Numerical Optimization, 2021 Fall
Homework 8 Solution

1 Curvature Condition

Show that if f is strongly convex, then the *curvature condition*

$$s_k^T y_k > 0 \tag{1}$$

holds for any vectors x_k and x_{k+1}, where $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$. [20pts]
A function $f(x)$ is strongly convex if all eigenvalues of $\nabla^2 f(x)$ are positive and bounded away from zero. This implies that there exists $\sigma > 0$ such that

$$d^T \nabla^2 f(x) d \geq \sigma \|d\|^2 \quad \text{for any } d \tag{2}$$

By Taylor's theorem, if $x_{k+1} = x_k + \alpha_k d_k$, then

$$\nabla f(x_{k+1}) = \nabla f(x_k) + \int_0^1 \left[\nabla^2 f(x_k + z\alpha_k d_k)\alpha_k d_k\right] dz \tag{3}$$

Then we have

$$\begin{aligned} s_k^T y_k &= \alpha_k d_k^T y_k \\ &= \alpha_k d_k^T \left[\nabla f(x_{k+1} - \nabla f(x_k)\right] \\ &= \alpha_k^2 \int_0^1 \left[d_k^T \nabla^2 f(x_k + z\alpha_k d_k) d_k\right] dz \\ &\geq \sigma \|d_k\|^2 \alpha_k^2 > 0 \end{aligned} \tag{4}$$

2 Second Strong Wolfe Condition

Show that the second *strong Wolfe condition*

$$\left|\nabla f(x_k + \alpha_k p_k)^T p_k\right| \leq c_2 \left|\nabla f_k^T p_k\right| \tag{5}$$

(where α_k is the step length, p_k is a descent direction and $0 < c_1 < c_2 < 1$), implies the curvature condition as described in (1). [20pts]
The second strong Wolfe condition (5) implies that

$$\begin{aligned} \nabla f(x_k + \alpha_k p_k)^T p_k &\geq -c_2 \left|\nabla f(x_k)^T p_k\right| \\ &= c_2 \nabla f(x_k)^T p_k \end{aligned} \tag{6}$$

Since p_k is a descent direction. Thus

$$\begin{aligned}\nabla f(x_k + \alpha_k p_k)^T p_k - \nabla f(x_k)^T p_k &\geq (c_2 - 1)\nabla f(x_k)^T p_k \\ &> 0\end{aligned} \tag{7}$$

The inequality holds because we assume $c_2 < 1$. Hence, the result follows by multiplying both sides by α_k and noting

$$\begin{aligned}s_k &= x_{k+1} - x_k = \alpha_k p_k \\ y_k &= \nabla f(x_k + \alpha_k p_k)^T - \nabla f(x_k)^T\end{aligned} \tag{8}$$

3 SR1 Update

If the square nonsingular matrix A undergoes a rank-one update to become

$$\bar{A} = A + ab^T \tag{9}$$

where $a, b \in \mathbb{R}^n$, then if $\bar{A}$ is nonsingular, we have

$$\bar{A}^{-1} = A^{-1} - \frac{A^{-1}ab^T A^{-1}}{1 + b^T A^{-1} a}. \tag{10}$$

which is known as the *Sherman–Morrison formula*. Use this formula to show that in SR1 update

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k} \tag{11}$$

is the inverse of

$$H_{k+1} = H_k + \frac{(s_k - H_k y_k)(s_k - H_k y_k)^T}{(s_k - H_k y_k)^T y_k}. \tag{12}$$

where B_k is the Hessian approximation and H_k is the inverse Hessian approximation. [20pts]
First we note that the rank-1 update of B_k can be written as

$$B_{k+1} = B_k + \alpha(y_k - B_k s_k)(y_k - B_k s_k)^T \tag{13}$$

where $\alpha = \left((y_k - B_k s_k)^T s_k\right)^{-1}$. So in the Sherman–Morrison formula, we have

$$\begin{aligned}a &= \alpha(y_k - B_k s_k) \\ b &= y_k - B_k s_k\end{aligned} \tag{14}$$

With formula (10) and the following relations

$$\begin{aligned}B_k &= H_k^{-1} \\ H^T &= H \\ B^T &= B\end{aligned} \tag{15}$$

we have

$$\begin{aligned}H_{k+1} = B_{k+1}^{-1} &= B_k^{-1} - \frac{B_k^{-1}\alpha(y_k - B_k s_k)(y_k - B_k s_k)^T B_k^{-1}}{1 + (y_k - B_k s_k)^T B_k^{-1}\alpha(y_k - B_k s_k)} \\ &= H_k - \frac{\alpha(H_k y_k - s_k)\left(y_k^T H_k^T - s_k^T B_k B_k^{-1}\right)}{1 + \alpha\left(y_k^T H_k - s_k^T B_k B_k^{-1}\right)(y_k - B_k s_k)}\end{aligned}$$

$$
\begin{aligned}
&= H_k - \frac{\left(s_k - H_k y_k\right)\left(s_k^T - y_k^T H_k^T\right)}{\alpha^{-1} + \left(y_k^T H_k y_k - y_k^T s_k - s_k^T y_k + s_k^T B_k s_k\right)} \\
&= H_k - \frac{\left(s_k - H_k y_k\right)\left(s_k^T - \left(H_k y_k\right)^T\right)}{\left(y_k - B_k s_k\right)^T s_k + \left(y_k^T H_k y_k - y_k^T s_k - s_k^T y_k + s_k^T B_k s_k\right)} \\
&= H_k + \frac{\left(s_k - H_k y_k\right)\left(s_k - H_k y_k\right)^T}{s_k^T y_k - y_k^T H_k y_k} \\
&= H_k + \frac{\left(s_k - H_k y_k\right)\left(s_k - H_k y_k\right)^T}{s_k^T y_k - \left(H_k y_k\right)^T y_k} \\
&= H_k + \frac{\left(s_k - H_k y_k\right)\left(s_k - H_k y_k\right)^T}{\left(s_k - H_k y_k\right)^T y_k}.
\end{aligned} \tag{16}
$$

4 Coding

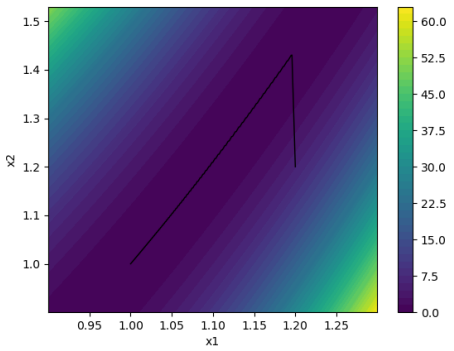
Please solve the following problem using *Quasi-Newton Method* with *BFGS Update*. [40pts]

$$
\min_{x_1, x_2} 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2 \tag{17}
$$

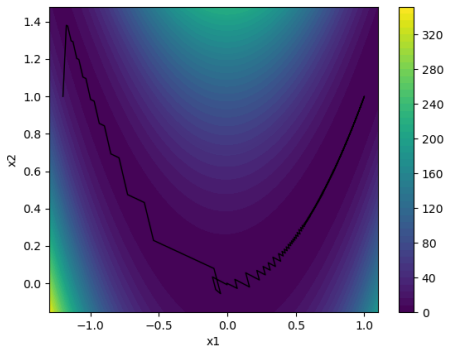
Requirements:

1. Use two starting points ($\boldsymbol{x}^0 = [1.2, 1.2]^T$ and $\boldsymbol{x}^0 = [-1.2, 1]^T$) to initiate your algorithms. For each case, draw the path of the algorithm from start to convergence;
2. Use Wolfe line search. You may use Wolfe line search from the beginning, or you can use Armijo line search at first and switch to Wolfe line search when the curvature condition is not met;
3. Your program may be implemented with Python or MATLAB. Please paste your images down below, then put your code in an independent file and submit it to Blackboard alongside the PDF.

Hint: You may set initial step size $\alpha_0 = 1$, and termination condition as $\|\nabla f(\boldsymbol{x}^k)\|_\infty \leq 10^{-4}$.



(a) $\boldsymbol{x}^0 = [1.2, 1.2]^T$



(b) $\boldsymbol{x}^0 = [-1.2, 1]^T$

Figure 1: Paths for two initial points.

Code:

```
import copy
import numpy as np
import argparse
from numpy import linalg as LA
import matplotlib.pyplot as plt


# Calculate the objective function
def calcObj(point):
    return 100 * ((point[1] - point[0] ** 2) ** 2) + (1 - point[0]) ** 2


# Calculate the gradient at current point
def calcJacobian(point):
    dx1 = -2 * (1 - point[0]) + 200 * (point[1] - point[0] ** 2) * (-2 * point[0])
    dx2 = 200 * (point[1] - (point[0] ** 2))
    return np.array([dx1, dx2])


# Calculate the Hessian matrix
def calcHessian(x):
    return np.array([[-400 * x[1] + 1200 * x[0] ** 2 + 2, -400 * x[0]],
                     [-400 * x[0],                200]])


# Find alpha^k with wolfe backtracking
def armijo(point, direction):
    alpha = 1
    c1 = 1e-4
    rho = 0.8
    f = calcObj(point)
    grad = calcJacobian(point)
    while calcObj(point + alpha * direction) > f + c1 * alpha * grad.T @ direction:
        alpha *= rho
    return alpha


# Find alpha^k with wolfe line search
def wolfe(point, direction, s_k, y_k):
    c1 = 1e-4
    c2 = 0.9
    rho = 0.8
    gamma = 1.2

    alpha = armijo(point, direction)
```

```
    f = calcObj(point)
    grad = calcJacobian(point)

    if y_k.T @ s_k <= 0:
        cond1 = (calcObj(point + alpha * direction) > f + c1 * alpha * grad.T @
            direction).any()
        cond2 = ((calcJacobian(point + alpha * direction).T @ direction) < c2 * grad.T
            @ direction).any()
        while (not cond1) or (not cond2):
            if not cond1:
                alpha *= rho
            elif not cond2:
                alpha *= gamma
            cond1 = (calcObj(point + alpha * direction) > f + c1 * alpha * grad.T @
                direction).any()
            cond2 = ((calcJacobian(point + alpha * direction).T @ direction) < c2 *
                grad.T @ direction).any()

    return alpha


# Visualize path
def drawPath(points):
    # Unzip points
    x1s = np.array([p[0] for p in points])
    x2s = np.array([p[1] for p in points])
    max_x1, max_x2 = max(x1s) + 0.1, max(x2s) + 0.1
    min_x1, min_x2 = min(x1s) - 0.1, min(x2s) - 0.1

    step = 0.001
    X1, X2 = np.meshgrid(np.arange(min_x1, max_x1, step), np.arange(min_x2, max_x2,
        step))
    F = 100 * ((X2 - X1 ** 2) ** 2) + (1 - X1) ** 2

    plt.contourf(X1, X2, F, 50); plt.colorbar(); plt.contour(X1, X2, F)
    plt.plot(x1s, x2s, color="black", linewidth=1, linestyle="-")
    plt.xlabel("x1"); plt.ylabel("x2")
    plt.show()


# Visualize alpha/gradient updates
def drawUpdates(pairs, ylabel):
    i = np.array([p[0] for p in pairs])
    value = np.array([p[1] for p in pairs])

    plt.plot(i, np.log10(value), color="blue", linewidth=1, linestyle="-")
    plt.xlabel("iteration"); plt.ylabel(ylabel)
```

```
    plt.show()


def main(x1, x2):
    point = np.array([x1, x2]) # The initial point

    tol = 1e-4
    epochs = 100000

    x_f_record = []
    alpha_record = []
    norm_record = []

    m = np.shape(point)[0]
    I = np.eye(m)
    H_k = LA.inv(calcHessian(point))
    s_k = np.array([1, 1])
    y_k = np.array([1, 1])

    for i in range(epochs):
        f = calcObj(point)
        grad = calcJacobian(point)
        direction = - H_k @ grad
        alpha_k = wolfe(point, direction, s_k, y_k)

        point_old = copy.deepcopy(point)
        grad_old = copy.deepcopy(grad)

        x_f_record.append([copy.deepcopy(point_old), f])
        norm_record.append([i, LA.norm(grad_old, np.inf)])
        alpha_record.append([i, alpha_k])

        print("Iteration:", i, x_f_record[i])
        if LA.norm(grad, np.inf) < tol:
            break
        else:
            point += direction * alpha_k
            grad = calcJacobian(point)
            s_k = point - point_old
            y_k = grad - grad_old

            H_k = (I - (s_k @ y_k.T) / (y_k.T @ s_k) ) @ H_k @ (I - (y_k @ s_k.T) /
                (y_k.T @ s_k)) + ((s_k @ s_k.T) // (y_k.T @ s_k))

    print("The minimum is: {0:.4} at point ({1:.4}, {2:.4})".format(x_f_record[-1][1],
        x_f_record[-1][0][0], x_f_record[-1][0][1]))
    drawPath([x_f[0] for x_f in x_f_record])
```

```
    drawUpdates(norm_record, ylabel="norm (log10)")
    drawUpdates(alpha_record, ylabel="alpha (log10)")


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Initial point.')
    parser.add_argument('x1', type=float, help='value for x1')
    parser.add_argument('x2', type=float, help='value for x2')
    args = parser.parse_args()
    main(args.x1, args.x2)
```