MECH597 Assignment 2
Python Code Documentation

Optimization Algorithms & Brequet Range Analysis

October 13, 2025

Contents

1 Line Search Optimizers

1.1 Source Code: line_search_optimizers.py

```python
#%%
#imports
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize


#minimize the rosenbrock function
#1 steepest descent
#2 nonlinear conjugate gradient
#3 quasi newton
#4 newtons method
#Use backtracking linesearch for all methods

def rosenbrock(x):
    return (1-x[0])**2 + 100*(x[1]-x[0]**2)**2

grad_rosenbrock = lambda x: np.array([-2*(1-x[0]) - 400*(x[1]-x[0]**2)*x[0], 200*(x[1]-x[0]**2)])

hess_rosenbrock = lambda x: np.array([[2-400*(x[1]-x[0]**2) + 800*x[0]**2, -400*x[0]],
                                      [-400*x[0], 200]])

def alpha_backtracking(func, grad_func, pk, xk, alpha=1.0, c=1e-4, rho=0.8, max_iter=100, alpha_min=1e-6):
    i = 0
    gk = grad_func(xk)
    if gk.T @ pk >= 0:
        pk = -gk  # ensure descent direction

    while func(xk + alpha * pk) > func(xk) + c * alpha * gk.T @ pk and alpha > alpha_min:
        alpha *= rho
        i += 1
        if i > max_iter:
            print(f"Alpha backtracking did not converge in {max_iter} iterations at {xk}")
            return alpha
    return alpha




def steepest_descent(x0,func,grad_func,tol=1e-6,max_iter=50000, alpha=1.0, c=10e-4, rho=0.8):
    xk = x0
    grad_history = [np.linalg.norm(grad_func(xk))]
    path_history = [xk.copy()]
    for i in range(max_iter):
        if i % 1000 == 0 and i > 0:
            print(f"Iteration {i}: xk = {xk}, grad norm = {np.linalg.norm(grad_func(xk))}")
        pk = -grad_func(xk)
        grad_history.append(np.linalg.norm(grad_func(xk)))
        alpha = alpha_backtracking(func,grad_func,pk,xk,alpha=alpha,c=c,rho=rho,max_iter=1000)
        xk = xk + alpha*pk
```

```
        path_history.append(xk.copy())
        if np.linalg.norm(grad_func(xk)) < tol:
            print(f"Steepest descent converged in {i} iterations at {xk}")
            return xk,i,grad_history,path_history
    print(f"Steepest descent did not converge in {max_iter} iterations at {xk}")
    return xk,i,grad_history,path_history


def nonlin_conj_grad(x0, func, grad_func, tol=1e-6, max_iter=30000):
    xk = x0
    pk = -grad_func(xk)
    grad_history = [np.linalg.norm(grad_func(xk))]
    path_history = [xk.copy()]
    for i in range(max_iter):
        if i % 1000 == 0 and i > 0:
            print(f"Iteration {i}: xk = {xk}, grad norm = {np.linalg.norm(grad_func(xk))}")
        gk = grad_func(xk)
        alpha = alpha_backtracking(func,grad_func,pk,xk)
        xk = xk + alpha*pk
        path_history.append(xk.copy())
        gk1 = grad_func(xk)
        bk = (gk1.T@gk1)/(gk.T@gk)
        pk = -gk1 + bk*pk
        grad_history.append(np.linalg.norm(grad_func(xk)))
        if np.linalg.norm(grad_func(xk)) < tol:
            print(f"Nonlinear conjugate gradient converged in {i} iterations at {xk}")
            return xk,i,grad_history,path_history
    print(f"Nonlinear conjugate gradient did not converge in {max_iter} iterations at {xk}")
    return xk,i,grad_history,path_history


def quasi_newton_bfgs(x0, func, grad_func, tol=1e-6, max_iter=3000, alpha=1.0, c=1e-4, rho=0.8):
    #H represents inverse of the Hessian
    xk = x0
    gk = grad_func(xk)
    Hk = np.eye(2)
    grad_history = [np.linalg.norm(gk)]
    path_history = [xk.copy()]
    for i in range(max_iter):
        if i % 1000 == 0 and i > 0:
            print(f"Iteration {i}: xk = {xk}, grad norm = {np.linalg.norm(gk)}")
        pk = -Hk@gk
        alpha = alpha_backtracking(func,grad_func,pk,xk)
        #alpha = 0.05
        xk = xk + alpha*pk
        path_history.append(xk.copy())
        gk1 = grad_func(xk)
        del_gk = gk1 - gk
        del_xk = alpha*pk

        # Check curvature condition: s_k^T y_k > 0
        # Only update H if this condition is satisfied
        sk_yk = del_xk.T @ del_gk
        if sk_yk > 1e-10:  # Add small threshold to avoid numerical issues
            # BFGS update
            rho = 1.0 / sk_yk
            Vk = np.eye(len(xk)) - rho * np.outer(del_xk, del_gk)
```

```
            Hk = Vk @ Hk @ Vk.T + rho * np.outer(del_xk, del_xk)

        gk=gk1
        grad_history.append(np.linalg.norm(grad_func(xk)))
        if np.linalg.norm(grad_func(xk)) < tol:
            print(f"Quasi-Newton BFGS converged in {i} iterations at {xk}")
            return xk,i,grad_history,path_history
    print(f"Quasi-Newton BFGS did not converge in {max_iter} iterations at {xk}")
    return xk,i,grad_history,path_history


def newtons_method(x0, func, grad_func, hess_func, tol=1e-6, max_iter=30000):
    xk =x0
    gk = grad_func(xk)
    grad_history = [np.linalg.norm(gk)]
    path_history = [xk.copy()]
    for i in range(max_iter):
        if i % 1000 == 0 and i > 0:
            print(f"Iteration {i}: xk = {xk}, grad norm = {np.linalg.norm(gk)}")
        pk = -np.linalg.inv(hess_func(xk))@gk
        alpha = alpha_backtracking(func,grad_func,pk,xk)
        xk = xk + alpha*pk
        path_history.append(xk.copy())
        gk = grad_func(xk)
        grad_history.append(np.linalg.norm(grad_func(xk)))
        if np.linalg.norm(grad_func(xk)) < tol:
            print(f"Newtons method converged in {i} iterations at {xk}")
            return xk,i,grad_history,path_history
    print(f"Newtons method did not converge in {max_iter} iterations at {xk}")
    return xk,i,grad_history,path_history

def plot_contour_with_path(func, path_history, title, x_range=(-2, 4), y_range=(-1, 5)):
    """Plot contour of function with optimization path overlaid"""
    # Create grid
    x = np.linspace(x_range[0], x_range[1], 400)
    y = np.linspace(y_range[0], y_range[1], 400)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros_like(X)

    # Evaluate function on grid
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Z[i, j] = func(np.array([X[i, j], Y[i, j]]))

    # Create plot
    fig, ax = plt.subplots(figsize=(10, 8))

    # Plot contours (log scale for Rosenbrock function)
    levels = np.logspace(-1, 3.5, 35)
    contour = ax.contour(X, Y, Z, levels=levels, cmap='viridis', alpha=0.6)
    ax.clabel(contour, inline=True, fontsize=8)

    # Plot optimization path
    path_array = np.array(path_history)
    ax.plot(path_array[:, 0], path_array[:, 1], 'ro-', linewidth=2,
            markersize=4, label='Optimization Path', alpha=0.8)
    ax.plot(path_array[0, 0], path_array[0, 1], 'go', markersize=10,
            label=f'Start: ({path_array[0, 0]:.2f}, {path_array[0, 1]:.2f})')
    ax.plot(path_array[-1, 0], path_array[-1, 1], 'r*', markersize=15,
            label=f'End: ({path_array[-1, 0]:.3f}, {path_array[-1, 1]:.3f})')
```

```python


    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_title(f'{title}\nIterations: {len(path_history)-1}')
    ax.legend()
    ax.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    #%%
    #steepest descent
    x0 = np.array([2.0, 2.0])

    xk,i,grad_history,path_history = steepest_descent(x0,rosenbrock,
    grad_rosenbrock)

    #plot log(grad_history) vs iteration
    plt.plot(np.log(grad_history))
    plt.xlabel("Iteration")
    plt.ylabel("Log(Gradient Norm)")
    plt.title("Steepest Descent")
    plt.show()

    #plot contour with path
    plot_contour_with_path(rosenbrock, path_history, "Steepest Descent")

    #%%
    #nonlinear conjugate gradient
    x0 = np.array([2.0, 2.0])
    xk,i,grad_history,path_history = nonlin_conj_grad(x0,rosenbrock,
    grad_rosenbrock)

    #plot log(grad_history) vs iteration
    plt.plot(np.log(grad_history))
    plt.xlabel("Iteration")
    plt.ylabel("Log(Gradient Norm)")
    plt.title("Nonlinear Conjugate Gradient")
    plt.show()

    #plot contour with path
    plot_contour_with_path(rosenbrock, path_history, "Nonlinear Conjugate
    Gradient")

    #%%
    #quasi newton
    x0 = np.array([2.0, 2.0])
    xk,i,grad_history,path_history = quasi_newton_bfgs(x0,rosenbrock,
    grad_rosenbrock)

    #plot log(grad_history) vs iteration
    plt.plot(np.log(grad_history))
    plt.xlabel("Iteration")
    plt.ylabel("Log(Gradient Norm)")
    plt.title("Quasi-Newton BFGS")
    plt.show()

    #plot contour with path
    plot_contour_with_path(rosenbrock, path_history, "Quasi-Newton BFGS")

    #%%
    #newtons method
```

```
    x0 = np.array([2.0, 2.0])
    xk,i,grad_history,path_history = newtons_method(x0,rosenbrock,
   grad_rosenbrock,hess_rosenbrock)

    #plot log(grad_history) vs iteration
    plt.plot(np.log(grad_history))
    plt.xlabel("Iteration")
    plt.ylabel("Log(Gradient Norm)")
    plt.title("Newtons Method")
    plt.show()

    #plot contour with path
    plot_contour_with_path(rosenbrock, path_history, "Newton's Method")
```

2 Brequet Range Optimizer

2.1 Source Code: brequet_range_optimizer.py

```
#%%
#Imports
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize
from line_search_optimizers import alpha_backtracking, steepest_descent,
    plot_contour_with_path, quasi_newton_bfgs
#%%
#Define functions
def brequet_range(V,ct,CL,CD,Wi,Wf):
    return V/ct*CL/CD*(np.log(Wi/Wf))

get_L = lambda CL, rho, V, S: 0.5*rho*V**2*S*CL
get_D = lambda CD, rho, V, S: 0.5*rho*V**2*S*CD
get_CL = lambda L,rho,V,S: L/(0.5*rho*V**2*S)
get_rho = lambda h: 1.2*np.maximum(1-0.0065*h/288, 0.01)**5.26  # Clamp to
    avoid negative base
get_CD = lambda CD0,CL,AR,e,CWD: CD0 + CL**2/np.pi/AR/e + CWD
get_CWD = lambda V,c: 10*(np.arctan(10*((V/0.7/c)**2-1))+np.pi/2)
get_ct = lambda m_dot, T : m_dot/T + 10e-5
get_m_dot_f = lambda rho, At, V, FAR: rho*At*V*FAR

eps = 1e-10
e=0.8
CD0=0.0083
AR = 10
S = 100 #m^2
Wf = 162400 #kg
Wfuel = 146571 #kg
At = 1.3295 #m^2
FAR =0.1
c=343
Wi = Wf + Wfuel
fuel_fraction = 0.25
Wf_75 = Wf + fuel_fraction*Wfuel
Lift = (Wf_75)*9.81
#wfuel_capacity = 1/.75*Wfuel#???
#plot range 0,300ms^-1 X 0,25000m

#plot the design space of the range of the aircraft by manipulating
#the cruisting altittude and velocity for the given variables with 75% fuel
#%%
#Plot design space of the range of the aircraft
V = np.linspace(0,300,100)
h = np.linspace(0,25000,100)
Range_arr = np.zeros((len(V),len(h)))
for i in range(len(V)):
    for j in range(len(h)):
        rho = get_rho(h[j])
        m_dot_f = get_m_dot_f(rho,At,V[i],FAR)
        CL = get_CL(Lift,rho,V[i],S)
        CWD = get_CWD(V[i],c)
        CD = get_CD(CD0,CL,AR,e,CWD)
        T = get_D(CD,rho,V[i],S)
        ct = get_ct(m_dot_f,T)

        Range_arr[i,j] = brequet_range(V[i],ct,CL,CD,Wi,Wf_75)

plt.contourf(V,h,Range_arr)
```

```python
plt.colorbar()
plt.xlabel("Velocity (m/s)")
plt.ylabel("Altitude (m)")
plt.title("Range of the Aircraft")
plt.show()


# V = np.linspace(0,300,100)
# h = np.linspace(0,25000,100)
# Range_arr = np.zeros((len(V),len(h)))
# for i in range(len(V)):
#     for j in range(len(h)):
#         x = np.array([V[i],h[j]])
#         Range_arr[i,j] = brequet_range_wrapper(x)
# plt.contourf(V,h,Range_arr)
# plt.colorbar()
# plt.xlabel("Velocity (m/s)")
# plt.ylabel("Altitude (m)")
# plt.title("Range of the Aircraft 2")
# plt.show()
#%%
#define functions

def grad_brequet(x):
    return scipy.optimize.approx_fprime(x,brequet_range_wrapper,1e-6)


#Could also use forward euler method to get gradient at specific points
#steepest descent
def brequet_range_wrapper(x): #x = [V,h]
    V,h = x
    # Safety check: V must be positive to avoid division by zero
    if V <= 0:
        return 1e10  # Return large penalty value for infeasible V

    rho = get_rho(h)
    m_dot_f = get_m_dot_f(rho,At,V,FAR)
    #Lift required = mass * 9.81 = (162400+0.75*146571)*9.81
    CL = get_CL(Lift,rho,V,S)
    CWD = get_CWD(V,c)
    CD = get_CD(CD0,CL,AR,e,CWD)
    T = get_D(CD,rho,V,S)
    ct = get_ct(m_dot_f,T)
    #want to minimize range, so we want to maximize negative range
    return -brequet_range(V,ct,CL,CD,Wi,Wf_75)

def plot_brequet_contour_with_path(func, path_history, x_range=(10, 300),
    y_range=(0, 25000)):
    """Plot contour of Brequet range with optimization path overlaid"""
    print("Generating contour plot (this may take a moment)...")

    # Create grid (using fewer points for speed)
    x = np.linspace(x_range[0], x_range[1], 100)
    y = np.linspace(y_range[0], y_range[1], 100)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros_like(X)

    # Evaluate function on grid
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Z[i, j] = func(np.array([X[i, j], Y[i, j]]))

    # Convert to actual range (remove negative sign)
```

```
    Z_range = -Z

    # Create plot
    fig, ax = plt.subplots(figsize=(10, 10))

    # Plot filled contours
    contourf = ax.contourf(X, Y, Z_range, levels=30, cmap='viridis', alpha=0.8)
    cbar = plt.colorbar(contourf, ax=ax, label='Range (m)')

    # Plot contour lines
    contour = ax.contour(X, Y, Z_range, levels=15, colors='white', alpha=0.3, linewidths=0.5)
    ax.clabel(contour, inline=True, fontsize=8, fmt='%.0f')

    # Plot optimization path
    path_array = np.array(path_history)
    ax.plot(path_array[:, 0], path_array[:, 1], 'r-', linewidth=2.5,
            label='Optimization Path', alpha=0.9, zorder=5)
    ax.plot(path_array[:, 0], path_array[:, 1], 'wo', markersize=3,
            alpha=0.6, zorder=6)

    # Mark start and end points
    ax.plot(path_array[0, 0], path_array[0, 1], 'go', markersize=12,
            label=f'Start: V={path_array[0, 0]:.1f} m/s, h={path_array[0, 1]:.0f} m',
            markeredgecolor='white', markeredgewidth=2, zorder=7)
    ax.plot(path_array[-1, 0], path_array[-1, 1], 'r*', markersize=18,
            label=f'End: V={path_array[-1, 0]:.1f} m/s, h={path_array[-1, 1]:.0f} m',
            markeredgecolor='white', markeredgewidth=1.5, zorder=7)

    # Labels and formatting
    ax.set_xlabel('Velocity (m/s)', fontsize=12, fontweight='bold')
    ax.set_ylabel('Altitude (m)', fontsize=12, fontweight='bold')
    ax.set_title(f'Brequet Range Optimization Path\n({len(path_history)-1} iterations)',
                 fontsize=14, fontweight='bold')
    ax.legend(loc='best', fontsize=10, framealpha=0.9)
    ax.grid(True, alpha=0.2, linestyle='--')

    plt.tight_layout()
    plt.show()

    # Print path statistics
    print(f"\nPath Statistics:")
    print(f"  Total iterations: {len(path_history)-1}")
    print(f"  Start: V={path_array[0,0]:.2f} m/s, h={path_array[0,1]:.2f} m")
    print(f"  End:   V={path_array[-1,0]:.2f} m/s, h={path_array[-1,1]:.2f} m")
    print(f"  Range improvement: {-func(path_array[-1]) - (-func(path_array[0])):.2f} m")


#%%
#Steepest Descent

x0 = np.array([175,15000])
run_steepest = True
if run_steepest == True:
    xk,i,grad_history,path_history = steepest_descent(x0,brequet_range_wrapper, grad_brequet,tol =1e-4,max_iter=400000, alpha=1.0, c=10e-4, rho=0.6)


    #plot log(grad_history) vs iteration
```

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

    #%%
    # Extract data from optimization history
    iterations = np.arange(len(path_history))
    velocities = np.array([p[0] for p in path_history])
    altitudes = np.array([p[1] for p in path_history])
    objective_values = np.array([brequet_range_wrapper(p) for p in path_history])
    # Convert negative objective to actual range
    range_values = -objective_values

    # Plot 1: Objective function (range), velocity, and altitude vs iterations
    fig, ax1 = plt.subplots(figsize=(12, 6))

    # Plot range on primary y-axis
    color1 = 'tab:blue'
    ax1.set_xlabel('Iteration', fontsize=12)
    ax1.set_ylabel('Range (m)', color=color1, fontsize=12)
    line1 = ax1.plot(iterations, range_values, color=color1, linewidth=2, label='Range')
    ax1.tick_params(axis='y', labelcolor=color1)
    ax1.grid(True, alpha=0.3)

    # Create second y-axis for velocity and altitude
    ax2 = ax1.twinx()
    color2 = 'tab:orange'
    color3 = 'tab:green'
    ax2.set_ylabel('Velocity (m/s) / Altitude (m)', fontsize=12)
    line2 = ax2.plot(iterations, velocities, color=color2, linewidth=2, linestyle='--', label='Velocity')
    line3 = ax2.plot(iterations, altitudes, color=color3, linewidth=2, linestyle='-.', label='Altitude')
    ax2.tick_params(axis='y')

    # Combine legends
    lines = line1 + line2 + line3
    labels = [l.get_label() for l in lines]
    ax1.legend(lines, labels, loc='best', fontsize=10)

    plt.title('Optimization Progress: Range, Velocity, and Altitude vs Iteration', fontsize=14, fontweight='bold')
    plt.tight_layout()
    plt.show()

    #%%
    # Print final results
    print("=" * 60)
    print("OPTIMIZATION RESULTS")
    print("=" * 60)
    print(f"Initial point:")
    print(f"  V = {path_history[0][0]:.2f} m/s, h = {path_history[0][1]:.2f} m")
    print(f"  Range = {-brequet_range_wrapper(path_history[0]):.2f} m")
    print(f"\nFinal point (after {len(path_history)-1} iterations):")
    print(f"  V = {xk[0]:.2f} m/s, h = {xk[1]:.2f} m")
```

```python
    print(f"  Range = {-brequet_range_wrapper(xk):.2f} m")
    print(f"\nImprovement:")
    print(f"  ΔRange = {-brequet_range_wrapper(xk) - (-brequet_range_wrapper(
    path_history[0])):.2f} m")
    print(f"  ΔV = {xk[0] - path_history[0][0]:.2f} m/s")
    print(f"  Δh = {xk[1] - path_history[0][1]:.2f} m")
    print(f"\nFinal gradient norm: {grad_history[-1]:.6e}")
    print("=" * 60)

#%%
#Quasi Newton Method

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)
#plot log(grad_history) vs iteration
plt.plot(np.log(grad_history))
plt.xlabel("Iteration")
plt.ylabel("Log(Gradient Norm)")
plt.title("Quasi Newton Method")
plt.show()
#%%
# Extract data from optimization history
iterations = np.arange(len(path_history))
# Multiply velocity by 100 for plotting
velocities = np.array([p[0] for p in path_history]) * 100
altitudes = np.array([p[1] for p in path_history])
objective_values = np.array([brequet_range_wrapper(p) for p in path_history])
# Convert negative objective to actual range
range_values = -objective_values

# Plot 1: Objective function (range), velocity (x100), and altitude vs
    iterations
fig, ax1 = plt.subplots(figsize=(12, 6))

# Plot range on primary y-axis
color1 = 'tab:blue'
ax1.set_xlabel('Iteration', fontsize=12)
ax1.set_ylabel('Range (m)', color=color1, fontsize=12)
line1 = ax1.plot(iterations, range_values, color=color1, linewidth=2, label='
    Range')
ax1.tick_params(axis='y', labelcolor=color1)
ax1.grid(True, alpha=0.3)

# Create second y-axis for velocity (x100) and altitude
ax2 = ax1.twinx()
color2 = 'tab:orange'
color3 = 'tab:green'
ax2.set_ylabel('Velocity (x100 m/s) / Altitude (m)', fontsize=12)
line2 = ax2.plot(iterations, velocities, color=color2, linewidth=2, linestyle='
    --', label='Velocity (x100)')
line3 = ax2.plot(iterations, altitudes, color=color3, linewidth=2, linestyle='
    -.', label='Altitude')
ax2.tick_params(axis='y')

# Combine legends
lines = line1 + line2 + line3
labels = [l.get_label() for l in lines]
ax1.legend(lines, labels, loc='best', fontsize=10)

plt.title('Optimization Progress: Range, Velocity (x100), and Altitude vs
    Iteration', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()
```

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