最优化大作业

张晋

北京航空航天大学，数学与系统科学学院

2017 年 12 月 25 日

目录

目录 3

1 Problem 5.6

对于 $q(\boldsymbol{x}) = (10x_1^2 - 18x_1x_2 + 10x_2^2)/2 + 4x_1 - 15x_2 + 13$

1.1 重要参数

$$G = \begin{bmatrix} 10 & -9 \\ -9 & 10 \end{bmatrix}, \qquad \lambda_1 = 19, \lambda_2 = 1, \qquad (\frac{\lambda_1 - \lambda_2}{\lambda_1 + x_2})^2 = 0.81$$

1.2 算法伪代码

Algorithm 1 Steepest-denscent method for problem(5.6)

Given $\boldsymbol{x}^{(0)}$ and G
Set $\boldsymbol{p}^{(0)} = -\boldsymbol{g}^{(0)}, k = 0$
while $\|\boldsymbol{g}^{(k)}\| > \epsilon$ **do**
    Set $\alpha_k = -\dfrac{{\boldsymbol{p}^{(k)}}^T \boldsymbol{g}^{(k)}}{{\boldsymbol{p}^{(k)}}^T \boldsymbol{G} \boldsymbol{p}^{(k)}}$
    Set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{p}^{(k)}$
    Set $\boldsymbol{g}^{(k+1)} = \boldsymbol{g}(\boldsymbol{x}^{(k+1)})$
    Set $\boldsymbol{p}^{(k)} = -\boldsymbol{g}^{(k)}$
    Set $k = k + 1$
end while

1.3 计算结果展示

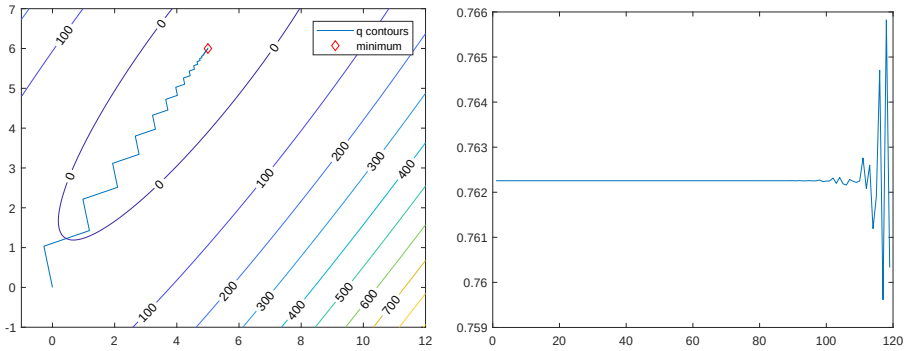


图 1: Steepest-denscent in (0,0)

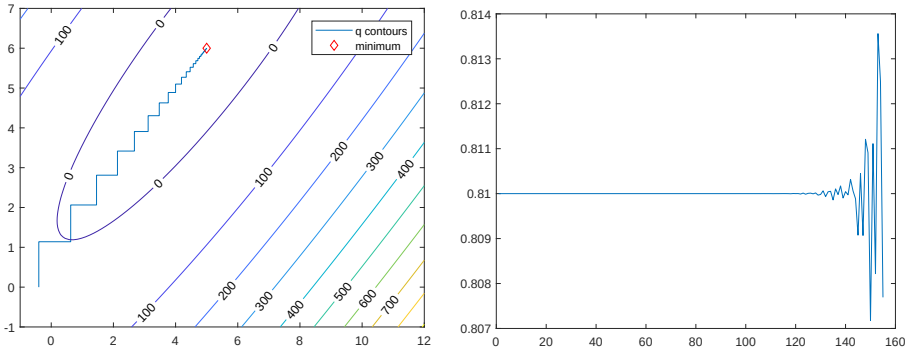


图 2: Steepest-denscent in (-0.4,0)

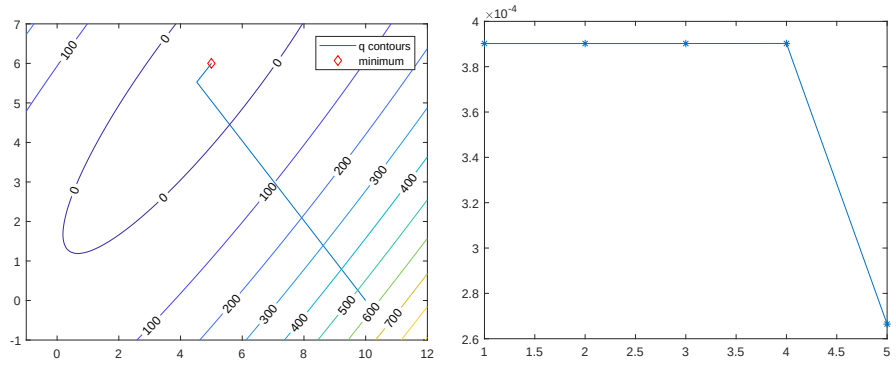


图 3: Steepest-denscent in (10,0)

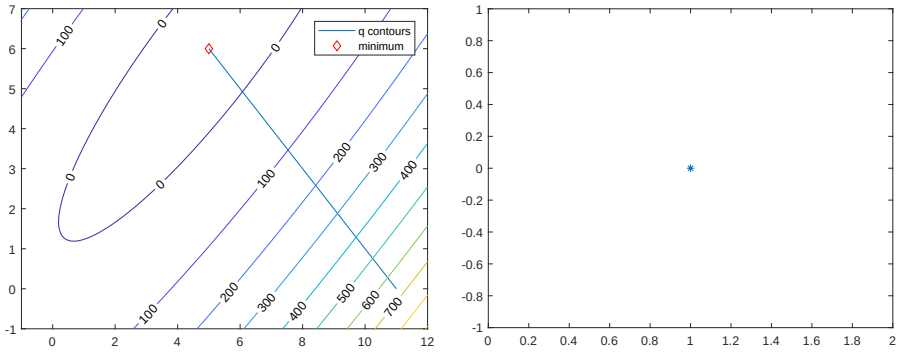


图 4: Steepest-denscent in (11,0)

表 1: 收敛因子比较

起始点	$(0,0)$	$(-0.4,0)$	$(10,0)$	$(11,0)$
收敛因子	0.762255	0.810000	0.000390	0

1.4 分析

由于目标函数为凸函数，故使用梯度下降法从这四个不同的起始点出发都能收敛到全局最优点，然而线性收敛因子却互不相同.

由图像可知：在迭代开始后，函数值的收敛速度稳定在一个值左右，直到接近最优点时，收敛速度开始较大幅度波动。

而且，可以看出，初始点越接近等值线椭圆的狭长端，线性收敛因子越大，在点 $(-0.4, 0)$ 处甚至达到了线性收敛因子的上界 0.81, 而离狭长端越远，收敛因子越小。这是由于梯度下降在构造搜索方向时没有充分利用到函数的二阶导数信息，在面临“峡谷”状的函数时，会反复震荡到“峡谷”的另一端，而不能直接向最优值方向前进。

容易看出，等值线椭圆的长轴端斜率为 $9/10$，且 $(-0.4, 0) = (5, 6) - 0.6 * (9, 10)$，这说明点 $(-0.4, 0)$ 刚好处在等值线椭圆的长轴上，因此也是震荡最剧烈的地方，收敛速度达到了最坏收敛速度。

2 Problem 5.7

2.1 重要参数

$$f(x) = 9x - 4\ln(x-7)$$
$$g(x) = f'(x) = 9 - \frac{4}{x-7}$$
$$G(x) = g'(x) = \frac{4}{(x-7)^2}$$

其迭代公式为:

$$x^{(k+1)} = x^{(k)} - \frac{1}{4}(x^{(k)} - 7)(9x^{(k)} - 67) \tag{1}$$

2.2 算法伪代码

Algorithm 2 Newton method for problem(5.7)

1: Given $x^{(0)}$ and compute $g(x) = f'(x)$
2: Compute $G(x) = g'(x)$
3: Set $g^{(0)} = g(x^{(0)}), k = 0$
4: **while** $|g^{(k)}| > \epsilon$ **do**
5: Set $s^{(k)} = -g^{(k)}/G^{(k)}$
6: Set $x^{(k+1)} = x^{(k)} + s^{(k)}$
7: Set $k = k + 1$
8: **end while**

2.3 计算结果展示

由 MATLAB 函数 fminsearch 求得最优解为:

$$x^\star = 7.444421386718750, \quad f(x^\star) = 70.243720870248540$$

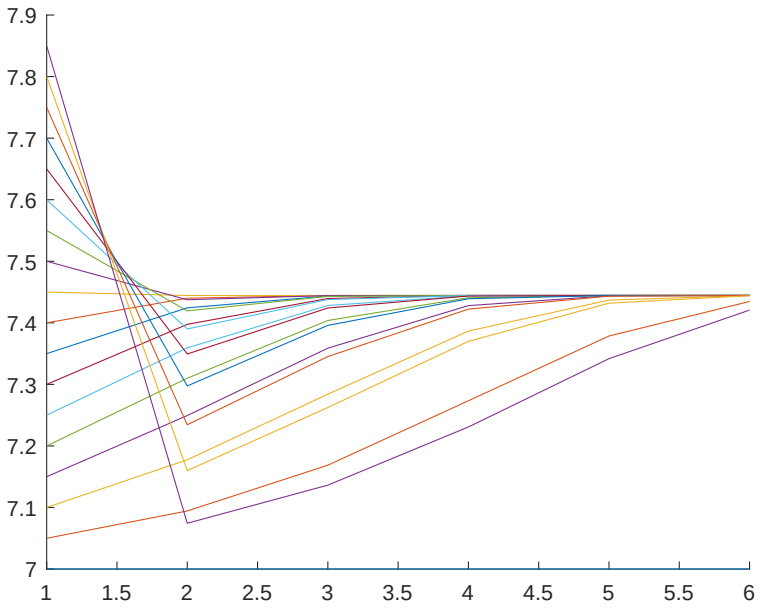
表 2: 迭代 5 次过程

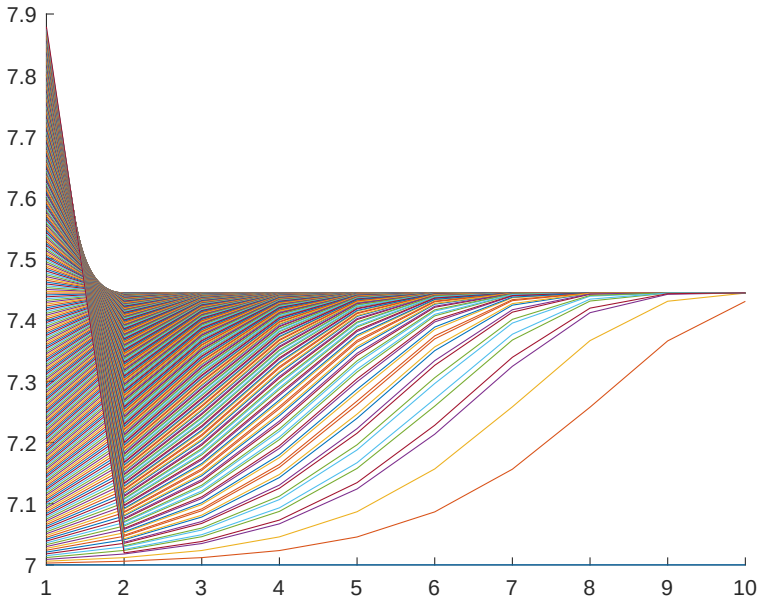
$x^{(0)}$	7.4	7.2	7.01	7.80	7.88
$x^{(1)}$	7.44	7.31	7.019775	7.16	7.0176
$x^{(2)}$	7.4444	7.403775	7.038670136	7.2624	7.03450304
$x^{(3)}$	7.44444444	7.440722936	7.073975668	7.36987904	7.066327546
$x^{(4)}$	7.444444444	7.444413283	7.135638438	7.431934445	7.122756569
$x^{(5)}$	7.444444444	7.444444442	7.229881858	7.444092319	7.211607493
$f(x^{(5)})$	70.24372086	70.24372086	70.94969577	70.24372212	71.11655611

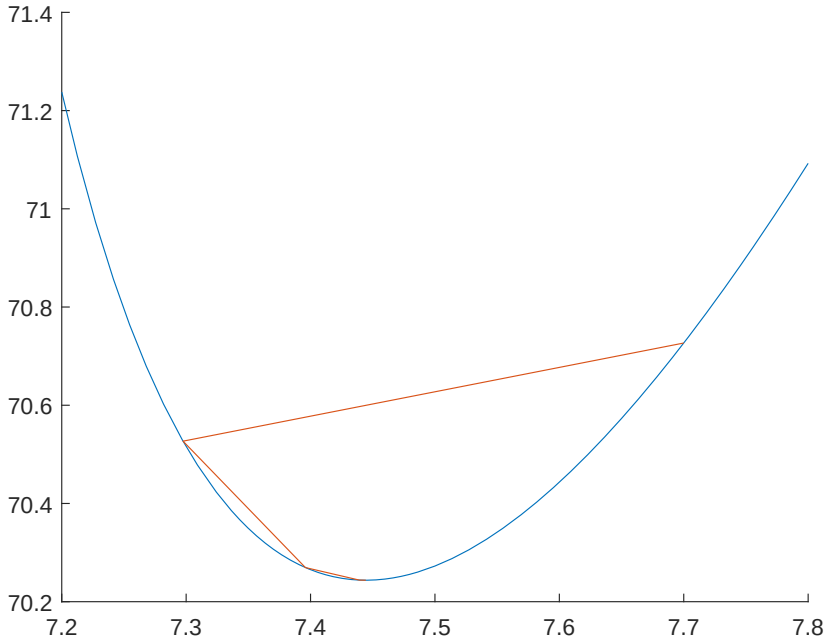
表 3: 大范围迭代结果

初始点	迭代终止点	初始点	迭代终止点
5	-Inf	9	-Inf
5.25	-Inf	9.25	-Inf
5.5	-Inf	9.5	-Inf
5.75	-Inf	9.75	-Inf
6	-Inf	10	-Inf
6.25	-Inf	10.25	-Inf
6.5	-Inf	10.5	-Inf
6.75	-Inf	10.75	-Inf
7	7	11	-Inf
7.25	7.44444444444445	11.25	-Inf
7.5	7.44444444444445	11.5	-Inf
7.75	7.44444444444445	11.75	-Inf
8	-Inf	12	-Inf
8.25	-Inf	12.25	-Inf
8.5	-Inf	12.5	-Inf
8.75	-Inf	12.75	-Inf

2.4 收敛域内点的迭代情况







2.5 总结分析

关于收敛域的分析

首先由于 $G(x) > 0$，故在定义域内总能收敛。
然后我们观察迭代公式 (equation:1)

$$x^{(k+1)} = x^{(k)} - \frac{1}{4}(x^{(k)} - 7)(9x^{(k)} - 67)$$

可以得出以下结论：

- 容易看出两个稳定点为 $x' = 7\ and\ x^\star = 67/9$
- 若初始点在定义域内，则要求 $x_0 >= 7$
- 然后要避免 x_k 迭代时使得 $x^{(k+1)}$ 跳出定义域，因此解方程

  $$7 = x^{(k)} - \frac{1}{4}(x^{(k)} - 7)(9x^{(k)} - 67)$$

  得到解:$x_k = 7\ or\ 71/9$
- 因此该迭代公式的收敛域为 $(7, 71/9)$

点的迭代性质分析

观察点的迭代图像可以看到：

- 在收敛域 $(7, 71/9)$ 内 x 将逐渐收敛到稳定点 $x^\star = 67/9$
- 若起始点 $x_0 \in (67/9, 71/9)$, 则其迭代第一步的步长非常大，使得一下子就跃到 $67/9$ 以下，然后慢慢朝稳定点方向靠近，而且一开始离稳定点越大，步长越大, 这可由公式 (equation:1) 解释
- 但当 x_0 离稳定点太远，超出 $71/9$ 时，将会使得下一步迭代步长过大，导致超出定义域。
- 若起始点 $x_0 \in (7, 67/9)$ 时，x_0 不会像之前那样猛烈地向稳定点 $x^\star$ 迈进，而是缓慢地迭代，而越远离稳定点 $x^\star$，迭代得越慢
- 透过公式 (equation:1)，我们可以用一种形象的语言来描述这种现象：当 $x_0 > 67/9$ 时，它受到了两个稳定点 $x', x^\star$ 的“吸引”，因此迈出的步长特别大。而当 $7 < x_0 < 67/9$ 时，$x^\star$ 对它的“吸引力”被另一个稳定点 x' 的“吸引力”抵消了一部分，而离 x' 越近，“引力”被抵消得越厉害，迭代也就越慢.

3 Problem 5.8

3.1 重要参数

$$
\begin{aligned}
f(x_1, x_2) = &-9x_1 - 10x_2 \\
&- \mu\big[\ln(-x_1 - x_2 + 100) + \ln(-x_1 + x_2 + 50) + \ln(x_1) + \ln(x_2)\big]
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{g}(x_1, x_2) &= \nabla f(x_1, x_2) \\
&= \begin{bmatrix} -9 - \dfrac{\mu}{x_1} + \dfrac{\mu}{100 - x_1 - x_2} + \dfrac{\mu}{50 - x_1 + x_2} \\ -10 - \dfrac{\mu}{x_2} + \dfrac{\mu}{100 - x_1 - x_2} - \dfrac{\mu}{50 - x_1 + x_2} \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&\boldsymbol{G}[x_1, x_2] \\
=&\nabla \boldsymbol{g}(x_1, x_2) \\
=&\begin{bmatrix} \dfrac{1}{{x_1}^2} + \dfrac{1}{(100 - x_1 - x_2)^2} + \dfrac{1}{(50 - x_1 + x_2)^2} & \dfrac{1}{(100 - x_1 - x_2)^2} - \dfrac{1}{(50 - x_1 + x_2)^2} \\ \dfrac{1}{(100 - x_1 - x_2)^2} - \dfrac{1}{(50 - x_1 + x_2)^2} & \dfrac{1}{{x_2}^2} + \dfrac{1}{(100 - x_1 - x_2)^2} + \dfrac{1}{(50 - x_1 + x_2)^2} \end{bmatrix}
\end{aligned}
$$

3.2 算法伪代码

Algorithm 3 Backtracking-Armijo Line Search

Choose $\bar{\alpha} > 0, \quad \gamma, \rho \in (0, 1)$
Set $\alpha = \bar{\alpha}$
while $\phi(\alpha) > \phi(0) + \rho\phi'(0)\alpha$ **do**
    Set $\alpha = \gamma\alpha$
end while
return α as α_k

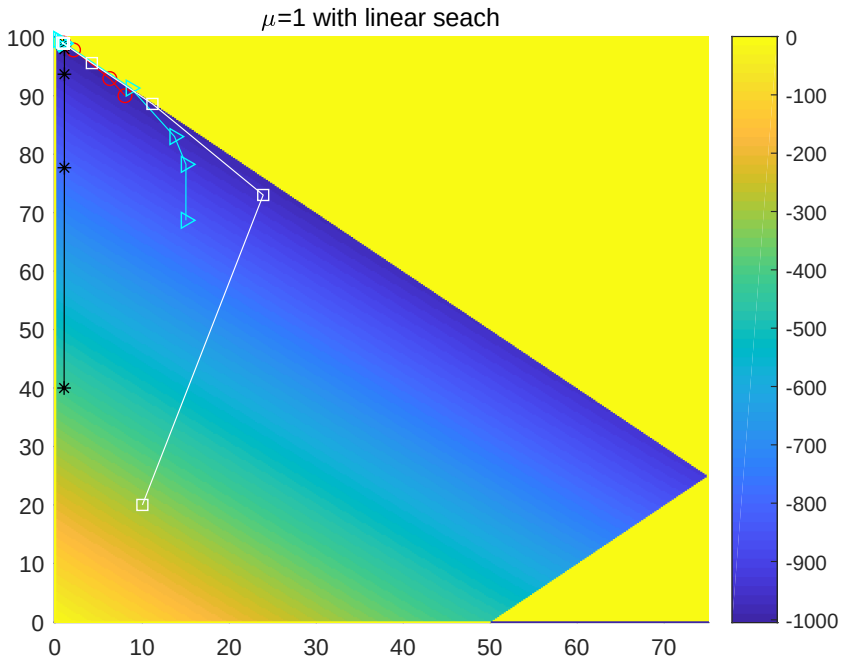
Algorithm 4 Newton method without Armijo Line Searchfor problem(5.8)

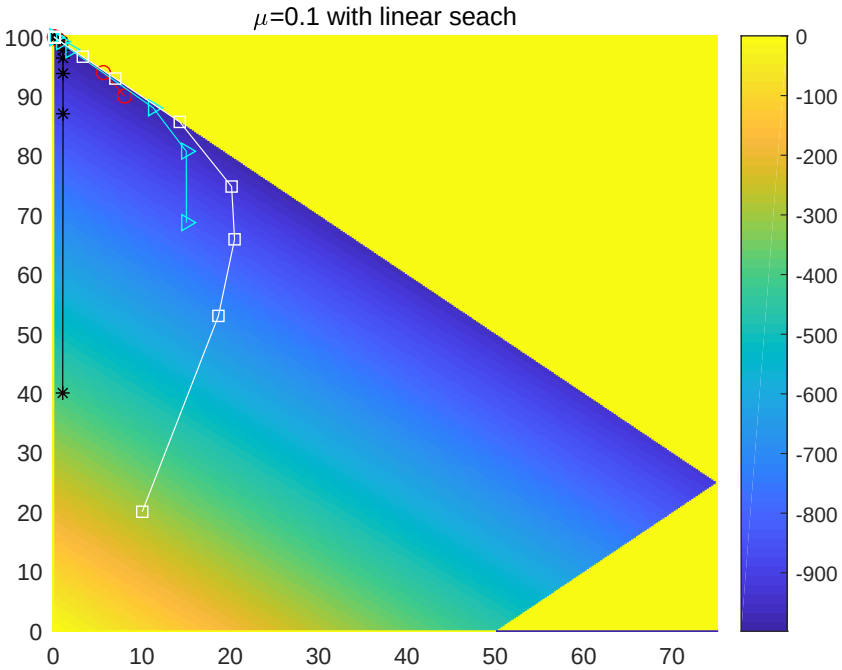
Given $\boldsymbol{x}^{(0)}$ and compute $\boldsymbol{g}(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$
Compute $\boldsymbol{G}(\boldsymbol{x}) = \nabla \boldsymbol{g}(\boldsymbol{x})^T$
Set $\boldsymbol{g}^{(0)} = \boldsymbol{g}(\boldsymbol{x}^{(0)}), k = 0$
while $\|\boldsymbol{g}^{(k)}\|_2 > \epsilon$ **do**
    Set $\boldsymbol{s}^{(k)} = -{\boldsymbol{G}^{(k)}}^{-1}\boldsymbol{g}^{(k)}$
    Set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{s}^{(k)}$
    Set $k = k + 1$
end while

Algorithm 5 Newton method with Armijo Line Searchfor problem(5.58)

Given $\boldsymbol{x}^{(0)}$ and compute $\boldsymbol{g}(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$
Compute $\boldsymbol{G}(\boldsymbol{x}) = \nabla \boldsymbol{g}(\boldsymbol{x})^T$
Set $\boldsymbol{g}^{(0)} = \boldsymbol{g}(\boldsymbol{x}^{(0)}), k = 0$
while $\|\boldsymbol{g}^{(k)}\|_2 > \epsilon$ **do**
    Set $\boldsymbol{s}^{(k)} = -{\boldsymbol{G}^{(k)}}^{-1}\boldsymbol{g}^{(k)}$
    Compute α_k by Line Search(**Algorithm** 3)
    Set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{s}^{(k)}$
    Set $k = k + 1$
end while

3.3 迭代点运动轨迹展示





3.4 总结分析

本题中 Armijo 线搜索的参数为 $\gamma = 0.5, \rho = 0.01$

没加线搜索和越界判定之前，牛顿法总是一两次就冲到了定义域外，然后无法收敛。

然后我加了个越界判定: 若下一步的迭代点不在定义域内，则将步长 α 缩小一半，牛顿法很好地收敛到了全局最优点。

```
%Check函数检查是否越界，以缩短步长ak
while(Check(x+ak*double(p')))
    ak=0.5*ak;
    xk=x+ak*double(p');
end
```

而后在越界判定的基础上加了个线搜索：

```
%采用Armijo法则计算近似步长ak
while(F(xk(1),xk(2)) > (F(x(1),x(2))+0.01*double(p'*g)*ak)
||Check(x+ak*double(p')))%Check函数检查是否越界
    ak=0.5*ak;
    xk=x+ak*double(p');
end
```

然而得到的结果与加上线搜索之前一样，可见在迭代的前期，越界判定起到了一定类似于线搜索缩短步长的效果，而到了迭代的后期，牛顿法的基本步长已经满足 Armijo 法则，还原成了基本牛顿法。

3.5 牛顿法总结分析

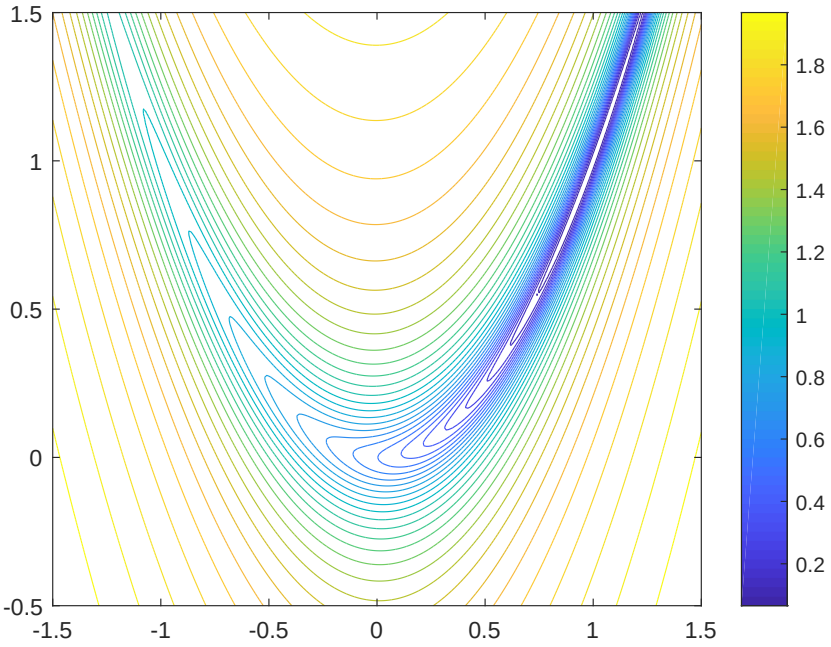
在面临梯度变化特别大的情况时，最速下降法显得非常不尽人意，即使在离稳定点非常近的地方依旧不能快速收敛，呈锯齿状，由我们所学的知识可以知道，最速下降法的收敛速度强烈依赖于 Hessian 矩阵的条件数。

而牛顿法显得更加智能

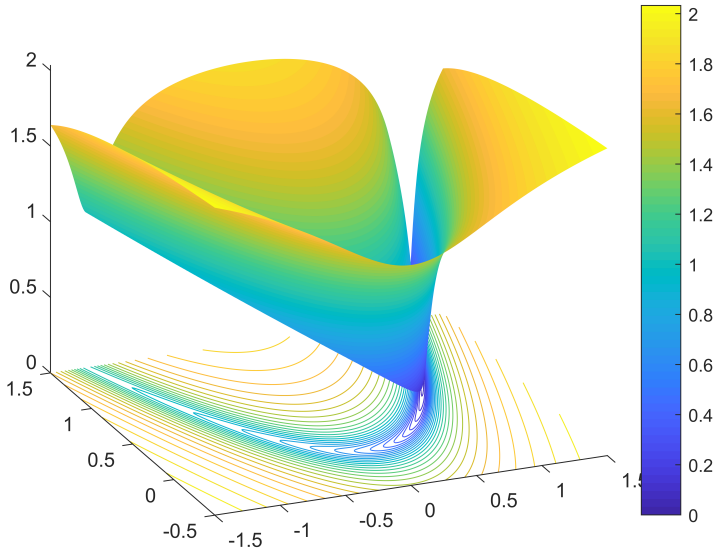
4 Problem5.9

4.1 Rosenbrock 函数图像

首先画出 Rosenbrock 函数的图像及等值线如下[1]:



[1]由于 Rosenbrock 函数过于陡峭，因此对原函数进行了取对数处理，以便于观察其特点。



4.2 算法伪代码

Armijo 线搜索法参看上节的 (**Algorithm** 3)

带线搜索的 Newton 法算法参看上节的 (**Algorithm** 5)

Algorithm 6 Steepest-denscent-Armijo method for problem(5.9)

1: Given $\boldsymbol{x}^{(0)}$ and G
2: Set $\boldsymbol{p}^{(0)} = -\boldsymbol{g}^{(0)}, k = 0$
3: **while** $\|\boldsymbol{g}^{(k)}\| > \epsilon$ **do**
4: Compute α_k by Line Search(**Algorithm** 3)
5: Set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{p}^{(k)}$
6: Set $\boldsymbol{g}^{(k+1)} = g(\boldsymbol{x}^{(k+1)})$
7: Set $\boldsymbol{p}^{(k)} = -\boldsymbol{g}^{(k)}$
8: Set $k = k + 1$
9: **end while**

4.3 计算结果展示

本题中 Armijo 线搜索的参数为 $\gamma = 0.5, \rho = 0.01$, 并设置最大搜索步长为 200.

然后分别以梯度下降法和牛顿法迭代，并画出等高线、运动轨迹、迭代值如下:

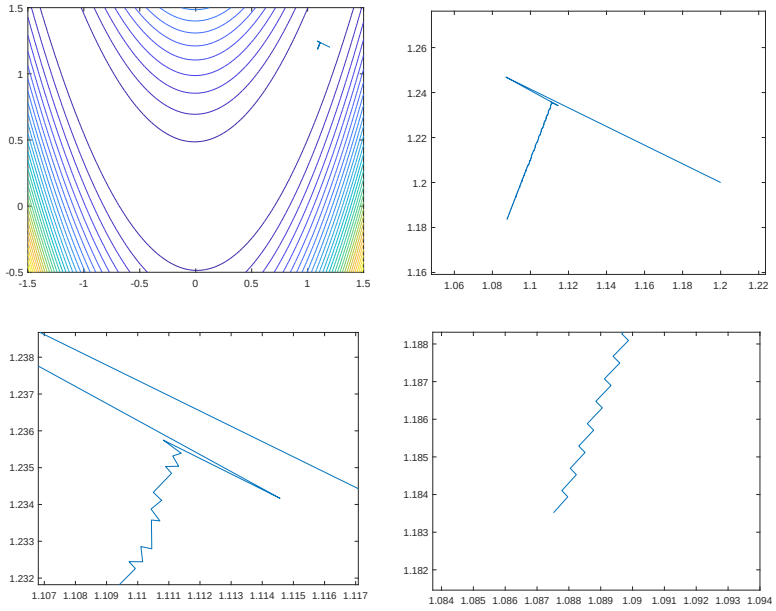


图 5: Steepest-denscent in (1.2,1.2)

```
%Result for Steepest-denscent in (1.2,1.2)
Step[1]:  x=[ 1.200000 1.200000 ] optim_fx=5.800000
Step[2]:  x=[ 1.087109 1.246875 ] optim_fx=0.430975
Step[3]:  x=[ 1.114571 1.234166 ] optim_fx=0.019689
...
...
...
Step[198]:  x=[ 1.083582 1.174725 ] optim_fx=0.007019
Step[199]:  x=[ 1.083742 1.174500 ] optim_fx=0.007013
Step[200]:  x=[ 1.083580 1.174500 ] optim_fx=0.006998
%最速下降法,共迭代 200 步
%最优解:
x=[ 1.083580e+00 1.174500e+00 ] optim_fx=0.006998
```

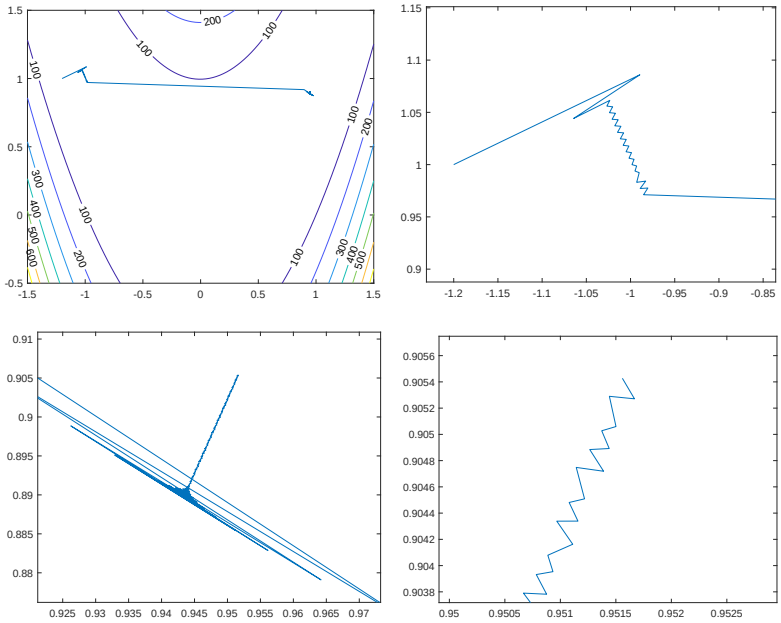


图 6: Steepest-denscent in (-1.2,1)

```
%Result for Steepest-denscent in (-1.2,1)
Step[1]:  x=[ -1.200000 1.000000 ] optim_fx=24.200000
Step[2]:  x=[ -0.989453 1.085938 ] optim_fx=5.101113
Step[3]:  x=[ -1.026893 1.065055 ] optim_fx=4.119416
Step[4]:  x=[ -1.027979 1.056815 ] optim_fx=4.112700
Step[5]:  x=[ 0.984742 1.049394 ] optim_fx=0.635080
Step[6]:  x=[ 1.015421 1.033832 ] optim_fx=0.000995
...
...
...
Step[195]:  x=[ 1.005171 1.010402 ] optim_fx=0.000027
Step[196]:  x=[ 1.005177 1.010389 ] optim_fx=0.000027
Step[197]:  x=[ 1.005163 1.010386 ] optim_fx=0.000027
Step[198]:  x=[ 1.005169 1.010373 ] optim_fx=0.000027
Step[199]:  x=[ 1.005155 1.010370 ] optim_fx=0.000027
Step[200]:  x=[ 1.005161 1.010357 ] optim_fx=0.000027
%最速下降法,共迭代 200 步
%最优解:
x=[ 1.005161e+00 1.010357e+00 ] optim_fx=0.000027
```

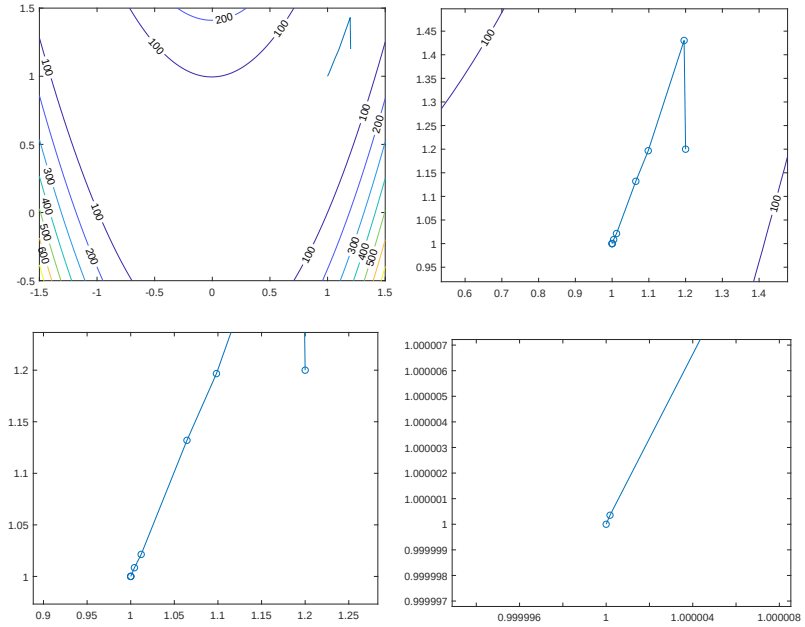


图 7: Newton-Armijo in (1.2,1.2)

```
%Result for Newton-Armijo in (1.2,1.2)
Step[1]: x=[ 1.200000 1.200000 ] optim_fx=5.800000
Step[2]: x=[ 1.195918 1.430204 ] optim_fx=0.038384
Step[3]: x=[ 1.098284 1.196688 ] optim_fx=0.018762
Step[4]: x=[ 1.064488 1.131993 ] optim_fx=0.004289
Step[5]: x=[ 1.011992 1.021372 ] optim_fx=0.000903
Step[6]: x=[ 1.004261 1.008481 ] optim_fx=0.000019
Step[7]: x=[ 1.000050 1.000083 ] optim_fx=0.000000
Step[8]: x=[ 1.000000 1.000000 ] optim_fx=0.000000
Step[9]: x=[ 1.000000 1.000000 ] optim_fx=0.000000
%牛顿 Armijo 回溯法,,共迭代 9 步
%最优解:
x=[ 1.000000e+00 1.000000e+00 ] optim_fx=0.000000
```

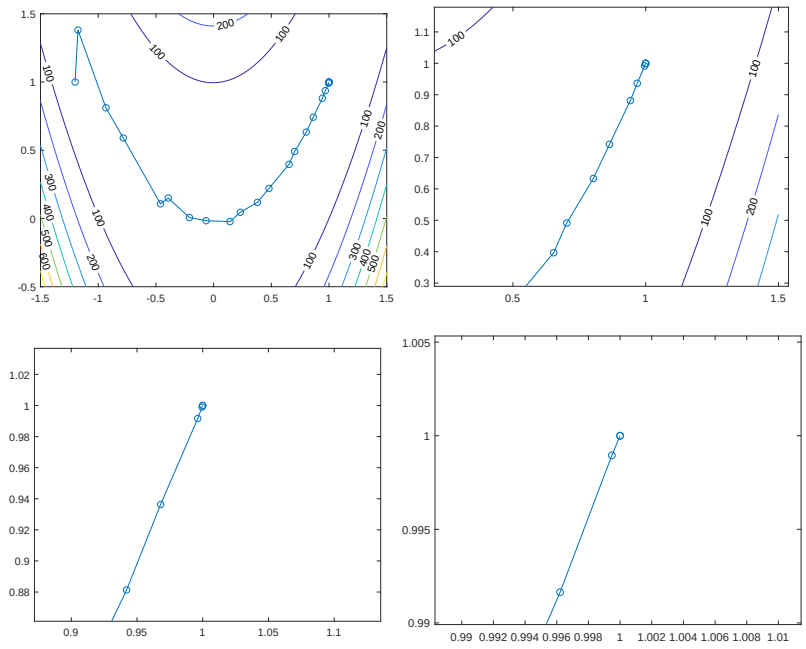


图 8: Newton-Armijo in (-1.2,1)

```
%Result for Newton-Armijo in (-1.2,1)
Step[1]:  x=[ -1.200000 1.000000 ] optim_fx=24.200000
Step[2]:  x=[ -1.175281 1.380674 ] optim_fx=4.731884
Step[3]:  x=[ -0.932981 0.811211 ] optim_fx=4.087399
Step[4]:  x=[ -0.782540 0.589736 ] optim_fx=3.228673
Step[5]:  x=[ -0.459997 0.107563 ] optim_fx=3.213898
Step[6]:  x=[ -0.393046 0.150002 ] optim_fx=1.942585
...
...
...
Step[17]:  x=[ 0.942079 0.881336 ] optim_fx=0.007169
Step[18]:  x=[ 0.967992 0.936337 ] optim_fx=0.001070
Step[19]:  x=[ 0.996210 0.991639 ] optim_fx=0.000078
Step[20]:  x=[ 0.999479 0.998948 ] optim_fx=0.000000
Step[21]:  x=[ 0.999999 0.999998 ] optim_fx=0.000000
Step[22]:  x=[ 1.000000 1.000000 ] optim_fx=0.000000
Step[22]: x=[ 1.000000 1.000000 ] optim_fx=0.000000
%牛顿 Armijo 回溯法,,共迭代 22 步
%最优解:
x=[ 1.000000e+00 1.000000e+00 ] optim_fx=0.000000
```

5 Problem 5.19

5.1 重要数据展示

求解对称正定的方程组 $\boldsymbol{Gx} = \boldsymbol{b}$ 可以看作极小化二次函数

$$q(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x^T G x} - \boldsymbol{b^T x}$$

$$\boldsymbol{g}(\boldsymbol{x}) = \nabla q(\boldsymbol{x}) = \boldsymbol{Gx} - \boldsymbol{b}$$

其中，$\nabla^2 q(\boldsymbol{x}) = \boldsymbol{G}$ 为 Hilbert 矩阵。

希尔伯特矩阵是一种系数都是单位分数的方块矩阵，希尔伯特矩阵 $\boldsymbol{H}$ 的第 i 横行第 j 纵列的系数是 $H_{ij} = \dfrac{1}{i+j-1}$

以 5 阶的 Hilbert 矩阵为例，其矩阵如下:

$$H_5 = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{pmatrix}$$

希尔伯特矩阵是一种数学变换矩阵，正定，且高度病态（即，任何一个元素发生一点变动，整个矩阵的行列式的值和逆矩阵都会发生巨大变化），病态程度和阶数相关。

希尔伯特矩阵的一个特点就是条件数特别大，以上面的 5 阶 Hilbert 矩阵为例，当范数为 l_2 矩阵范数时，其条件数大约是 4.8×10^5

经证明: 当 $n \to \infty$ 的时候，$n \times n$ 的希尔伯特矩阵的条件数近似为 $O((1+\sqrt{2})^{4n}/\sqrt{n})$

MATLAB 中有直接生成 N 阶希尔伯特矩阵的函数: $\boxed{hilb(N)}$

5.2 算法伪代码

Algorithm 7 Conjugate gradient method method for problem(5.19)

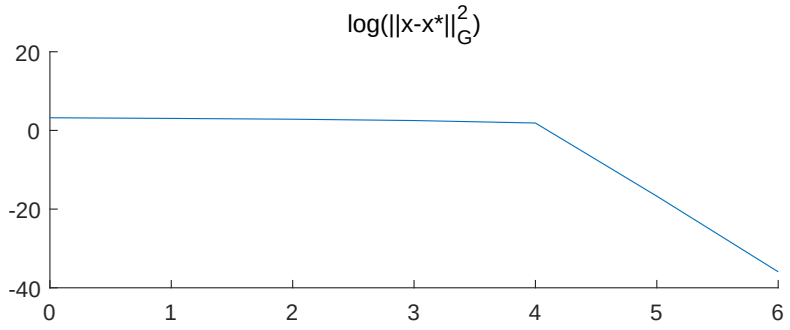
Given $\boldsymbol{x}^{(0)}$ and $\boldsymbol{G}$
Set $\boldsymbol{g}^{(0)} = \boldsymbol{G}\boldsymbol{x}^{(0)} - \boldsymbol{b}$
Set $\boldsymbol{p}^{(0)} = -\boldsymbol{g}^{(0)}, k = 0$
while $\|\boldsymbol{g}^{(k)}\| > \epsilon$ **do**
  Set $\boldsymbol{d} = \boldsymbol{G}\boldsymbol{p}^{(k)}$
  Set $\alpha_k = \dfrac{{\boldsymbol{g}^{(k)}}^T \boldsymbol{g}^{(k)}}{{\boldsymbol{p}^{(k)}}^T \boldsymbol{d}}$
  Set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{p}^{(k)}$
  Set $\boldsymbol{g}^{(k+1)} = \boldsymbol{g}^{(k)} + \alpha_k \boldsymbol{d}$
  Set $\beta_{k+1} = \dfrac{{\boldsymbol{g}^{(k+1)}}^T \boldsymbol{g}^{(k+1)}}{{\boldsymbol{g}^{(k)}}^T \boldsymbol{g}^{(k)}}$
  Set $\boldsymbol{p}^{(k+1)} = -\boldsymbol{g}^{(k+1)} + \beta_{k+1} \boldsymbol{p}^{(k)}$
  Set $k = k + 1$
end while
return $\boldsymbol{x}^{(k)}$ as $\boldsymbol{x}^\star$

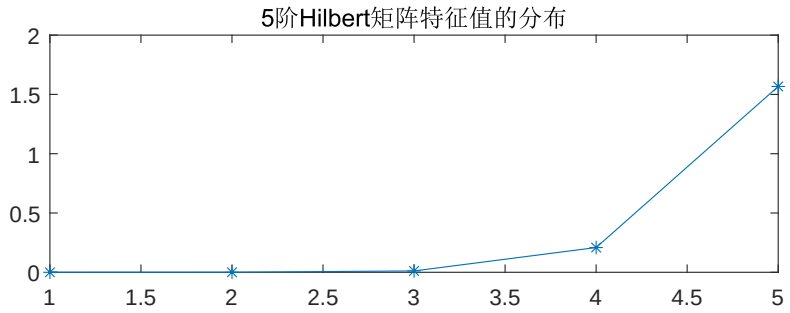
5.3 迭代结果展示

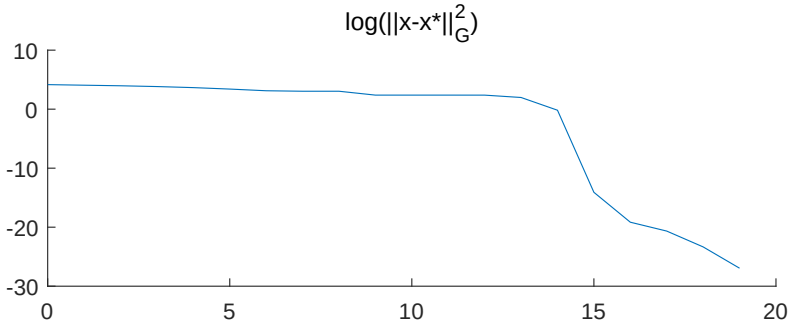
求解不同阶数 Hilbert 矩阵的迭代次数列成表格如下:

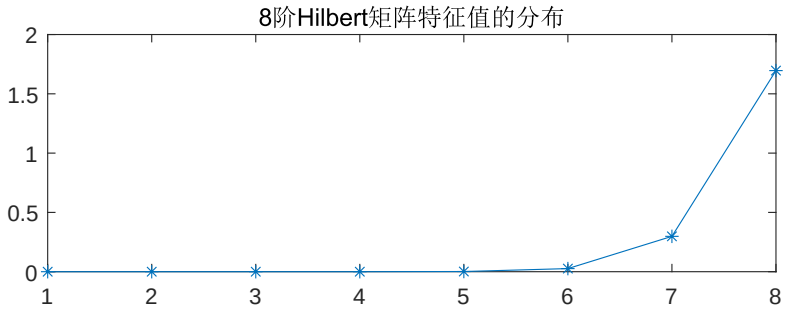
表 4: 迭代次数

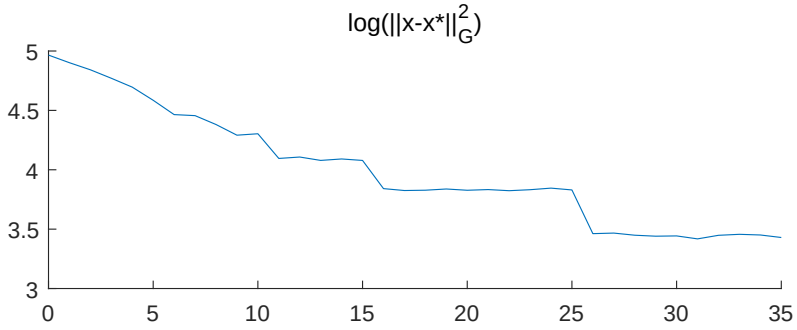
n	5	8	12	20
step	6	19	35	66

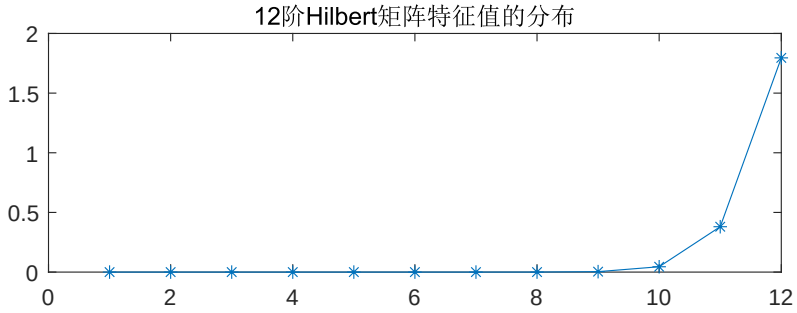

log(||x-x*||^2_G)
20
0
-20
-40
0
1
2
3
4
5
6

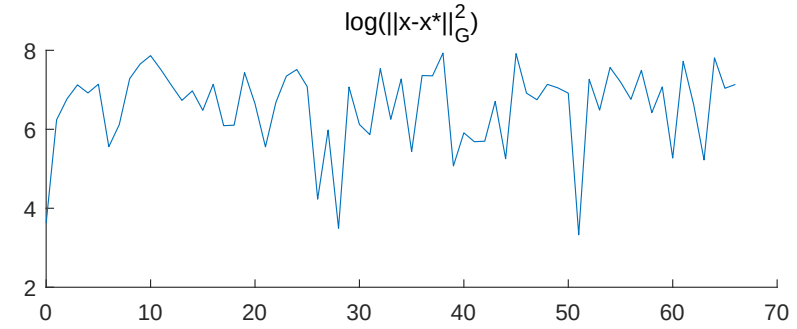

5阶Hilbert矩阵特征值的分布
2
1.5
1
0.5
0
1
1.5
2
2.5
3
3.5
4
4.5
5

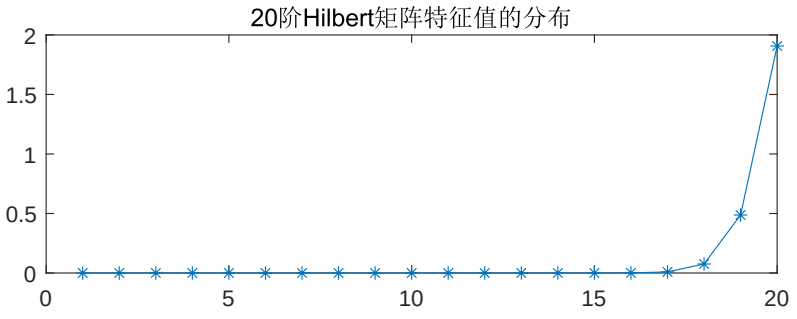

log(||x-x*||^2_G)
10
0
-10
-20
-30
0
5
10
15
20


8阶Hilbert矩阵特征值的分布
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8


log(||x-x*||$_G^2$)
5
4.5
4
3.5
3
0
5
10
15
20
25
30
35


12阶Hilbert矩阵特征值的分布
2
1.5
1
0.5
0
0
2
4
6
8
10
12


log(||x-x*||$_G^2$)
8
6
4
2
0
10
20
30
40
50
60
70


20阶Hilbert矩阵特征值的分布
2
1.5
1
0.5
0
0
5
10
15
20

5.4 总结分析

首先我们通过一个小程序考察一下 Hilbert 矩阵的病态性质:

```
N=20;
G=random('unif',0,1,[N N]);
G1=inv(G);
I=eye(N);
norm(G*G1-I)

%ans =
%
%  2.9348e-14
```

先生成一个 20×20 的随机矩阵，然后用 MATLAB 自带的求逆函数 inv() 求逆，然后将原矩阵与其逆相乘再减去单位阵，之后求矩阵范数，求得结果为 2.9348e-14，可见 MATLAB 自带的求逆函数在面对一般问题时还是比较精确的。

```
N=20;
H=hilb(N);
H_inv=inv(H);
I=eye(N);
norm(H*H_inv-I)

%ans =
%
%  33.8220
```

然后我们对 Hilbert 矩阵进行相同操作，结果为 33.8220，远远大于上面那个结果，足以见 Hilbert 矩阵的高度病态性质。

另外，我们由书本知识可以知道: 共轭梯度法与特征值的分布有关，当特征值分布较密集时，相同步数迭代精度更高。所以我们可以对 Hilbert 矩阵进行预处理，降低其条件数。

嗯，那如何对 Hilbert 矩阵进行预处理呢？我在网上经过不懈的搜索，终于在 Alfi Quarteroni Fausto Saleri 的 Scientifi Computing with MATLAB and Octave 中找到这么一个题目 (见图9)，题目中点出可以用 Hilbert 矩阵对角元素构建一个对角矩阵，作为预处理矩阵。

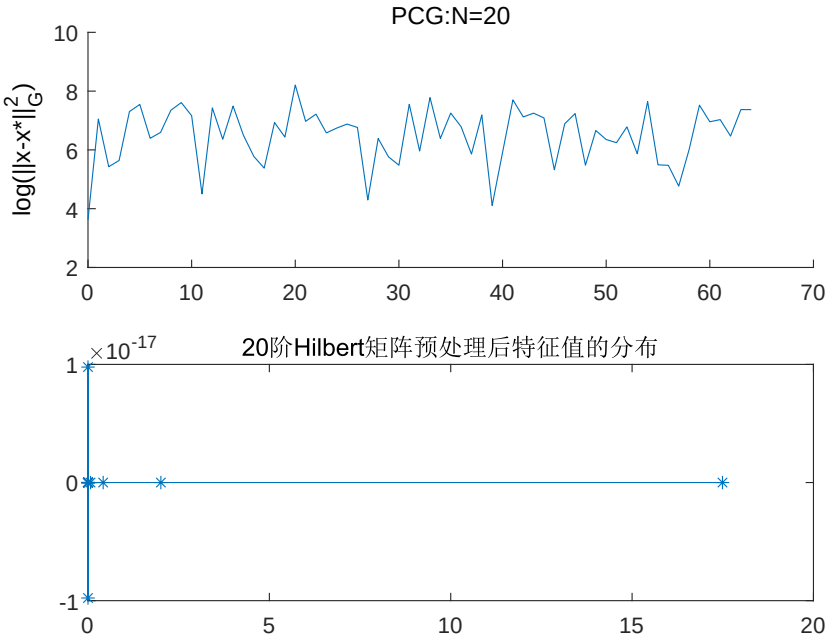
Example 5.16 Let us go back to Example 5.9 on the Hilbert matrix and solve the related system (for different values of n) by the preconditioned gradient (PG) and the preconditioned conjugate gradient (PCG) methods, using as preconditioner the diagonal matrix D made of the diagonal entries of the Hilbert matrix. We fix $\mathbf{x}^{(0)}$ to be the null vector and iterate until the relative residual (5.64) is less than 10^{-6}. In Table 5.4 we report the absolute errors (with respect to the exact solution) obtained with PG and PCG methods, as well as the errors obtained using the MATLAB command \. For the latter, the error degenerates when n gets large. On the other hand, we can appreciate the beneficial effect that a suitable iterative method such as the PCG scheme can have on the number of iterations. ■

图 9: 从书上摘的题

也就是说，设 C 是由 G 的对角线元素开方构成的对角矩阵，令 $\hat{G} = C^{-T}GC^{-1}$, $\hat{x} = Cx$,经过变换可调整为 $M^{-1}Gx = M^{-1}b$,其中 $M = C^TC$,即由 G 对角线元素构成的对角阵。

经过这么一番预处理后，不难看出不难看出该矩阵仍然是对称正定矩阵，而且其对角元素全为 1.

那么我就使用书上的预处理共轭梯度法首先对 $N = 20$ 的 Hilbert 矩阵进行检验：



可见这个预处理确实把特征值都聚集到一起了，但是迭代次数却为 65 次，仅仅比之前减少了一步，优化不太明显，这令我非常困惑，于是我改变终止精度进行测试，结果如下:

表 5: 不同精度下迭代次数的比较 ($N = 20$)

Error	10^{-6}	10^{-7}	10^{-8}
Steps of CG	66	121	520
Steps of PCG	65	115	314

可见在精度要求较低的情况下，两者的相差并不明显。

然后我试着将 N 调到 100，观察其迭代情况，发现在精度较低时，PCG 法所迭代的次数有时候甚至比 CG 法还要多，但在精度要求达 10^{-8} 时，CG 法无论如何迭代都无法满足精度要求，而 PCG 略胜一筹，精度高出了一个数量级，且迭代次数更少。

表 6: 不同精度下迭代次数的比较 ($N = 100$)

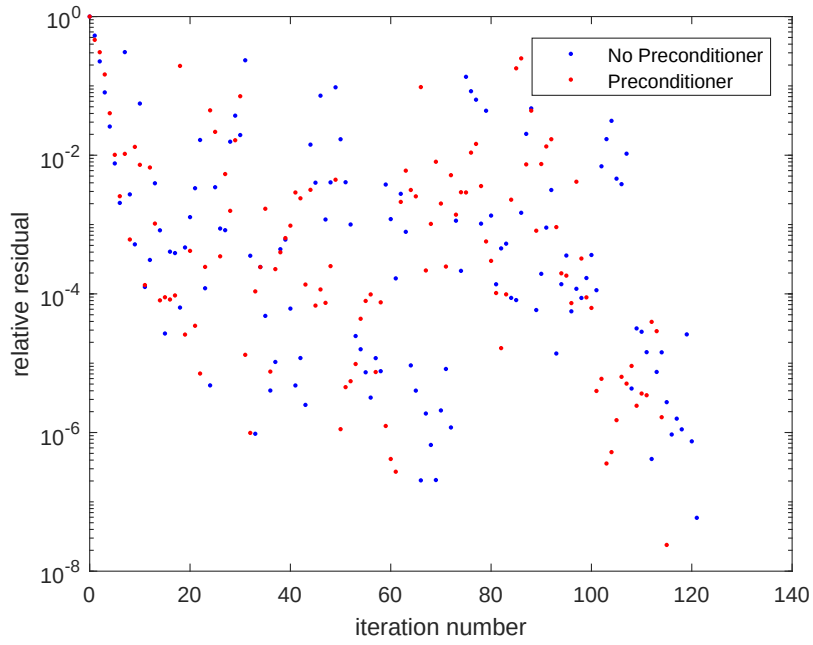
Error	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}
Steps of CG	12	25	47	175	748	2132(error=1.132e-08)
Steps of PCG	15	27	65	111	756	1686(error=9.683e-09)

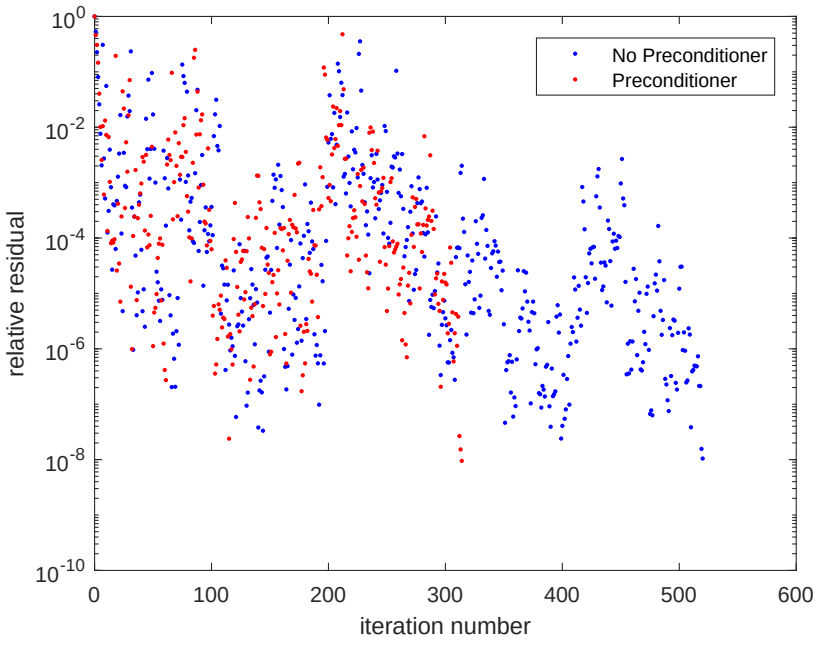
此外，该书上还列举了梯度下降法 (PG) 和预处理共轭梯度法 (PCG) 在求解该问题上的比较，我认为比较有借鉴意义，贴在下面：

Table 5.4. Errors obtained using the preconditioned gradient method (PG), the preconditioned conjugate gradient method (PCG), and the direct method implemented in the MATLAB command \ for the solution of the Hilbert system. For the iterative methods also the number of iterations is reported

		\	PG		PCG	
n	$K(\mathrm{A}_n)$	Error	Error	Iter	Error	Iter
4	1.55e+04	7.72e-13	8.72e-03	995	1.12e-02	3
6	1.50e+07	7.61e-10	3.60e-03	1813	3.88e-03	4
8	1.53e+10	6.38e-07	6.30e-03	1089	7.53e-03	4
10	1.60e+13	5.24e-04	7.98e-03	875	2.21e-03	5
12	1.70e+16	6.27e-01	5.09e-03	1355	3.26e-03	5
14	6.06e+17	4.12e+01	3.91e-03	1379	4.32e-03	5

$N = 20$ 时，有/无预处理的共轭梯度法迭代情况如下：[2]

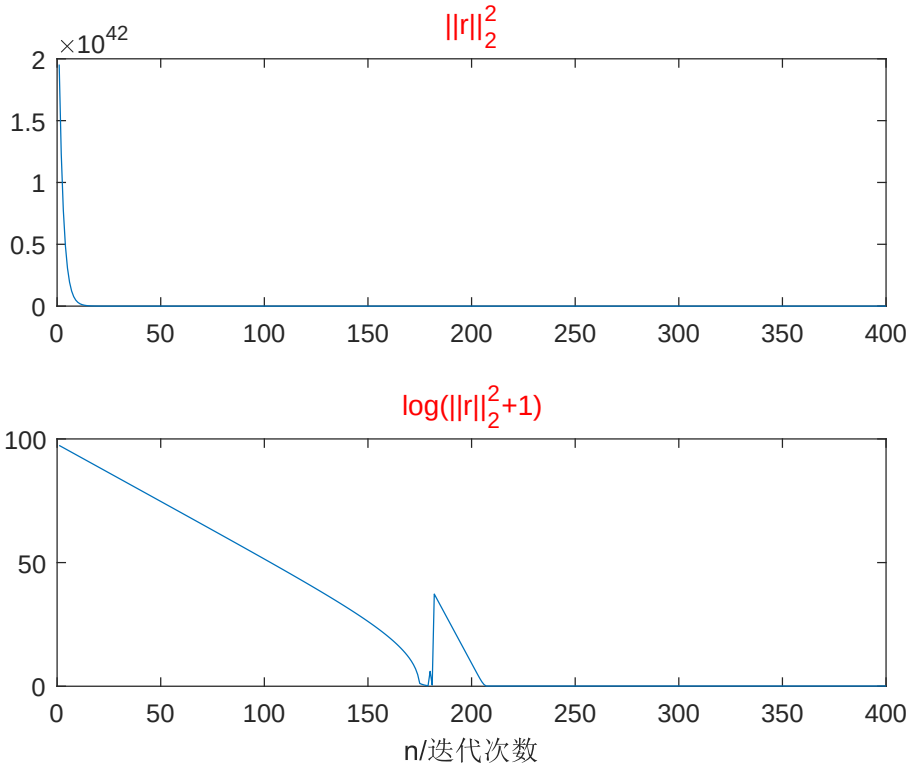




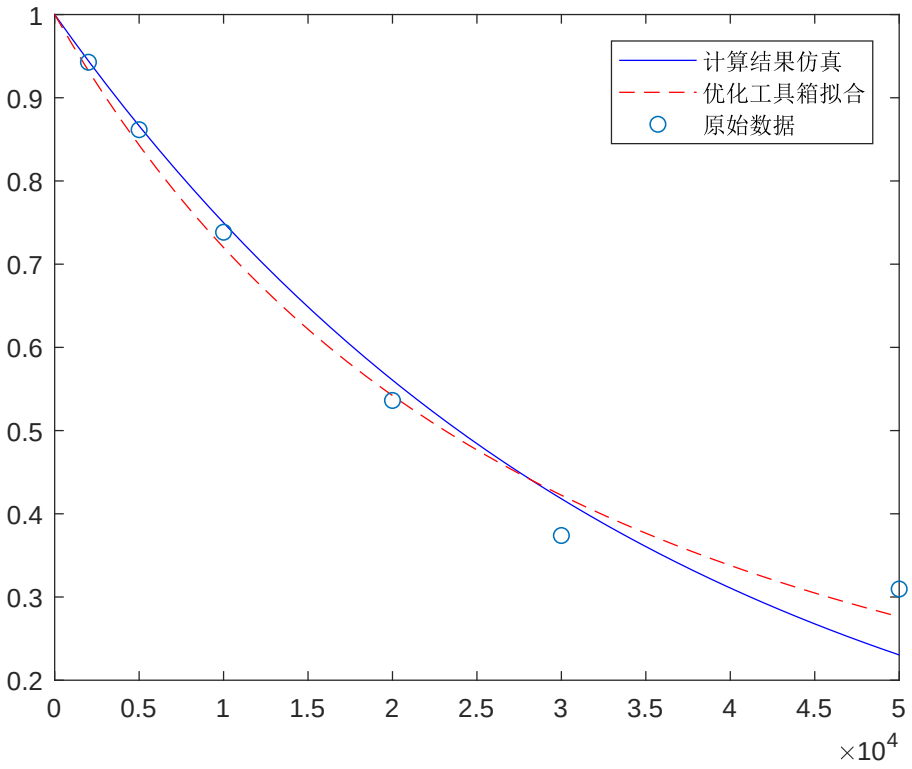
[2]第一个图的精度要求为 10^{-7}，第二个图的精度要求为 10^{-8}

6 Problem 5.27

此题采用线搜索确定步长时，得到的结果误差极大，因为 $\phi'(0)$ 在离稳定点较远时数量级高达 10^{40} 量级，导致线搜索得到的步长极小，几乎为 0，无法收敛, 经反复调整参数都没能取得好的结果，最后只好手动确定步长 $\alpha_k = 0.05$，此时效果良好，残量的 2-范数随迭代次数的下降情况如下：（由于数量级巨大，为了更好的显示残差的波动，将原图中将残差取对数处理后并排参考）



又采用 MATLAB 中优化工具箱中的 lsqnonlin 函数进行拟合，得到的结果比我的程序算出来的略好，将两者进行比较，比较结果如下：



6.1 计算结果展示

表 7: 结果比较

	程序计算	工具箱拟合
stv	0.125639950119876	0.104420208306470
x_1	3.323336983976929e-04	-0.009615612533368
x_2	3.516673367231929e+02	-19.446505801429495

6.2 算法伪代码

Algorithm 8 Gauss-Newton method for problem(5.27)

1: Given $\boldsymbol{x}^{(0)}$ and G
2: Set $\boldsymbol{p}^{(0)} = -\boldsymbol{g}^{(0)}, k = 0$
3: **while** $\|\boldsymbol{g}^{(k)}\| > \epsilon$ **do**
4: Set $\alpha_k = -\dfrac{{\boldsymbol{p}^{(k)}}^T \boldsymbol{g}^{(k)}}{{\boldsymbol{p}^{(k)}}^T \boldsymbol{G} \boldsymbol{p}^{(k)}}$
5: Set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{p}^{(k)}$
6: Set $\boldsymbol{g}^{(k+1)} = g(\boldsymbol{x}^{(k+1)})$
7: Set $\boldsymbol{p}^{(k)} = -\boldsymbol{g}^{(k)}$
8: k=k+1
9: **end while**