

MECH597 Assignment 2

Python Code Documentation

Optimization Algorithms & Brequet Range Analysis

October 13, 2025

Contents

| | | |
|----------|---|----------|
| 1 | Line Search Optimizers | 2 |
| 1.1 | Source Code: line_search_optimizers.py | 2 |
| 2 | Brequet Range Optimizer | 7 |
| 2.1 | Source Code: brequet_range_optimizer.py | 7 |

1 Line Search Optimizers

1.1 Source Code: line_search_optimizers.py

```

1  ###
2  #imports
3  import numpy as np
4  import matplotlib.pyplot as plt
5  import scipy.optimize
6
7
8  #minimize the rosenbrock function
9  #1 steepest descent
10 #2 nonlinear conjugate gradient
11 #3 quasi newton
12 #4 newtons method
13 #Use backtracking linesearch for all methods
14
15 def rosenbrock(x):
16     return (1-x[0])**2 + 100*(x[1]-x[0]**2)**2
17
18 grad_rosenbrock = lambda x: np.array([-2*(1-x[0]) - 400*(x[1]-x[0]**2)*x[0],
19     200*(x[1]-x[0]**2)])
20
21 hess_rosenbrock = lambda x: np.array([[2-400*(x[1]-x[0]**2) + 800*x[0]**2,
22     -400*x[0]],
23     [-400*x[0],
24     200]])
25
26 def alpha_backtracking(func, grad_func, pk, xk, alpha=1.0, c=1e-4, rho=0.8,
27     max_iter=100, alpha_min=1e-6):
28     i = 0
29     gk = grad_func(xk)
30     if gk.T @ pk >= 0:
31         pk = -gk # ensure descent direction
32
33     while func(xk + alpha * pk) > func(xk) + c * alpha * gk.T @ pk and alpha >
34     alpha_min:
35         alpha *= rho
36         i += 1
37         if i > max_iter:
38             print(f"Alpha backtracking did not converge in {max_iter}
39             iterations at {xk}")
40             return alpha
41     return alpha
42
43 def steepest_descent(x0, func, grad_func, tol=1e-6, max_iter=50000, alpha=1.0, c=10
44     e-4, rho=0.8):
45     xk = x0
46     grad_history = [np.linalg.norm(grad_func(xk))]
47     path_history = [xk.copy()]
48     for i in range(max_iter):
49         if i % 1000 == 0 and i > 0:
50             print(f"Iteration {i}: xk = {xk}, grad norm = {np.linalg.norm(
51             grad_func(xk))}")
52         pk = -grad_func(xk)
53         grad_history.append(np.linalg.norm(grad_func(xk)))
54         alpha = alpha_backtracking(func, grad_func, pk, xk, alpha=alpha, c=c, rho=rho
55         , max_iter=1000)
56         xk = xk + alpha*pk

```

```

51     path_history.append(xk.copy())
52     if np.linalg.norm(grad_func(xk)) < tol:
53         print(f"Steepest descent converged in {i} iterations at {xk}")
54         return xk,i,grad_history,path_history
55     print(f"Steepest descent did not converge in {max_iter} iterations at {xk}")
56     return xk,i,grad_history,path_history
57
58
59 def nonlin_conj_grad(x0, func, grad_func, tol=1e-6, max_iter=30000):
60     xk = x0
61     pk = -grad_func(xk)
62     grad_history = [np.linalg.norm(grad_func(xk))]
63     path_history = [xk.copy()]
64     for i in range(max_iter):
65         if i % 1000 == 0 and i > 0:
66             print(f"Iteration {i}: xk = {xk}, grad norm = {np.linalg.norm(
grad_func(xk))}")
67             gk = grad_func(xk)
68             alpha = alpha_backtracking(func,grad_func,pk,xk)
69             xk = xk + alpha*pk
70             path_history.append(xk.copy())
71             gk1 = grad_func(xk)
72             bk = (gk1.T@gk1)/(gk.T@gk)
73             pk = -gk1 + bk*pk
74             grad_history.append(np.linalg.norm(grad_func(xk)))
75             if np.linalg.norm(grad_func(xk)) < tol:
76                 print(f"Nonlinear conjugate gradient converged in {i} iterations at
{xk}")
77                 return xk,i,grad_history,path_history
78     print(f"Nonlinear conjugate gradient did not converge in {max_iter}
iterations at {xk}")
79     return xk,i,grad_history,path_history
80
81
82 def quasi_newton_bfgs(x0, func, grad_func, tol=1e-6, max_iter=3000, alpha=1.0,
c=1e-4, rho=0.8):
83     #H represents inverse of the Hessian
84     xk = x0
85     gk = grad_func(xk)
86     Hk = np.eye(2)
87     grad_history = [np.linalg.norm(gk)]
88     path_history = [xk.copy()]
89     for i in range(max_iter):
90         if i % 1000 == 0 and i > 0:
91             print(f"Iteration {i}: xk = {xk}, grad norm = {np.linalg.norm(gk)}")
92
93     pk = -Hk@gk
94     alpha = alpha_backtracking(func,grad_func,pk,xk)
95     #alpha = 0.05
96     xk = xk + alpha*pk
97     path_history.append(xk.copy())
98     gk1 = grad_func(xk)
99     del_gk = gk1 - gk
100     del_xk = alpha*pk
101
102     # Check curvature condition: s_k^T y_k > 0
103     # Only update H if this condition is satisfied
104     sk_yk = del_xk.T @ del_gk
105     if sk_yk > 1e-10: # Add small threshold to avoid numerical issues
106         # BFGS update
107         rho = 1.0 / sk_yk
108         Vk = np.eye(len(xk)) - rho * np.outer(del_xk, del_gk)

```

```

108         Hk = Vk @ Hk @ Vk.T + rho * np.outer(del_xk, del_xk)
109
110         gk=gk1
111         grad_history.append(np.linalg.norm(grad_func(xk)))
112         if np.linalg.norm(grad_func(xk)) < tol:
113             print(f"Quasi-Newton BFGS converged in {i} iterations at {xk}")
114             return xk,i,grad_history,path_history
115         print(f"Quasi-Newton BFGS did not converge in {max_iter} iterations at {xk}")
116         return xk,i,grad_history,path_history
117
118
119 def newtons_method(x0, func, grad_func, hess_func, tol=1e-6, max_iter=30000):
120     xk =x0
121     gk = grad_func(xk)
122     grad_history = [np.linalg.norm(gk)]
123     path_history = [xk.copy()]
124     for i in range(max_iter):
125         if i % 1000 == 0 and i > 0:
126             print(f"Iteration {i}: xk = {xk}, grad norm = {np.linalg.norm(gk)}")
127
128         pk = -np.linalg.inv(hess_func(xk))@gk
129         alpha = alpha_backtracking(func,grad_func,pk,xk)
130         xk = xk + alpha*pk
131         path_history.append(xk.copy())
132         gk = grad_func(xk)
133         grad_history.append(np.linalg.norm(grad_func(xk)))
134         if np.linalg.norm(grad_func(xk)) < tol:
135             print(f"Newtons method converged in {i} iterations at {xk}")
136             return xk,i,grad_history,path_history
137         print(f"Newtons method did not converge in {max_iter} iterations at {xk}")
138         return xk,i,grad_history,path_history
139
140 def plot_contour_with_path(func, path_history, title, x_range=(-2, 4), y_range
141                             =(-1, 5)):
142     """Plot contour of function with optimization path overlaid"""
143     # Create grid
144     x = np.linspace(x_range[0], x_range[1], 400)
145     y = np.linspace(y_range[0], y_range[1], 400)
146     X, Y = np.meshgrid(x, y)
147     Z = np.zeros_like(X)
148
149     # Evaluate function on grid
150     for i in range(X.shape[0]):
151         for j in range(X.shape[1]):
152             Z[i, j] = func(np.array([X[i, j], Y[i, j]]))
153
154     # Create plot
155     fig, ax = plt.subplots(figsize=(10, 8))
156
157     # Plot contours (log scale for Rosenbrock function)
158     levels = np.logspace(-1, 3.5, 35)
159     contour = ax.contour(X, Y, Z, levels=levels, cmap='viridis', alpha=0.6)
160     ax.clabel(contour, inline=True, fontsize=8)
161
162     # Plot optimization path
163     path_array = np.array(path_history)
164     ax.plot(path_array[:, 0], path_array[:, 1], 'ro-', linewidth=2,
165             markersize=4, label='Optimization Path', alpha=0.8)
166     ax.plot(path_array[0, 0], path_array[0, 1], 'go', markersize=10,
167             label=f'Start: ({path_array[0, 0]:.2f}, {path_array[0, 1]:.2f})')
168     ax.plot(path_array[-1, 0], path_array[-1, 1], 'r*', markersize=15,
169             label=f'End: ({path_array[-1, 0]:.3f}, {path_array[-1, 1]:.3f})')

```

```

168
169
170     ax.set_xlabel('x1')
171     ax.set_ylabel('x2')
172     ax.set_title(f'{title}\nIterations: {len(path_history)-1}')
173     ax.legend()
174     ax.grid(True, alpha=0.3)
175     plt.tight_layout()
176     plt.show()
177
178 if __name__ == "__main__":
179     ###
180     #steepest descent
181     x0 = np.array([2.0, 2.0])
182
183     xk,i,grad_history,path_history = steepest_descent(x0,rosenbrock,
184     grad_rosenbrock)
185
186     #plot log(grad_history) vs iteration
187     plt.plot(np.log(grad_history))
188     plt.xlabel("Iteration")
189     plt.ylabel("Log(Gradient Norm)")
190     plt.title("Steepest Descent")
191     plt.show()
192
193     #plot contour with path
194     plot_contour_with_path(rosenbrock, path_history, "Steepest Descent")
195
196     ###
197     #nonlinear conjugate gradient
198     x0 = np.array([2.0, 2.0])
199     xk,i,grad_history,path_history = nonlin_conj_grad(x0,rosenbrock,
200     grad_rosenbrock)
201
202     #plot log(grad_history) vs iteration
203     plt.plot(np.log(grad_history))
204     plt.xlabel("Iteration")
205     plt.ylabel("Log(Gradient Norm)")
206     plt.title("Nonlinear Conjugate Gradient")
207     plt.show()
208
209     #plot contour with path
210     plot_contour_with_path(rosenbrock, path_history, "Nonlinear Conjugate
211     Gradient")
212
213     ###
214     #quasi newton
215     x0 = np.array([2.0, 2.0])
216     xk,i,grad_history,path_history = quasi_newton_bfgs(x0,rosenbrock,
217     grad_rosenbrock)
218
219     #plot log(grad_history) vs iteration
220     plt.plot(np.log(grad_history))
221     plt.xlabel("Iteration")
222     plt.ylabel("Log(Gradient Norm)")
223     plt.title("Quasi-Newton BFGS")
224     plt.show()
225
226     #plot contour with path
227     plot_contour_with_path(rosenbrock, path_history, "Quasi-Newton BFGS")
228
229     ###
230     #newtons method

```

```
227     x0 = np.array([2.0, 2.0])
228     xk,i,grad_history,path_history = newtons_method(x0,rosenbrock,
grad_rosenbrock,hess_rosenbrock)
229
230     #plot log(grad_history) vs iteration
231     plt.plot(np.log(grad_history))
232     plt.xlabel("Iteration")
233     plt.ylabel("Log(Gradient Norm)")
234     plt.title("Newtons Method")
235     plt.show()
236
237     #plot contour with path
238     plot_contour_with_path(rosenbrock, path_history, "Newton's Method")
```

2 Brequet Range Optimizer

2.1 Source Code: brequet_range_optimizer.py

```

1  ###
2  #Imports
3  import numpy as np
4  import matplotlib.pyplot as plt
5  import scipy.optimize
6  from line_search_optimizers import alpha_backtracking, steepest_descent,
   plot_contour_with_path, quasi_newton_bfgs
7  ###
8  #Define functions
9  def brequet_range(V,ct,CL,CD,Wi,Wf):
10     return V/ct*CL/CD*(np.log(Wi/Wf))
11
12 get_L = lambda CL, rho, V, S: 0.5*rho*V**2*S*CL
13 get_D = lambda CD, rho, V, S: 0.5*rho*V**2*S*CD
14 get_CL = lambda L,rho,V,S: L/(0.5*rho*V**2*S)
15 get_rho = lambda h: 1.2*np.maximum(1-0.0065*h/288, 0.01)**5.26 # Clamp to
   avoid negative base
16 get_CD = lambda CD0,CL,AR,e,CWD: CD0 + CL**2/np.pi/AR/e + CWD
17 get_CWD = lambda V,c: 10*(np.arctan(10*((V/0.7/c)**2-1))+np.pi/2)
18 get_ct = lambda m_dot, T : m_dot/T + 10e-5
19 get_m_dot_f = lambda rho, At, V, FAR: rho*At*V*FAR
20
21 eps = 1e-10
22 e=0.8
23 CD0=0.0083
24 AR = 10
25 S = 100 #m^2
26 Wf = 162400 #kg
27 Wfuel = 146571 #kg
28 At = 1.3295 #m^2
29 FAR =0.1
30 c=343
31 Wi = Wf + Wfuel
32 fuel_fraction = 0.25
33 Wf_75 = Wf + fuel_fraction*Wfuel
34 Lift = (Wf_75)*9.81
35 #wfuel_capacity = 1/.75*Wfuel#???
36 #plot range 0,300ms^-1 X 0,25000m
37
38 #plot the design space of the range of the aircraft by manipulating
39 #the cruising altitude and velocity for the given variables with 75% fuel
40 ###
41 #Plot design space of the range of the aircraft
42 V = np.linspace(0,300,100)
43 h = np.linspace(0,25000,100)
44 Range_arr = np.zeros((len(V),len(h)))
45 for i in range(len(V)):
46     for j in range(len(h)):
47         rho = get_rho(h[j])
48         m_dot_f = get_m_dot_f(rho,At,V[i],FAR)
49         CL = get_CL(Lift,rho,V[i],S)
50         CWD = get_CWD(V[i],c)
51         CD = get_CD(CD0,CL,AR,e,CWD)
52         T = get_D(CD,rho,V[i],S)
53         ct = get_ct(m_dot_f,T)
54
55         Range_arr[i,j] = brequet_range(V[i],ct,CL,CD,Wi,Wf_75)
56
57 plt.contourf(V,h,Range_arr)

```

```

58 plt.colorbar()
59 plt.xlabel("Velocity (m/s)")
60 plt.ylabel("Altitude (m)")
61 plt.title("Range of the Aircraft")
62 plt.show()
63
64
65 # V = np.linspace(0,300,100)
66 # h = np.linspace(0,25000,100)
67 # Range_arr = np.zeros((len(V),len(h)))
68 # for i in range(len(V)):
69 #     for j in range(len(h)):
70 #         x = np.array([V[i],h[j]])
71 #         Range_arr[i,j] = brequet_range_wrapper(x)
72 # plt.contourf(V,h,Range_arr)
73 # plt.colorbar()
74 # plt.xlabel("Velocity (m/s)")
75 # plt.ylabel("Altitude (m)")
76 # plt.title("Range of the Aircraft 2")
77 # plt.show()
78 %%
79 #define functions
80
81 def grad_brequet(x):
82     return scipy.optimize.approx_fprime(x,brequet_range_wrapper,1e-6)
83
84
85 #Could also use forward euler method to get gradient at specific points
86 #steepest descent
87 def brequet_range_wrapper(x): #x = [V,h]
88     V,h = x
89     # Safety check: V must be positive to avoid division by zero
90     if V <= 0:
91         return 1e10 # Return large penalty value for infeasible V
92
93     rho = get_rho(h)
94     m_dot_f = get_m_dot_f(rho,At,V,FAR)
95     #Lift required = mass * 9.81 = (162400+0.75*146571)*9.81
96     CL = get_CL(Lift,rho,V,S)
97     CWD = get_CWD(V,c)
98     CD = get_CD(CD0,CL,AR,e,CWD)
99     T = get_D(CD,rho,V,S)
100     ct = get_ct(m_dot_f,T)
101     #want to minimize range, so we want to maximize negative range
102     return -brequet_range(V,ct,CL,CD,Wi,Wf_75)
103
104 def plot_brequet_contour_with_path(func, path_history, x_range=(10, 300),
y_range=(0, 25000)):
105     """Plot contour of Brequet range with optimization path overlaid"""
106     print("Generating contour plot (this may take a moment)...")
107
108     # Create grid (using fewer points for speed)
109     x = np.linspace(x_range[0], x_range[1], 100)
110     y = np.linspace(y_range[0], y_range[1], 100)
111     X, Y = np.meshgrid(x, y)
112     Z = np.zeros_like(X)
113
114     # Evaluate function on grid
115     for i in range(X.shape[0]):
116         for j in range(X.shape[1]):
117             Z[i, j] = func(np.array([X[i, j], Y[i, j]]))
118
119     # Convert to actual range (remove negative sign)

```



```

120     Z_range = -Z
121
122     # Create plot
123     fig, ax = plt.subplots(figsize=(10, 10))
124
125     # Plot filled contours
126     contourf = ax.contourf(X, Y, Z_range, levels=30, cmap='viridis', alpha=0.8)
127     cbar = plt.colorbar(contourf, ax=ax, label='Range (m)')
128
129     # Plot contour lines
130     contour = ax.contour(X, Y, Z_range, levels=15, colors='white', alpha=0.3,
131                          linewidths=0.5)
132     ax.clabel(contour, inline=True, fontsize=8, fmt='%.0f')
133
134     # Plot optimization path
135     path_array = np.array(path_history)
136     ax.plot(path_array[:, 0], path_array[:, 1], 'r-', linewidth=2.5,
137            label='Optimization Path', alpha=0.9, zorder=5)
138     ax.plot(path_array[:, 0], path_array[:, 1], 'wo', markersize=3,
139            alpha=0.6, zorder=6)
140
141     # Mark start and end points
142     ax.plot(path_array[0, 0], path_array[0, 1], 'go', markersize=12,
143            label=f'Start: V={path_array[0, 0]:.1f} m/s, h={path_array[0, 1]:.0'
144            f} m',
145            markeredgewidth=2, zorder=7)
146     ax.plot(path_array[-1, 0], path_array[-1, 1], 'r*', markersize=18,
147            label=f'End: V={path_array[-1, 0]:.1f} m/s, h={path_array[-1, 1]:.0'
148            f} m',
149            markeredgewidth=1.5, zorder=7)
150
151     # Labels and formatting
152     ax.set_xlabel('Velocity (m/s)', fontsize=12, fontweight='bold')
153     ax.set_ylabel('Altitude (m)', fontsize=12, fontweight='bold')
154     ax.set_title(f'Brequet Range Optimization Path\n({len(path_history)-1}'
155            f' iterations)',
156            fontsize=14, fontweight='bold')
157     ax.legend(loc='best', fontsize=10, framealpha=0.9)
158     ax.grid(True, alpha=0.2, linestyle='--')
159
160     plt.tight_layout()
161     plt.show()
162
163     # Print path statistics
164     print(f"\nPath Statistics:")
165     print(f"    Total iterations: {len(path_history)-1}")
166     print(f"    Start: V={path_array[0,0]:.2f} m/s, h={path_array[0,1]:.2f} m")
167     print(f"    End:    V={path_array[-1,0]:.2f} m/s, h={path_array[-1,1]:.2f} m")
168     print(f"    Range improvement: {-func(path_array[-1]) - (-func(path_array[0]))'
169            f':.2f} m")
170
171     ##
172     #Steepest Descent
173
174     x0 = np.array([175, 15000])
175     run_steepest = True
176     if run_steepest == True:
177         xk, i, grad_history, path_history = steepest_descent(x0, brequet_range_wrapper,
178             grad_brequet, tol=1e-4, max_iter=400000, alpha=1.0, c=10e-4, rho=0.6)
179
180     #plot log(grad_history) vs iteration

```

```

177 plt.plot(np.log(grad_history))
178 plt.xlabel("Iteration")
179 plt.ylabel("Log(Gradient Norm)")
180 plt.title("Steepest Descent")
181 plt.show()
182 #plot contour with path
183 plot_brequet_contour_with_path(brequet_range_wrapper, path_history, x_range
=(10,300), y_range=(0,25000))

184
185 ###
186 # Extract data from optimization history
187 iterations = np.arange(len(path_history))
188 velocities = np.array([p[0] for p in path_history])
189 altitudes = np.array([p[1] for p in path_history])
190 objective_values = np.array([brequet_range_wrapper(p) for p in path_history
])
191 # Convert negative objective to actual range
192 range_values = -objective_values
193
194 # Plot 1: Objective function (range), velocity, and altitude vs iterations
195 fig, ax1 = plt.subplots(figsize=(12, 6))
196
197 # Plot range on primary y-axis
198 color1 = 'tab:blue'
199 ax1.set_xlabel('Iteration', fontsize=12)
200 ax1.set_ylabel('Range (m)', color=color1, fontsize=12)
201 line1 = ax1.plot(iterations, range_values, color=color1, linewidth=2, label
='Range')
202 ax1.tick_params(axis='y', labelcolor=color1)
203 ax1.grid(True, alpha=0.3)
204
205 # Create second y-axis for velocity and altitude
206 ax2 = ax1.twinx()
207 color2 = 'tab:orange'
208 color3 = 'tab:green'
209 ax2.set_ylabel('Velocity (m/s) / Altitude (m)', fontsize=12)
210 line2 = ax2.plot(iterations, velocities, color=color2, linewidth=2,
linestyle='--', label='Velocity')
211 line3 = ax2.plot(iterations, altitudes, color=color3, linewidth=2,
linestyle='-.', label='Altitude')
212 ax2.tick_params(axis='y')
213
214 # Combine legends
215 lines = line1 + line2 + line3
216 labels = [l.get_label() for l in lines]
217 ax1.legend(lines, labels, loc='best', fontsize=10)
218
219 plt.title('Optimization Progress: Range, Velocity, and Altitude vs
Iteration', fontsize=14, fontweight='bold')
220 plt.tight_layout()
221 plt.show()
222
223 ###
224 # Print final results
225 print("=" * 60)
226 print("OPTIMIZATION RESULTS")
227 print("=" * 60)
228 print(f"Initial point:")
229 print(f"  V = {path_history[0][0]:.2f} m/s, h = {path_history[0][1]:.2f} m"
)
230 print(f"  Range = {-brequet_range_wrapper(path_history[0]):.2f} m")
231 print(f"\nFinal point (after {len(path_history)-1} iterations):")
232 print(f"  V = {xk[0]:.2f} m/s, h = {xk[1]:.2f} m")

```

```

233     print(f"   Range = {-brequet_range_wrapper(xk):.2f} m")
234     print(f"\nImprovement:")
235     print(f"   ΔRange = {-brequet_range_wrapper(xk) - (-brequet_range_wrapper(
path_history[0])):.2f} m")
236     print(f"   ΔV = {xk[0] - path_history[0][0]:.2f} m/s")
237     print(f"   Δh = {xk[1] - path_history[0][1]:.2f} m")
238     print(f"\nFinal gradient norm: {grad_history[-1]:.6e}")
239     print("=" * 60)
240
241     ###
242     #Quasi Newton Method
243
244     xk,i,grad_history,path_history = quasi_newton_bfgs(x0, brequet_range_wrapper,
grad_brequet, tol=1e-6, max_iter=50000, alpha=1.0, c=1e-4, rho=0.8)
245     #plot log(grad_history) vs iteration
246     plt.plot(np.log(grad_history))
247     plt.xlabel("Iteration")
248     plt.ylabel("Log(Gradient Norm)")
249     plt.title("Quasi Newton Method")
250     plt.show()
251     ###
252     # Extract data from optimization history
253     iterations = np.arange(len(path_history))
254     # Multiply velocity by 100 for plotting
255     velocities = np.array([p[0] for p in path_history]) * 100
256     altitudes = np.array([p[1] for p in path_history])
257     objective_values = np.array([brequet_range_wrapper(p) for p in path_history])
258     # Convert negative objective to actual range
259     range_values = -objective_values
260
261     # Plot 1: Objective function (range), velocity (x100), and altitude vs
iterations
262     fig, ax1 = plt.subplots(figsize=(12, 6))
263
264     # Plot range on primary y-axis
265     color1 = 'tab:blue'
266     ax1.set_xlabel('Iteration', fontsize=12)
267     ax1.set_ylabel('Range (m)', color=color1, fontsize=12)
268     line1 = ax1.plot(iterations, range_values, color=color1, linewidth=2, label='
Range')
269     ax1.tick_params(axis='y', labelcolor=color1)
270     ax1.grid(True, alpha=0.3)
271
272     # Create second y-axis for velocity (x100) and altitude
273     ax2 = ax1.twinx()
274     color2 = 'tab:orange'
275     color3 = 'tab:green'
276     ax2.set_ylabel('Velocity (x100 m/s) / Altitude (m)', fontsize=12)
277     line2 = ax2.plot(iterations, velocities, color=color2, linewidth=2, linestyle='
--', label='Velocity (x100)')
278     line3 = ax2.plot(iterations, altitudes, color=color3, linewidth=2, linestyle='
-.', label='Altitude')
279     ax2.tick_params(axis='y')
280
281     # Combine legends
282     lines = line1 + line2 + line3
283     labels = [l.get_label() for l in lines]
284     ax1.legend(lines, labels, loc='best', fontsize=10)
285
286     plt.title('Optimization Progress: Range, Velocity (x100), and Altitude vs
Iteration', fontsize=14, fontweight='bold')
287     plt.tight_layout()
288     plt.show()

```

```
289 #plot contour with path
290 plot_brequet_contour_with_path(brequet_range_wrapper, path_history, x_range
    =(10,300), y_range=(0,25000))
```