MECH 579: Numerical Optimization
Department of Mechanical Engineering, McGill University
Project #2: Unconstrained Optimization
13th. October, 2025

Name : Chiyoung Kwon
ID : 261258263
Department : Mechanical Engineering
Program : PhD

1. Write a numerical code to minimize the Rosenbrock function defined below using a slew of gradient based optimization algorithms. The function is a non-convex function and has a global minimum at (1, 1). The minimum is at the bottom of a narrow parabolic valley that is curved on the x-y plane. The function is often used to test the performance of optimization algorithms.

$$f(x, y) = (1 - x)2 + 100(y - x2)2$$

(a) Write four codes using the steepest descent, nonlinear conjugate gradient, quasi-Newton, and Newton method to solve for the minimum of the function. Use a simple backtracking line search method to compute the step length. You may choose one particular algorithm for the conjugate gradient (either the Hestenes-Stiefel, Polak-Ribiere, or Fletcher-Reeves) and quasi-Newton (DFP or BFGS) method.

➔ **Please refer to the attached codes on the Appendix pages.**

(b) Provide the following in a written report:

(a) Convergence of the gradient (y-axis: log of the gradient, x-axis: iteration) and a comparison of the convergence. Compare against your steepest descent result from Project 1.

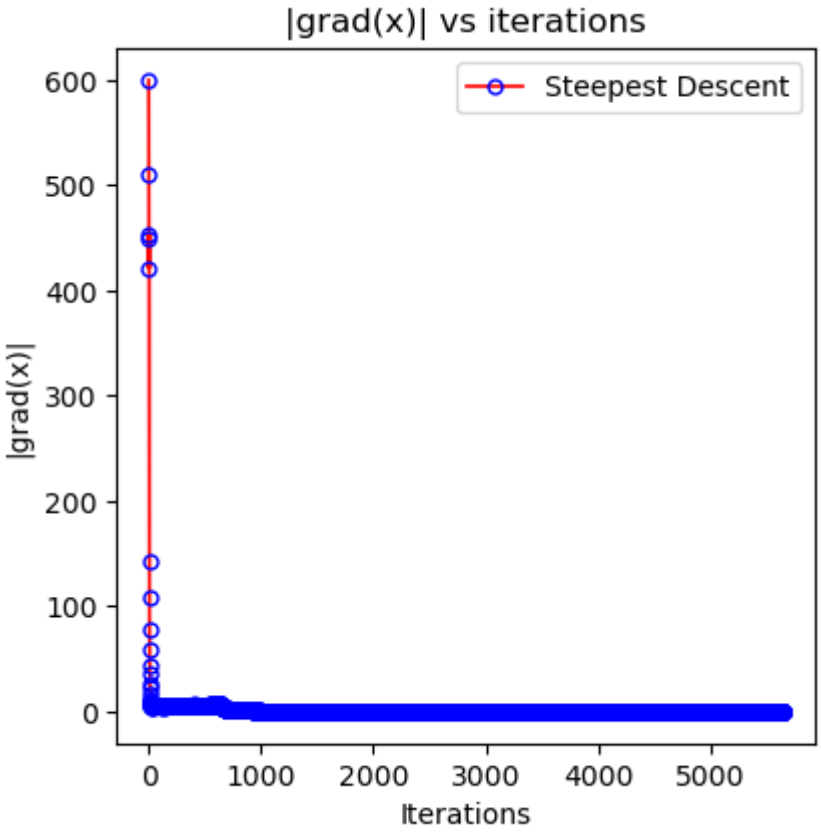

Figure 1. Steepest descent – norm of gradient vs iterations

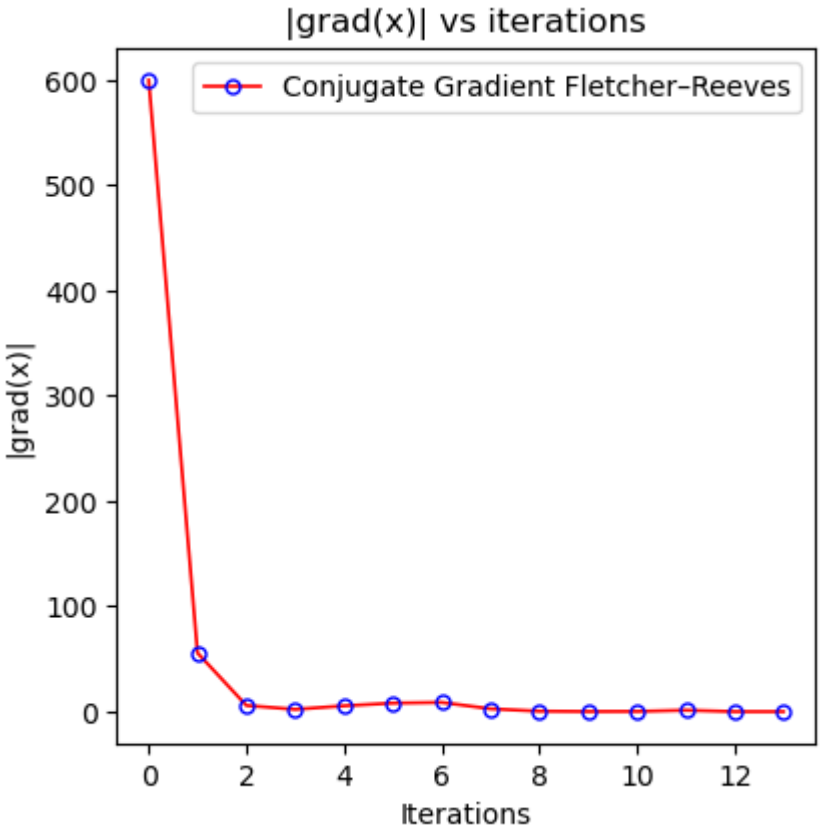

Figure 2. Conjugate gradient Fletcher-Reeves – norm of gradient vs iteration

(b) Contour plot of the path of the optimization algorithm.

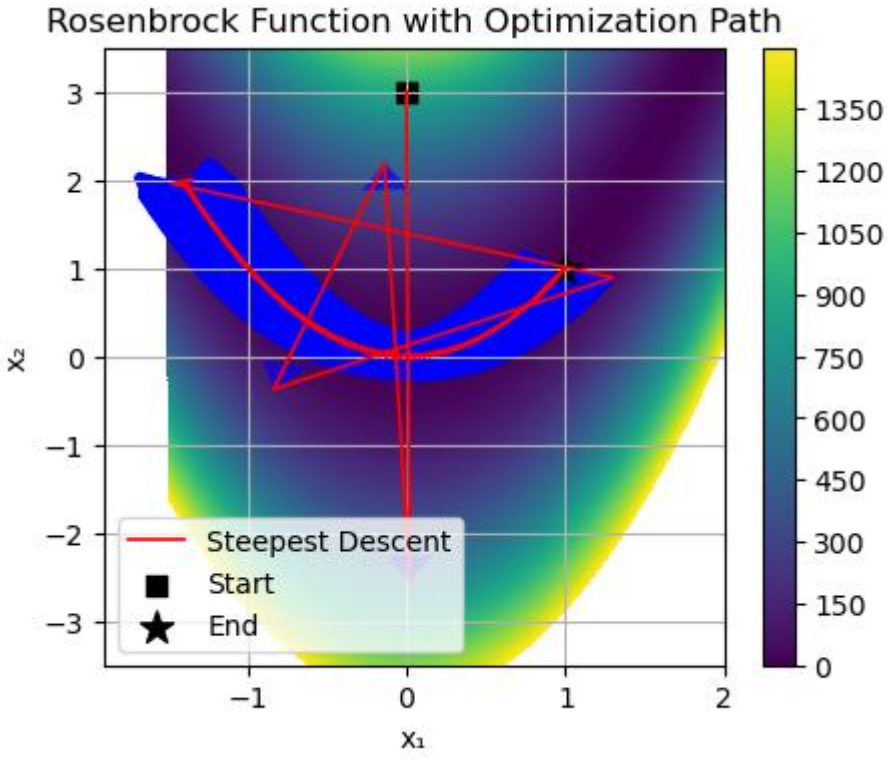

Figure 3. Steepest descent – contour plot of the path

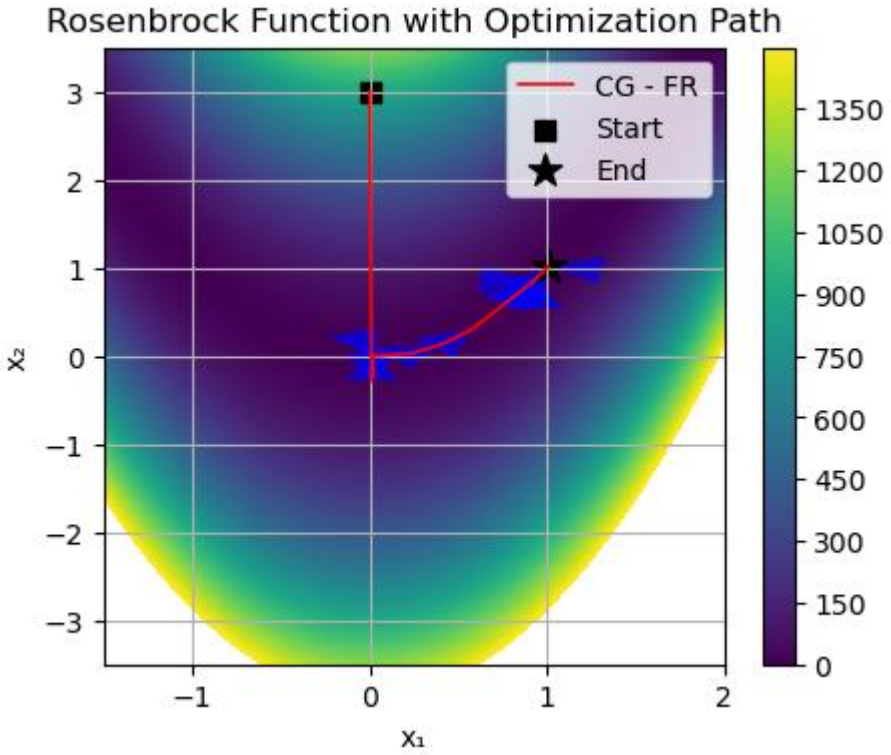

Figure 4. Conjugate gradient Fletcher-Reeves – contour plot of the path

2. The maximum range of an aircraft can be calculated using the the Brequet Range equation.

(a) Plot the design space of the range of the aircraft by manipulating the cruising altitude and velocity for the given variables with 75% of fuel remaining. Have this plot be shown between [0, 300]ms−1 × [0, 25000]m.

➔ Because the range tended to diverge at velocities above 400 m/s and irregular behavior was observed in the air-density equation at very low or very high altitudes, manipulation of the velocity and altitude ranges was performed. Specifically, the lower and upper bounds for velocity were set to 0.01 and 400, respectively, and those for altitude were set to 0.01 and 25,000.
When the computed velocity exceeded 400 m/s, it was fixed at 400 m/s, effectively flattening the function outside the defined bounds before performing the optimization.

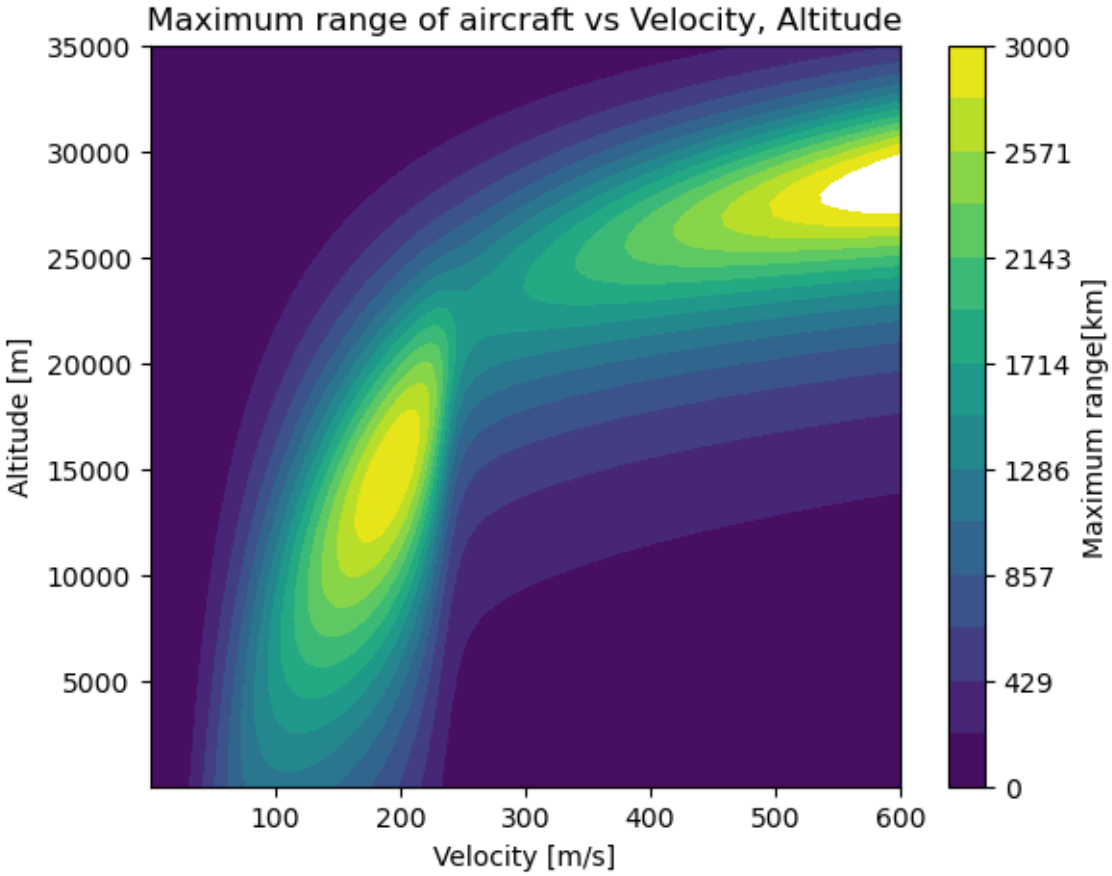

Figure 5. Maximum range of unlimited range of velocity, altitude

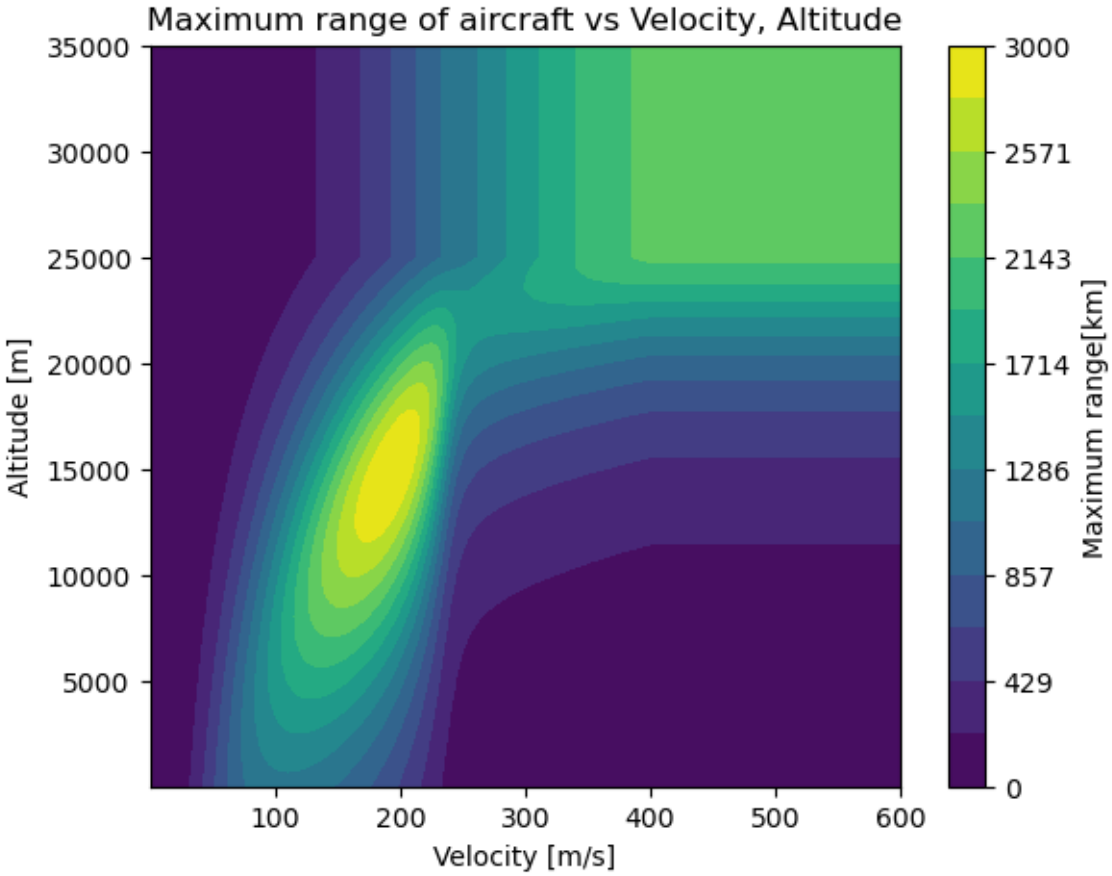

Figure 6. Maximum range of limited range of velocity([1e-2, 400]), altitude([1e-2, 25000])

(b) Find the maximum range of the aircraft with respect to the velocity, V and altitude h using the codes written for problem 1. Use at least two methods (steepest descent and one other method from problem 1). What are the final cruising altitude and velocity to 4 significant digits? Plot the objective function (range), velocity, and altitude as a function of iterations on the same plot. Also, plot the gradient of the objective function as a function of the number of iterations. Comment on the analysis of your results from the two plots.

➔ Since the objective was to maximize the function, the negative of the original function was used as the objective function. The optimization was considered converged when the norm of the gradient became smaller than 0.01. The initial point was selected as a velocity of 50 m/s and an altitude of 5,000 m.

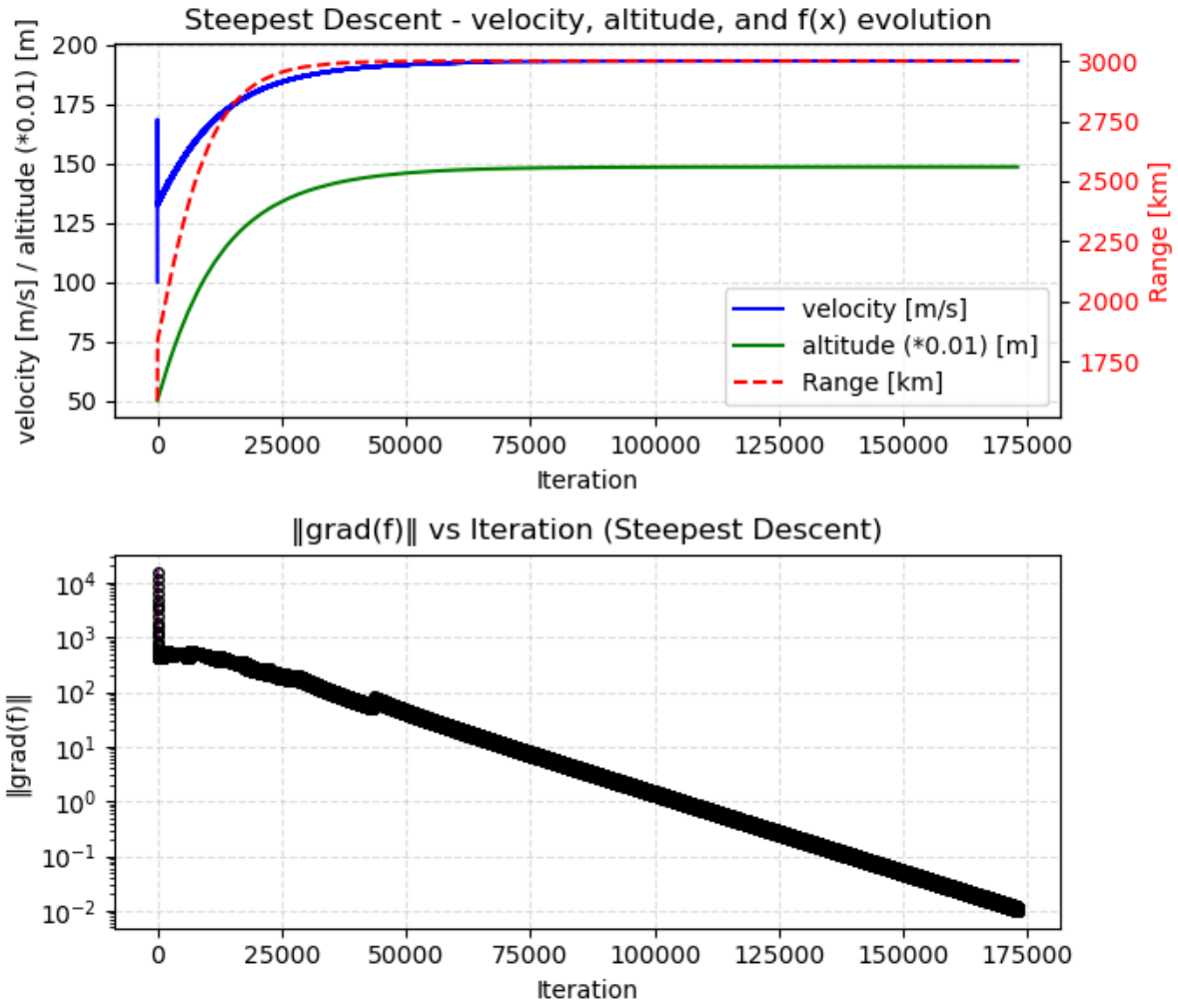

Figure 7. Optimization of range for 75% remaining fuel (using steepest descent).

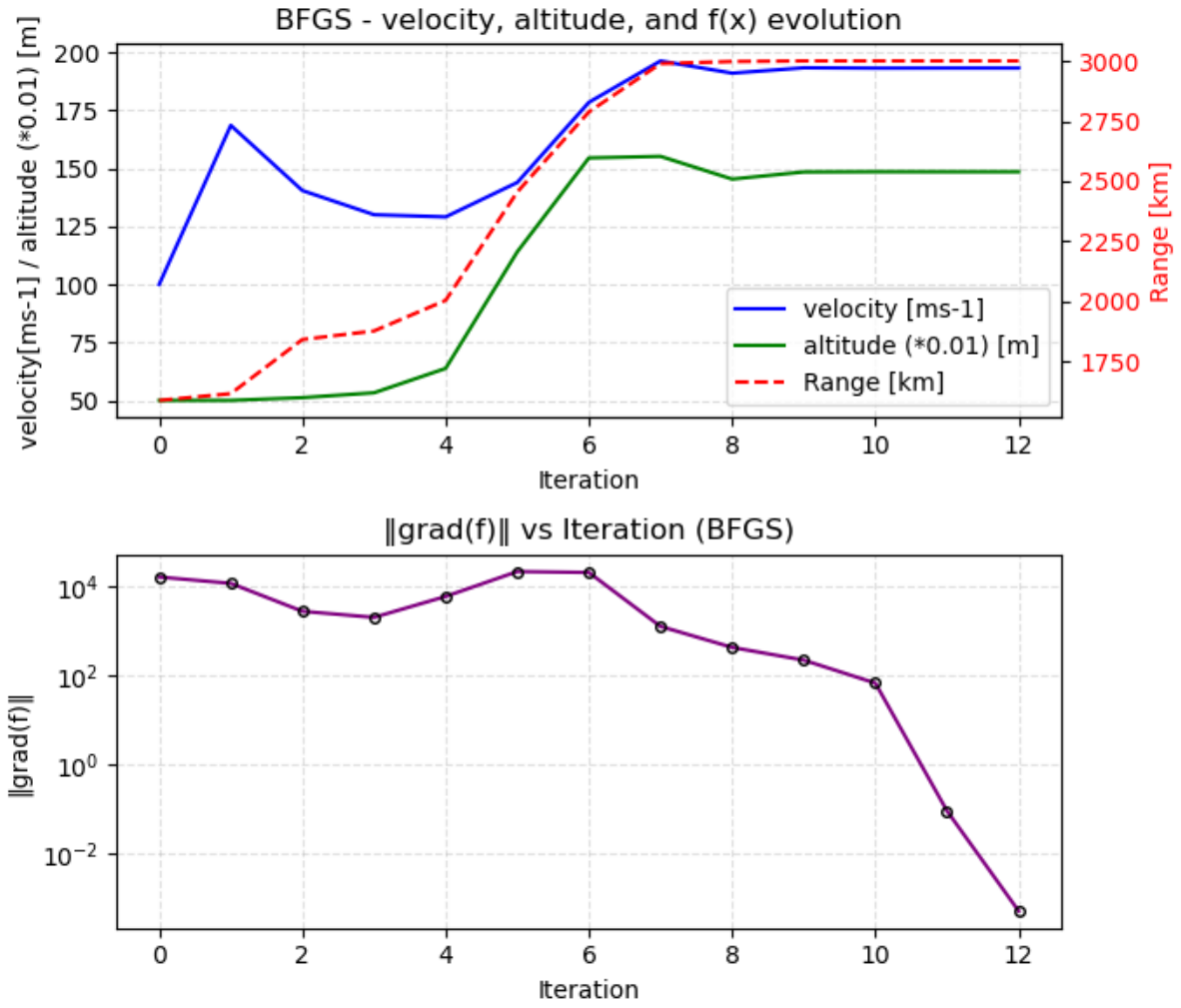

Figure 8. Optimization of range for 75% remaining fuel(using quasi-Newton's BFGS).

The BFGS algorithm converged to the optimal cruising condition in 12 iterations, whereas the steepest descent method required more than 170,000 iterations. The superior performance of BFGS is attributed to its quasi-Newton approximation, which updates the inverse Hessian to capture curvature information efficiently. In contrast, the steepest descent relies only on the gradient direction, resulting in slow zig-zag convergence along the narrow valley of the objective function.

(c) How does the solution differ if 25% of the fuel remains? Repeat (b) for the new condition.

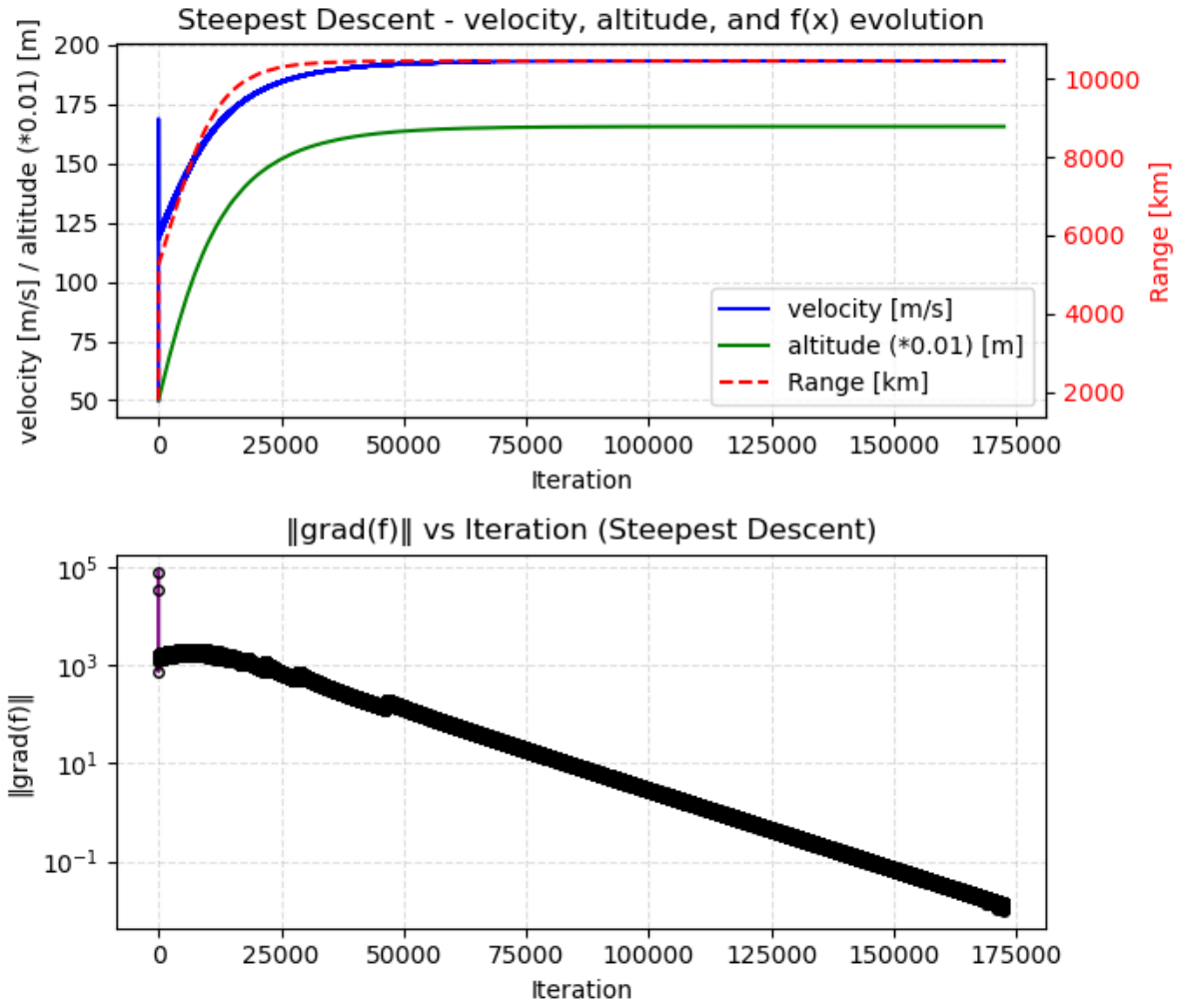

Figure 9. Optimization of range for 25% remaining fuel(using steepest descent).

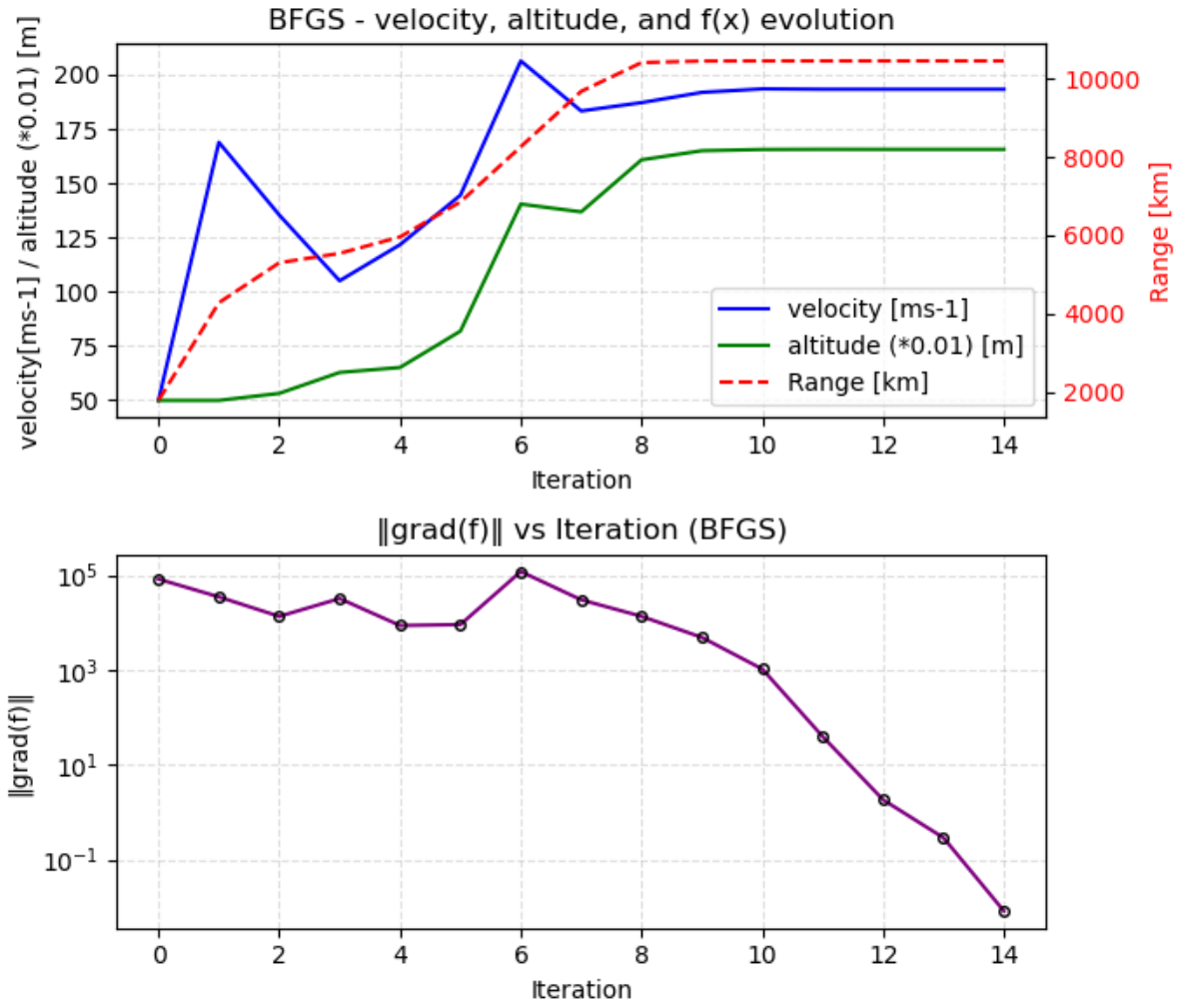

Figure 10. Optimization of range for 25% remaining fuel(using quasi-Newton's BFGS). It showed almost same optimum point with that of 75% fuel remaining case.

Appendix - Codes

```
############################################ Numerical Optimization
모듈(완성본 모음) ############################################
import numpy as np

######################### Central Difference Method - Scalar func / n-dim
point x
### scipy.differentiate.derivative(f, x, ...) 함수 사용 가능
def grad_centraldiff(f, x):
    x = np.atleast_1d(x)
    rel_step = 1e-6
    dfdx = np.zeros(len(x))
    for i in range(len(x)):
        h = rel_step*np.max([np.abs(x[i]), 1])
        e_unit = np.zeros(len(x)); e_unit[i] = 1
        dx = h*e_unit
        num = f(x+dx) - f(x-dx)
        den = 2*h
        dfdx[i] = num/den
    if not np.isfinite(dfdx).all(): # Check finitude
        raise ValueError('At least one component of gradient is not finite !')
    return dfdx

######################### Central Difference Method로 함수의 근사 hessian
계산하는 함수
### scipy.differentiate.hessian(f, x, ...) 함수 사용 가능
def hessian_centraldiff(func, x):

    n = len(x)
    H = np.zeros((n, n))
    h = 1e-5

    for i in range(n):
        for j in range(n):
            x_ij_plus_plus = np.array(x, dtype=float)
            x_ij_plus_minus = np.array(x, dtype=float)
            x_ij_minus_plus = np.array(x, dtype=float)
            x_ij_minus_minus = np.array(x, dtype=float)

            x_ij_plus_plus[i] += h
            x_ij_plus_plus[j] += h

            x_ij_plus_minus[i] += h
            x_ij_plus_minus[j] -= h

            x_ij_minus_plus[i] -= h
```

```
            x_ij_minus_plus[j] += h

            x_ij_minus_minus[i] -= h
            x_ij_minus_minus[j] -= h

            H[i, j] = (func(x_ij_plus_plus) - func(x_ij_plus_minus) -
func(x_ij_minus_plus) + func(x_ij_minus_minus)) / (4 * h**2)
    if not np.isfinite(H).all():
        print(f'Warning : Hessian approximation includes NaN !')
        # raise ValueError('Warning : Hessian approximation includes NaN !')
    return H

########################## Search direction using Steepest Descent Method -
Scalar func / n-dim point x
def search_direction_stp_descent(func, x):
    p = -grad_centraldiff(func, x)

    if not np.isfinite(p).all():
        raise ValueError("Non-finite number detected in
search_direction_stp_descent (NaN or inf)")

    return p

########################## Search direction using Conjugate Gradient Method -
Hestenes Stiefel Formula supplemented by Steepest Descent Method for numerical
stability
######## Nonlinear CG 중 Hestenes-Stiefel algorithm 사용 시 beta 계산에서 분모가
0 되면 폭주 가능.
######## 따라서 Steepest descent랑 섞어서 그런 부분 방지해야 함.
### scipy.optimize.minimize(..., method='CG', ...) 함수 사용 가능
def search_direction_cg_hs(k, grad_old, grad_cur, p_old):
    if k == 0:
        p = -grad_cur
    else:
        num = (grad_cur - grad_old) @ grad_cur
        den = (grad_cur - grad_old) @ p_old
        if abs(den) < 1e-12 or np.isnan(den) or np.isnan(num): # 분모(den)가
0에 가까워지면 beta, p, x_new가 차례로 폭주하는 걸 방지하기 위해 이 경우 steepest
descent method를 대신 사용.
            p = -grad_cur
        else:
            beta = num/den
            p = -grad_cur + beta*p_old

    if not np.isfinite(p).all():
        raise ValueError("Non-finite number detected in search_direction_cg_hs
(NaN or inf)")
```

```
    return p

######################### Search direction using Conjugate Gradient Method - Fletcher-Reeves Formula
######## Nonlinear CG 중 Fletcher-Reeves algorithm 사용 시 beta 계산에서 분모가 0 될 일이 없기에 폭주 가능성 없음.
######## 따라서 Steepest descent랑 섞어서 안 써도 됨. -> 수정 : grad_old가 0에 가까울 시 분모 0 될 수 있음.
### scipy.optimize.minimize(..., method='CG', ...) 함수 사용 가능
def search_direction_cg_fr(k, grad_old, grad_cur, p_old):
    if k == 0:
        p = -grad_cur
    else:
        num = grad_cur @ grad_cur
        den = grad_old @ grad_old

        ####### steepest descent 부분 무효(주석)처리 ########
        if abs(den) < 1e-12 or np.isnan(den) or np.isnan(num): # 분모(den)가 0에 가까워지면 beta, p, x_new가 차례로 폭주하는 걸 방지하기 위해 이 경우 steepest descent method를 대신 사용.
            p = -grad_cur
        else:
            beta = num/den
            p = -grad_cur + beta*p_old

        beta = num/den
        p = -grad_cur + beta*p_old

    if not np.isfinite(p).all():
        raise ValueError("Non-finite number detected in search_direction_cg_fr (NaN or inf)")

    return p

######################### Search direction using Newton's Method
######## Hessian이 PD가 아니거나, x0가 x*에서 너무 먼 경우 수렴 보장 X.
def search_direction_newton(grad, hessian):
    p = -np.linalg.solve(hessian, grad)
    return p

######################### Search direction using Conjugate Gradient Method - Fletcher-Reeves Formula
### scipy.optimize.minimize(..., method='BFGS', ...) 함수 사용 가능
def search_direction_quasi_newton_bfgs(k, x_old, x_cur, grad_old, grad_cur, hessian_inv_aprx_old):
    dim_x = len(x_cur)
    if k == 0: # 첫 iteration의 근사 Hessian inverse는 I로 설정
        hessian_inv_aprx = np.eye(dim_x)
```

```
    else: # 2번째 이후 iteration부터의 근사 Hessian inverse부터는 BFGS로 구함
        dx = x_cur - x_old
        dg = grad_cur - grad_old
        dgdx = dg @ dx
        if abs(dgdx) < 1e-10:  # to avoid division by zero
            hessian_inv_aprx = np.eye(dim_x)
        else:
            I = np.eye(dim_x)
            rho = 1.0 / dgdx
            V = I - rho * np.outer(dx, dg) # 벡터로 행렬 생성 연산은 @ 연산 쓰지
말고 대신 np.outer(a, b) 함수 써라.
            hessian_inv_aprx = V @ hessian_inv_aprx_old @ V.T + rho *
np.outer(dx, dx) # BFGS formula

    p = -hessian_inv_aprx @ grad_cur

    if not np.isfinite(p).all():
        raise ValueError("Non-finite number detected in
search_direction_quasi_newton_bfgs (NaN or inf)")

    return p, hessian_inv_aprx

########################## Step length search A) Backtracking algorithm -
Scalar func / n-dim point x / n-dim grad_x / n-dim search direction p /
curruent iteration k
# backtracking 알고리즘, 더 포괄적으로 step size alpha를 찾는 line search
algorithm은 반드시 함수가 명시적으로 주어져야 한다.
# alpha를 찾기 위해서는 every alpha_try에서 function evaluation을 거쳐야 하기
때문이다.
def backtracking(func, x, grad_x, p, k):
    c1 = 1e-4
    c2 = 0.5

    alpha_try = 1
    x_try = x + alpha_try*p

    i = 0

    print(f"func(x)={func(x)}, func(x_try)={func(x_try)},
grad·p={grad_x.T@p}")

    while func(x_try) > (func(x) + c1*alpha_try*grad_x.T@p):
        i = i + 1
        alpha_try = c2*alpha_try
        x_try = x + alpha_try*p

    alpha = alpha_try
```

```
    print(f'alpha_{k}_{i} = {alpha}\n')

    return alpha

######################### Step length search B) Strong Wolfe's Conditions + Interpolation algorithm- Scalar func / n-dim point x / n-dim grad_x / n-dim search direction p / curruent iteration k
# interpol_alpha ⊂ bracketing_alpha ⊂ wolfe_strong_interpol
def interpol_alpha(f, x_cur, p_cur, a, b):

    phi_a = f(x_cur + a*p_cur)
    phi_b = f(x_cur + b*p_cur)
    dphi_a = grad_centraldiff(f, x_cur + a*p_cur)@p_cur

    dem = 2*(phi_b - phi_a - dphi_a*(b - a)) # 2 차 함수 근사식 기반 minimum point 계산식의 분모

    if (abs(dem) < 1e-12) | (not np.isfinite(dem)): # 분모 수치적으로 불안정하면
        alpha_min = .5*(a + b) # 그냥 구간의 중심점을 minimum point 로 상정하고 return
        return alpha_min
    else: # 분모 수치적으로 안정하면
        num = dphi_a*(b - a) # 분자 계산해주고
        alpha_min = a - num/dem # 2 차근사식 minimum point 계산식 기반 minimum point 구해준다.
        alpha_min = np.clip(alpha_min, a + 0.1*(b - a), b - 0.1*(b - a)) # 만약 minimum point 가 너무 작은 값이면(유의미한 point 이동 못 이뤄냄) 0.1 로, 너무 큰 값(너무 많이 point 이동해도 문제)이면 0.9 로 치환한다.
        return alpha_min

# bracketing_alpha ⊂ wolfe_strong_interpol
def bracketing_alpha(f, x_cur, p_cur, c2_dphi0, phi_armijo, alpha_lo, alpha_hi):
    phi_lo = f(x_cur+alpha_lo*p_cur)

    for _ in range(50):
        alpha_new = interpol_alpha(f, x_cur, p_cur, alpha_lo, alpha_hi) # 다항식 보간함수가 최소가 되는 점 alpha_new 구함
        phi_new = f(x_cur + alpha_new*p_cur) # alpha_new 에서의 함수값
        dphi_new = grad_centraldiff(f, x_cur + alpha_new*p_cur)@p_cur # alpha_new 에서의 기울기

        if (phi_new > phi_armijo(alpha_new)) | (phi_new >= phi_lo): # alpha_new 에서의 함수값이 오히려 증가했다
            alpha_hi = alpha_new # alpha_optm 은 alpha_lo 와 alpha_new 사이 존재 -> 구간 [alpha_lo, alpha_new]로 업뎃
            # alpha_lo unchanged
        else:
```

```
            if abs(dphi_new) <= c2_dphi0: # alpha_new 에서의 함수값이 감소했고
기울기까지 작다
                alpha_optm = alpha_new # alpha_new 가 alpha_optm
                return alpha_optm
            elif dphi_new > 0: # alpha_new 에서의 함수값이 감소했는데 기울기는
여전히 양수다
                alpha_hi = alpha_new # alpha_optm 은 alpha_lo 와 alpha_new 사이
존재 -> 구간 [alpha_lo, alpha_new]로 업뎃
                # alpha_lo unchanged
            else: # alpha_new 에서의 함수값이 감소했는데 기울기가 음수다
                alpha_lo = alpha_new # alpha_optm 은 alpha_new 와 alpha_hi 사이
존재 -> 구간 [alpha_new, alpha_hi]로 업뎃
                # alpha_hi unchanged

        phi_lo = f(x_cur + alpha_lo*p_cur) # 업뎃된 alpha_lo 에서의 함수값 계산

        if abs(alpha_hi - alpha_lo) < 1e-8: # 만약 구간이 충분히 줄어들었으면 그냥
탈출해서 구간의 절반지점을 alpha_optm 으로 return
            break

    return 0.5*(alpha_lo + alpha_hi)

# wolfe_strong_interpol
def wolfe_strong_interpol(f, x_cur, f_cur, grad_cur, p_cur, c2):
    c1 = 1e-4 # Armijo 조건, Curvature 조건용 factors
    alpha_try_old, alpha_try = 0, 1 # Initial bracket of alpha

    phi0 = f_cur # Armijo 함수 생성용
    dphi0 = grad_cur@p_cur # Armijo 함수 생성용 및 Curvature 조건 비교용

    phi_armijo = lambda alpha : phi0 + c1*alpha*dphi0 # Armijo 람다 함수 정의

    for _ in range(50):
        x_try = x_cur + alpha_try*p_cur
        phi_try = f(x_try)
        dphi_try = grad_centraldiff(f, x_try)@p_cur

        phi_armijo_try = phi_armijo(alpha_try)

        x_try_old = x_cur + alpha_try_old*p_cur
        phi_try_old = f(x_try_old)

        if (phi_try > phi_armijo_try) | (phi_try > phi_try_old): # phi_try 가
충분히 크다면 -> alpha_optm 이 alpha_try_old 와 alpha_try 사이 존재
            alpha_lo, alpha_hi = alpha_try_old, alpha_try
            alpha_optm = bracketing_alpha(f, x_cur, p_cur, abs(c2*dphi0),
phi_armijo, alpha_lo, alpha_hi) # bracketing 하고 interpolation iteration
돌려서 alpha_optm 뽑아내자
```

```
            return alpha_optm

        elif abs(dphi_try) <= -c2*dphi0: # phi_try 가 충분히 작고 기울기까지
작다면
            alpha_optm = alpha_try # 그 점이 alph_optm 이다
            return alpha_optm

        elif dphi_try >= 0: # phi_try 가 충분히 작긴 한데 기울기가 양수라면 더 작은
phi 값을 가지는 alpha_optm 이 alpha_try_old 와 alpha_try 사이 존재
            alpha_lo, alpha_hi = alpha_try_old, alpha_try
            alpha_optm = bracketing_alpha(f, x_cur, p_cur, abs(c2*dphi0),
phi_armijo, alpha_lo, alpha_hi) # bracketing 하고 interpolation iteration
돌려서 alpha_optm 뽑아내자
            return alpha_optm

        else: # phi_try 가 충분히 작긴 한데 기울기가 음수라면 더 작은 phi 값을
가지는 alpha_optm 은 alpha_try 보다 뒤의 구간에 존재 -> 구간 업데이트
            alpha_try_old = alpha_try # 다음 구간의 하한 = 현재 구간의 상한
            alpha_try = min(alpha_try * 2, 10.0) # 다음 구간의 상한 = 현재 구간
상한의 2 배. 최대는 10 으로 제한

    if not np.isfinite(alpha_try):
        alpha_try = 1e-3
    return max(min(alpha_try, 1.0), 1e-6)

##############################################################################
########################################
########################## Main Optimization Algorithm
##########################
def stp_descent(f, x0, tol):
    if type(x0) != np.ndarray:
        raise ValueError('Please input ndarray type !!')
    elif len(x0.shape) >= 2:
        raise ValueError('Please input vector type ndarray !! ')

    f0 = f(x0)
    if not np.isfinite(f0).all():
        raise ValueError('Function value at x0 is not finite. Try another
x0 !')
    else:
        pass

    ### Check gradient of x0
    grad0 = grad_centraldiff(f, x0)
    if np.linalg.norm(grad0) < tol: # Check optimality
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} < {tol}, x0 : {x0}
is optimum point !')
        # return x0
```

```
    else:
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} > {tol}, x0 : {x0}
is not an optimum point. Optimization begins !')
        pass

    ### Initialization for searching iterations
    x_new = x0
    f_new = f0
    grad_new = grad0
    err = 1
    k = 0

    ############### Searching iterations
    ###### Update info of current point
    while err > tol:
        x_cur = x_new
        f_cur = f_new
        grad_cur = grad_new

        ###### Line search
        ### Search direction - by steepest gradient method
        p_cur = search_direction_stp_descent(f, x_cur)
        # Search direction check
        if grad_cur@p_cur > 0: # search direction이 증가 방향이면 경고 내보내고
steepest descent direction으로 search direction 변경
            print(f'Warning : grad(x_k)·p_k={grad_cur@p_cur} > 0 : Search
direction p_k would likely make function increase !')
            print(f'Warning : p_k would be replaced with steepest descent
direction grad(x0) : {-grad_cur} !')
            p_cur = search_direction_stp_descent(f, x_cur)

        ### Step length - by Strong Wolfe + interpolation
        alpha_cur = wolfe_strong_interpol(f, x_cur, f_cur, grad_cur, p_cur,
0.9)

        ##### x_new update
        k = k + 1
        x_new = x_cur + alpha_cur*p_cur
        f_new = f(x_new)
        grad_new = grad_centraldiff(f, x_new)
        err = np.linalg.norm(grad_new)
        print(f'x_{k} : {x_new}')
        print(f'f_{k} : {f_new}')
        print(f'norm(grad(x_{k})) : {err}')
        print(f'recent alpha : {alpha_cur}')
        print(f'recent p : {p_cur}')
        print()
```

```
    print(f'Optimization converges -> Iteration : {k} / x* : {x_new} / f(x*) :
{f(x_new)} / norm(grad(x*)) : {np.linalg.norm(grad_new)} ')
    return x_new

def cg_hs(f, x0, tol):
    if type(x0) != np.ndarray:
        raise ValueError('Please input ndarray type !!')
    elif len(x0.shape) >= 2:
        raise ValueError('Please input vector type ndarray !! ')

    f0 = f(x0)
    if not np.isfinite(f0).all():
        raise ValueError('Function value at x0 is not finite. Try another
x0 !')
    else:
        pass

    ### Check gradient of x0
    grad0 = grad_centraldiff(f, x0)
    if np.linalg.norm(grad0) < tol: # Check optimality
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} < {tol}, x0 : {x0}
is optimum point !')
        # return x0
    else:
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} > {tol}, x0 : {x0}
is not an optimum point. Optimization begins !')
        pass

    ### Initialization for searching iterations
    p_cur = None
    grad_cur = None

    x_new = x0
    f_new = f0
    grad_new = grad0

    err = 1
    k = 0

    ############### Searching iterations
    ###### Update info of current point
    while err > tol:
        grad_old = grad_cur
        p_old = p_cur

        x_cur = x_new
        f_cur = f_new
        grad_cur = grad_new
```

```
        ###### Line search
        ### Search direction - by Conjugate Gradient_HS method
        p_cur = search_direction_cg_hs(k, grad_old, grad_cur, p_old)
        # Search direction check
        if grad_cur@p_cur > 0: # search direction이 증가 방향이면 경고 내보내고
steepest descent direction으로 search direction 변경
            print(f'Warning : grad(x_k)·p_k={grad_cur@p_cur} > 0 : Search
direction p_k would likely make function increase !')
            print(f'Warning : p_k would be replaced with steepest descent
direction grad(x0) : {-grad_cur} !')
            p_cur = search_direction_stp_descent(f, x_cur)

        ### Step length - by Strong Wolfe + interpolation
        alpha_cur = wolfe_strong_interpol(f, x_cur, f_cur, grad_cur, p_cur,
0.1)

        ##### x_new update
        k = k + 1
        x_new = x_cur + alpha_cur*p_cur
        f_new = f(x_new)
        grad_new = grad_centraldiff(f, x_new)
        err = np.linalg.norm(grad_new)
        print(f'x_{k} : {x_new}')
        print(f'f_{k} : {f_new}')
        print(f'norm(grad(x_{k})) : {err}')
        print(f'recent alpha : {alpha_cur}')
        print(f'recent p : {p_cur}')
        print()

    print(f'Optimization converges -> Iteration : {k} / x* : {x_new} / f(x*) :
{f(x_new)} / norm(grad(x*)) : {np.linalg.norm(grad_new)} ')
    return x_new

def cg_fr(f, x0, tol):
    if type(x0) != np.ndarray:
        raise ValueError('Please input ndarray type !!')
    elif len(x0.shape) >= 2:
        raise ValueError('Please input vector type ndarray !! ')

    f0 = f(x0)
    if not np.isfinite(f0).all():
        raise ValueError('Function value at x0 is not finite. Try another
x0 !')
    else:
        pass

    ### Check gradient of x0
```

```
    grad0 = grad_centraldiff(f, x0)
    if np.linalg.norm(grad0) < tol: # Check optimality
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} < {tol}, x0 : {x0}
is optimum point !')
        # return x0
    else:
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} > {tol}, x0 : {x0}
is not an optimum point. Optimization begins !')
        pass

    ### Initialization for searching iterations
    p_cur = None
    grad_cur = None

    x_new = x0
    f_new = f0
    grad_new = grad0

    err = 1
    k = 0

    ############### Searching iterations
    ###### Update info of current point
    while err > tol:
        grad_old = grad_cur
        p_old = p_cur

        x_cur = x_new
        f_cur = f_new
        grad_cur = grad_new

        ###### Line search
        ### Search direction - by Conjugate Gradient_FR method
        p_cur = search_direction_cg_fr(k, grad_old, grad_cur, p_old)
        # Search direction check
        if grad_cur@p_cur > 0: # search direction이 증가 방향이면 경고 내보내고
steepest descent direction으로 search direction 변경
            print(f'Warning : grad(x_k)·p_k={grad_cur@p_cur} > 0 : Search
direction p_k would likely make function increase !')
            print(f'Warning : p_k would be replaced with steepest descent
direction grad(x0) : {-grad_cur} !')
            p_cur = search_direction_stp_descent(f, x_cur)

        ### Step length - by Strong Wolfe + interpolation
        alpha_cur = wolfe_strong_interpol(f, x_cur, f_cur, grad_cur, p_cur,
0.1)

        ##### x_new update
```

```
        k = k + 1
        x_new = x_cur + alpha_cur*p_cur
        f_new = f(x_new)
        grad_new = grad_centraldiff(f, x_new)
        err = np.linalg.norm(grad_new)
        print(f'x_{k} : {x_new}')
        print(f'f_{k} : {f_new}')
        print(f'norm(grad(x_{k})) : {err}')
        print(f'recent alpha : {alpha_cur}')
        print(f'recent p : {p_cur}')
        print()

    print(f'Optimization converges -> Iteration : {k} / x* : {x_new} / f(x*) :
{f(x_new)} / norm(grad(x*)) : {np.linalg.norm(grad_new)} ')
    return x_new

def newton(f, x0, tol):
    if type(x0) != np.ndarray:
        raise ValueError('Please input ndarray type !!')
    elif len(x0.shape) >= 2:
        raise ValueError('Please input vector type ndarray !! ')

    f0 = f(x0)
    if not np.isfinite(f0).all():
        raise ValueError('Function value at x0 is not finite. Try another
x0 !')
    else:
        pass

    ### Check gradient of x0
    grad0 = grad_centraldiff(f, x0)
    if np.linalg.norm(grad0) < tol: # Check optimality
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} < {tol}, x0 : {x0}
is optimum point !')
        # return x0
    else:
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} > {tol}, x0 : {x0}
is not an optimum point. Optimization begins !')
        pass

    ### Initialization for searching iterations
    p_cur = None
    grad_cur = None

    x_new = x0
    f_new = f0
    grad_new = grad0
```

```
    err = 1
    k = 0

    ############### Searching iterations
    ###### Update info of current point
    while err > tol:
        x_cur = x_new
        f_cur = f_new
        grad_cur = grad_new
        hessian_cur = hessian_centraldiff(f, x_cur)

        ###### Line search
        ### Search direction - by Newton method
        p_cur = search_direction_newton(grad_cur, hessian_cur)

        # Search direction check
        if grad_cur@p_cur > 0: # search direction 이 증가 방향이면 경고 내보내고
steepest descent direction 으로 search direction 변경
            print(f'Warning : grad(x_k)·p_k={grad_cur@p_cur} > 0 : Search
direction p_k would likely make function increase !')
            print(f'Warning : p_k would be replaced with steepest descent
direction grad(x0) : {-grad_cur} !')
            p_cur = search_direction_stp_descent(f, x_cur)

        ### Step length - by Strong Wolfe + interpolation
        alpha_cur = wolfe_strong_interpol(f, x_cur, f_cur, grad_cur, p_cur,
0.9)

        ##### x_new update
        k = k + 1
        x_new = x_cur + alpha_cur*p_cur
        f_new = f(x_new)
        grad_new = grad_centraldiff(f, x_new)
        err = np.linalg.norm(grad_new)
        print(f'x_{k} : {x_new}')
        print(f'f_{k} : {f_new}')
        print(f'norm(grad(x_{k})) : {err}')
        print(f'recent alpha : {alpha_cur}')
        print(f'recent p : {p_cur}')
        print()

    print(f'Optimization converges -> Iteration : {k} / x* : {x_new} / f(x*) :
{f(x_new)} / norm(grad(x*)) : {np.linalg.norm(grad_new)} ')
    return x_new

def quasi_newton_bfgs(f, x0, tol):
    if type(x0) != np.ndarray:
        raise ValueError('Please input ndarray type !!')
```

```
    elif len(x0.shape) >= 2:
        raise ValueError('Please input vector type ndarray !! ')

    f0 = f(x0)
    if not np.isfinite(f0).all():
        raise ValueError('Function value at x0 is not finite. Try another
x0 !')
    else:
        pass

    ### Check gradient of x0
    grad0 = grad_centraldiff(f, x0)
    if np.linalg.norm(grad0) < tol: # Check optimality
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} < {tol}, x0 : {x0}
is optimum point !')
        # return x0
    else:
        print(f'Since |grad(x0)| = {np.linalg.norm(grad0)} > {tol}, x0 : {x0}
is not an optimum point. Optimization begins !')
        pass

    ### Initialization for searching iterations
    x_cur = None
    grad_cur = None
    hessian_inv_aprx_cur = np.identity(len(x0))

    x_new = x0
    f_new = f0
    grad_new = grad0

    err = 1
    k = 0

    ############### Searching iterations
    ###### Update info of current point
    while err > tol:
        x_old = x_cur
        grad_old = grad_cur
        hessian_inv_aprx_old = hessian_inv_aprx_cur

        x_cur = x_new
        f_cur = f_new
        grad_cur = grad_new

        ###### Line search
        ### Search direction - by Quasi-Newton's(BFGS) method
        p_cur, hessian_inv_aprx_cur = search_direction_quasi_newton_bfgs(k,
x_old, x_cur, grad_old, grad_cur, hessian_inv_aprx_old)
```

```
        # Search direction check
        if grad_cur@p_cur > 0: # search direction 이 증가 방향이면 경고 내보내고
steepest descent direction 으로 search direction 변경
            print(f'Warning : grad(x_k)·p_k={grad_cur@p_cur} > 0 : Search
direction p_k would likely make function increase !')
            print(f'Warning : p_k would be replaced with steepest descent
direction grad(x0) : {-grad_cur} !')
            p_cur = search_direction_stp_descent(f, x_cur)

        ### Step length - by Strong Wolfe + interpolation
        alpha_cur = wolfe_strong_interpol(f, x_cur, f_cur, grad_cur, p_cur,
0.9)

        ##### x_new update
        k = k + 1
        x_new = x_cur + alpha_cur*p_cur
        f_new = f(x_new)
        grad_new = grad_centraldiff(f, x_new)
        err = np.linalg.norm(grad_new)
        print(f'x_{k} : {x_new}')
        print(f'f_{k} : {f_new}')
        print(f'norm(grad(x_{k})) : {err}')
        print(f'recent alpha : {alpha_cur}')
        print(f'recent p : {p_cur}')
        print()

    print(f'Optimization converges -> Iteration : {k} / x* : {x_new} / f(x*) :
{f(x_new)} / norm(grad(x*)) : {np.linalg.norm(grad_new)} ')
    return x_new
```