va

Advanced Optimization

Homework 3.

—

Ali Izadi

810199102

سوالات تحلیلی

سوال ۱

سوال ۲

طبق صورت سوال داریم:

$$\bar{x}_k = \arg\min_{x \in X} \left\{ \nabla f(x_k)^T (x - x_k) + \frac{1}{2s_k}(x - x_k)^T H_k (x - x_k) \right\}$$

عبارت داخل کروشه را در $2s_k$ ضرب میکنیم و عبارت را مطابق با روابط نوشته شده بسط میدهیم تا به فرم norm خواسته شده برسیم.. بنابراین خواهیم داشت:

$$\left(\nabla f(x_k)^T (x - x_k) + \frac{1}{2s_k}(x - x_k)^T H_k (x - x_k) \right) \times 2s_k$$

$$= 2s_k \nabla f(x_k)^T (x - x_k) + (x - x_k)^T H_k (x - x_k)$$

$$= 2s_k (x - x_k)^T H_k H_k^{-1} \nabla f(x_k) + (x - x_k)^T H_k (x - x_k)$$

$$= (x - x_k)^T H_k (x - x_k) + (x - x_k)^T H_k s_k H_k^{-1} \nabla f(x_k) + \left(s_k H_k^{-1} \nabla f(x_k) \right)^T H_k (x - x_k)^T$$

$$+ \cancel{\left(s_k H_k^{-1} \nabla f(x_k) \right)^T H_k\, s_k H_k^{-1} \nabla f(x_k)} \; constant$$

$$= \left(x - x_k + s_k H_k^{-1} \nabla f(x_k) \right)^T H_k \left(x - x_k + s_k H_k^{-1} \nabla f(x_k) \right)$$

بنابراین مسئله اصلی مطابق با مسئله زیر خواهد بود:

$$\bar{x}_k = \arg\min_{x \in X} \left\{ \left(x - x_k + s_k H_k^{-1} \nabla f(x_k) \right)^T H_k \left(x - x_k + s_k H_k^{-1} \nabla f(x_k) \right) \right\}$$

عبارت داخل کروشه به فرم نرم داده شده است یعنی:

$$\left(x - x_k + s_k H_k^{-1} \nabla f(x_k)\right)^T H_k \left(x - x_k + s_k H_k^{-1} \nabla f(x_k)\right) \;=\; \left|\left|x - x_k + s_k H_k^{-1} \nabla f(x_k)\right|\right|_{H_k}^2$$

بنابراین خواهیم داشت:

$$\begin{aligned} \bar{x}_k =&\; \arg\min_{x \in X} \left|\left|x - x_k + s_k H_k^{-1} \nabla f(x_k)\right|\right|_{H_k}^2 \\ =&\; \arg\min_{x \in X} \frac{1}{2} \left|\left|x - x_k + s_k H_k^{-1} \nabla f(x_k)\right|\right|_{H_k}^2 \end{aligned}$$

و در نتیجه به عبارت خواسته شده در مسئله میرسیم:

$$\bar{x}_k = \; \arg\min_{x \in X} \left|\left|x - \;\left(x_k - s_k H_k^{-1} \nabla f(x_k)\right)\right|\right|_{H_k}^2$$

- نحوه حل مسئله:

برای به دست آوردن روابط از حکم به فرض روابط را بسط دادم و در آخر عکس مراحل را به عنوان جواب در بالا آوردم.

سوالات پیاده سازی

سوال ۱

روش manifolds sub-optimization:

برای پیاده سازی این روش مطابق با الگوریتم روش مجموعه فعال که حالت خاصی از الگوریتم manifold و برای تابع درجه ۲ است و با استفاده از الگوریتم گفته شده در کتاب nocedal عمل کرده ایم.

شبه کد الگوریتم مطابق با کتاب در زیر آورده شده است که در ادامه به توضیح هر قسمت و پیاده سازی انجام شده خواهیم پرداخت.

Algorithm 16.3 (Active-Set Method for Convex QP).

Compute a feasible starting point x_0;
Set $\mathcal{W}_0$ to be a subset of the active constraints at x_0;
for $k = 0, 1, 2, \ldots$
    Solve (16.39) to find p_k;
    if $p_k = 0$
        Compute Lagrange multipliers $\hat{\lambda}_i$ that satisfy (16.42),
            with $\hat{\mathcal{W}} = \mathcal{W}_k$;
        if $\hat{\lambda}_i \geq 0$ for all $i \in \mathcal{W}_k \cap \mathcal{I}$
            stop with solution $x^* = x_k$;
        else
            $j \leftarrow \arg\min_{j \in \mathcal{W}_k \cap \mathcal{I}} \hat{\lambda}_j$;
            $x_{k+1} \leftarrow x_k$; $\mathcal{W}_{k+1} \leftarrow \mathcal{W}_k \backslash \{j\}$;
    else (* $p_k \neq 0$ *)
        Compute α_k from (16.41);
        $x_{k+1} \leftarrow x_k + \alpha_k p_k$;
        if there are blocking constraints
            Obtain $\mathcal{W}_{k+1}$ by adding one of the blocking
                constraints to $\mathcal{W}_k$;
        else
            $\mathcal{W}_{k+1} \leftarrow \mathcal{W}_k$;
end (for)

- قیود و هم چنین تابع درجه ۲ داده شده به صورت ماتریسی مطابق با زیر پیاده سازی شده اند.

```
self.Q = np.array([[1, 0, 0], [0, 2, 0], [0, 0, 3]])
if active == 'a':
    self.A = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 1]])
    self.b = np.array([0, 0, 0, 1])
    self.active = self.active_a

elif active == 'b':
    self.A = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 1], [-1, 0, 0]])
    self.b = np.array([0, 0, 0, 1, -0.5])
    self.active = self.active_b
```

که در نتیجه با استفاده از این ماتریس ها تابع f و هم چنین قیود فعال به صورت زیر محاسبه میشوند.

```
def f(self, x):
    f = np.dot(np.dot(x.T, self.Q), x)
    return f
```

```
def active_a(self, x):
    active_a = np.dot(self.A, x) - self.b
    return np.where(active_a == 0)[0]

def active_b(self, x):
    active_b = np.dot(self.A, x) - self.b
    return np.where(active_b == 0)[0]
```

- نقطه feasible اولیه:

چون نقطه اولیه داده شده در شروط صدق میکند بنابراین نیاز به حل مسئله بهینه سازی در این قسمت نداریم.

- ابتدا با استفاده از توابع active بالا برای نقطه x0 مجموعه فعال را به دست می آوریم.

- سپس در iterationهای مختلف الگوریتم مراحل زیر را طی میکنیم.

۱- ابتدا مسئله بهینه سازی زیر را حل میکنیم.

$$\begin{aligned} \min_{p} \quad & \tfrac{1}{2}p^T G p + g_k^T p \\ \text{subject to} \quad & a_i^T p = 0, \quad i \in \mathcal{W}_k. \end{aligned}$$

برای حل این مسئله بهینه سازی مقید از روش حل دستگاه معادلات kkt یعنی روش زیر استفاده میکنیم.

- شرایط لازم بهینگی KKT:

$$\begin{cases} Gx^* - A^T\lambda^* & = & -c \\ Ax^* & = & b \end{cases} \Longrightarrow \begin{pmatrix} G & -A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} x^* \\ \lambda^* \end{pmatrix} = \begin{pmatrix} -c \\ b \end{pmatrix}$$

که پیاده سازی آن مطابق با زیر انجام شده است.

```
H = self.hessian_f(x)
g = self.grad_f(x)

kkt = np.block([[H, -A.T], [A, np.zeros((len(active_set), len(active_set)))]])
rhs_kkt = np.block([-g, np.zeros(len(b))])
sol = np.linalg.solve(kkt, rhs_kkt)
x = sol[:len(H)]
```

۲- برای به دست آوردن ضرایب لاگرانژ و حذف متغیری با کمترین عدد لاگرانژ منفی نیز مطابق بالا دستگاه معادلات زیر را حل میکنیم که مطابق با قیود فعال مسئله اصلی یعنی Ax = b است.

```
kkt = np.block([[self.Q, -A.T], [A, np.zeros((len(active_set), len(active_set)))]])
rhs_kkt = np.block([np.zeros((len(self.Q))), b])

sol = np.linalg.solve(kkt, rhs_kkt)
```

در هر دو دستگاه معادله بالا قیود از active set فعلی انتخاب شده اند یعنی:

```
A = self.A[active_set]
b = self.b[active_set]
```

بر اساس جهت و ضرایب لاگرانژ به دست آمده از زیر مسئله حالات مختلف الگوریتم نیز مطابق با زیر پیاده سازی شده است.

```
p, l = self.sub_problem_solution(x, active_set)

if np.all(p <= 0.000001):
    if np.all(l >= 0):
        return fs

    else:
        j = active_set[np.argmin(l)]
        active_set.remove(j)
else:
    not_active_set = [i for i in range(len(self.A)) if i not in active_set]
    has_constraint = [i for i in not_active_set if self.A[i, :].dot(p) < 0]
    blocking = [(self.b[i] - self.A[i, :].dot(x)) / self.A[i, :].dot(p) for i in has_constraint]
    alpha = min([1] + blocking)
    if alpha != 1:
        active_set.append(has_constraint[np.argmin(blocking)])
    x = x + alpha * p
```

- در قسمت اول شرط p=0 به p نزدیک به صفر تبدیل شده است زیرا در حل دستگاه معدلات خطای عددی داریم.

- همان طور که در بالا مشاهده میشود در شرط دوم نیز کمترین عدد منفی لاگرانژ نیز از active set مطابق با الگوریتم حذف شده است.

- در شرط آخر نیز alpha طبق فرمول زیر محاسبه شده است تا حتما جهت حرکت جهت feasibleای باشد و هم چنین اگر alphaای غیر از یک به دست آمد مطابق با الگوریتم آن قید blocking باید به active set اضافه شود.

$$\alpha_k \stackrel{\text{def}}{=} \min\left(1, \min_{i \notin \mathcal{W}_k,\, a_i^T p_k < 0} \frac{b_i - a_i^T x_k}{a_i^T p_k}\right).$$

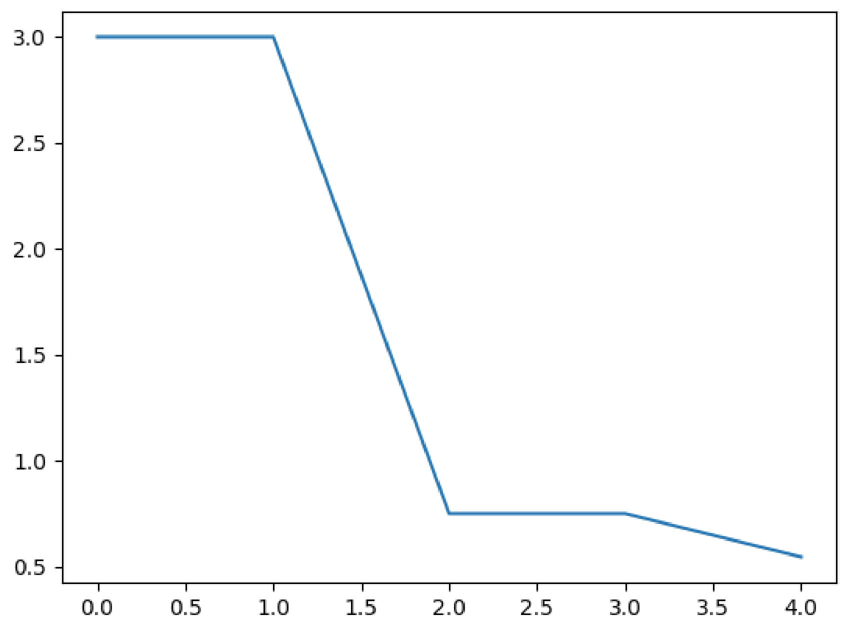
در ادامه نتایج را بررسی خواهیم کرد.

نتایج:

قید اول:

$$\text{(a)}\ \ \mathbf{x_1 \geq 0,\ \ x_2 \geq 0,\ \ x_3 \geq 0,\ \ x_1 + x_2 + x_3 \geq 1}$$

همان طور که در نمودار زیر مشاهده میشود الگوریتم بعد از ۴ iteration به نقطه بهینه رسیده است و پایان یافته است.



و نقطه بهینه و مقدار بهینه تابع نیز مطابق با زیر است. که در قیود نیز صدف میکنند همه بزرگتر از صفر و جمعشان بزگتر از ۱ است.

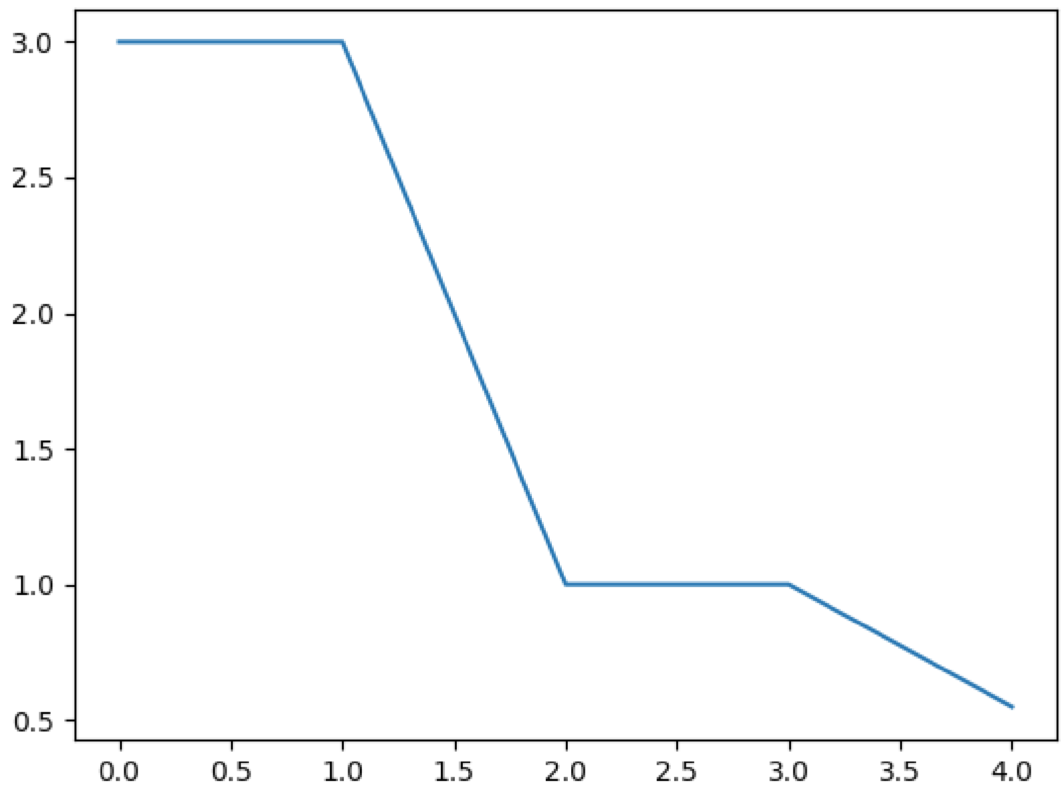
x* = [0.54545455 0.27272727 0.18181818]

f* = 0.5454

قید دوم:

$$\text{(b)}\ \mathbf{x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0,\ x_1 + x_2 + x_3 \geq 1,\ x_1 \leq 0.5}$$

همان طور که در نمودار زیر مشاهده میشود الگوریتم بعد از ۴ iteration به نقطه بهینه رسیده است و پایان یافته است.



و نقطه بهینه و مقدار بهینه تابع نیز مطابق با زیر است. که در شروط صدق میکنند.

در اجرای این قسمت بر خلاف حالت قبل حالتی به وجود آمد که alpha کمتر از ۱ به دست آید.

x* = [0.5 0.3 0.2]

f* = 0.5499

روش two metric projection:

شبه کد اصلی این روش مطابق با زیر پیاده سازی شده است.

```
alpha = 1
x = x0
fs = []
fs.append(f(x))
for iteration in range(100):
    g = grad_f(x)
    H = hessian_f(x)
    D = np.linalg.inv(H)
    D = modified_D(x, D)
    alpha = backtracking_line_search(x, -D @ g, alpha)
    x = x - alpha * np.dot(D, g)
    x = c_c_projection(x)
    fs.append(f(x))
```

تفاوت اصلی این الگوریتم با روشی مثل نیوتن در دو بخش است.

۱- تغییر ماتریس وارون hessian بر اساس روش زیر

Let us denote for all $x \geq 0$

$$I^{+}(x) = \left\{ i \;\middle|\; x_i = 0,\ \frac{\partial f(x)}{\partial x_i} > 0 \right\}. \tag{2.65}$$

We say that a symmetric $n \times n$ matrix D with elements d_{ij} is *diagonal with respect to a subset of indices* $I \subset \{1, \ldots, n\}$, if

$$d_{ij} = 0, \qquad \forall\, i \in I,\ j = 1, \ldots, n,\ j \neq i.$$

که پیاده سازی آن مطابق با زیر انجام شده است.

- ابتدا مجموعه I بر اساس گرادیان و مقدار نقطه به دست می آید.

```
def I_plus(x):
    g = grad_f(x)
    indices = []
    for i, (x_i, g_i) in enumerate(zip(x, g)):
        if x_i == 0 and g_i > 0:
            indices.append(i)

    return indices
```

- سپس با استفاده از این مجموعه مقادیر مشخصی از ماتریس عکس Hessian صفر میشود.

```
def modified_D(x, D):
    I_plus_indices = I_plus(x)
    for i in I_plus_indices:
        for j in range(D.shape[1]):
            if j != i:
                D[i][j] = 0
    return D
```

۲- تصویر x جدید به دست آمده بر اساس قیود:

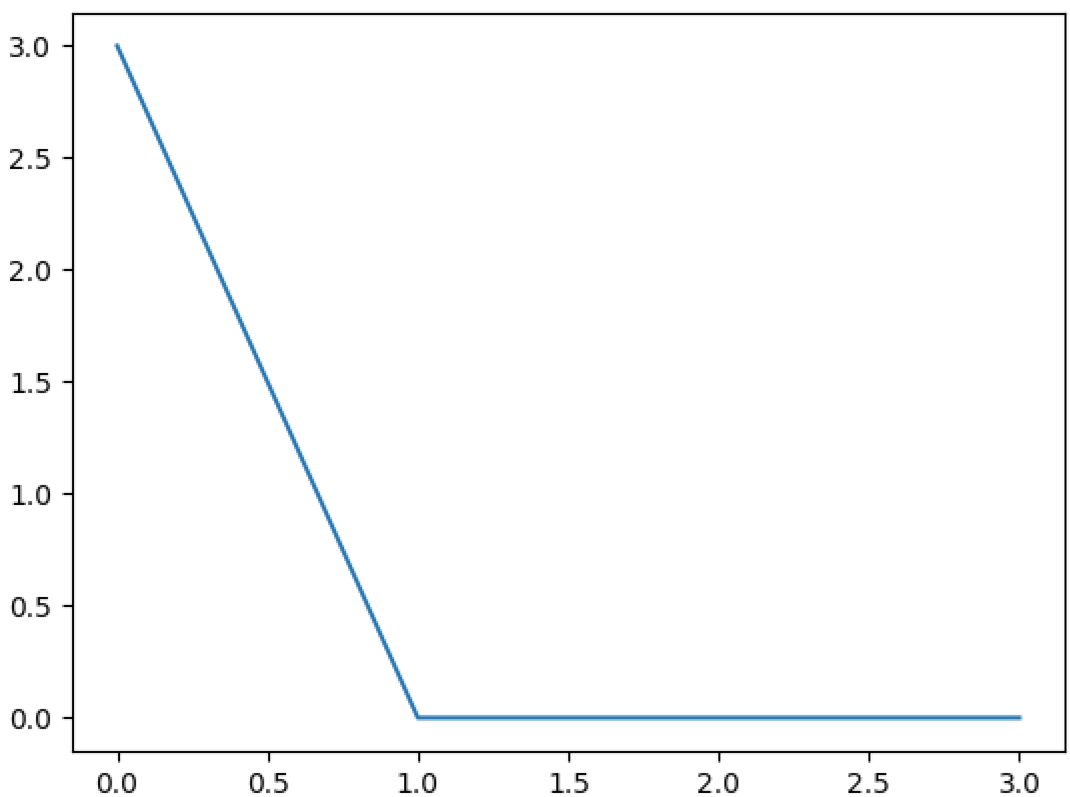
که چون قیود bound است به راحتی انجام میشود.

```
def c_c_projection(x):
    projected = np.zeros(3)
    for i in range(3):
        if x[i] < 0:
            projected[i] = 0
        else:
            projected[i] = x[i]
    return projected
```

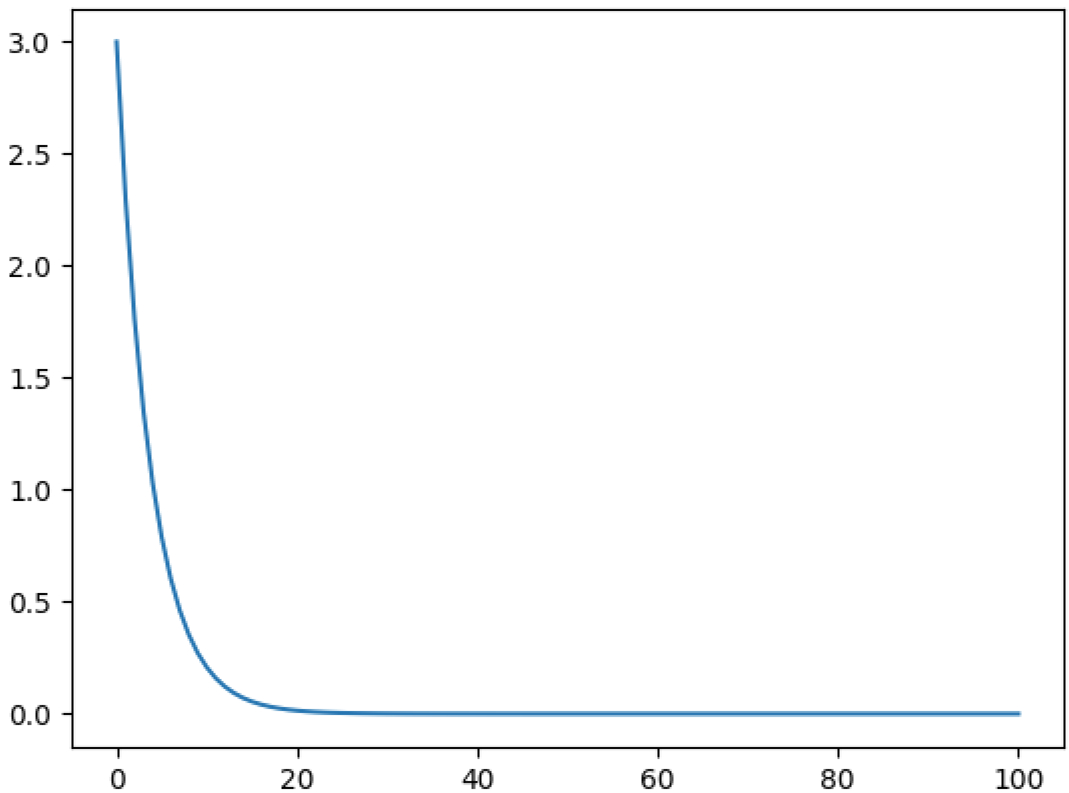
- هم چنین alpha مورد نیاز در هر مرحله با backtracking و بر اساس قاعده آرمیجو به دست می آید.

نتایج

برای حالت ضریب c1=0.01 نمودار iterationها و مقدار تابع در زیر آورده شده است. همان طور که مشاهده میشود در ۱ مرحله به مقدار بهینه 0 با x بهینه 0, 0 ,0 رسیده است.



برای حالت c1=0.9 نیز تابع بعد از iterationهای بیشتر به مقدار 0, 0, 0 همگرا شده است.



سوال ۲

تفاوت دو روش cauchy و dogleg در محاسبه جهت نزول p در هر مرحله است.

شبه کد اصلی روش trust regsion مطابق با زیر پیاده سازی شده است

```
rho = (f(x) - f(x + p)) / (-(g @ p + 0.5 * p.T @ B @ p))

if rho < 0.25:
    delta = 0.25 * delta
else:
    if rho > 0.75 and np.linalg.norm(p) == delta:
        delta = min(2.0 * delta, delta_hat)
    else:
        delta = delta

if rho > eta:
    x = x + p
xs.append(x)

if np.linalg.norm(g) < 0.0001:
    break

fs.append(f(x))
```

در ادامه به توضیح پیاده سازی دو روش مختلف cauchy و dogleg برای محاسبه p خواهیم پرداخت.

```
if p_func == 'dogleg':
    p = dogleg(H, g, B, delta)
elif p_func == 'cauchy':
    p = cauchy_point(x, B, delta)
```

نقطه کوشی مطابق به روابط زیر به دست می آید.

$$p_k^C = -\tau_k \frac{\Delta_k}{\|g_k\|} g_k$$

$$\tau_k = \begin{cases} 1, & \text{if } g_k^T B_k g_k \leq 0 \\ \min\left(\frac{\|g_k\|^3}{\Delta_k g_k^T B_k g_k}, 1\right) & \text{otherwise} \end{cases}$$

که پیاده سازی آن دقیقا مطابق با روابط بالا انجام شده است.

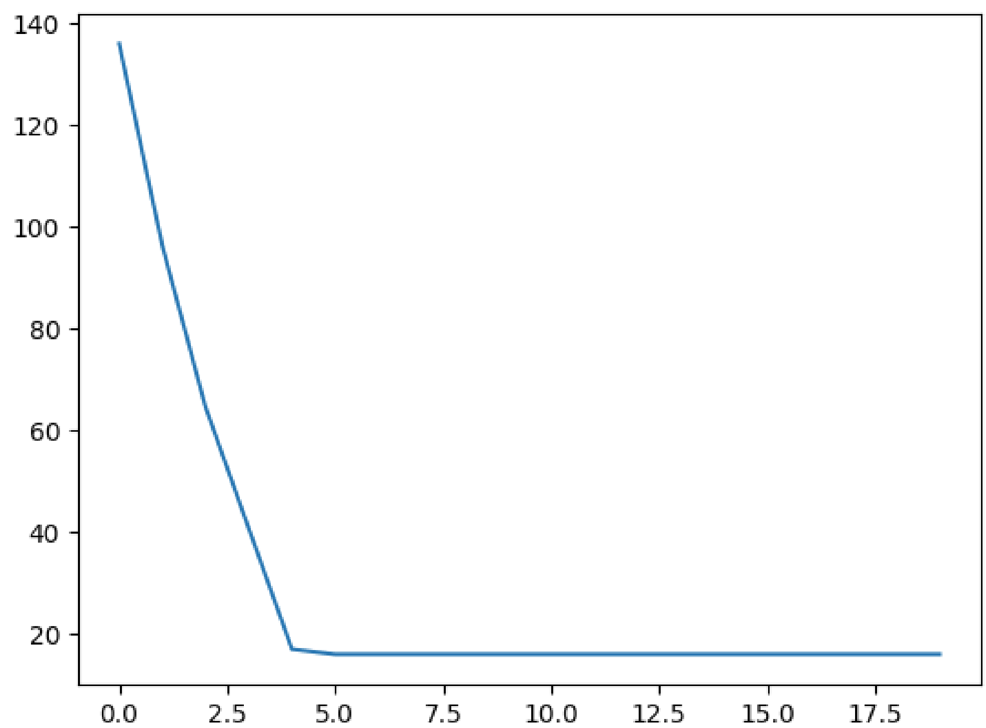
```
def cauchy_point(x, B, delta):
    g = grad_f(x)
    gT_B_g = np.dot(np.dot(g, B), g)
    g_norm = np.linalg.norm(g)

    if gT_B_g <= 0:
        taw = 1
    else:
        taw = min(g_norm ** 3. / (delta * gT_B_g), 1.)
    return -1. * (taw * delta / g_norm) * g
```

جهت dogleg نیز مطابق با زیر پیاده سازی شده است که بر اساس حالات مختلف الگوریتم مقادیر متخلف مطابق با زیر پیاده سازی شده است.

```python
def dogleg(H, g, B, delta):
    pb = -H @ g

    if np.linalg.norm(pb) <= delta:
        return pb

    pu = - (np.dot(g, g) / np.dot(g, B @ g)) * g
    dot_pu = np.dot(pu, pu)
    norm_pu = np.sqrt(dot_pu)
    if norm_pu >= delta:
        return delta * pu / norm_pu

    # ||pu**2 +(tau-1)*(pb-pu)**2|| = delta**2
    pb_pu = pb - pu
    pb_pu_sq = np.dot(pb_pu, pb_pu)
    pu_pb_pu_sq = np.dot(pu, pb_pu)
    d = pu_pb_pu_sq ** 2 - pb_pu_sq * (dot_pu - delta ** 2)
    tau = (-pu_pb_pu_sq + np.sqrt(d)) / pb_pu_sq + 1
    if tau < 1:
        return pu * tau
    return pu + (tau - 1) * pb_pu
```

نتایج:

نمودار همگرایی dogleg درزیر آورده شده است.

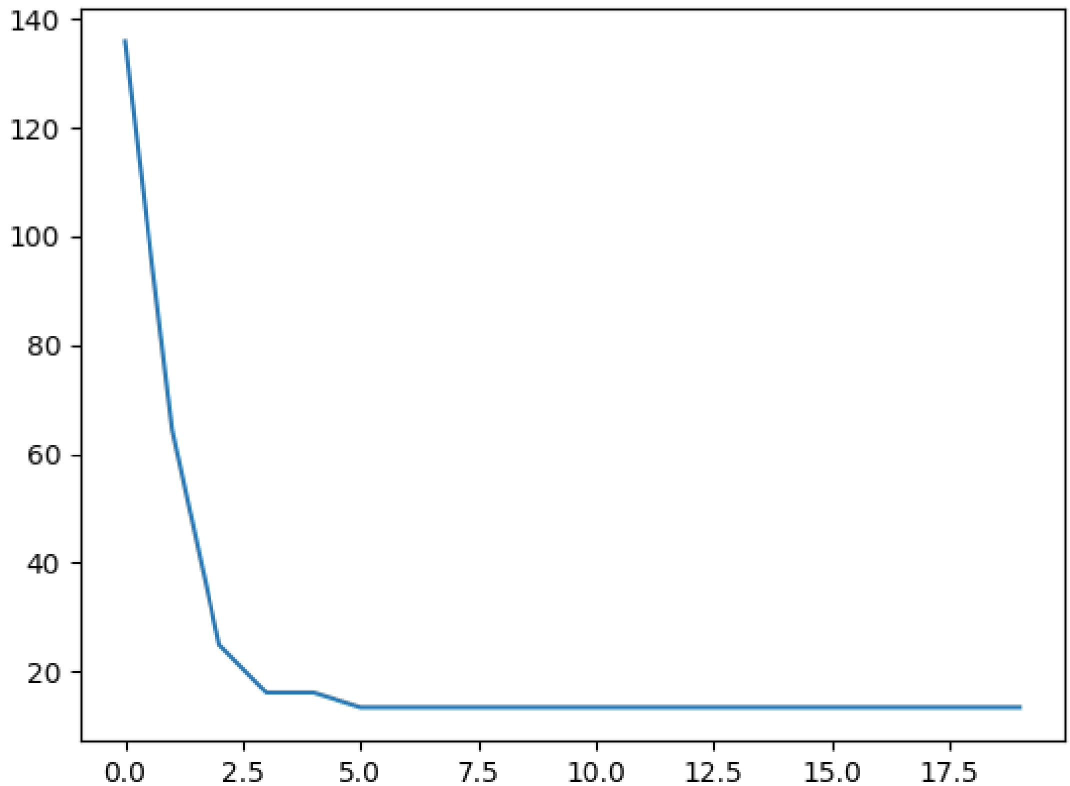


هم چنین x , f بهینه در زیر آورده شده اند.

x* = [6.45695584 1.04650873]

f* = 15.978

نمودار همگرایی cauchy درزیر آورده شده است.



هم چنین x , f بهینه در زیر آورده شده اند.

x* = [5.29824811 1.08066644]

f* = 13.36

سوال ۳

در روش Marquardet هدف مینیمم کردن تابع باقیمانده مربعات خطی زیر است.

$$f(x) = \frac{1}{2}\|r(x)\|^2 = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x)$$

که در این مسئله f مطابق با زیر است.

$$\mathbf{f}\left(\theta\right) = \sum_{\mathbf{q}=1}^{\mathbf{Q}} \left\|\mathbf{x_q} - \mathbf{D}\left(\mathbf{E}(\mathbf{x_q})\right)\right\|^{\mathbf{2}}$$

در این روش گرادیان و تقریب Hessian مطابق با زیر به دست می آید که تنها به محاسبه گرادیان f برای هر داده m نیاز است.

$$\nabla f(x) = \sum_{i=1}^{m} r_j(x) \nabla r_j(x)$$

$$\nabla^2 f(x) = \underbrace{\sum_{i=1}^{m} \nabla r_j(x) \nabla r_j(x)^T}_{J(x)^T J(x)}$$

پارامترهای این مسئله W, b_e, b_d هستند و باید از f یک x خاص (نمونه mام)نسبت به این پارامتر ها مشتق گرفته شود.

برای مشتق گیری از autograd پایتورچ مطابق با زیر استفاده میکنیم که از تابع نرم ۲ داده شده نسبت به پارامترهای خواسته شده مشتق میگیرد. تا در محاسبه گرادیان اصلی تابع برای همه داده ها استفاده شود.

```python
def auto_grad_f(x, w, b_e, b_d):
    w = torch.tensor(w, dtype=torch.float32, requires_grad=True)
    b_e = torch.tensor(b_e, dtype=torch.float32, requires_grad=True)
    b_d = torch.tensor(b_d, dtype=torch.float32, requires_grad=True)
    x = torch.tensor(x, dtype=torch.float32, requires_grad=False)

    w.grad = None
    b_e.grad = None
    b_d.grad = None

    x = x.reshape(x.shape[0], -1)
    error = torch.norm(x - w.T * (1 / (1 + torch.exp(-w @ x - b_e))) + b_d)
    error.backward(torch.ones(error.shape))
    return error.detach().numpy(), w.grad.detach().numpy(), b_e.grad.detach().numpy(), b_d.grad.detach().numpy()
```

گرادیان و hessian برای داده های mتایی حساب میشود و در نهایت مطابق با زیر جمع به ازای mهای مختلف جمع میشوند.

```python
for j, x_train in enumerate(X_train):
    error, w_grad, b_e_grad, b_d_grad = auto_grad_f(x_train, w, b_e, b_d)
    ws += error * w_grad
    b_es += error * b_e_grad
    b_ds += error * b_d_grad

    H_ws += w_grad.T @ w_grad
    H_b_es += b_e_grad.T @ b_e_grad
    H_b_ds += b_d_grad @ b_d_grad.T
```

جهت حرکت برای سه پارامتر خواسته شده نیز مطابق با رابطه زیر و گرادیان و hessian محاسبه شده در بالا به دست می آید.

$$J_k J_k^T p_k^{GN} = -J_k r_k$$

که پیاده سازی آن مطابق با زیر انجام شده است.

```
p_w = - (np.linalg.pinv(temp_H_w) @ g_w.T).T
p_b_e = - np.linalg.pinv(temp_H_b_e) @ g_b_e
p_b_d = - np.linalg.pinv(temp_H_b_d) @ g_b_d
```

در ادامه نیز مقدار لامبدا بر اساس میزان کاهش ویا افزایش تابع به ازای lambda یا lambda/v و مطابق با پیشنهاد مارکوات طبق زیر پیاده سازی شده است.

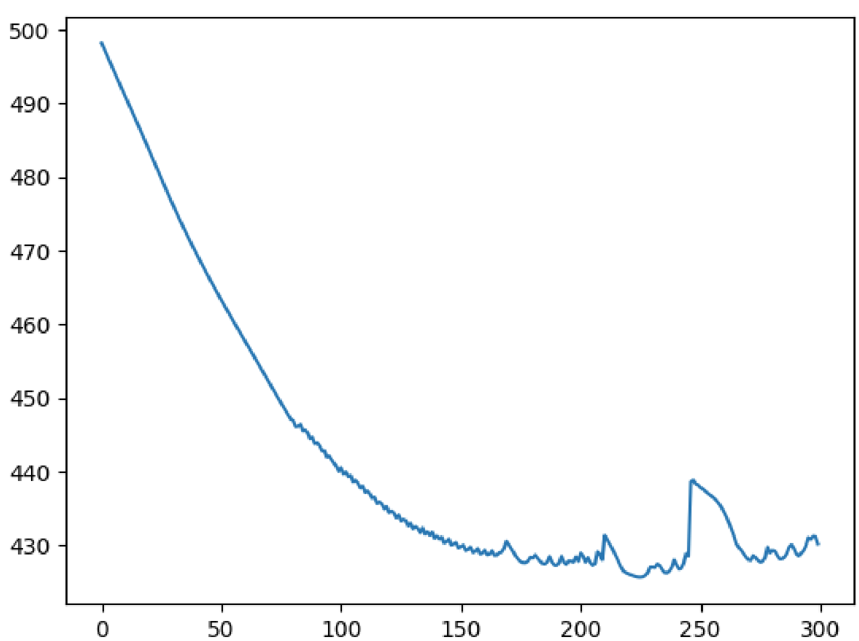
```
f_last = compute_f(w, b_e, b_d)
l1 = deepcopy(l)
l2 = l / v

f_l1 = f_next(w, b_e, b_d, H_w, H_b_e, H_b_d, g_w, g_b_e, g_b_d, l1)
f_l2 = f_next(w, b_e, b_d, H_w, H_b_e, H_b_d, g_w, g_b_e, g_b_d, l2)

decreased_f_l1 = f_last - f_l1
decreased_f_l2 = f_last - f_l2

# levenberg-marquardt suggestion:

if decreased_f_l1 < 0 and decreased_f_l2 < 0:
    l = l * v

else:
    if decreased_f_l1 >= decreased_f_l2:
        l = l1

    else:
        l = l2
```

در نهایت نیز بر اساس lamda به دست آمده پارامترها در جهت p مطابق با زیر حرکت خواهند کرد.

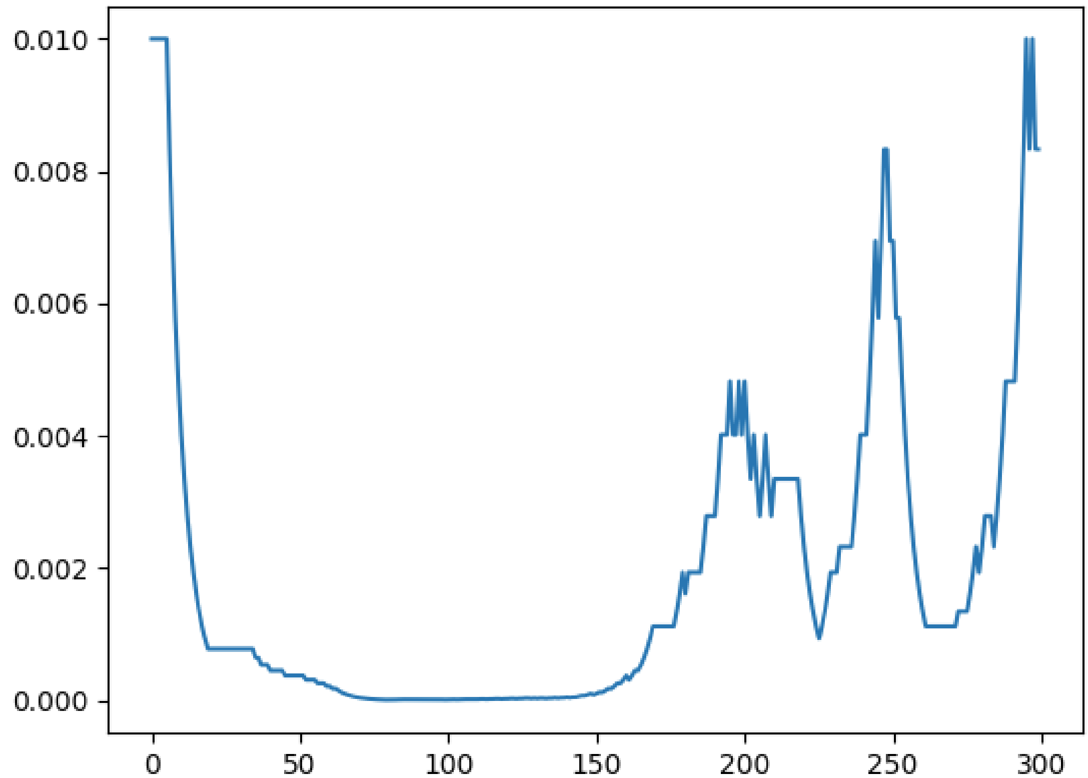
```
H_w = H_w + l * np.diag(np.diag(H_w))
H_b_e = H_b_e + l * np.diag(np.diag(H_b_e))
H_b_d = H_b_d + l * np.diag(np.diag(H_b_d))

p_w = - (np.linalg.pinv(H_w) @ g_w.T).T
p_b_e = - np.linalg.pinv(H_b_e) @ g_b_e
p_b_d = - np.linalg.pinv(H_b_d) @ g_b_d

w += alpha * p_w
b_e += alpha * p_b_e
b_d += alpha * p_b_d
```

نتایج:

در نمودار زیر مقدار تابع f در iterationهای مختلف آموزش آورده شده است.



در نمودار زیر مقدار lamda به ازای iterationهای مختلف آورده شده است.



هم چنین مقدار CCR بر روی داده های تست برابر مقدار زیر به دست آمده است.

CCR(test) = 0.259