作业 1

1、证明对任意的非奇异矩阵B，有：$\|Bx\| \geq \frac{\|x\|}{\|B^{-1}\|}$

Pf. $\|B^{-1}\| = \max_{\|x\|\neq 0} \frac{\|B^{-1}x\|}{\|x\|}$

令 $B^{-1}x = y$，即 $x = By$，代入后有：

$$\|B^{-1}\| = \max_{\|y\|\neq 0} \frac{\|y\|}{\|By\|} \geq \frac{\|y\|}{\|By\|}$$

即：$\|By\| \geq \frac{\|y\|}{\|B^{-1}\|}$，证毕。

2、(a).

$$\Leftrightarrow (A+ab^T)^{-1} = A^{-1} - \frac{A^{-1}ab^TA^{-1}}{1+b^TA^{-1}a}$$

右乘 $(A+ab^T)$：$\Leftrightarrow E = A^{-1}(A+ab^T) - \frac{A^{-1}ab^TA^{-1}(A+ab^T)}{1+b^TA^{-1}a}$

$$\Leftrightarrow 0 = A^{-1}ab^T - \frac{A^{-1}ab^T + A^{-1}ab^TA^{-1}ab^T}{1+b^TA^{-1}a}$$

$$\Leftrightarrow A^{-1}ab^T \cdot b^TA^{-1}a = A^{-1}ab^TA^{-1}ab^T$$

($b^TA^{-1}a$是一个数) $\Leftrightarrow (b^TA^{-1}a)\cdot A^{-1}ab^T = (b^TA^{-1}a)\cdot A^{-1}ab^T$

上式显然成立，故验证得

$$\bar{A}^{-1} = A^{-1} - \frac{A^{-1}ab^TA^{-1}}{1+b^TA^{-1}a}$$ 成立。

(b) 令 $A = H_k$，$a = s_k - H_ky_k$，$b^T = \frac{(s_k-H_ky_k)^T}{(s_k-H_ky_k)^Ty_k}$

则由(a)中公式：$\bar{A}^{-1} = A^{-1} - \frac{A^{-1}ab^TA^{-1}}{1+b^TA^{-1}a}$ 代入有：

$$H_{k+1}^{-1} = (H_k+ab^T)^{-1} = B_{k+1} = B_k - \frac{B_k(s_k-H_ky_k)\frac{(s_k-H_ky_k)^T}{(s_k-H_ky_k)^Ty_k}B_k}{1+\frac{(s_k-H_ky_k)^T}{(s_k-H_ky_k)^Ty_k}B_k\cdot(s_k-H_ky_k)}$$

$$= B_k + \frac{(y_k-B_ks_k)\cdot(s_k-H_ky_k)^TB_k}{(s_k-H_ky_k)^Ty_k + (s_k-H_ky_k)^TB_k(s_k-H_ky_k)}$$

$$= B_k + \frac{(y_k-B_ks_k)\cdot(s_k-B_k^{-1}y_k)^TB_k^T}{(s_k-H_ky_k)^Ty_k + (s_k-H_ky_k)^T(B_ks_k-y_k)}$$

$$= B_k + \frac{(y_k-B_ks_k)(y_k-B_ks_k)^T}{(s_k-H_ky_k)^TB_k^Ts_k}$$

$$= B_k + \frac{(y_k-B_ks_k)^T(y_k-B_ks_k)^T}{(y_k-B_ks_k)^Ts_k}$$ 证毕.

第三题

Rosenbrock function （the steepest descent method）

关于steepest descent method中步长的选择，我使用Armijo规则进行一维线搜索去找，具体见代码：

```
import numpy as np
from sympy import symbols, diff, hessian
x, y = symbols('x y')
f = 100*(y-x**2)**2 + (1-x)**2
# f = (1.5-x+x*y)**2+(2.25-x+x*y*y)**2+(2.625-x+x*y*y*y)**2
# 计算在(x1,x2)处的偏导数值
def partial_fx(x_k):
    x1 = x_k[0];x2 = x_k[1]
    df_dx = diff(f, x)
    df_dy = diff(f, y)
    df_dx_value = df_dx.subs({x: x1, y: x2})
    df_dy_value = df_dy.subs({x: x1, y: x2})
    return np.array([df_dx_value,df_dy_value])

x0 = np.array([-1.2,1])
x_k = x0
for i in range(1,10):
    print("iter:",i," ",x_k)
    g_k = partial_fx(x_k)
    t = 1
    # 使用Armijo规则进行一维线搜索
    while f.subs({x: x_k[0]-t*g_k[0], y: x_k[1]-t*g_k[1]}) > (f.subs({x: x_k[0], y: x_k[1]}) - 0.1*t*np.dot(g_k.T, g_k)):
        t *= 0.5
        # 防止步长过小导致无限循环
        if t <  1e-6:
            break
    x_k = x_k - t*g_k
```

同样我运行了10次，得到较好的结果：

```
iter: 1   [-1.2  1. ]
iter: 2   [-0.989453125000000 1.08593750000000]
iter: 3   [-1.02689260772895 1.06505468487740]
iter: 4   [-1.02797918645513 1.05681542154222]
iter: 5   [0.984741960435344 1.04939404581511]
iter: 6   [1.01542082282326 1.03383206980475]
iter: 7   [1.01648252828014 1.03329444824582]
iter: 8   [1.01618573632181 1.03293371107682]
iter: 9   [1.01627331117593 1.03287506647274]
```

Rosenbrock function （Newton)

```
import numpy as np
from sympy import symbols, diff, hessian
x, y = symbols('x y')
f = 100*(y-x**2)**2 + (1-x)**2
# 计算在(x1,x2)处的偏导数值
def partial_fx(x_k):
    x1 = x_k[0];x2 = x_k[1]
    df_dx = diff(f, x)
    df_dy = diff(f, y)
    df_dx_value = df_dx.subs({x: x1, y: x2})
    df_dy_value = df_dy.subs({x: x1, y: x2})
    return np.array([df_dx_value,df_dy_value])
# 计算在(x1,x2)处的海森矩阵
def hes_fx(x_k):
    x1 = x_k[0];x2 = x_k[1]
    H = hessian(f, [x, y])
    H_value = H.subs({x: x1, y: x2})
    H_value = np.array(H_value).astype(np.float64)
    return H_value
x0 = np.array([-1.2,1])
x_k = x0
for i in range(1,10):
    print("iter:",i," ",x_k)
    H_T = np.linalg.inv(hes_fx(x_k))
    g_k = partial_fx(x_k)
    x_k = x_k - np.dot(H_T,g_k)
```

iter: 1 [-1.2 1.]
iter: 2 [-1.17528089887641 1.38067415730337]
iter: 3 [0.763114871176195 -3.17503385474755]
iter: 4 [0.763429678883799 0.582824775496728]
iter: 5 [0.999995311085005 0.944027323853239]
iter: 6 [0.999995695653681 0.999991391325742]
iter: 7 [1.00000000000001 0.999999999981502]
iter: 8 [0.999999999999986 0.999999999999972]
iter: 9 [0.999999999999986 0.999999999999972]

可以看到，经过9轮迭代已经有很高的精度，最后得到：

$$x^* = (0.999999999999986, 0.999999999999972)$$

Beale function（the steepest descent method)

> 对于 Beale function，代码是相同的，只不过将函数f修改了运行，所以我不放上代码，直接上结果：

在999次迭代后，x的值来到了$(-2.78326800725535 1.27835877754421)$，并且还在一路下降。而函数值也一直缓慢下降。而这不是beale函数的全局最小值，分析应该是这种方法一开始走错了方向，后来一直没走到正确的地方去。

Beale function（Newton）

```
iter: 1   [-1.2  1. ]
iter: 2   [0 1.00000000000000]
iter: 3   [0 1.00000000000000]
iter: 4   [0 1.00000000000000]
iter: 5   [0 1.00000000000000]
iter: 6   [0 1.00000000000000]
iter: 7   [0 1.00000000000000]
iter: 8   [0 1.00000000000000]
iter: 9   [0 1.00000000000000]
```

经检验发现此时海塞矩阵负定，牛顿法无法收敛，所以一直卡在saddle point $(0,1)$上。

第4题

易知全局最优是$(0,0,0,0)^T$

(1) 当$\sigma=1$时, $x^{(0)}=(\cos 70^{\circ},\sin 70^{\circ},\cos 70^{\circ},\sin 70^{\circ})^T$ 时，

pure_newton 法:

```
iter: 1   [0.34202014 0.93969262 0.34202014 0.93969262]
iter: 2   [0.237039454426468 0.619948296782725 0.226616605139021 0.599778527355121]
iter: 3   [0.169930010412634 0.403358471185648 0.148591288767330 0.363302958382570]
iter: 4   [0.126459984956814 0.253765167881718 0.0944044885035707 0.197330848389114]
iter: 5   [0.0931221938605041 0.147170890444275 0.0547102694502769 0.0874496309945327]
iter: 6   [0.0580986012957210 0.0703385984122402 0.0250985157145790 0.0284977525939063]
iter: 7   [0.0209232887357403 0.0200351371518453 0.0064623148276907 0.00585063743831753]
iter: 8   [0.00121800666708339 0.000987373122393356 0.000279223616379252
0.000259145271946315]
iter: 9   [2.15769782754491e-7 1.57474918963509e-7 3.95049142612978e-8
4.24846663371311e-8]
iter: 10   [1.08605505705762e-18 7.41100889288309e-19 1.68916898528229e-19
2.11354572674649e-19]
```

收敛速度很快，精度也较高，10次迭代后误差已经到10^{-18}以下。

newton_linear_search 法:

```
iter: 1   [0.34202014 0.93969262 0.34202014 0.93969262]
0.03125
iter: 2   [0.338739496797569 0.929700610660809 0.338413782757336 0.929070305366196]
0.03125
iter: 3   [0.335495349112858 0.919810442800227 0.334844356959494 0.918551208540655]
0.03125
iter: 4   [0.332287325195313 0.910021030321040 0.331311469757643 0.908134217498598]
0.03125
iter: 5   [0.329115053374564 0.900331297345472 0.327814728866299 0.897818231793100]
0.03125
iter: 6   [0.325978165330915 0.890740178883454 0.324353745843983 0.887602163250822]
0.03125
iter: 7   [0.322876296040245 0.881246620716262 0.320928136048647 0.877484935883796]
0.03125
iter: 8   [0.319809083718965 0.871849579281431 0.317537518593535 0.867465485803165]
0.03125
```

```
iter: 9   [0.316776169769057 0.862548021558933 0.314181516303489 0.857542761134874]
0.03125
```

带线搜索的牛顿法收敛较慢，在9轮时结果差距还很大，这是因为它的步长因子取得很保守（一开始只有0.03，而pure_newton取的1）（当然保守的程度这也和选取的一维搜索方式有关，这里我沿用了上面的Armojio条件来刻画）

```
iter: 57   [1.24077091882954e-24 1.24077091882954e-24 3.10192729707385e-25
3.10192729707385e-25]
```

而最后的收敛精度两个都很优秀，在57轮线搜索的牛顿法已经效果很好。

而当$\sigma = 10^4$ 时，带线搜索的速度慢的特点更加明显，而pure牛顿法依然能以很快的速度收敛到全局最优。

(1) 当$\sigma = 1, 10^4$时, $x^{(0)} = (\cos 50°, \sin 50°, \cos 50°, \sin 50°)^T$ 时,

跟上面情况一样，**两种方法都能给出很好的收敛结果，但带线搜索的方法因为步长取得保守而收敛速度较慢。**

```
'''
带线搜索的newton法求解
'''
import numpy as np
import math
from sympy import symbols, diff, hessian, Matrix
sigma = 10000
x1, x2, x3, x4 = symbols('x1 x2 x3 x4')
# 计算在(x1,x2)处的偏导数值
X = Matrix([x1, x2, x3, x4])
A = Matrix([[5, 1, 0,1], [1,4,0.5,0], [0,0.5,3,0],[0.5,0,0,2]])
f = 0.5*X.T*X+0.25*sigma*(X.T*A*X)**2
def partial_fx(x_k):
    a = x_k[0];b = x_k[1];c = x_k[2];d= x_k[3]
    df_da = diff(f, x1)
    df_db = diff(f, x2)
    df_dc = diff(f, x3)
    df_dd = diff(f, x4)
    df_da_value = df_da.subs({x1: a, x2: b,x3:c,x4:d}).tolist()
    df_db_value = df_db.subs({x1: a, x2: b,x3:c,x4:d}).tolist()
    df_dc_value = df_dc.subs({x1: a, x2: b,x3:c,x4:d}).tolist()
    df_dd_value = df_dd.subs({x1: a, x2: b,x3:c,x4:d}).tolist()
    return np.array([df_da_value[0][0],df_db_value[0][0],df_dc_value[0]
[0],df_dd_value[0][0]])

# 计算海森矩阵
def hes_fx(x_k):
    H = hessian(f, [x1, x2,x3,x4])
    H_value = H.subs({x1: x_k[0], x2: x_k[1],x3:x_k[2],x4:x_k[3]})
    H_value = np.array(H_value).astype(np.float64)
    return H_value
x0 =
np.array([math.cos(math.radians(50)),math.sin(math.radians(50)),math.cos(math.radians(50
)),math.sin(math.radians(50))])
x_k = x0
for i in range(1,200):
    print("iter:",i," ",x_k)
```

```
    H_T = np.linalg.inv(hes_fx(x_k))
    g_k = partial_fx(x_k)
    t = 1
    # 使用Armijo规则进行一维线搜索
    while f.subs({x1: x_k[0]-t*g_k[0], x2: x_k[1]-t*g_k[1],x3: x_k[2]-t*g_k[2],x4:
x_k[3]-t*g_k[3]}).tolist()[0][0] > (f.subs({x1: x_k[0], x2: x_k[1],x3: x_k[2],x4:
x_k[3]}).tolist()[0][0] - 0.1*t*np.dot(g_k.T, g_k)):
        t *= 0.5
        # 防止步长过小导致无限循环
        if t <  1e-6:
            break
    print(t)
    x_k = x_k - t*np.dot(H_T,g_k)
```

第5题

$$f(x) = \min f(x) = \frac{1}{2}\|Ax - b\|^2 + \mu L_\delta(x)$$

在梯度下降法中，需要计算∇f：

$$\begin{aligned}\frac{d}{dx}\|Ax - b\|^2 &= \frac{d}{dx}\left[(Ax - b)^T(Ax - b)\right] \\ &= 2(Ax - b)^T A\end{aligned}$$

Barzilar-Borwein method用如下的迭代公式:

$$x_{k+1} = x_k - \frac{f(x_k) - f(x_{k-1})}{\|\nabla f(x_k) - \nabla f(x_{k-1})\|^2}(\nabla f(x_k))$$

> sorry，这道题对我来说有些苦难，我不是很明白A，x随机跟后面的是什么意思......