دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

تمر ین دوم درس بهینه سازی(بخش عملی)

دکتر امیرمزلقانی

غلامرضا دار ۴۰۰۱۳۱۰۱۸

بهار ۱۴۰۱

فهرست مطالب

سوال ۹)

ابتدا مانند خواسته سوال متغیرهای مسئله بهینه سازی را تعریف میکنیم.

```python
m = 300
n = 100
A = np.random.rand(m,n)
b = A@np.ones((n,1))/2
c = -np.random.rand(n, 1)
```

سپس مسئله شماره ۲ را برای بدست آوردن x^{rlx} حل میکنیم.

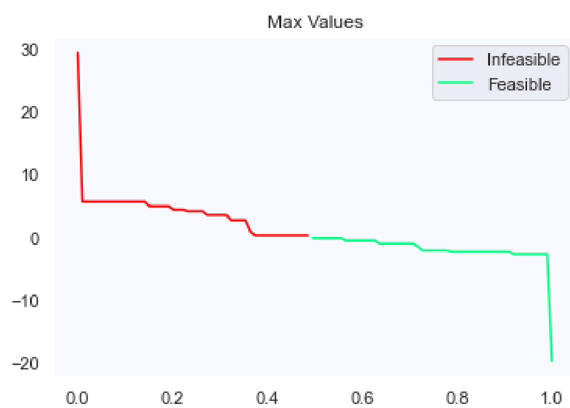
```
Optimal value:  -34.58971131665731
X_rlx solution:
 [[0.3602951]
 [1.       ]
 [1.       ]
 [0.       ]
 [0.       ]
 [1.       ]
```

مقدار L برابر با همین عدد منفی ۳۴ و خوردی است.

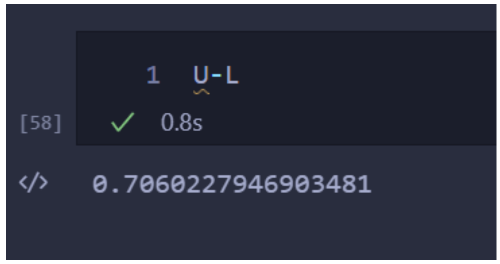
در ادامه با در نظر گرفتن مقادیر مختلف بین ۰ تا ۱ (۱۰۰ عدد مختلف) برای t، موارد خواسته شده سوال را با x_hat که به صورت باینری تبدیل شده است، بدست می‌آوریم.

```python
# Generate 100 t values between 0 and 1
N = 100
t_values = np.linspace(0, 1, N)
max_values = np.zeros(N)
u_values = np.zeros(N)

feasible_Us = []

for i in range(N):
    t = t_values[i]

    # Calculate x_hat based on x_rlx (x_hat is binary)
    x_hat = (x_rlx>t).astype(float)

    # Calculate upperbound
    u = c.T@x_hat

    # Calculate maximum element of AX-b (X=x_hat)
    max_val = np.max(A@x_hat-b)

    if max_val<=0:
        feasible_Us.append(u)

    # save the maximum value and the corresponding u
    max_values[i] = max_val
    u_values[i] = u.flatten()[0]
```

همچنین نمودار خواسته شده در سوال را در تصویر زیر مشاهده میکنید.



با انجام آزمایش‌های قبل به این نتبجه می‌رسیم که مقدار U برابر با 33.8836- است که طبق خواسته سوال تفاضل LوU را محاسبه میکنیم.



سوال ۱۰)

در این سوال ابتدا به پیاده سازی توابعی برای تولید ماتریس های به فرم Hilbert و Tridiag پرداختیم.

```
[67]
tridiag()
```

```
array([[ 4., -1.,  0.,  0.,  0.],
       [-1.,  4., -1.,  0.,  0.],
       [ 0., -1.,  4., -1.,  0.],
       [ 0.,  0., -1.,  4., -1.],
       [ 0.,  0.,  0., -1.,  4.]])
```

```
[68]
hilbert()
```

```
array([[1.        , 0.5       , 0.33333333, 0.25      , 0.2       ],
       [0.5       , 0.33333333, 0.25      , 0.2       , 0.16666667],
       [0.33333333, 0.25      , 0.2       , 0.16666667, 0.14285714],
       [0.25      , 0.2       , 0.16666667, 0.14285714, 0.125     ],
       [0.2       , 0.16666667, 0.14285714, 0.125     , 0.11111111]])
```

سپس توابع مربوط به تابع داده شده و الگوریتم Backtracking برای یافتن اندازه قدم بهینه را پیاده کردیم.

```
# 1/2 * X^T * A * X - b^T * X
def f(x, A, b):
    return 0.5 * x.T @ A @ x - b.T @ x
```
[162] ✓ 0.2s

```
def f_prim(x, A, b):
    return A@x-b

def f_prim_norm(x, A, b):
    return np.linalg.norm(f_prim(x, A, b))
```
[163] ✓ 0.1s

```
def backtracking_algorithm(x, A, b, pk, rho, alpha_0, c):
    alpha = alpha_0
    # Check sufficient decrease condition
    while f(x+alpha*pk, A, b) > f(x, A, b) + c * alpha * f_prim(x, A, b).T @ pk:
        alpha *= rho
    return alpha

```
[164] ✓ 0.1s

در ادامه، به پیاده‌سازی الگوریتم‌های بهینه سازی Steepest Descent و Newton و BFGS پرداختیم.

```python
def steepest_descent(x0, A, b, rho, alpha_0, c, max_iter=100):
    x = x0
    for _ in range(max_iter):
        grad = f_prim(x, A, b)
        pk = -grad
        alpha = backtracking_algorithm(x, A, b, pk, rho, alpha_0, c)
        x += alpha * pk
    return x
```

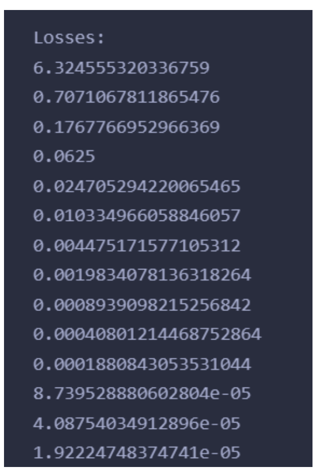
```python
def newtons_method(x0, A, b, rho, alpha_0, c, max_iter=100):
    x = x0
    for _ in range(max_iter):
        grad = f_prim(x, A, b)
        hess = A
        pk = -(np.linalg.inv(hess)@grad)

        alpha = backtracking_algorithm(x, A, b, pk, rho, alpha_0, c)
        # Update the current iterate
        x += alpha * pk
    return x
```

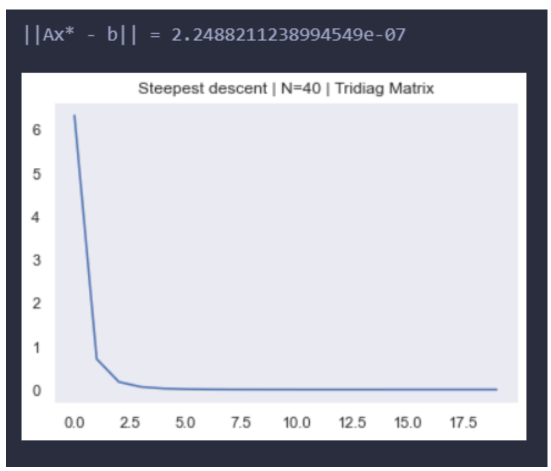
```python
def BFGS(x0, A, b, rho, alpha_0, c, max_iter=100):
    x = x0
    # Initialize Hessian approx to I_n
    x_prev = x
    hess_approx = np.eye(A.shape[0])
    for _ in range(max_iter):
        grad = f_prim(x, A, b)
        # Update the Hessian approximation
        hess_approx = approximate_hessian(x0, x_prev, hess_approx, A, b)
        # Calculate the BFGS pk = -(hess_approx @ grad)
        pk = -(np.linalg.inv(hess_approx)@grad)
        # Calculate the step size
        alpha = backtracking_algorithm(x, A, b, pk, rho, alpha_0, c)
        # Update the current iterate
        x += alpha * pk
        x_prev = x
    return x
```

سپس آزمایش های گفته شده در سوال را انجام دادیم.

به عنوان مثال چند نمونه از نتایج آزمایش ها را با هم در گزارش میبینیم. برای مشاهده سایر آزمایش ها به نوت بوک مراجعه کنید.

در تصویر زیر نمودار میزان خطای ماتریس tridiag و N=40 را مشاهده میکنید که به روش Steepest Descent انجام شده است. میبینیم که پس از تعداد کمی iteration به همگرایی رسیده ایم.

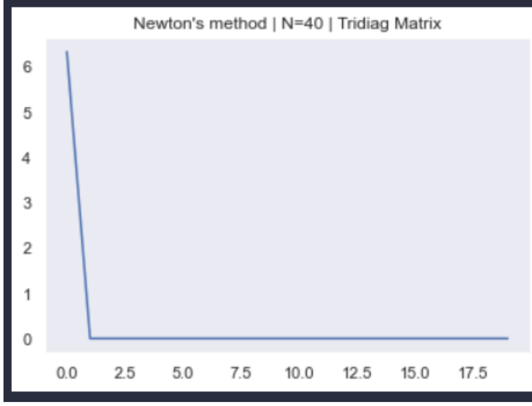




به عنوان مثالی دیگر نمودار tridiag, N=40 و روش Newton را مشاهده میکنید. همانطور که واضح است، روش نیوتن با تعداد ایتریشن کمتری و با سرعت بالاتری همگرا می‌شود.

```
Losses:
6.324555320336759
1.4261072965499199e-15
1.318316798730503e-15
1.2462222543165673e-15
1.2462222543165673e-15
1.2462222543165673e-15
1.2462222543165673e-15
1.2462222543165673e-15
1.2462222543165673e-15
1.2462222543165673e-15
1.2462222543165673e-15
```

```
||Ax* - b|| = 1.2462222543165673e-15
```



جهت مشاهده سایر نمودارها به نوت بوک مراجعه شود.

الف) در جدول زیر خلاصه نتیجه آزمایش های مختلف موجود است. دقت کنید که هایپرپارامتر هایی مانند $\rho. \alpha_0. c$ که در الگوریتم های مختلف مورد استفاده قرار میگیرد به میزان قابل توجهی در میزان خطای نهایی تاثیر دارند.

اسم روش	i	اندازه n	$\|\|A_i x^* - b\|\|_2$
Steepest Descent	1	40	**2.24 e-7**
Newton's Method	1	40	**1.24 e-15**
BFGS	1	40	**2.34 e-5**
Steepest Descent	1	400	**2.24 e-7**
Newton's Method	1	400	**0**
BFGS	1	400	**2.35 e-5**
Steepest Descent	2	40	**0.26**
Newton's Method	2	40	**3.24**
BFGS	2	40	**0.104**
Steepest Descent	2	400	**0.49**
Newton's Method	2	400	**19.77**
BFGS	2	400	**1.34**

ب) بخش محاسبه x-x_hat ممکن است دچار باگ باشد.

اسم روش	i	اندازه n	$\|\|A_i x^* - b\|\|_2$	$\left\|\left\|A_i - \widehat{A}_i\right\|\right\|_2$	$\|\|x^* - \widehat{x}^*\|\|_2$
Steepest Descent	1	40	**2.25 e-7**	4 e-5	6.10 e-5
Newton's Method	1	40	**0**	4 e-5	6.10 e-5
BFGS	1	40	**2.34 e-5**	4 e-5	6.10 e-5
Steepest Descent	1	400	**6.49 e-7**	4 e-4	1.99 e-3
Newton's Method	1	400	**7 e-14**	4 e-4	1.99 e-3
BFGS	1	400	**2.36 e-5**	4 e-4	1.99 e-3
Steepest Descent	2	40	**0.26**	4 e-5	3.11 e-3
Newton's Method	2	40	**5.1**	4 e-5	1253275844
BFGS	2	40	**0.104**	4 e-5	0.01465
Steepest Descent	2	400	**0.49**	4 e-4	0.24641
Newton's Method	2	400	**19.8**	4 e-4	11806472648187
BFGS	2	400	**2.94**	4 e-4	0.13460

ج) همانطور که از نتایج جدول و مخصوصا نمودارهای همگرایی میبینیم، روش نیوتن نسبت به روش steepest descent سرعن همگرایی بسیار بالاتری دارد. مشکلی که روش نیوتن دارد نیازمند بودن به محاسبه ماتریس Hessian است که باعث می-شود نتوان آن را در همه مسائل استفاده کرد. روش BFGS و سایر روش های شبه نیوتن، با تخمین زدن ماتریس Hessian مشکلات روش نیوتن را رفع میکنند. روش های شبه نیوتن از روش Steepest Descent همگرایی بهتری دارند و از روش نیوتن همگرایی کمتری دارند.

نکته دیگری که مشاهده می‌شود، تفاوت بین i=1 و i=2 است. در ماتریس Tridiag در ازمایش های مختلف هم نتیجه همگرایی بهتر بود یعنی خطای کمتر و هم فاصله بین نقاط بین بخش A,B سوال کمتر بود. اما در ماتریس Hilbert هم خطای همگرایی بیشتری داشتیم و هم تفاوت اندکی در ماتریس A باعث شد نقاط بهینه بسیار از یکدیگر دور شوند.

علاوه بر این ها، ابعاد ماتریس A نیز هم در سرعت همگرایی و هم در خطای آن تاثیر عکس داشت.