m*m ) / ( y*m*m - 0.5*(y-1) )
    m2 = np.sqrt(m2)
    return m2

#----------------------------------------------------------------------------------------------------#

def Mach_number_after_oblique_shock(t,b,m,y):
    m = np.abs(m)
    m2 = m * np.sin(b)
    m2 = Mach_number_after_normal_shock(m2,y)
    m2 = m2/np.sin(b-t)
    return np.abs(m2)

#----------------------------------------------------------------------------------------------------#

def Pressure_after_normal_shock(p,m,y):
    # equation 3.57 from anderson
    m = np.abs(m)
```

```
    p2 = 2 * y * ( m*m - 1 ) / ( y + 1 )
    p2 = p*(1 + p2)
    return p2

#---------------------------------------Define the inputs here--------------------------------------#

y = 1.4
M1 = 3
T1 = 263
P1 = 30000
t12 = -4 * (np.pi/180)
t13 = 3 * (np.pi/180)
Pt1 = P1 * (1 + 0.5*(y-1)*M1*M1)**(y/y-1)

#----------------------------------------------------------------------------------------------------#


b12 = Beta(t12,M1,y)
P2 = Pressure_after_normal_shock ( P1 , M1*np.sin(b12) , y )
M2 = Mach_number_after_oblique_shock(t12,b12,M1,y)
print("M2 is",(M2))
print("P2 is",(P2))

b13 = Beta(t13,M1,y)
P3 = Pressure_after_normal_shock ( P1 , M1*np.sin(b13) , y )
M3 = Mach_number_after_oblique_shock(t13,b13,M1,y)
print("M3 is",(M3))
print("P3 is",(P3))

#----------------------------------------------------------------------------------------------------#

pressure_difference = []
delta = []
theta = -3*np.pi/180

while theta <= 3*np.pi/180:
```

```
    t24 = theta - t12
    b24 = Beta(t24,M2,y)
    P42 = Pressure_after_normal_shock ( P2 , M2*np.sin(b24) , y )

    t34 = theta - t13
    b34 = Beta(t34,M3,y)
    P43 = Pressure_after_normal_shock ( P3 , M3*np.sin(b34) , y )

    delta.append(theta)
    pressure_difference.append(P43-P42)
    theta = theta + 0.01*(np.pi/180)

#----------------------------------------------------------------------------------------------------#

pressure_difference_vs_theta = interp1d(pressure_difference,delta)
Final_flow_angle = pressure_difference_vs_theta(0)

t24 = Final_flow_angle - t12
b24 = Beta(t24,M2,y)
P42 = Pressure_after_normal_shock ( P2 , M2*np.sin(b24) , y )

Final_flow_angle = Final_flow_angle*180/np.pi
print("The  final pressure downstream of shock interaction is : ",(P42)," Pa. ")
print("The  final flow direction angle is: ",(Final_flow_angle)," degrees. ")
print("Note: negative angle indicates clockwise rotation wrt x axis.")
print(" ")
```

**OUTPUT:**

```
M2 is 2.798812188480194
P2 is 40566.19015027438
M3 is 2.848234514731154
P3 is 37683.56325575157
The final pressure downstream of shock interaction is :  50293.86464076658  Pa.
The final flow direction angle is:  -0.9995253374482028  degrees.
Note: negative angle indicates clockwise rotation wrt x axis.
```

**Q3) Shock expansion theory: Write a code (and run it) to solve Example 4.15 of Anderson (2003) involving calculation of supersonic flow over a flat plate using shock-expansion theory. Calculate the cl and cd. Also, calculate the precise slip-line angle (see Fig. 4.37 of the book). Find similar examples from other textbooks/references to validate your code.**

**CODE:**

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
import warnings

# Known parameters
m1 = float(input("Free Stream Mach M1: "))
aoa = float(input("Angle of Attack (in degrees): "))
y = 1.4      #float(input("Gamma (y): ")) #y is Gamma
theta = np.radians(aoa)  # aoa to radians

#-----------------------------------------------------------------------------------------------------#
# EXPANSION FAN REGION (1-3)

def prandtl_meyer_angle(m, y):  # Prandtl-Meyer function to calculate ν(Mach)
    G = (y + 1) / (y - 1)
    term1 = np.sqrt(G) * np.arctan(np.sqrt((m**2 - 1) / G))
    term2 = np.arctan(np.sqrt(m**2 - 1))
    return term1 - term2

v1 = prandtl_meyer_angle(m1, y) # Find v1 from m1
v2 = v1 + theta  # v2 = v1 + θ

def solve_mach(m, y, v):# Find M2 from V2
    return prandtl_meyer_angle(m, y) - v

m2_guess = m1 + 1  # Reasonable initial guess for m2
m2 = fsolve(solve_mach, m2_guess, args=(y, v2)) # Solve for Mach number m2 using fsolve
```

```python
def free_stream_pressure_ratio(m, y): #Free stream pressure ratio for a given Mach number
    PR = (1 + (((y - 1) / 2) * (m**2))) ** (y / (y - 1))
    return PR

# Calculate pressure ratios before and after the expansion fan
PR01 = free_stream_pressure_ratio(m1, y)
PR02 = free_stream_pressure_ratio(m2[0], y)

p21 = PR01 / PR02 # Overall pressure ratio across the expansion fan

#----------------------------------------------------------------------------------------------------#
# SHOCK WAVE REGION (1-3)

# Function to solve theta-beta-Mach relationship for shock waves
def theta_beta_mach(B, m, y, theta):
    B = np.radians(B)  # Convert beta to radians for computation
    term1 = (m**2 * np.sin(B)**2 - 1)
    term2 = (m**2 * (y + np.cos(2 * B)) + 2)
    lhs = np.tan(theta)
    rhs = 2 * (1 / np.tan(B)) * (term1 / term2)
    return lhs - rhs

B_guess = 30  # reasonable initial guess for Beta

# Solve for beta using fsolve
B_Sol = fsolve(theta_beta_mach, B_guess, args=(m1, y, theta))

def upstream_normal_mach(m, B):# Finding Normal Mach number Mn
    β = np.radians(B)  # Convert beta to radians
    Mn1 = m * np.sin(β)
    return Mn1

def downstream_mach(m1, B, y):
    β = np.radians(B)
    Mn1 = m1 * np.sin(β)  # Normal component of upstream Mach number
```

```
    Mn2 = np.sqrt((1 + ((y - 1) / 2) * Mn1 ** 2) / (y * Mn1 ** 2 - (y - 1) / 2))
    M2 = Mn2 / np.sin(β - theta)  # Downstream Mach number after shock
    return M2

Mn1 = upstream_normal_mach(m1, B_Sol[0])
m3 = downstream_mach(m1, B_Sol[0], y)

# Function to calculate pressure ratio across the shock
def shock_pressure_ratio(Mn, y):
    PR = 1 + ((2 * y) / (y + 1)) * (Mn**2 - 1)
    return PR

p31 = shock_pressure_ratio(Mn1, y)

#----------------------------------------------------------------------------------------------------#
# Calculating Cl & Cd
s = np.tan(theta)  # Cd/Cl = tan(AOA)
cl = (2 / (y * (m1**2)) * (np.cos(theta))) * (p31 - p21)  # Lift Coefficient
cd = cl * s  # Drag Coefficient

print(f"Lift Coefficient (Cl): {cl:.4f}")
print(f"Drag Coefficient (Cd): {cd:.4f}")

#----------------------------------------------------------------------------------------------------#

#Iteration Fuction
def iteration(sl):
    # SLIPLINE REGION (4-5)
    def theta_for_slipline(aoa,sl):
        tsl = aoa + sl
        return tsl

    tsld = theta_for_slipline(aoa,sl)
    tsl = np.radians(tsld)

    #REGION 2-4
```

```
    B_Sol4 = fsolve(theta_beta_mach, B_guess, args=(m2, y, tsl))
    Mn2 = upstream_normal_mach(m2, B_Sol4[0])

    p42 = shock_pressure_ratio(Mn2, y)
    p41 = p42*p21

    #REGION 3-5
    v3 = prandtl_meyer_angle(m3, y) # Find v1 from m1
    v5 = v3 + tsl  # v5 = v3 + tsl
    m5 = fsolve(solve_mach, 3, args=(y, v5))

    PR05 = free_stream_pressure_ratio(m5, y)
    PR03 = free_stream_pressure_ratio(m3, y)

    p51 = (PR03*p31)/(PR05)
    PD = p51-p41
    return PD[0] if isinstance(PD, np.ndarray) else PD

#----------------------------------------------------------------------------------------------------#
#BY USING GRAPHICAL METHOD

# Vectorize the iteration function so it works element-wise on arrays
vectorized_iteration = np.vectorize(iteration)

# Create the sl values ranging from -1 to 2
sl = np.linspace(-1, 2, 100)  # sl is a NumPy array

# Compute PD values using the vectorized iteration function
PD = vectorized_iteration(sl)

# Now let's define a function that finds sl when PD = 0
def find_sl_at_pd_zero():
    # Define a function that gives PD - we want this to be 0
    def objective(sl):
        return iteration(sl)  # PD(sl) needs to be 0
```

```
    # Use fsolve to find sl where PD = 0, initial guess of 2.5 (adjust if necessary)
    sl_solution = fsolve(objective, 2.5)
    return sl_solution

# Find sl where PD = 0
sl_at_pd_zero = find_sl_at_pd_zero()
print(f"The value of sl at PD = 0 is: {sl_at_pd_zero[0]}")

# Plot PD vs sl
plt.plot(sl, PD, label="PD vs sl")

# Draw a short horizontal line at PD = 0 near sl_at_pd_zero
plt.plot([sl[0], sl_at_pd_zero[0]], [0, 0], color='gray', linestyle='--', linewidth=1)

# Draw a short vertical line at sl_at_pd_zero near PD = 0
plt.plot([sl_at_pd_zero[0], sl_at_pd_zero[0]], [0, min(PD)], color='gray', linestyle='--', linewidth=1)

plt.text(sl_at_pd_zero[0], plt.ylim()[0], f'{sl_at_pd_zero[0]:.2f}',
         ha='center', va='bottom', color='red', fontsize=10)
# Label the axes
plt.xlabel("sl")
plt.ylabel("PD")

# Add a title
plt.title("Plot of PD vs sl with Lines at PD=0 and sl at PD=0")

# Show a legend
plt.legend()

# Display the plot
plt.show()
#----------------------------------------------------------------------------------------------------#
"""#BY USING STRAIGHT LINE EQUATION

#1 Point
sl1=0
```

```
PD1 = iteration(sl1)

#2nd Point
sl2=1
PD2 = iteration(sl2)


#STRAIGHT LINE EQUATION
PD=0
sl = sl1 + (((sl2-sl1)*(PD-PD1))/(PD2-PD1))
print(f"SLIP LINE ANGLE : {sl}") """

#----------------------------------------------------------------------------------------------------#
"""
#BY USING SECANT METHOD
def secant_method(func, x0, x1, tol=1e-6, max_iter=100):
    for i in range(max_iter):
        # Evaluate function at the current guesses
        f_x0 = func(x0)
        f_x1 = func(x1)

        # Secant formula to update the guess
        x2 = x1 - f_x1 * (x1 - x0) / (f_x1 - f_x0)

        # Check for convergence
        if abs(x2 - x1) < tol:
            return x2

        # Update for the next iteration
        x0, x1 = x1, x2

    raise ValueError("Secant method did not converge")

# Define the objective function for the Secant method
def objective(sl):
    return iteration(sl)  # PD(sl) needs to be 0
```

```
# Use Secant method to find sl where PD = 0
initial_guess_1 = 1.0  # First initial guess
initial_guess_2 = 2.0  # Second initial guess
sl_at_pd_zero_secant = secant_method(objective, initial_guess_1, initial_guess_2)

# Output the result
print(f"The value of sl at PD = 0 using the Secant method is: {sl_at_pd_zero_secant:.6f}") """
```

**OUTPUT:**

```
Free Stream Mach M1: 3
Angle of Attack (in degrees): 20
Lift Coefficient (Cl): 0.5387
Drag Coefficient (Cd): 0.1961
The value of sl at PD = 0 is: 0.8925983677948616
```