最优化方法：

牛顿法，共轭梯度法。最速下降法比较

李梓铉

学号 3118103163

指导老师: 阮小娥教授

2019 年 10 月 31 日

1 问题简介

Find the minimizer of the higher-dimensional quadratic optimization for the respective consistent convex, bounded convex and convex cases with different initial points by any 3 methods of conjugate gradient method, steepest descent method, Newton’s method and Quasi Newton method.

2 方法

2.1 生成三种情况下的凸优化问题

对于二次型优化问题 $\min f(x) = \frac{1}{2}x^{\mathrm{T}}Ax - b^{\mathrm{T}}x$ 根据 consistent convex 的定义，构造一个 hessin 阵为实对称正定的即满足一致连续的定义。首先设计生成一个随机的实对称正定矩阵的算法算法的主要过程总结如下：

1. 生成一个随机对角阵 diag(rand（N,1）)
2. 生成一个随机的矩阵 B(N,N)
3. 求矩阵 B 的正交基 U
4. 得到随机实对称正定矩阵 A=U’*D*U

在实对称的基础上，构造半正定实对称阵实现第二种情况 convex，只需要将 A 减去 A 的最小特征值大小的对角阵，即可得到有特征值为 0 的半正定 convex。

算法的主要过程总结如下：

1. 生成一个随机对角阵 diag(rand（N,1）)
2. 生成一个随机的矩阵 B(N,N)
3. 求矩阵 B 的正交基 U
4. 得到随机实对称正定矩阵 A=U’*D*U
5. 得到实对称半正定矩阵 A=A-max(lam)*eye

类似的可以得到 bounded convex 的情况，只需得到不定 hessian 矩阵的 f（x）算法的主要过程总结如下：

1. 生成一个随机对角阵 diag(rand（N,1）)
2. 生成一个随机的矩阵 B(N,N)
3. 求矩阵 B 的正交基 U
4. 得到随机实对称正定矩阵 A=U’*D*U

5. 得到实对称半正定矩阵 A=A-（max(lam)+min(lam))/2*eye

可证明对于实对称正定矩阵 A 的最优化问题 $minf(x)=\frac{1}{2}x^{\mathrm{T}}Ax-b^{\mathrm{T}}x$, 解即为方程 $Ax=b$ 的解。因此为了方便检验算法的精确性，设最优解为元素均为 1 的向量，根据方程 $Ax=b$ 求出 b。

2.2 共轭梯度法

对于最优化问题 $\min f(x)=\frac{1}{2}x^{\mathrm{T}}Ax-b^{\mathrm{T}}x$ 使用共轭梯度法，可证明在问题的一组正交方向上进行精确线搜索即可在有限步迭代后得到极小值。证明可得以下迭代公式

1. 优化方向 $d^{(k)}=r^{(k)}-\frac{r^{(k)T}Ad^{(0)}}{d^{(0)T}Ad^{(0)}}d^{(0)}-\cdots-\frac{r^{(k)T}Ad^{(k-1)}}{d^{(k-1)T}Ad^{(k-1)}}d^{(k-1)}$
2. 优化步长 $\alpha_k=\frac{r^{(k)\mathrm{T}}d^{(k)}}{d^{(k)\mathrm{T}}Ad^{(k)}}$
3. 迭代误差 $r^{(k)}=b-Ax^{(k)}$

2.3 最速下降法

最速下降法是梯度方法的一种实现，它的理念是在每次的迭代过程中，选取一个合适的步长，使得目标函数的值能够最大程度的减小。步长为 $\alpha_k=\arg\min f\left(x^{(k)}-\alpha\nabla f\left(x^{(k)}\right)\right),\alpha\geq 0$ 在目标函数为二次型的情况下

1. 优化方向 $g^{(k)}=\nabla f\left(\boldsymbol{x}^{(k)}\right)=\boldsymbol{Q}\boldsymbol{x}^{(k)}-\boldsymbol{b}$
2. 优化步长 $\alpha_k=\frac{g^{(k)T}g^{(k)}}{g^{(k)T}\boldsymbol{Q}g^{(k)}}$
3. 迭代公式 $x^{(k+1)}=x^{(k)}-\frac{g^{(k)T}g^{(k)}}{g^{(k)T}\boldsymbol{Q}g^{(k)}}g(k)$

2.4 牛顿法

最速下降法只用到了函数的一阶导数，这种方法并不总是最高效的。而这里说的牛顿法用到了二阶导数，它的效率可能比最速下降法更优。应用局部极小点的一阶必要条件：$\mathbf{0}=\nabla q(\boldsymbol{x})=\boldsymbol{g}^{(k)}+\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right)\left(\boldsymbol{x}-\boldsymbol{x}^{(k)}\right)$

如果 hessian 矩阵正定，可得函数的极小值点为：$\boldsymbol{x}^{(k+1)}=\boldsymbol{x}^{(k)}-\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right)^{-1}\boldsymbol{g}^{(k)}$

3 结果与讨论

下图是在距离最小点比较远的情况下，三种方法的收敛曲线

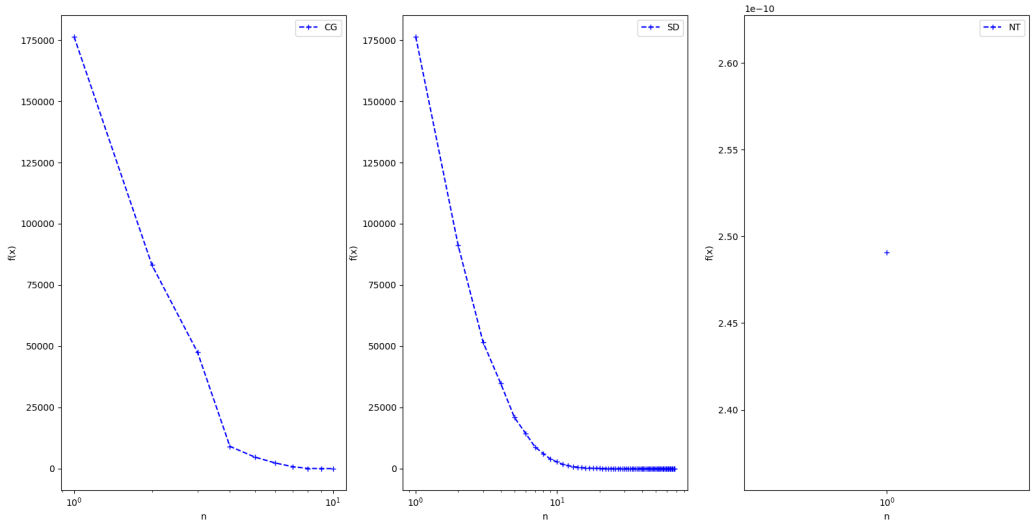

图 1: consistent convex（实对称正定）收敛曲线

可以看出对于 consistent convex，三种方法都可以计算收敛，其中牛顿法收敛最快，一步即可得到最优值点，梯度下降法醉眠，迭代次数远高于另外两种方法。

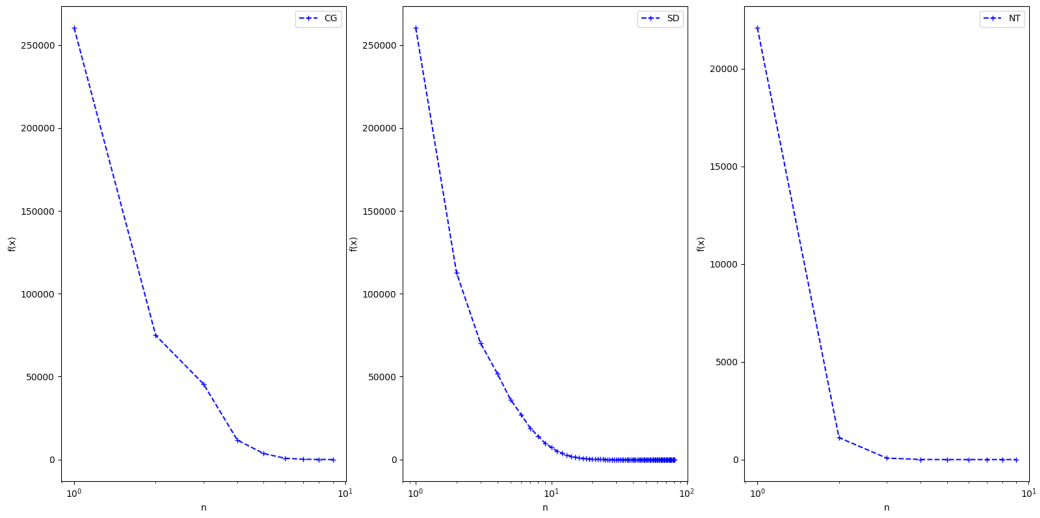

图 2: convex（实对称半正定）收敛曲线

可以看出对于一般的 convex，三种方法都可以计算收敛，这时牛顿法的优势没有那么大了，共轭梯度法仍然保持了较快的速度，牛顿法与共轭梯度法都比梯度下降法表现除了极大的优势。

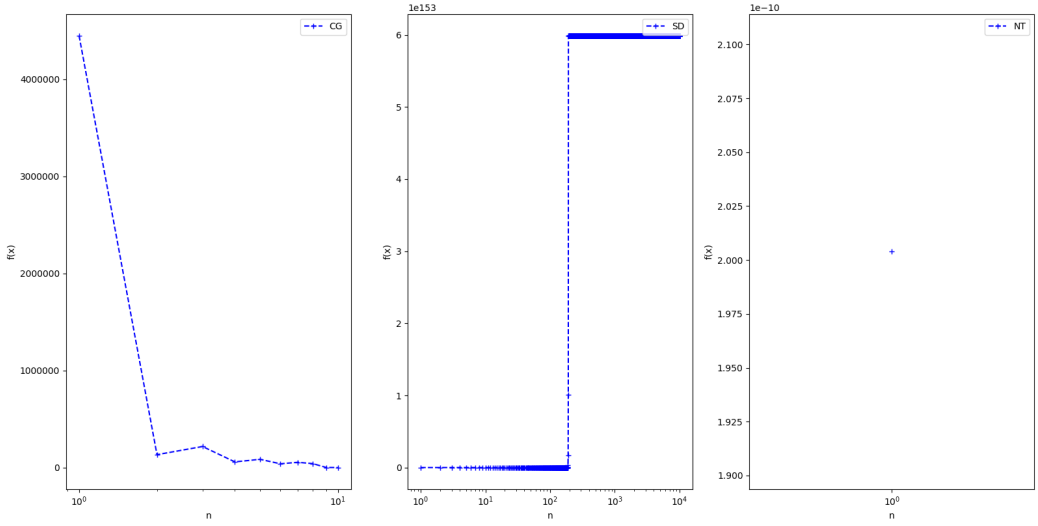

图 3: bound convex（不定）收敛曲线

对于 bound convex，梯度下降法没有办法到达极小值点，陷入了局部极值点，而牛顿法仍然收敛了，可能是初始点距离目标极小值点比较近。

总结来说，共轭梯度法是最稳定的算法，牛顿法会因为目标函数的 hessin 阵性质而影响收敛性，如果 hessin 是非正定的，那么牛顿法牛顿法的搜索方向并不一定是目标函数值的下降方向。而最速下降法由于只用到了一阶导数信息（不像共轭梯度用到了函数的阶数，牛顿法用到二阶导数信息），表现是三个算法中最差的。

4 源代码

使用 python 编写代码如下.

```
#-*- coding: utf-8 -*-
"""
@author: Starklee created on 2019-10-13
"""
import random
import sympy
import numpy as np
import math
import matplotlib.pyplot as pl
from scipy.linalg import orth
from mpl_toolkits.mplot3d import Axes3D as ax3

def random_int_list(start, stop, length):
  start, stop = (int(start), int(stop)) if start <= stop else (int(stop), int(start))
```

```
    length = int(abs(length)) if length else 0
    random_list = []
    for i in range(length):
     random_list.append(random.randint(start, stop))
    return random_list


def random_SSDP(N):
    # 生成随机对称正定矩阵
    D = np.diag(random_int_list(1, 100, N))
    a = np.array(random_int_list(1, 100, N ** 2))
    u = a.reshape((N, N))
    U = orth(u)
    A = np.dot(U.T, D)
    A = np.dot(A, U)
    lam = np.linalg.eigvals(A)
    A = A - max(lam)*np.eye(N)
    print('是半正定矩阵')
    return A

def random_UD(N):
    # 生成随机对称正定矩阵
    D = np.diag(random_int_list(1, 100, N))
    a = np.array(random_int_list(1, 100, N ** 2))
    u = a.reshape((N, N))
    U = orth(u)
    A = np.dot(U.T, D)
    A = np.dot(A, U)
    lam = np.linalg.eigvals(A)
    A = A - ((max(lam)+min(lam))/2)*np.eye(N)
    B = np.linalg.eigvals(A)
    if np.all(B < 0):
        print('是负定␣矩阵')
    elif np.all(B > 0):
        print('是正定␣矩阵')
    elif np.all(B >= 0):
        print('是半正定␣矩阵')
    else:
        print('是不定矩阵')
    return A

def random_SDP(N):
    # 生成随机对称正定矩阵
    D = np.diag(random_int_list(1, 100, N))
    a = np.array(random_int_list(1, 100, N ** 2))
    u = a.reshape((N, N))
    U = orth(u)
    A = np.dot(U.T, D)
    A = np.dot(A, U)
    B = np.linalg.eigvals(A)
    if np.all(B > 0):
        print('是正定␣矩阵')
    return A




# 共轭梯度法
def CG(x0, N, E, f, f_d):
    X = [];
    Y = [];
    Y_d = [];
    #初始化
    n = 0
    r0 = -f_d(x0)
    rk0=r0
    rk=r0
    dk=0
    x=x0
    e=1
```

```
    while n < N and e > E:
        n = n + 1
        dk = rk + dk*np.linalg.norm(rk) ** 2/np.linalg.norm(rk0) ** 2
        ak = np.linalg.norm(rk) ** 2/np.dot(np.dot(dk.T, A), dk)
        x = x + dk * ak
        rk0=rk
        rk =-f_d(x)
        X.append(n)
        Y.append(f(x)[0, 0])
        e = np.linalg.norm(f_d(x))
        Y_d.append(e)
        print ('第%2s次迭代: f(x)=%f e=%f' % (n, f(x), e))
    return X, Y, Y_d

# 最速下降法
def SD(x0, N, E, f, f_d):
    X = [];
    Y = [];
    Y_d = [];
    #初始化
    n = 0
    x=x0
    e=1
    #f = lambda x: 0.5 * (np.dot(np.dot(x.T, A), x)) - np.dot(b.T, x)
    #f_d = lambda x: np.dot(A, x) - b
    while n < N and e > E:
        n = n + 1
        dk = f_d(x)
        ak = np.dot(dk.T, dk)/np.dot(np.dot(dk.T, A), dk)
        x = x - dk * ak
        X.append(n)
        Y.append(f(x)[0, 0])
        e = np.linalg.norm(f_d(x))
        Y_d.append(e)
        print ('第%2s次迭代: f(x)=%f e=%f' % (n, f(x), e))
    return X, Y, Y_d

# 牛顿法
def NT(x0, N, E, f, f_d):
    X = [];
    Y = [];
    Y_d = [];
    #初始化
    n = 0
    x=x0
    e=1
    #f = lambda x: 0.5 * (np.dot(np.dot(x.T, A), x)) - np.dot(b.T, x)
    #f_d = lambda x: np.dot(A, x) - b -> gx
    while n < N and e > E:
        n = n + 1
        dk = f_d(x)
        x = x - np.dot(np.linalg.inv(A),dk)
        X.append(n)
        Y.append(f(x)[0, 0])
        e = np.linalg.norm(f_d(x))
        Y_d.append(e)
        print ('第%2s次迭代: f(x)=%f e=%f' % (n, f(x), e))
    return X, Y, Y_d



if __name__ == '__main__':
    N=40
#    A=random_SSDP(N)
    A = random_SDP(N)
#    A=random_UD(N)
    x_goal = np.ones((N, 1))
    b = np.dot(A, x_goal)
    print(np.dot(b.T, x_goal))
```

```
    c = 0
    f = lambda x: 0.5 * (np.dot(np.dot(x.T, A), x)) - np.dot(b.T, x)
    f_d = lambda x: np.dot(A, x) - b
    x0 = x_goal + np.random.rand(N, 1) * 100000
    N = 10000#最大迭代次数
    E = 10 ** (-6)#计算精度要求
    print ('共轭梯度')
    X1, Y1, Y_d1 = CG(x0, N, E, f, f_d)
    print('最速下降')
    X2, Y2, Y_d2 = SD(x0, N, E, f, f_d)
    print('牛顿法')
    X3, Y3, Y_d3 = NT(x0, N, E, f, f_d)

    figure1 = pl.figure('SSDP')
    pl.subplot(1,3,1)
    pl.semilogx(X1, Y_d1, color='b',linestyle='dashed', marker='+', label='CG')#绘制收敛图像
    pl.legend()
    pl.xlabel('n')
    pl.ylabel('f(x)')

    pl.subplot(1, 3, 2)
    pl.semilogx(X2, Y_d2, color='b',linestyle='dashed', marker='+', label='SD')#绘制收敛图像
    pl.legend()
    pl.xlabel('n')
    pl.ylabel('f(x)')

    pl.subplot(1, 3, 3)
    pl.semilogx(X3, Y_d3, color='b',linestyle='dashed', marker='+', label='NT')#绘制收敛图像
    pl.legend()
    pl.xlabel('n')
    pl.ylabel('f(x)')

    pl.show()
```