

## Homework 3

### Number 2

Upon using the Falkner-Skan laminar boundary layer equations, the following plots were made for different values of  $\alpha$ . The figure below shows the change of  $f'(\eta)$  or  $u/U_\infty$  with respect to  $\eta$ . The domain used is only until  $\eta = 10$  and is because if the shear stress was right, the solution converges fast to 1. Other reasons are that the equation is sensitive to the initial condition, so marching along will produce errors which can grow fast that will not reflect the expected behavior. The shear stress was found by using a binary split guess shooting method which gives the an accurate guess within 50 iterations. In order to increase the speed of the code and it's accuracy, the shooting method will stop evaluating the ordinary differential equation (ODE) if  $f'$  is above a threshold over 1. After finding the necessary shear stress, the shape factor was calculated by integrating along the entire domain by the use of a left Reimann sum. This doesn't affect the accuracy that much due to the chosen  $\Delta\eta = 0.001$ .

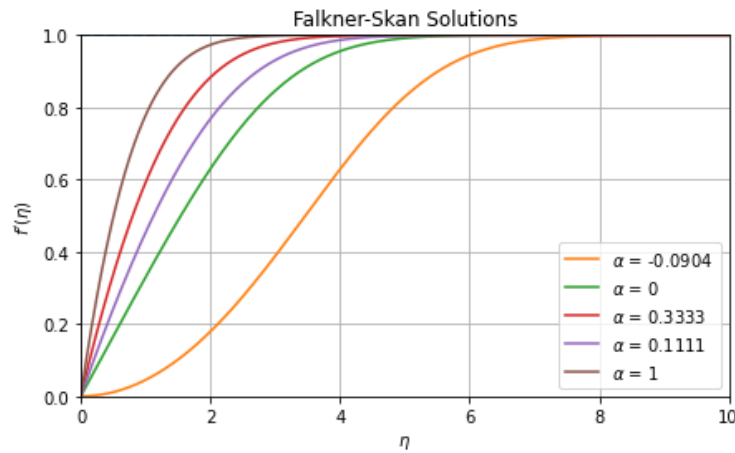


Figure 1: Numerical solution of Falker-Skan in terms of  $\eta$

$\alpha$	$f''(0)$	$H$
-0.0904	0	4.0282
0	0.3321	2.5931
1/3	.7574	2.2893
1/9	0.5118	2.409
1	1.2326	2.2334

Table 1: Vertex Coordinates of the y-direction prism

In order to match the figure 10.8 in Kundu's textbook, it was noted that the  $x$  scale isn't along  $\eta$ . Therefore it was necessary to scale it properly. It can be seen from the figure below that the produced plot looked similar to the textbook figures.

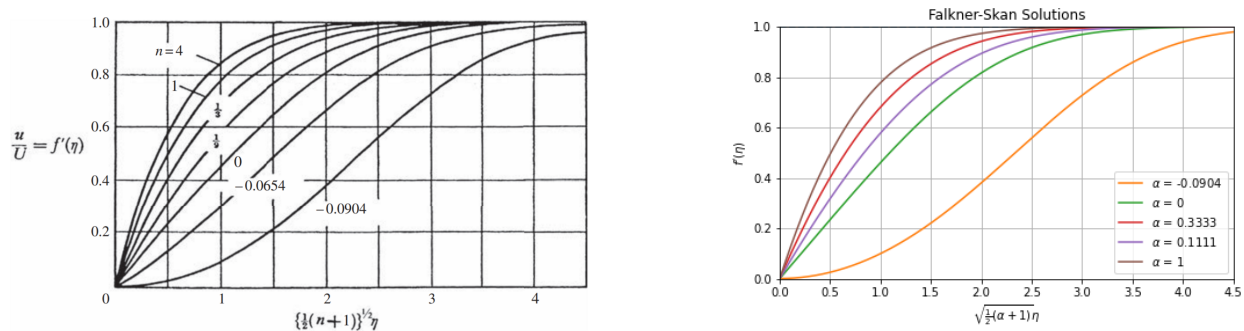


Figure 2: Comparison of Figure 10.8 from KCD and Scaled Numerical solution of Falkner-Skan

### Number 3 c)

To find the velocity profile of the Non-newtonian boundary layer, a different ODE was solved. By using the same shooting method, a value of  $f'''(0) = 0.619$  was found. Afterwards the velocity profile was compared with the Newtonian Blasius boundary layer. It can be seen that the velocity increases faster along the  $y$  axis.

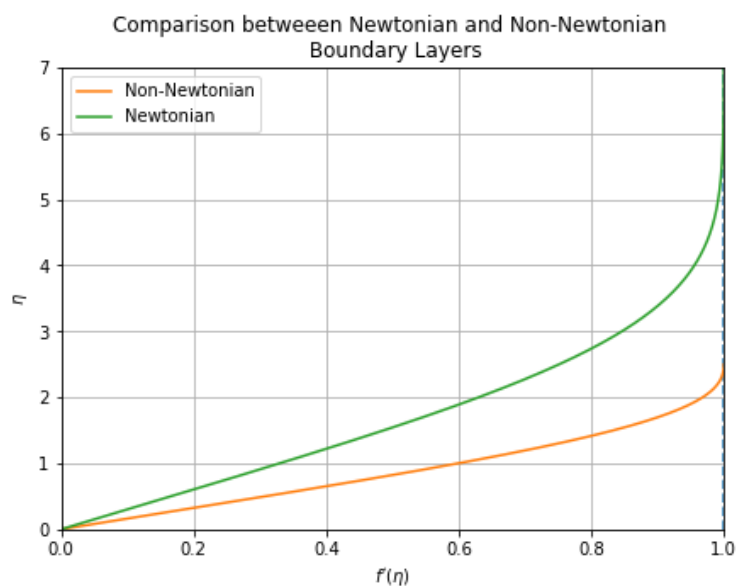


Figure 3: Newtonian and Non-Newtonian Boundary Layers

# Appendix)

## Python Code for Problems 2

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Tue Apr 6 21:15 2023
4
5 @author: jjser
6 Justine John A. Serdoncillo
7 AEM 8202 Fluids II
8 Homework3_P2_v4
9     - No more functions
10    - Summarized all of the plots
11    - Working shooting method
12    - shapeFactor()
13    - added shooting
14 """
15 import numpy as np
16 import matplotlib.pyplot as plt
17
18 # %%
19 """
20     Personal Created Functions
21     - g1, g2 and g3 represents 3 linear ODEs
22     - rk4 is the Runge-Kutta 4th Order Method
23     - shoot is the shooting method needed
24     - shapeFactor()
25 """
26
27 # Define personal created functions
28 def g1(f, df, ddf, a, n):
29     """Returns the value of the first ODE"""
30     return df
31
32 def g2(f, df, ddf, a, n):
33     """Returns the value of the second ODE"""
34     return ddf
35
36 def g3(f, df, ddf, a, n):
37     """Returns the value of the third ODE"""
38     return -0.5 * (a+1) * f * ddf - a * (1 - df**2)
39
40 def rk4(ni, nf, f0, df0, ddf0, dn, a):
41     """
42     Returns the numerical solutions for the Falkner-Skan equations using
43     the fourth-order Runge-Kutta method.
44     """
45     N = int((nf - ni) / dn)
46     n = np.linspace(ni, nf, N+1)
47     scaled = ( ( 0.5 * (a+1) )**(1/2) ) * n
48     F = np.zeros(len(n))
49     dF = np.zeros(len(n))
50     ddF = np.zeros(len(n))
51     F[0] = f0
52     dF[0] = df0
53     ddF[0] = ddf0
54     for i in range(len(n)-1):
55         k01 = g1(F[i], dF[i], ddF[i], a, n[i])
56         k02 = g2(F[i], dF[i], ddF[i], a, n[i])
57         k03 = g3(F[i], dF[i], ddF[i], a, n[i])
58
59         k11 = g1(F[i]+dn/2*k01, dF[i]+dn/2*k02, ddF[i]+dn/2*k03, a, n[i]+dn/2)
60         k12 = g2(F[i]+dn/2*k01, dF[i]+dn/2*k02, ddF[i]+dn/2*k03, a, n[i]+dn/2)
61         k13 = g3(F[i]+dn/2*k01, dF[i]+dn/2*k02, ddF[i]+dn/2*k03, a, n[i]+dn/2)
62
63         k21 = g1(F[i]+dn/2*k11, dF[i]+dn/2*k12, ddF[i]+dn/2*k13, a, n[i]+dn/2)
64         k22 = g2(F[i]+dn/2*k11, dF[i]+dn/2*k12, ddF[i]+dn/2*k13, a, n[i]+dn/2)
65         k23 = g3(F[i]+dn/2*k11, dF[i]+dn/2*k12, ddF[i]+dn/2*k13, a, n[i]+dn/2)
66
67         k31 = g1(F[i]+dn*k21, dF[i]+dn*k22, ddF[i]+dn*k23, a, n[i]+dn)
```

```

68     k32 = g2(F[i]+dn*k21, dF[i]+dn*k22, ddF[i]+dn*k23, a, n[i]+dn)
69     k33 = g3(F[i]+dn*k21, dF[i]+dn*k22, ddF[i]+dn*k23, a, n[i]+dn)
70
71     F[i+1] = F[i] + dn/6 * (k01 + 2*k11 + 2*k21 + k31)
72     dF[i+1] = dF[i] + dn/6 * (k02 + 2*k12 + 2*k22 + k32)
73     ddF[i+1] = ddF[i] + dn/6 * (k03 + 2*k13 + 2*k23 + k33)
74
75     if dF[i+1] > 3:
76         #print("bruh you high messed up")
77         dF[i+1] = 3
78         break
79     elif dF[i+1] < -1:
80         #print("bruh you low messed up")
81         dF[i+1] = 0
82         break
83     return n, F, dF, ddF, scaled
84
85 def shoot(ni, nf, f0, df0, dn, a, up, down, show = False, showShow = True):
86     ite = 0
87     MAXITE = 100
88     bad = True
89     TOL = 10E-14
90     target = 1
91     while bad and ite < MAXITE:
92         skrt = (up + down)/2
93         ite += 1
94         n, F, dF, ddF, s = rk4(ni, nf, f0, df0, skrt, dn, a)
95         if np.max(dF) > target + TOL:
96             down = skrt
97         elif np.max(dF) < target - TOL:
98             up = skrt
99         else:
100             bad = False
101         if show:
102             print(f"I shot with {skrt}")
103             print(f"It's been {ite} days since cookies")
104             print(f"This is my {np.around(np.max(dF),6)} power \n ")
105             print(f"For the last time {np.around(dF[-1],6)} ")
106         if showShow:
107             plt.plot(n, dF, label=str(ite))
108     dff0 = skrt
109     if show:
110         plt.title(f"This is the {ite} time bro")
111         plt.legend()
112     return dff0, ite
113
114 def shapeFactor(n, dF):
115     deltaStar = 0
116     theta = 0
117     # Left Reimann Sum
118     for i in range(len(n)-1):
119         dn = n[i+1] - n[i]
120         deltaStar += (1 - dF[i]) * dn
121         theta += dF[i] * (1 - dF[i]) * dn
122     H = deltaStar/theta
123     return H
124
125 # %%
126 """
127 Falkner-Skan Similarity Solutions of the Laminar Boundary-Layer Equations
128 - alphas = [-0.0904, 0, 1/3, 1/9, 1]
129 - fppp + (a+1)/2 * f * fpp - a * fp**2 + a = 0
130   - Given: f(0) = 0, fp(0) = 0
131   - Shoot with: fpp(0) = idk
132   - Target: fp(inf) = 1
133 """
134
135 # Define problem-specific variables
136 f0 = 0          # initial value of the dependent variable f
137 df0 = 0         # initial value of the first derivative of f
138 ni = 0.0        # initial value of the independent variable n
139 nf = 10.0       # final value of the independent variable n

```

```

140 dn = 0.001      # step size
141 alphas = [-0.0904, 0, 1/3, 1/9, 1] # constant parameter of the Falkner-Skan equations
142
143 # Calculate wall shear stress using shooting method
144 #ddf0s = np.zeros(len(alphas))
145 #for i in range(len(alphas)-1):
146 #    ddf0s[i+1], ite = shoot(ni, nf, f0, df0, dn, alphas[i+1], 0, 2)
147 ddf0s = [0, 0.3321, 0.7574, 0.5118, 1.2326] # initial value of the second derivative of f
148
149 # Create two subplots to display the solutions
150 fig, ax = plt.subplots(figsize=(7,4))
151 fig1, ax1 = plt.subplots(figsize=(7,4))
152
153 # Set the titles and axis labels for the plots
154 ax.set_title('Falkner-Skan Solutions')
155 ax.set_xlabel('$ \eta $')
156 ax.set_ylabel('$ f''(\eta) $')
157 ax1.set_title('Falkner-Skan Solutions')
158 ax1.set_xlabel('$ \sqrt{\frac{1}{2}} (\alpha + 1) \eta $')
159 ax1.set_ylabel('$ f''(\eta) $')
160
161 # Set the x and y limits for the plots
162 ax.set_xlim([0,10])
163 ax.set_ylim([0,1.0])
164 ax1.set_xlim([0,4.5])
165 ax1.set_ylim([0,1.0])
166
167 # Plot a horizontal dashed line at y=1 to show the target value
168 ax.plot(np.linspace(0,100,100,True), np.ones(100), '--')
169 ax1.plot(np.linspace(0,100,100,True), np.ones(100), '--')
170
171 # Loop over the different values of alpha to solve the Falkner-Skan equations
172 for i in range(len(alphas)):
173     # Solve the Falkner-Skan equations using the Runge-Kutta method
174     n, F, dF, ddF, scaled = rk4(ni, nf, f0, df0, ddf0s[i], dn, alphas[i])
175     H = shapeFactor(n, dF)
176
177     # Print the value of alpha and f''(0) for this solution
178     print(f"For alpha = {np.around(alphas[i],4)}, ddf(0) = {np.around(ddf0s[i],4)}")
179     # Print the value of the shape factor
180     print(f"The corresponding shape factor is {np.around(H,4)}")
181
182     # Add a label for the current value of alpha to the plot
183     label = "$\alpha$ = " + str(np.around(alphas[i],4))
184
185     # Plot the solution for f''(eta) on both subplots
186     ax.plot(n, dF, label=label)
187     ax1.plot(scaled, dF, label=label)
188
189 # Add a legend and gridlines to both subplots
190 ax.grid()
191 ax.legend(loc='lower right')
192 ax1.grid()
193 ax1.legend(loc='lower right')
194
195 # %%
196 """
197     Shooting
198 """
199 for i in range(len(alphas)-1):
200     yay, skrt = shoot(0, 10, 0, 0, 0.001, i+1, 0, 2, True)
201     print(f"For alpha = {np.around(alphas[i],4)} \n I got ddF0 = {yay} and it only took me {
202         skrt}")
203
204 ## using (0,2) and 10E-14, (-1, 3)
205 # 0 -> 0.332057337203 in 43 days
206 # 1 -> 1.232587656820 in 45 days
207 # 1/3 -> 0.7574475807215 in 47 days
208 # 1/9 -> 0.511842057835 in 45 days
209 # -0.0904 -> 0

```

## Python Code for Problems 3

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr  3 21:44:28 2023
4
5 @author: jjser
6 Justine John A. Serdoncillo
7 AEM 8202 Fluids II
8 Homework 3 Problem 3 v2
9     - Solve the Non-Newtonian Similarity ODE
10    - Working shooting method
11    - added shooting
12 """
13 import numpy as np
14 import matplotlib.pyplot as plt
15 import warnings
16
17 # %%
18 """
19     Personal Created Functions
20     - g1, g2 and g3 represents 3 linear ODEs
21     - rk4 is the Runge-Kutta 4th Order Method
22     - shoot is the shooting method needed
23 """
24 def g1(f, df, ddf, n):
25     """Returns the value of the first ODE"""
26     return df
27
28 def g2(f, df, ddf, n):
29     """Returns the value of the second ODE"""
30     return ddf
31
32 def g3(f, df, ddf, n):
33     """Returns the value of the third ODE"""
34     try:
35         with warnings.catch_warnings(record=True) as w:
36             warnings.simplefilter("always")
37             result = -f * ddf / (ddf**(1/2))
38             if w:
39                 print(f"Warning occurred with: \n f={np.around(f,6)},\n df={np.around(df,6)}
40                 ,\n ddf={np.around(ddf,6)},\n n={np.around(n,6)}")
41                 print(f"Warning message: {w[0].message}")
42     except ZeroDivisionError:
43         print(f"ZeroDivisionError occurred with \n f={np.around(f,6)},\n df={np.around(df
44         ,6)},\n ddf={np.around(ddf,6)},\n n={np.around(n,6)}")
45         result = 0 # or set a suitable value when a division by zero occurs
46     return result
47
48 def rk4(ni, nf, f0, df0, ddf0, dn):
49     N = int((nf - ni) / dn)
50     n = np.linspace(ni, nf, N+1)
51     F = np.ones(len(n))
52     dF = np.ones(len(n))
53     ddF = np.ones(len(n))
54     F[0] = f0
55     dF[0] = df0
56     ddF[0] = ddf0
57     for i in range(len(n)-1):
58         k01 = g1(F[i], dF[i], ddF[i], n[i])
59         k02 = g2(F[i], dF[i], ddF[i], n[i])
60         k03 = g3(F[i], dF[i], ddF[i], n[i])
61
62         k11 = g1(F[i]+dn/2*k01, dF[i]+dn/2*k02, ddF[i]+dn/2*k03, n[i]+dn/2)
63         k12 = g2(F[i]+dn/2*k01, dF[i]+dn/2*k02, ddF[i]+dn/2*k03, n[i]+dn/2)
64         k13 = g3(F[i]+dn/2*k01, dF[i]+dn/2*k02, ddF[i]+dn/2*k03, n[i]+dn/2)
65
66         k21 = g1(F[i]+dn/2*k11, dF[i]+dn/2*k12, ddF[i]+dn/2*k13, n[i]+dn/2)
67         k22 = g2(F[i]+dn/2*k11, dF[i]+dn/2*k12, ddF[i]+dn/2*k13, n[i]+dn/2)
68         k23 = g3(F[i]+dn/2*k11, dF[i]+dn/2*k12, ddF[i]+dn/2*k13, n[i]+dn/2)
69
70         k31 = g1(F[i]+dn*k21, dF[i]+dn*k22, ddF[i]+dn*k23, n[i]+dn)
```

```

69     k32 = g2(F[i]+dn*k21, dF[i]+dn*k22, ddF[i]+dn*k23, n[i]+dn)
70     k33 = g3(F[i]+dn*k21, dF[i]+dn*k22, ddF[i]+dn*k23, n[i]+dn)
71
72     F[i+1] = F[i] + dn/6 * (k01 + 2*k11 + 2*k21 + k31)
73     dF[i+1] = dF[i] + dn/6 * (k02 + 2*k12 + 2*k22 + k32)
74     ddF[i+1] = ddF[i] + dn/6 * (k03 + 2*k13 + 2*k23 + k33)
75
76     if dF[i+1] > 3:
77         #print("bruh you high messed up")
78         dF[i+1] = 3
79         break
80     elif dF[i+1] < -1:
81         #print("bruh you low messed up")
82         dF[i+1] = 0
83         break
84
85     return n, F, dF, ddF
86
87 def shoot(ni, nf, f0, df0, dn, up, down, show = False):
88     ite = 0
89     MAXITE = 100
90     bad = True
91     TOL = 10E-14
92     target = 1
93     while bad and ite < MAXITE:
94         skrt = (up + down)/2
95         ite += 1
96         n, F, dF, ddF = rk4(ni, nf, f0, df0, skrt, dn)
97         if np.max(dF) > target + TOL:
98             down = skrt
99         elif np.max(dF) < target - TOL:
100             up = skrt
101         else:
102             bad = False
103         if show:
104             print(f"I shot with {skrt}")
105             print(f"It's been {ite} days since cookies")
106             print(f"This is my {np.around(np.max(dF),6)} power \n ")
107             #print(f"For the last time {np.around(dF[-1],6)} ")
108             plt.plot(n, dF, label=str(ite))
109     ddf0 = skrt
110     if show:
111         plt.title(f"This is the {ite} time bro")
112         plt.legend()
113     return ddf0, ite
114
115
116 # %%
117 """
118     Non-Newtonian Boundary Layer Profile
119     - dddf * ddf^(1/2) + f * ddf = 0
120 """
121
122 # Define problem-specific variables
123 f0 = 0          # initial value of the dependent variable f
124 df0 = 0         # initial value of the first derivative of f
125 ni = 0.0        # initial value of the independent variable n
126 nf = 2.5        # final value of the independent variable n
127 dn = 0.001      # step size
128
129 # Calculate wall shear stress using shooting method
130 #ddf0, ite = shoot(ni, nf, f0, df0, dn, 0.5, 0.75)
131 ddf0 = 0.619    # initial value of the second derivative of f
132
133 # Solve the Falkner-Skan equations using the Runge-Kutta method
134 n, F, dF, ddF = rk4(ni, nf, f0, df0, ddf0, dn)
135 print(f"For Non-Newtonian Boundary Layers, ddf(0) = {np.around(ddf0,4)}")
136
137 # Create a plot to display the solution
138 fig, ax = plt.subplots(figsize=(7,5))
139
140 # Set the title and axis labels for the plot

```

```

141 ax.set_title('Comparison between Newtonian and Non-Newtonian \n Boundary Layers')
142 ax.set_ylabel('$ \eta $')
143 ax.set_xlabel('$ f'(\eta) $')
144
145 # Set the x and y limits for the plot
146 ax.set_ylim([0,7.0])
147 ax.set_xlim([0,1.0])
148
149 # Plot a horizontal dashed line at y=1 to show the target value
150 ax.plot(np.ones(100), np.linspace(0,100,100,True), '--')
151
152 # Plot the solution for f'(\eta)
153 ax.plot(dF, n, label='Non-Newtonian')
154
155 # Add gridlines to the plot
156 ax.grid()
157
158 # %%
159 """
160     Blasius Newtonian Solution Taken from Problem 2
161 """
162 # Define personal created functions
163 def g1(f, df, ddf, a, n):
164     """Returns the value of the first ODE"""
165     return df
166
167 def g2(f, df, ddf, a, n):
168     """Returns the value of the second ODE"""
169     return ddf
170
171 def g3(f, df, ddf, a, n):
172     """Returns the value of the third ODE"""
173     return -0.5 * (a+1) * f * ddf - a * (1 - df**2)
174
175 def rk4(ni, nf, f0, df0, ddf0, dn, a):
176     """
177     Returns the numerical solutions for the Falkner-Skan equations using
178     the fourth-order Runge-Kutta method.
179     """
180     N = int((nf - ni) / dn)
181     n = np.linspace(ni, nf, N+1)
182     scaled = ( ( 0.5 * (a+1) )**(1/2) ) * n
183     F = np.zeros(len(n))
184     dF = np.zeros(len(n))
185     ddF = np.zeros(len(n))
186     F[0] = f0
187     dF[0] = df0
188     ddF[0] = ddf0
189     for i in range(len(n)-1):
190         k01 = g1(F[i], dF[i], ddF[i], a, n[i])
191         k02 = g2(F[i], dF[i], ddF[i], a, n[i])
192         k03 = g3(F[i], dF[i], ddF[i], a, n[i])
193
194         k11 = g1(F[i]+dn/2*k01, dF[i]+dn/2*k02, ddF[i]+dn/2*k03, a, n[i]+dn/2)
195         k12 = g2(F[i]+dn/2*k01, dF[i]+dn/2*k02, ddF[i]+dn/2*k03, a, n[i]+dn/2)
196         k13 = g3(F[i]+dn/2*k01, dF[i]+dn/2*k02, ddF[i]+dn/2*k03, a, n[i]+dn/2)
197
198         k21 = g1(F[i]+dn/2*k11, dF[i]+dn/2*k12, ddF[i]+dn/2*k13, a, n[i]+dn/2)
199         k22 = g2(F[i]+dn/2*k11, dF[i]+dn/2*k12, ddF[i]+dn/2*k13, a, n[i]+dn/2)
200         k23 = g3(F[i]+dn/2*k11, dF[i]+dn/2*k12, ddF[i]+dn/2*k13, a, n[i]+dn/2)
201
202         k31 = g1(F[i]+dn*k21, dF[i]+dn*k22, ddF[i]+dn*k23, a, n[i]+dn)
203         k32 = g2(F[i]+dn*k21, dF[i]+dn*k22, ddF[i]+dn*k23, a, n[i]+dn)
204         k33 = g3(F[i]+dn*k21, dF[i]+dn*k22, ddF[i]+dn*k23, a, n[i]+dn)
205
206         F[i+1] = F[i] + dn/6 * (k01 + 2*k11 + 2*k21 + k31)
207         dF[i+1] = dF[i] + dn/6 * (k02 + 2*k12 + 2*k22 + k32)
208         ddF[i+1] = ddF[i] + dn/6 * (k03 + 2*k13 + 2*k23 + k33)
209     return n, F, dF, ddF, scaled
210
211 # Define problem-specific variables
212 f0 = 0 # initial value of the dependent variable f

```



```

213 df0 = 0          # initial value of the first derivative of f
214 ni = 0.0         # initial value of the independent variable n
215 nf = 10.0        # final value of the independent variable n
216 dn = 0.001       # step size
217 a = 0            # constant parameter of the Falkner-Skan equations
218
219 ddf0 = 0.3321     # initial value of the second derivative of f
220
221 # Solve the Falkner-Skan equations using the Runge-Kutta method
222 n, F, dF, ddF, scaled = rk4(ni, nf, f0, df0, ddf0, dn, a)
223 print(f"For Newtonian Boundary Layers, ddf(0) = {np.around(ddf0,4)}")
224
225 # Plot the solution for f'(eta)
226 ax.plot(dF, n, label='Newtonian')
227
228 ax.legend(loc='upper left')
229
230
231 # %%
232 """
233     Shooting
234 """
235 yay, skrt = shoot(0, 2.5, 0, 0, 0.001, 0.5, 0.75, True)
236 print(f"After shooting \n I got ddF0 = {yay} and it only took me {skrt}")
237 ## using (0.5,0.75) and 10E-14, (-1, 3)
238 # -> 0.618964749629 in 41 days

```