1 第一题

1. Prove that $\|Bx\| \geq \|x\|/\|B^{-1}\|$, for any non singular matrix B.

证明. $\|B^{-1}\| = \sup_{y\neq 0} \frac{\|B^{-1}y\|}{\|y\|} \quad \Rightarrow \quad \|B^{-1}y\| \leq \|B^{-1}\| \cdot \|y\|$

令 $y = Bx$, 则 $B^{-1}y = x$.

代入上式得 $\|x\| \leq \|B^{-1}\| \cdot \|Bx\|$.

即 $\|Bx\| \geq \|x\|/\|B^{-1}\|$.

□

2 第二题

(a) Verify that when $\bar{A}$ is nonsingular, we have

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

证明. 右乘 $(A + ab^T)$, 右边 $= A^{-1}(A + ab^T) - \frac{A^{-1}ab^TA^{-1}(A+ab^T)}{1+b^TA^{-1}a}$

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

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

$$= I$$

左边 = 右边得证。

□

(b) Using the above formula to show that

$$B_{k+1} = B_k + \frac{(y_k - B_ks_k)(y_k - B_ks_k)^T}{(y_k - B_ks_k)^Ts_k}$$

is the inverse of

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

where $H_k^{-1} = B_k$ is symmetric, $s_k = x_{k+1} - x_k$ and $y_{k+1} = \nabla f(x_{k+1}) - \nabla f(x_k)$.

证明. 令 $A = H_k$, $a = s_k - H_k y_k$, $b^T = \frac{(s_k - H_k y_k)^T}{(s_k - H_k y_k)^T y_k}$

$$
\begin{aligned}
H_{k+1}^{-1} &= (H_k + ab^T)^{-1} = B_{k+1} \\
&= (H_k)^{-1} - \frac{H_k^{-1} a b^T H_k^{-1}}{1 + b^T H_k^{-1} a} \\
&= B_k - \frac{B_k a b^T B_k}{1 + b^T B_k a} \\
&= B_k - \frac{B_k (s_k - H_k y_k) \cdot \frac{(s_k - H_k y_k)^T}{(s_k - H_k y_k)^T y_k} \cdot B_k}{1 + \frac{(s_k - H_k y_k)^T}{(s_k - H_k y_k)^T y_k} \cdot B_k (s_k - H_k y_k)} \\
&= B_k + \frac{(y_k - B_k s_k)(s_k - H_k y_k)^T B_k}{(s_k - H_k y_k)^T y_k + (s_k - H_k y_k)^T B_k (s_k - H_k y_k)} \\
&= B_k + \frac{(y_k - B_k s_k)(B_k s_k - B_k H_k y_k)^T}{(s_k - H_k y_k)^T (y_k + B_k s_k - B_k H_k y_k)} \\
&= B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k}.
\end{aligned}
$$

□

3 第三题

我用最速下降法和牛顿法分别求解了 Rosenbrock 函数和 Beale 函数的最优解，结果如下表所示，其中最速下降法步长的回溯线搜索我使用了 Armigo condition，即 $f(x + \alpha p) \leq f(x) + c\alpha \nabla f(x)^T p$。

表 1: 最速下降法和牛顿法求最优解

Function	Method	Iterations	x^*	$f(x^*)$	$\|\nabla f(x^*)\|$
Rosenbrock	Steepest Descent	100 (max)	$[0.934384, 0.872610]$	0.004327	1.02×10^{-1}
	Newton	7	$[1.000000, 1.000000]$	0.000000	8.29×10^{-9}
Beale	Steepest Descent	100 (max)	$[-1.596594, 1.428288]$	1.198996	5.10×10^{-1}
	Newton	2	$[0.000000, 1.000000]$	14.203125	0.00×10^{0}

从表 1可以看出，对于 Rosenbrock 函数，最速下降法和牛顿法都找到了较好的（近似）最优解，但牛顿法只用了 7 次迭代就找到了精确的最优解，而最速下降法在 100 次迭代后仍未收敛。

对于 Beale 函数，最速下降法似乎在一开始就找错了方向导致无法找到正确的最优解 (如图 3)，而牛顿法由于 Hessian 矩阵是负定的无法收敛，在 2 次迭代后一直卡在点 $(0, 1)$。

下面是迭代路径图：

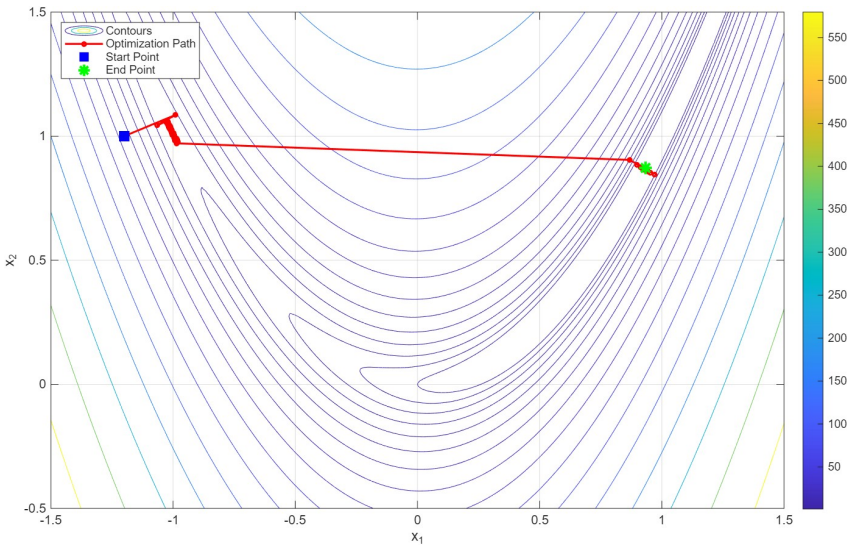

图 1: Rosenbrock(steepest descent)

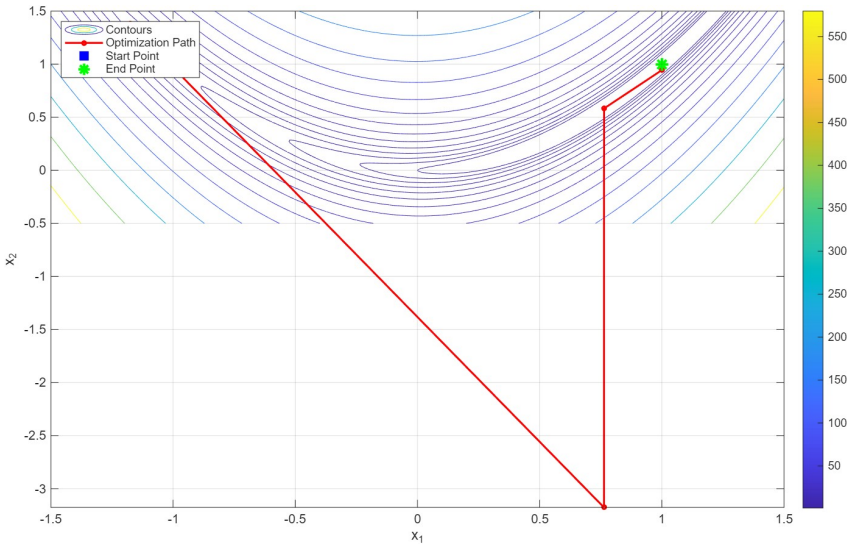

图 2: Rosenbrock(newton)

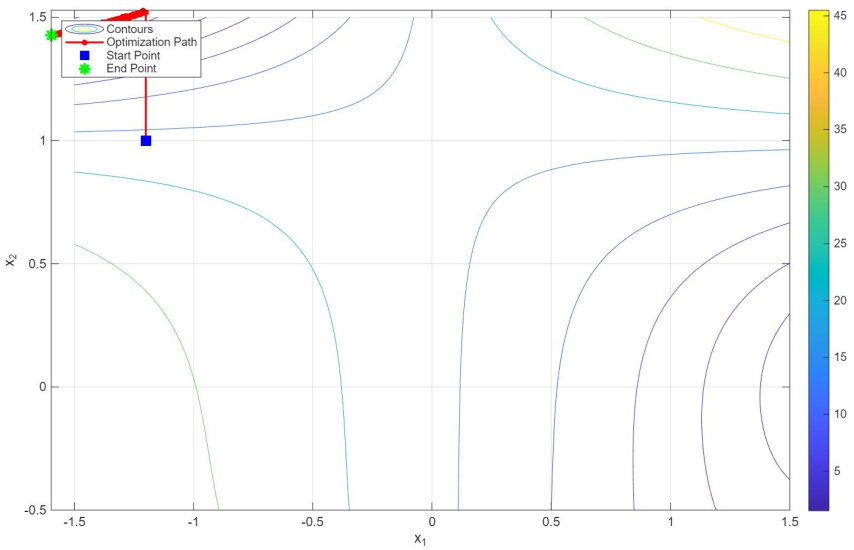

图 3: Beale(steepest descent)

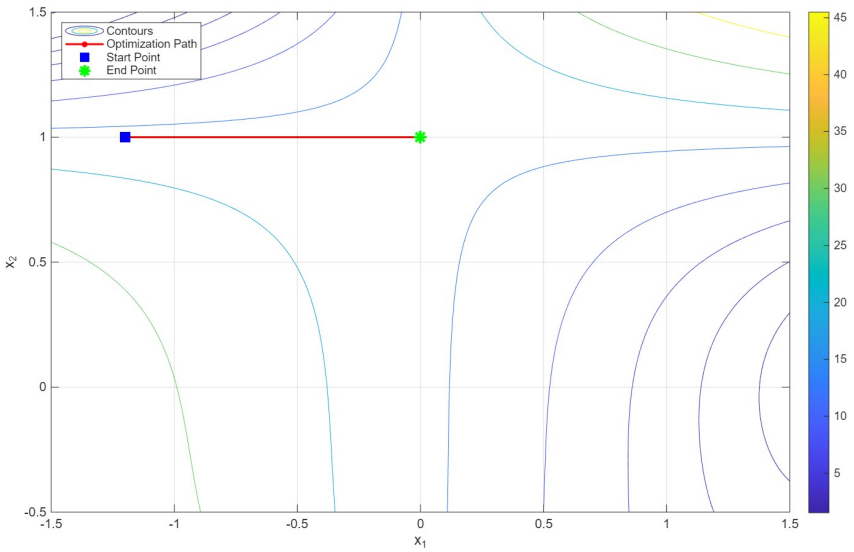

图 4: Beale(newton)

4 第四题

易知全局最优是 $(0,0,0,0)^T$.

根据牛顿法收敛定理，$f(x)$ 二次连续可微，x^* 为极小点且满足 $\nabla f(x^*)=0$。若 Hessian 矩阵 $\nabla^2 f(x^*)$ 正定，且在 x^* 的邻域内满足 Lipschitz 条件：

$$|G_{ij}(x)-G_{ij}(y)| \le L\|x-y\|, \quad \forall i,j, \tag{1}$$

则对充分靠近 x^* 的初始点 x_0，牛顿法产生的序列 $\{x_k\}$ 收敛到 x^*，且具有二次收敛速度：

$$\lim_{k\to\infty} \frac{\|x_{k+1}-x^*\|}{\|x_k-x^*\|^2} < \infty. \tag{2}$$

而在精确一维搜索条件下，带步长因子的牛顿法产生的迭代序列 $\{x_k\}$ 满足：

- 若 $\{x_k\}$ 为有限点列，则对某个 k，有 $\nabla f(x_k)=0$（算法在某一步停止）；

- 若 $\{x_k\}$ 为无穷点列，则 $\{x_k\}$ 收敛到 f 的唯一极小值点（全局收敛）。

我分别用普通的牛顿法和带步长因子的牛顿法求解了该问题，其中步长的回溯线搜索仍然使用了 Armigo condition。在不同的 σ 和初始点下，结果如下表所示：

表 2: 牛顿法和带步长因子的牛顿法

σ	Initial Point	Method	Iterations	x^*	$f(x^*)$
1.0×10^0	x0=(cos70,sin70,..)	Pure Newton	9	$[0.000000, 0.000000, 0.000000, 0.000000]$	7.694412×10^{-15}
1.0×10^0	x0=(cos70,sin70,..)	Newton with Line Search	24	$[0.000000, 0.000000, 0.000000, 0.000000]$	1.821270×10^{-13}
1.0×10^0	x0=(cos50,sin50,..)	Pure Newton	10	$[0.000000, 0.000000, 0.000000, 0.000000]$	1.667507×10^{-30}
1.0×10^0	x0=(cos50,sin50,..)	Newton with Line Search	24	$[0.000001, 0.000001, 0.000000, 0.000000]$	3.663454×10^{-13}
1.0×10^4	x0=(cos70,sin70,..)	Pure Newton	20	$[0.000000, 0.000000, 0.000000, 0.000000]$	4.277472×10^{-15}
1.0×10^4	x0=(cos70,sin70,..)	Newton with Line Search	29	$[0.000000, 0.000000, 0.000000, 0.000000]$	2.077721×10^{-13}
1.0×10^4	x0=(cos50,sin50,..)	Pure Newton	20	$[0.000001, 0.000000, 0.000000, 0.000000]$	3.686010×10^{-13}
1.0×10^4	x0=(cos50,sin50,..)	Newton with Line Search	28	$[0.000001, 0.000000, 0.000000, 0.000000]$	3.187173×10^{-13}

从表 2中可以看出，对于不同的初始点和 σ，两种方法都能收敛到全局最优解 $(0,0,0,0)^T$，且带步长因子的牛顿法在迭代次数上要远多于普通的牛顿法。这是因为当初始步长不满足回溯线搜索的 Armigo 条件时，步长会缩小，从而产生了更多的迭代次数。

初始点对收敛速度没有产生太大的影响，但 σ 的增大使迭代次数显著增加。这是因为，随着 σ 的增大，在远离最优解的区域，牛顿法依赖的局部二次函数近似变得不准确。

在这道题中我尝试了 $f(x+\alpha p) \le f(x) + c\alpha \nabla f(x)^T p$ 中不同的 c 的取值，当 c 取 0.5 时，初始步长取为 1 一直满足 Armigo 条件，导致步长没有变化，迭代过程与普通牛顿法完全相同；当 c 取 0.6 时，首次步长违背 Armigo 条件的迭代次数如下表所示：

表 3: 首次步长减小的迭代次数

$\boldsymbol{\sigma}$	**初始点**	**首次步长减小的迭代次数**
1.0×10^0	$(\cos 70°, \sin 70°, \ldots)$	6
1.0×10^0	$(\cos 50°, \sin 50°, \ldots)$	6
1.0×10^4	$(\cos 70°, \sin 70°, \ldots)$	17
1.0×10^4	$(\cos 50°, \sin 50°, \ldots)$	18

为了更好地比较收敛速度，我分别计算了最后几次迭代时 $R_l = \frac{\|x_{k+1}\|}{\|x_k\|}$ 和 $R_q = \frac{\|x_{k+1}\|}{\|x_k\|^2}$ 的值，如下表所示：

表 4: 收敛速度

σ	Initial Point	Method	Iter. (k)	$\|x_k\|$	线性率 $R_l = \frac{\|x_{k+1}\|}{\|x_k\|}$	二次率 $R_q = \frac{\|x_{k+1}\|}{\|x_k\|^2}$
1.0	$x_1^{(0)}$	Pure Newton	5	9.62×10^{-2}	0.2902	3.02
			6	2.79×10^{-2}	0.0448	1.60
			7	1.25×10^{-3}	10^{-4}	0.08
		Newton with Line Search	21	2.41×10^{-6}	0.5000	2.07×10^5
			22	1.21×10^{-6}	0.5000	4.14×10^5
	$x_2^{(0)}$	Pure Newton	6	4.07×10^{-2}	0.0892	2.19
			7	3.63×10^{-3}	0.0008	0.23
			8	3.05×10^{-6}	≈ 0	0.00
		Newton with Line Search	21	3.42×10^{-6}	0.5000	1.46×10^5
			22	1.71×10^{-6}	0.5000	2.92×10^5
10^4	$x_1^{(0)}$	Pure Newton	16	1.27×10^{-3}	0.3718	292.95
			17	4.72×10^{-4}	0.1118	236.95
			18	5.28×10^{-5}	0.0018	33.23
		Newton with Line Search	26	2.58×10^{-6}	0.5000	1.94×10^5
			27	1.29×10^{-6}	0.5000	3.88×10^5
	$x_2^{(0)}$	Pure Newton	16	1.47×10^{-3}	0.4265	290.14
			17	6.27×10^{-4}	0.1769	282.17
			18	1.11×10^{-4}	0.0077	69.74
		Newton with Line Search	25	3.19×10^{-6}	0.5000	1.57×10^5
			26	1.60×10^{-6}	0.5000	3.13×10^5

注: $x_1^{(0)} = (\cos 70^\circ, \sin 70^\circ, \cos 70^\circ, \sin 70^\circ)^T$, $x_2^{(0)} = (\cos 50^\circ, \sin 50^\circ, \cos 50^\circ, \sin 50^\circ)^T$。$R_l$ 和 R_q 分别代表线性和二次收敛率的估计值。

从表 4中可以看出，纯牛顿法具有二次收敛的性质，当趋于最优解时，$R_q = \frac{\|x_{k+1}\|}{\|x_k\|^2}$ 趋于一个有限的常数满足二次收敛的定义 $\lim_{k\to\infty} \frac{\|x_{k+1}-x^*\|}{\|x_k-x^*\|^2} < \infty$. 当 σ 变大时，二次收敛常数 R_q 也显著增大到 200-300 之间，与前面的分析相符合。

带线搜索的牛顿法在此问题上并未表现出二次收敛，而是退化为线性收敛，$R_q = \frac{\|x_{k+1}\|}{\|x_k\|^2}$ 趋于 ∞，而 $R_l = \frac{\|x_{k+1}\|}{\|x_k\|}$ 稳定在 0.5 附近。这是因为前面所提到的 Armigo 条件的 c 取得太严格，导致最后几步迭代的步长一直被缩小到 0.5。

5 第五题

Barzilai-Borwein 法的公式如下，令：

$$\mathbf{s}_k = \mathbf{x}_k - \mathbf{x}_{k-1}, \tag{3}$$

$$\mathbf{y}_k = \nabla f(\mathbf{x}_k) - \nabla f(\mathbf{x}_{k-1}), \tag{4}$$

则最速下降法的步长可以由下式计算：

$$\alpha_k = \frac{\mathbf{s}_k^T \mathbf{s}_k}{\mathbf{s}_k^T \mathbf{y}_k} = \frac{\|\mathbf{s}_k\|^2}{\mathbf{s}_k^T \mathbf{y}_k}. \tag{5}$$

对于不同的 μ，我分别用最速下降法和 Barzilai-Borwein 法求解了 Lasso 问题

$$\min_x \ f(x) = \frac{1}{2}\|Ax - b\|^2 + \mu L_\delta(x). \tag{6}$$

结果如下表所示：

表 5: 最速下降法和 Barzilai-Borwein 法求解 Lasso 问题

μ	Method	Iterations	$f(x^*)$	Error	Time (s)
1.0e-2	Steepest Descent	2000	1.471862	0.6952	0.9729
	Barzilai-Borwein	2000	1.311111	0.6252	0.1604
1.0e-3	Steepest Descent	2000	0.151870	0.7125	0.4076
	Barzilai-Borwein	2000	0.144063	0.6829	0.0875

其中误差由下式计算:

$$\mathrm{Err}_{L1} = \frac{\|x - x_{\mathrm{true}}\|_1}{\|x_{\mathrm{true}}\|_1}, \tag{7}$$

从表 5可以看出，Barzilai-Borwein 法在相同的迭代次数下，计算时间明显少于最速下降法，并且在目标函数值和相对误差方面也表现得更好一些。当然，两种方法的相对误差都比较大，超过了 0.6，说明在求解 Lasso 问题时，这两种方法的效果都不是特别理想。

下面是真实解与两种方法求得的解的对比图:

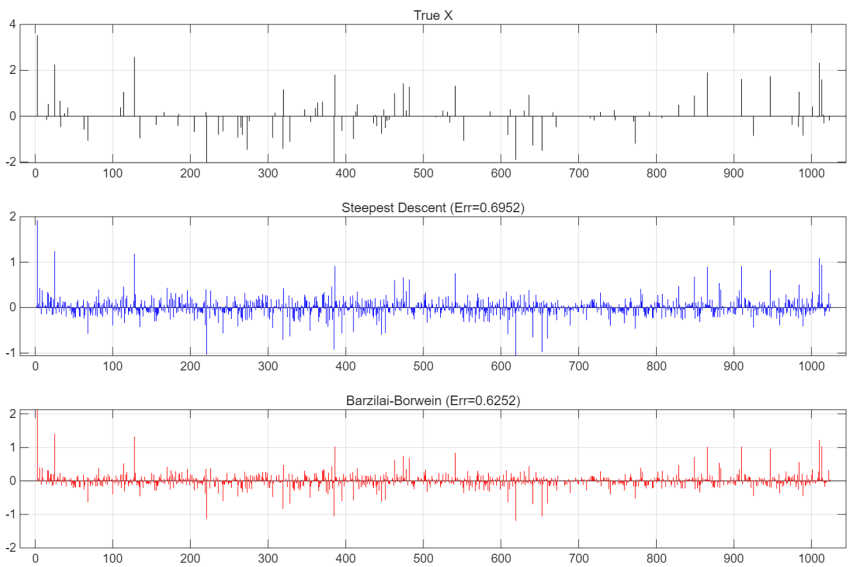

图 5: $\mu = 1.0e-02$

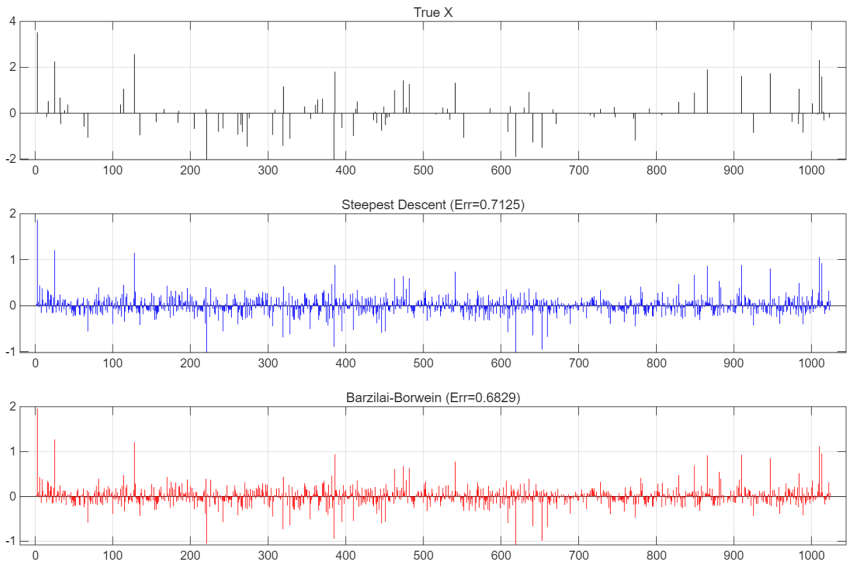

图 6: $\mu = 1.0e-03$

```
function [x, val, history] = steepest_descent(func, grad, x0,parameters)
    % STEEPEST_DESCENT
    %   Inputs:
    %       func         - 目标函数
    %       grad         - 目标函数的梯度
    %       x0           - 初始点
    %       parameters   - 参数
    %   Outputs:
    %       x         - 最优解
    %       val       - 最小值
    %       history   - 迭代历史

        % 设置默认参数
        if ~isfield(parameters, 'tol'), parameters.tol = 1e-6; end
        if ~isfield(parameters, 'max_iter'), parameters.max_iter = 1000; end

        x = x0;
        history.x_path = {x};
        history.f_path = {func(x)};

        for i = 1:parameters.max_iter
            g = grad(x);
            val = func(x);

            if norm(g) < parameters.tol
                fprintf('\nConverged successfully.\n');
                break;
            end

            % 回溯线搜索(满足Armigo condition)
            alpha = 1;
            beta = 0.5;
            c = 1e-4;
```

```
            p = -g;
            while func(x + alpha * p) > val + c * alpha * (g' * p)  % Armijo condition
                alpha = beta * alpha;
            end

            % 更新参数
            x = x + alpha * p;
            history.x_path{end+1} = x;
            history.f_path{end+1} = func(x);
        end

        if i == parameters.max_iter
            fprintf('\nReached max iterations without convergence.\n');
        end

        val = func(x);
        history.iterations = i;
    end
```

Listing 1: 最速下降法

```
function [x,val, history] = newton(func, grad, hessian, x0, parameters)
    % NEWTON法
    %   Inputs:
    %       func         - 目标函数
    %       grad         - 目标函数的梯度
    %       hessian      - 目标函数的Hessian矩阵
    %       x0           - 初始点
    %       parameters   - 参数
    %   Outputs:
    %       x         - 最优解
    %       val       - 最小值
    %       history   - 迭代历史

        % 设置默认参数
        if ~isfield(parameters, 'tol'), parameters.tol = 1e-6; end
        if ~isfield(parameters, 'max_iter'), parameters.max_iter = 100; end

        x = x0;
        % history.x_path = {x};
        n = length(x0);
        history.x_path = zeros(n, parameters.max_iter + 1);
        history.x_path(:, 1) = x;
        history.f_path = {func(x)};

        for i = 1:parameters.max_iter
            g = grad(x);
            H = hessian(x);

            if norm(g) < parameters.tol
                fprintf('\nConverged successfully.\n');
                break;
```

```
            end

            % 求解步长 p: H*p = -g
            p = -H \ g;

            x = x + p;
            % history.x_path{end+1} = x;
            history.x_path(:, i+1) = x;
            history.f_path{end+1} = func(x);
        end

        if i == parameters.max_iter
            fprintf('\nReached max iterations without convergence.\n');
        end

        val = func(x);
        history.iterations = i;
        history.x_path = history.x_path(:, 1:i+1);
    end
```

Listing 2: 牛顿法

```
function [x, val, history] = newton_line_search(func, grad, hessian, x0, parameters)

        if ~isfield(parameters, 'tol'), parameters.tol = 1e-6; end
        if ~isfield(parameters, 'max_iter'), parameters.max_iter = 100; end

        x = x0;
        n = length(x0);
        % history.x_path = {x};
        history.x_path = zeros(n, parameters.max_iter + 1);
        history.x_path(:, 1) = x;
        history.f_path = {func(x)};
        history.alphas = [];

        for i = 1:parameters.max_iter
            g = grad(x);
            H = hessian(x);
            f_val = func(x);

            if norm(g) < parameters.tol
                fprintf('\nConverged successfully.\n');
                break;
            end

            % 求解步长 p
            p = -H \ g;

            % 回溯线搜索 (满足Armijo Condition)
            alpha = 1;
            % alpha = alpha_init;      % 初始步长
            beta = 0.5;
```

```
            c = 0.6;
            while func(x + alpha * p) > f_val + c * alpha * (g' * p) % Armijo Condition
                alpha = beta * alpha;
            end
            % if(alpha ~= 1)
            %     fprintf('alpha reduced to %f at iteration %d\n', alpha, i);
            % end
            history.alphas(end+1) = alpha;

            % 更新
            x = x + alpha * p;
            history.x_path(:, i+1) = x;
            % history.x_path{end+1} = x;
            history.f_path{end+1} = func(x);
        end

        if i == parameters.max_iter
            fprintf('\nReached max iterations without convergence.\n');
        end

        val = func(x);
        history.iterations = i;
        history.x_path = history.x_path(:, 1:i+1);
    end
```

Listing 3: 带步长因子的牛顿法

```
function [x, val, history] = Barzilai_Borwein(func, grad, x0,parameters)
    % STEEPEST_DESCENT
    %   Inputs:
    %       func         - 目标函数
    %       grad         - 目标函数的梯度
    %       x0           - 初始点
    %       parameters   - 参数
    %   Outputs:
    %       x         - 最优解
    %       val       - 最小值
    %       history   - 迭代历史

        % 设置默认参数
        if ~isfield(parameters, 'tol'), parameters.tol = 1e-6; end
        if ~isfield(parameters, 'max_iter'), parameters.max_iter = 1000; end

        x = x0;
        history.x_path = {x};
        history.f_path = {func(x)};
        history.alphas = [];
        x_prev = x;
        g_prev = grad(x);
        for i = 1:parameters.max_iter
            g = grad(x);

```

```
            p = -g;
            if norm(g) < parameters.tol
                fprintf('\nConverged successfully.\n');
                break;
            end

            if(i == 1)
                % 回溯线搜索(满足Armigo condition)
                alpha = 1;
                beta = 0.5;
                c = 1e-4;
                val = func(x);
                while func(x + alpha * p) > val + c * alpha * (g' * p)  % Armijo condition
                    alpha = beta * alpha;
                end
            else
                s = x - x_prev;
                y = g - g_prev;
                alpha = (s' * s) / (s' * y);
            end

            x_prev = x;
            g_prev = g;

            % 更新参数
            x = x + alpha * p;
            history.x_path{end+1} = x;
            history.f_path{end+1} = func(x);
            history.alphas(end+1) = alpha;
        end

        if i == parameters.max_iter
            fprintf('\nReached max iterations without convergence.\n');
        end

        val = func(x);
        history.iterations = i;
    end
```

Listing 4: Barzilai-Borwein 法