数值优化算法学习报告

林立康　数学　25210180078

2025 年 12 月 24 日

摘要

...(此处填写摘要内容)

目录

1 第一题

1.1 问题描述

本题要求随机生成 Stiefel 流形的一个二次函数, 编程实现讲义中的 **Algorithm 4.3**(梯度下降法) 与 **Algorithm 4.4**(结合 BB 步长的梯度法) 极小化这个二次函数, 探索非单调的作用, 或者交替使用 BB 步长两个公式的作用. 具体问题为:

$$\min_{X \in St(n,p)} f(X) = \mathrm{tr}(X^T A X) + 2\mathrm{tr}(X^T B) \tag{1}$$

其中 $St(n,p) = \{X \in \mathbb{R}^{n\times p} : X^T X = I_p\}$, $A \in \mathbb{R}^{n\times n}$ 为对称矩阵, $B \in \mathbb{R}^{n\times p}$.

1.2 理论总结

根据讲义内容, 本题涉及的黎曼几何基础与优化理论如下:

1.2.1 黎曼梯度

Stiefel 流形 $St(n,p)$ 是欧氏空间 $\mathbb{R}^{n\times p}$ 的嵌入子流形. 对于定义在 $\mathbb{R}^{n\times p}$ 上的光滑函数 f, 其在 $X \in St(n,p)$ 处的黎曼梯度 $\mathrm{grad} f(X)$ 定义为欧氏梯度 $\nabla f(X)$ 在切空间 $T_X St(n,p)$ 上的正交投影. 根据讲义 **例 2.3.8**, 黎曼梯度的显式表达式为:

$$\mathrm{grad} f(X) = \nabla f(X) - X\mathrm{Sym}(X^T \nabla f(X)) \tag{2}$$

其中 $\mathrm{Sym}(Z) = (Z + Z^T)/2$. 对于本题的二次函数, 欧氏梯度为 $\nabla f(X) = 2(AX + B)$.

1.2.2 收缩映射

为了保证迭代点保持在流形上, 算法利用收缩映射 R 将切空间中的切向量映射回流形. 根据讲义 **例 4.1.15**, 本实验采用基于 SVD 的收缩映射 (也等价于基于极分解的收缩映射), 其具有良好的理论性质:

$$R_X(\xi) = UV^T, \quad \text{其中} X + \xi = U\Sigma V^T \tag{3}$$

此映射将切向量 $\xi \in T_X St(n,p)$ 映射为 $X + \xi$ 的极因子.

1.3 算法描述

1.3.1 算法 4.3: 梯度下降法与线搜索

本节描述黎曼流形上的梯度下降算法。该算法是欧氏空间最速下降法在流形上的自然推广，其核心思想是沿负黎曼梯度方向在切空间中寻找下降方向，并通过收缩映射将切向量映射回流形。

算法的第 k 步迭代格式如下：

$$x_{k+1} = R_{x_k}(-\alpha_k \mathrm{grad} f(x_k)) \tag{4}$$

其中，$\mathrm{grad} f(x_k) \in T_{x_k}\mathcal{M}$ 为目标函数在 x_k 处的黎曼梯度，R 为收缩映射（本实验中采用基于 SVD 的极分解收缩映射），$\alpha_k > 0$ 为步长。

为了保证算法的收敛性并获得良好的数值表现，步长 α_k 的选取至关重要。本实验实现了教材中所述的三种线搜索策略，分别对应单调与非单调的下降准则。

1.3.2 线搜索策略

设 $v_k = -\mathrm{grad} f(x_k)$ 为搜索方向，$\rho \in (0,1)$ 为回退因子，$c_1 \in (0,1)$ 为充分下降参数。

- **单调线搜索 (Armijo 条件)**
  该策略要求目标函数值在每一步都严格单调下降。步长 α_k 需满足如下充分下降条件：
  $$f(R_{x_k}(\alpha_k v_k)) \le f(x_k) + c_1 \alpha_k \langle \mathrm{grad} f(x_k), v_k \rangle \tag{5}$$
  在实际计算中，通常采用回退法：从初始步长开始，若不满足条件则令 $\alpha \leftarrow \rho\alpha$，直至满足为止。

- **Grippo 非单调线搜索**
  为了避免算法陷入局部极小值并允许函数值在迭代初期出现短暂上升，Grippo 等人提出了基于历史最大值的非单调准则。根据教材 **定义 4.2.2**，步长需满足：
  $$f(R_{x_k}(\alpha_k v_k)) \le \max_{0 \le j \le \min\{k, M\}} f(x_{k-j}) + c_1 \alpha_k \langle \mathrm{grad} f(x_k), v_k \rangle \tag{6}$$
  其中 $M \ge 0$ 为历史窗口大小。当 $M = 0$ 时，该策略退化为标准的 Armijo 单调线搜索。

- **Zhang-Hager 非单调线搜索**
  该策略利用历史函数值的加权平均来以此放宽下降条件，通常比 Grippo 策略表现更为平稳。根据教材 **定义 4.2.3**，步长需满足：
  $$f(R_{x_k}(\alpha_k v_k)) \le C_k + c_1 \alpha_k \langle \mathrm{grad} f(x_k), v_k \rangle \tag{7}$$
  其中参考值 C_k 按照如下递归方式更新：
  $$\begin{cases} Q_{k+1} = \eta_k Q_k + 1, \quad Q_0 = 1 \\ C_{k+1} = \dfrac{\eta_k Q_k C_k + f(x_{k+1})}{Q_{k+1}}, \quad C_0 = f(x_0) \end{cases} \tag{8}$$
  这里 $\eta_k \in [\eta_{\min}, \eta_{\max}] \subset (0,1)$ 为控制参数。

1.3.3 算法流程

基于上述理论，黎曼梯度下降法的完整流程如算法1所示。

Algorithm 1 黎曼梯度下降法

Require: 初始点 $x_0 \in \mathrm{St}(n,p)$，收缩映射 R，参数 $\rho, c_1 \in (0,1)$，精度 ϵ

$k \leftarrow 0$
while $\|\mathrm{grad} f(x_k)\| > \epsilon$ **do**
  计算黎曼梯度：$\xi_k = \mathrm{grad} f(x_k)$
  设置初始尝试步长 α (例如 $\alpha = 1$ 或上一轮步长)
  Line Search:
  while 不满足选定的线搜索条件 (Armijo / Grippo / Zhang-Hager) **do**
    $\alpha \leftarrow \rho\alpha$
  end while
  更新迭代点：$x_{k+1} = R_{x_k}(-\alpha\xi_k)$
  更新历史信息 (如 C_{k+1}, Q_{k+1} 或存储 $f(x_{k+1})$ 到历史窗口)
  $k \leftarrow k+1$
end while
输出：最优解 x^*

1.3.4 算法 4.4: BB 步长方法

为了加速收敛, 引入 Barzilai-Borwein (BB) 步长. 根据讲义 **§4.4**, 严格的黎曼 BB 方法需要利用向量传输算子 $\mathcal{T}$ 将 g_{k-1} 传输到 x_k 所在的切空间. 然而, 由于 $St(n,p)$ 是欧氏空间的嵌入子流形, 根据讲义 **公式 (5.2.118)**, 我们可以采用如下欧氏差分来近似, 从而避免高昂的传输计算成本:

$$s_{k-1} = x_k - x_{k-1}, \quad y_{k-1} = \mathrm{grad} f(x_k) - \mathrm{grad} f(x_{k-1}) \tag{9}$$

BB 步长 α_k 的计算公式为:

$$\alpha_k^{\mathrm{BB1}} = \frac{\langle s_{k-1}, s_{k-1}\rangle}{|\langle s_{k-1}, y_{k-1}\rangle|}, \quad \alpha_k^{\mathrm{BB2}} = \frac{|\langle s_{k-1}, y_{k-1}\rangle|}{\langle y_{k-1}, y_{k-1}\rangle} \tag{10}$$

算法在获得初始步长 α_k 后, 仍结合非单调线搜索以保证全局收敛性. 通常采用交替使用 BB1 和 BB2 的策略或自适应选取策略.

结合非单调技术与近似向量传输的 BB 算法流程详见算法 2.

Algorithm 2 带非单调线搜索的 BB 算法

Require: 初始点 X_0, 历史窗口大小 $M \geq 0$, 步长截断范围 $[\alpha_{\min}, \alpha_{\max}]$

$k \leftarrow 0$, 初始 BB 步长 $\alpha_0 = 1$
while $\|\mathrm{grad} f(X_k)\| > \epsilon$ **do**
    计算黎曼梯度:$\xi_k = \mathrm{grad} f(X_k)$
    非单调线搜索 (代码中使用了 Zhang-Hager 方法):
    获取历史最大值 $f_{\mathrm{ref}} = \max_{0 \leq j \leq \min(k,M)} f(X_{k-j})$
    令尝试步长 $\alpha = \alpha_k$
    while $f(R_{X_k}(-\alpha \xi_k)) > f_{\mathrm{ref}} - c_1 \alpha \|\xi_k\|^2$ **do**
        $\alpha \leftarrow \rho \alpha$
    end while
    更新迭代点:$X_{k+1} = R_{X_k}(-\alpha \xi_k)$
    计算下一轮 BB 步长 (近似策略):
    $S_k = X_{k+1} - X_k$ ▷ 利用欧氏差分近似
    $Y_k = \mathrm{grad} f(X_{k+1}) - \xi_k$
    if k is odd **then**
        $\alpha_{\mathrm{temp}} = \frac{\langle S_k, S_k \rangle}{|\langle S_k, Y_k \rangle|}$ ▷ BB1 步长
    else
        $\alpha_{\mathrm{temp}} = \frac{|\langle S_k, Y_k \rangle|}{\langle Y_k, Y_k \rangle}$ ▷ BB2 步长
    end if
    $\alpha_{k+1} = \min(\alpha_{\max}, \max(\alpha_{\min}, \alpha_{\mathrm{temp}}))$ ▷ 截断保护
    $k \leftarrow k + 1$
end while

1.4 测试与结果

(此处预留填写具体的数值实验结果, 包括: 目标函数值随时间/迭代次数的下降曲线对比图, 以及关于单调/非单调、BB1/BB2 步长效果的表格分析.)

2 第二题

2.1 凸二次规划的积极集算法总结

本节总结求解凸二次规划问题的经典算法——积极集算法. 该算法对应于 Nocedal & Wright 教材中的 Algorithm 16.3.

2.1.1 问题定义

我们要解决的凸二次规划问题标准形式如下:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & q(x)=\frac{1}{2}x^TGx+x^Tc \\ \text{s.t.} \quad & a_i^Tx=b_i, \quad i\in\mathcal{E} \quad \text{(等式约束)} \\ & a_i^Tx\geq b_i, \quad i\in\mathcal{I} \quad \text{(不等式约束)} \end{aligned} \tag{11}$$

其中矩阵 G 是半正定的, 这是保证算法收敛到全局最优解的关键条件.

2.1.2 工作集的定义

算法的核心思想是通过迭代, 动态调整 "当前起作用的约束集合". 在第 k 次迭代中, 我们定义一个**工作集** $\mathcal{W}_k$: 其一, $\mathcal{W}_k$ 是当前所有约束的一个子集. 其二, 它包含所有等式约束 $\mathcal{E}$, 以及部分在当前点 x_k 处有效 (即取等号) 的不等式约束. 其三, 假设 $\mathcal{W}_k$ 中的约束梯度向量 $\{a_i\}_{i\in\mathcal{W}_k}$ 是线性无关的.

算法在每一步都将 $\mathcal{W}_k$ 中的约束视为等式, 而暂时忽略其他不等式约束, 从而将原问题转化为一个较简单的等式约束子问题 (EQP).

2.1.3 算法的核心逻辑

算法的每一次迭代主要包含以下两个分支判断:

分支 1: 计算搜索方向与步长

首先求解以 $\mathcal{W}_k$ 为等式约束的子问题, 得到搜索方向 p_k. 若 $p_k\neq 0$, 说明还可以继续下降. 此时需计算最大步长 $\alpha_k\in[0,1]$ 以保证不违反工作集之外的约束:

$$\alpha_k=\min\left(1,\quad \min_{i\notin\mathcal{W}_k,\,a_i^Tp_k<0}\frac{b_i-a_i^Tx_k}{a_i^Tp_k}\right)$$

若 $\alpha_k<1$, 说明遇到了挡路约束, 需将其加入工作集; 若 $\alpha_k=1$, 则直接移动到子问题的极小值点, 工作集保持不变.

分支 2: 检验最优性与剔除约束

若 $p_k=0$, 说明在当前工作集子空间内已达最优. 此时需检查拉格朗日乘子 $\hat{\lambda}$:

$$\sum_{i\in\mathcal{W}_k}a_i\hat{\lambda}_i=Gx_k+c$$

若所有不等式约束的乘子 $\hat{\lambda}_i\geq 0$, 则满足 KKT 条件, 找到全局最优解. 若存在 $\hat{\lambda}_j<0$, 说明该约束阻碍了目标函数进一步下降, 需将其从工作集中剔除.

2.1.4 算法伪代码

算法的完整流程如下所示:

Algorithm 3 凸二次规划的有效集法

初始化: 计算一个可行初始点 x_0, 并设定初始工作集 $\mathcal{W}_0(\mathcal{W}_0 \subseteq \mathcal{A}(x_0))$.
for $k = 0, 1, 2, \ldots$ **do**
  步骤 1: 求解 EQP 子问题
  求解以下问题得到搜索方向 p_k:

$$\min_p \frac{1}{2} p^T G p + (G x_k + c)^T p \quad \text{s.t.} \quad a_i^T p = 0, \forall i \in \mathcal{W}_k$$

  if $p_k = 0$ **then**
    步骤 2: 检查乘子
    计算满足 $\sum_{i \in \mathcal{W}_k} a_i \hat{\lambda}_i = G x_k + c$ 的拉格朗日乘子 $\hat{\lambda}$
    if $\hat{\lambda}_i \geq 0, \quad \forall i \in \mathcal{W}_k \cap \mathcal{I}$ **then**
      停止: $x^* = x_k$ 即为全局最优解.
    else
      选择使得 $\hat{\lambda}_j < 0$ 的约束索引 j(通常选最负的那个)
      $x_{k+1} \leftarrow x_k$
      $\mathcal{W}_{k+1} \leftarrow \mathcal{W}_k \setminus \{j\}$ ▷ 剔除约束
    end if
  else
    步骤 3: 计算步长
    计算 $\alpha_k \leftarrow \min\left(1, \min_{i \notin \mathcal{W}_k, a_i^T p_k < 0} \frac{b_i - a_i^T x_k}{a_i^T p_k}\right)$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    if $\alpha_k < 1$ **then**
      找到限制步长的挡路约束索引 j
      $\mathcal{W}_{k+1} \leftarrow \mathcal{W}_k \cup \{j\}$ ▷ 添加约束
    else
      $\mathcal{W}_{k+1} \leftarrow \mathcal{W}_k$
    end if
  end if
end for

2.1.5 收敛性分析

1. **有限终止性**: 在非退化假设下, 该算法保证在有限步内终止. 原因是工作集的可能组合是有限的, 且算法确保目标函数值在非最优点的迭代中严格下降, 因此不会出现循环.

2. **全局最优性**: 由于问题是凸规划,KKT 条件是全局最优的充分必要条件. 当算法在 $p_k = 0$ 且所有乘子非负时终止, 即满足了 KKT 条件, 从而保证解是全局最优的.

2.2 数值实验与结果分析

2.2.1 实验设置

为验证手写有效集法的正确性与效率, 我们设计了多组数值实验, 并与成熟的二次规划求解器 OSQP 进行对比. 测试问题为标准形式的凸二次规划:

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}x^TGx + c^Tx \quad \text{s.t.} \quad Ax \geq b \tag{12}$$

其中 $G \in \mathbb{R}^{n\times n}$ 为随机生成的对称正定矩阵,$A \in \mathbb{R}^{m\times n}$,$c, b$ 为随机向量.

实验参数设置如下:

- **终止条件**:KKT 残差 $< 10^{-8}$ 或达到最大迭代次数
- **重复次数**: 每组实验重复 5 次取平均值
- **实验环境**:AMD Ryzen 5 5500U, 16GB RAM, Python 3.11

2.2.2 固定约束数量的性能测试

表 1 展示了固定约束数量 $m = 100$ 时, 变量维度 n 从 50 增加到 2000 的性能变化.

表 1: 固定约束数量 $m = 100$ 时，不同变量维度 n 的性能比较

n	有效集法		OSQP		目标差异	加速比
	时间 (s)	迭代次数	时间 (s)	迭代次数	(绝对值)	
50	0.2641±0.5097	74	0.0061±0.0069	230	1.92e-04	0.02
100	0.0240±0.0075	79	0.0045±0.0055	65	9.72e-05	0.19
200	0.0191±0.0097	60	0.0121±0.0018	95	4.73e-05	0.63
500	0.0639±0.0108	64	0.0802±0.0252	120	1.39e-05	1.26
1000	0.2319±0.0439	65	0.7803±0.0946	130	8.30e-05	3.36
1500	0.3923±0.0466	64	2.0990±0.0461	150	1.45e-04	5.35
2000	0.7728±0.2699	64	4.4285±0.1765	125	3.91e-04	5.73

从表 1 可以观察到: 当变量维度 n 较大 (如 $n \geq 500$) 时, 有效集法相比 OSQP 展现出明显的速度优势, 加速比可达 5.73 倍. 这是因为有效集法的每次迭代仅需求解一个等式约束子问题, 而 OSQP 作为一阶方法需要更多迭代才能达到相同精度.

2.2.3 固定变量维度的性能测试

表 2 展示了固定变量维度 $n = 500$ 时, 约束数量 m 从 50 增加到 500 的性能变化.

表 2: 固定变量维度 $n = 500$ 时，不同约束数量 m 的性能比较

m	有效集法		OSQP		目标差异	加速比
	时间 (s)	迭代次数	时间 (s)	迭代次数	(绝对值)	
50	0.0288±0.0078	32	0.0605±0.0160	125	1.50e-05	2.10
100	0.0592±0.0065	64	0.0775±0.0197	120	1.39e-05	1.31
200	0.1501±0.0218	139	0.0929±0.0116	120	4.78e-05	0.62
300	0.2814±0.0465	206	0.1205±0.0065	100	1.01e-03	0.43
400	0.4851±0.0268	292	0.2017±0.0113	110	9.05e-04	0.42
500	0.9082±0.0575	415	0.3455±0.0401	110	2.97e-03	0.38

从表 2 可以看出: 当约束数量 m 增大时, 有效集法的迭代次数显著增加 (从 32 次增至 415 次), 导致总时间增长. 这符合有效集法的理论特性——最坏情况下可能需要遍历所有约束组合. 相比之下,OSQP 的迭代次数相对稳定.

2.2.4 大规模问题测试

表 3 展示了大规模问题 (变量维度和约束数量同时增大) 下的算法性能.

表 3: 大规模问题的性能比较

n	m	有效集法		OSQP		目标差异	加速比
		时间 (s)	迭代次数	时间 (s)	迭代次数	(绝对值)	
1000	200	0.4268±0.0238	146	0.8082±0.1459	158	4.71e-07	1.89
1500	300	1.3937±0.0572	194	3.2763±0.1043	158	1.23e-03	2.35
2000	400	3.3922±0.4256	287	6.1394±0.2243	158	1.68e-04	1.81
2500	500	7.0515±1.2780	350	11.4530±0.6320	158	1.73e-04	1.62
3000	600	10.1802±1.0055	407	19.6626±1.4677	158	3.14e-05	1.93

实验结果表明: 在大规模问题上, 有效集法仍然保持了对 OSQP 的速度优势, 加速比稳定在 1.6–2.4 倍之间. 目标函数值的差异 (绝对值) 均在 10^{-3} 量级以下, 验证了算法的准确性.

2.2.5 条件数敏感性测试

表 4 展示了不同 Hessian 矩阵条件数 $\kappa(G)$ 下的算法稳定性.

表 4: 不同条件数下的算法性能比较 ($n = 300$, $m = 100$)

条件数	有效集法		OSQP		目标差异	加速比
$\kappa(G)$	时间 (s)	迭代次数	时间 (s)	迭代次数	(绝对值)	
$1e+01$	0.0230±0.0017	51	0.0182±0.0014	–	1.59e-07	0.79
$1e+02$	0.0267±0.0020	54	0.0181±0.0018	–	8.75e-06	0.68
$1e+03$	0.0236±0.0037	52	0.0171±0.0006	–	3.40e-04	0.72
$1e+04$	0.0243±0.0040	54	0.0186±0.0022	–	1.66e-03	0.77

从表 4 可以看出: 有效集法对条件数的变化表现出良好的鲁棒性, 迭代次数和运行时间几乎不受条件数影响. 然而, 随着条件数增大, 目标函数值的差异有所增加, 这主要是由于数值精度的限制.

2.2.6 综合结果汇总

表 5 汇总了所有测试类别的综合结果.

表 5: 有效集法数值实验综合结果汇总

测试类别	问题规模	平均时间 (s)	平均迭代	最大目标差异	平均加速比
固定约束数量	$n \in [50, 2000]$, $m = 100$	0.2526	67	1.02e-03	2.36
固定变量维度	$n = 500$, $m \in [50, 500]$	0.3188	191	5.92e-03	0.88
大规模问题	$n \in [1000, 3000]$	4.4889	277	3.69e-03	1.92
条件数测试	$\kappa \in [10, 10^4]$	0.0244	53	4.31e-03	0.74

2.2.7 结论

数值实验验证了手写有效集法的正确性与效率. 其一, 笔者所写代码的计算结果与 OSQP 求解器的目标函数值差异均在 10^{-3} 量级以下, 表明算法实现正确. 其二, 积极集算法更适合 $n \gg m$ 的问题; 当约束数量 m 接近或超过变量维度 n 时, 一阶方法 (如 OSQP) 可能更具优势. 其三, 该算法对 Hessian 矩阵条件数表现出良好的鲁棒性.

3 第三题

3.1 问题描述

本题要求使用 L-BFGS 算法在 Stiefel 流形 $\mathcal{M} = \mathrm{St}(n, p) = \{X \in \mathbb{R}^{n\times p} \mid X^\top X = I_p\}$ 上求解二次函数的最小化问题, 并进行相关性能评估, 包括代码运行时间、迭代次数、最

优解处梯度的范数, 最优解目标函数值. 目标函数定义如下:

$$\min_{X\in \mathrm{St}(n,p)} f(X) = \mathrm{Tr}(X^\top AX) + 2\mathrm{Tr}(B^\top X), \tag{13}$$

其中 $A \in \mathbb{R}^{n\times n}$ 为对称矩阵,$B \in \mathbb{R}^{n\times p}$ 为任意矩阵.

3.2 流形上的 L-BFGS 算法描述

流形上的 L-BFGS 算法通过利用最近 m 步的曲率信息 $\{s_k, y_k\}$ 来近似 Hessian 的逆算子, 从而避免了显式计算 Hessian 矩阵. 与欧氏空间不同, 流形上的 s_k 和 y_k 位于不同的切空间, 需要利用向量传输或隐式近似处理.

3.2.1 算法流程

算法的伪代码如算法4所示, 其核心步骤如下:

Step1 初始化: 选择初始点 $X_0 \in \mathrm{St}(n,p)$, 设定内存大小 m.

Step2 计算搜索方向: 在第 k 步, 计算黎曼梯度 $g_k = \mathrm{grad} f(X_k)$. 利用双循环递归算法, 结合存储的对 (s_i, y_i), 计算下降方向 $d_k = -H_k g_k$.

Step3 线搜索: 采用 Armijo 准则确定步长 α_k, 使得

$$f(R_{X_k}(\alpha_k d_k)) \le f(X_k) + c_1\alpha_k\langle g_k, d_k\rangle. \tag{14}$$

Step4 更新迭代点: 令 $X_{k+1} = R_{X_k}(\alpha_k d_k)$.

Step5 曲率信息更新: 计算位移 s_k 和梯度差 y_k. 由于 $g_k \in T_{X_k}\mathcal{M}$ 而 $g_{k+1} \in T_{X_{k+1}}\mathcal{M}$, 在实际代码实现中, 我们通过将切向量视为环境空间 $\mathbb{R}^{n\times p}$ 中的矩阵进行近似计算:

$$s_k = X_{k+1} - X_k, \quad y_k = g_{k+1} - g_k. \tag{15}$$

若满足曲率条件 $\langle s_k, y_k\rangle > 0$, 则将 (s_k, y_k) 存入内存, 移除最旧的一对.

Algorithm 4 黎曼 L-BFGS 算法 (含阻尼更新策略)

Require: 初始点 $X_0 \in St(n,p)$, 内存大小 m, 阻尼参数 δ (默认 1.0)

$k \leftarrow 0$, 初始化存储对列表 $\mathcal{M} = \emptyset$
while $\|\mathrm{grad} f(X_k)\| > \epsilon$ **do**
  计算黎曼梯度:$\xi_k = \mathrm{grad} f(X_k)$
  1. 双循环递归 (Two-Loop Recursion) 计算方向 d_k:
  $q \leftarrow \xi_k$
  for $i = |\mathcal{M}| - 1$ **to** 0 **do** ▷ Backward Pass
    $(s_i, y_i, \rho_i) \leftarrow \mathcal{M}[i]$
    $\alpha_i \leftarrow \rho_i \langle s_i, q \rangle$
    $q \leftarrow q - \alpha_i y_i$
  end for
  $\gamma_k \leftarrow \frac{\langle s_{last}, y_{last} \rangle}{\langle y_{last}, y_{last} \rangle}$ ▷ Scaling
  $r \leftarrow \gamma_k q$
  for $i = 0$ **to** $|\mathcal{M}| - 1$ **do** ▷ Forward Pass
    $(s_i, y_i, \rho_i) \leftarrow \mathcal{M}[i]$
    $\beta \leftarrow \rho_i \langle y_i, r \rangle$
    $r \leftarrow r + s_i(\alpha_i - \beta)$
  end for
  $d_k \leftarrow -r$
  2. 线搜索与更新:
  获取步长 t_k 并更新 $X_{k+1} = R_{X_k}(t_k d_k)$
  3. 阻尼更新 (Damped Update):
  计算位移与梯度差 (欧氏近似):$s_k = X_{k+1} - X_k, \quad y_k = \mathrm{grad} f(X_{k+1}) - \xi_k$
  计算投影曲率:$sy = \langle s_k, y_k \rangle, \quad ss = \delta \langle s_k, s_k \rangle$
  if $sy < 0.25 ss$ **then** ▷ 阻尼条件判断
    $\theta = \frac{0.75 ss}{ss - sy}$
    $r_k = \theta y_k + (1 - \theta) \delta s_k$ ▷ 修正梯度差
  else
    $r_k = y_k$
  end if
  若 $\langle s_k, r_k \rangle > 10^{-10}$, 将 $(s_k, r_k, 1/\langle s_k, r_k \rangle)$ 存入 $\mathcal{M}$ (若满则移除最旧)
  $k \leftarrow k + 1$
end while

在黎曼流形 $\mathcal{M}$ 上的拟牛顿法中, 为了提高算法在非凸问题上的鲁棒性以及在大规模问题上的计算效率, 常采用阻尼技术和子空间技术. 以下分别对这两种方法进行总结.

3.3 阻尼 L-BFGS 更新

在拟牛顿法中, 为保证拟牛顿矩阵 B_{k+1}(或其逆 H_{k+1}) 的正定性, 必须满足曲率条件 $s_k^{\mathrm{T}} y_k > 0$, 其中 s_k 为位移向量,y_k 为梯度差向量. 然而, 当步长由非精确线搜索确定时, 该条件可能不成立. 为此, 采用阻尼技术对梯度差向量进行修正.

类似讲义对欧氏空间中阻尼 L-BFGS 方法的描述, 给定对称正定矩阵 B_k、位移向量 s_k 和梯度差 y_k, 我们构造修正向量 r_k 来代替 y_k:

$$r_k = \theta_k y_k + (1 - \theta_k) B_k s_k, \tag{16}$$

其中 $\theta_k \in [0, 1]$ 是确保 B_{k+1} 保持正定的参数, 定义如下:

$$\theta_k = \begin{cases} 1, & 若 s_k^{\mathrm{T}} y_k \geq 0.25 s_k^{\mathrm{T}} B_k s_k, \\ \frac{0.75 s_k^{\mathrm{T}} B_k s_k}{s_k^{\mathrm{T}} B_k s_k - s_k^{\mathrm{T}} y_k}, & 若 s_k^{\mathrm{T}} y_k < 0.25 s_k^{\mathrm{T}} B_k s_k. \end{cases} \tag{17}$$

该构造保证了 $s_k^{\mathrm{T}} r_k \geq 0.25 s_k^{\mathrm{T}} B_k s_k > 0$, 即修正后的曲率条件严格成立. 在实际的 L-BFGS 算法中, 通常取 B_k 为初始近似矩阵 (如 $B_{k,0} = \delta I$). 利用修正后的对 (s_k, r_k) 执行标准的 BFGS 更新公式:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^{\mathrm{T}} B_k}{s_k^{\mathrm{T}} B_k s_k} + \frac{r_k r_k^{\mathrm{T}}}{s_k^{\mathrm{T}} r_k}. \tag{18}$$

此方法能够有效防止拟牛顿矩阵在非凸区域失去正定性, 从而增强算法的稳定性. 具体的算法流程如算法 4 所示, 其核心是利用双循环递归高效计算搜索方向 $d_k = -H_k g_k$.

3.4 子空间 L-BFGS 方法

注意, 笔者尝试将子空间方法和形上的 L-BFGS 方法结合起来, 代码中实现了这部分的算法, 由于时间关系, 来不及验证其正确性与优化代码, 故这里仅简要介绍笔者的思路.

在大规模问题中, 存储和更新全维拟牛顿矩阵代价过高. 讲义 **§5.2.2** 介绍了子空间方法. 其核心思想是将优化限制在由历史梯度张成的低维 Krylov 子空间 $G_k = \mathrm{span}\{g_0, \ldots, g_k\}$ 中.

根据讲义 **定理 5.2.7 (第 156 页)**, 如果我们维护子空间的一组正交基 Z_k, 并定义投影梯度 $\bar{g}_k = Z_k^T g_k$ 和投影拟牛顿矩阵 $\bar{H}_k = Z_k^T H_k Z_k$, 那么在低维空间直接对 $\bar{H}_k$ 进行 BFGS 更新, 在理论上等价于在全空间更新 H_k 后再投影. 这为算法的高效实现提供了理论依据.

定义梯度向量 $g_0, \cdots, g_k$ 张成的线性子空间为:

$$G_k = \mathrm{span}\{g_0, \cdots, g_k\}, \quad \forall k \geq 0. \tag{19}$$

设 l_k 为子空间 G_k 的维数. 令 $Z_k \in \mathbb{R}^{n \times l_k}$ 为 G_k 的一组单位正交基, 即 $Z_k^{\mathrm{T}} Z_k = I_{l_k}$. 将梯度 g_k 和拟牛顿矩阵 H_k 投影到子空间, 定义:

$$\bar{g}_k = Z_k^{\mathrm{T}} g_k \in \mathbb{R}^{l_k}, \quad \bar{H}_k = Z_k^{\mathrm{T}} H_k Z_k \in \mathbb{R}^{l_k \times l_k}. \tag{20}$$

在此框架下, 搜索方向 $p_k = -H_k g_k$ 可以通过低维计算得到:

$$H_k g_k = Z_k \bar{H}_k \bar{g}_k. \tag{21}$$

这意味着迭代公式可以写为 $x_{k+1} = x_k - \alpha_k Z_k \bar{H}_k \bar{g}_k$. 当 $l_k \ll n$ 时, 更新 $\bar{H}_k$ 的计算量远小于更新 H_k.

算法的关键在于从 $\bar{H}_k$ 递推得到 $\bar{H}_{k+1}$. 通过引入辅助向量 ϕ_{k+1} 和 u_{k+1}:

$$\phi_{k+1} = \|(I - Z_k Z_k^{\mathrm{T}}) g_{k+1}\|, \quad u_k = Z_k^{\mathrm{T}} g_{k+1}. \tag{22}$$

如果 $\phi_{k+1} > 0$, 则扩展基矩阵 $Z_{k+1} = [Z_k, z_{k+1}]$, 其中 $z_{k+1} = (g_{k+1} - Z_k u_k)/\phi_{k+1}$. 此时, 低维矩阵 $\bar{H}_{k+1}$ 可通过如下分块矩阵的 BFGS 更新得到:

$$\bar{H}_{k+1} = \left(I - \tilde{\rho}_k \tilde{s}_k \tilde{y}_k^{\mathrm{T}}\right) \tilde{H}_k \left(I - \tilde{\rho}_k \tilde{y}_k \tilde{s}_k^{\mathrm{T}}\right) + \tilde{\rho}_k \tilde{s}_k \tilde{s}_k^{\mathrm{T}}, \tag{23}$$

其中 $\tilde{s}_k, \tilde{y}_k$ 分别是 s_k, y_k 在扩展子空间上的投影,$\tilde{H}_k$ 是 $\bar{H}_k$ 的扩展矩阵. 为了控制计算成本, 通常采用周期性重启策略或限制子空间维数 (类似于 L-BFGS 的显式截断). 具体流程详见算法5.

Algorithm 5 子空间 L-BFGS 算法

Require: 最大子空间维数 $dim_{\max}$

初始化:$Z = [\xi_0/\|\xi_0\|]$ (基矩阵), $H_{sub} = [1]$ (子空间逆 Hessian), $dim = 1$

while 未收敛 **do**

1. 子空间投影求方向:

将梯度投影到子空间:$g_{sub} = Z^T \mathrm{grad} f(X_k)$

计算子空间方向:$p_{sub} = -H_{sub} g_{sub}$

恢复全空间方向:$d_k = Z p_{sub}$

2. 更新迭代点:

$X_{k+1} = R_{X_k}(t_k d_k)$

计算 $s_k = X_{k+1} - X_k, \quad y_k = \mathrm{grad} f(X_{k+1}) - \mathrm{grad} f(X_k)$

3. 扩展子空间基 Z:

计算 $g_{next} = y_k + \mathrm{grad} f(X_k)$ (利用梯度递推关系)

正交化:$u = g_{next} - Z(Z^T g_{next})$

if $\|u\| > \epsilon$ **and** $dim < dim_{\max}$ **then**

$Z \leftarrow [Z, u/\|u\|]$, $dim \leftarrow dim + 1$

扩展 H_{sub}:$H_{sub} \leftarrow \mathrm{diag}(H_{sub}, 1)$

else

若已达最大维数, 重置子空间

end if

4. 更新子空间矩阵 H_{sub}:

投影 s, y:$\bar{s} = Z^T s_k, \quad \bar{y} = Z^T y_k$

使用 BFGS 公式更新 H_{sub}:

$$H_{sub} \leftarrow (I - \rho \bar{s} \bar{y}^T) H_{sub} (I - \rho \bar{y} \bar{s}^T) + \rho \bar{s} \bar{s}^T$$

end while

3.5 数值实验与结果分析

3.5.1 实验设置

为了评估黎曼 L-BFGS 及阻尼 L-BFGS 在 Stiefel 流形上的性能，我们构建了如下的二次规划测试问题：

$$\min_{X \in St(n,p)} f(X) = \mathrm{Tr}(X^T A X) + 2\mathrm{Tr}(B^T X) \tag{24}$$

其中 $A \in \mathbb{R}^{n \times n}$ 为生成的对称矩阵，$B \in \mathbb{R}^{n \times p}$ 为全零矩阵（主要考察二次项性质）。

参数设置如下：

- **算法对比:**标准黎曼 L-BFGS 与阻尼黎曼 L-BFGS (以下简称为 Damped, $\delta = 20.0$)。
- **存储对数：**$m = 10$。
- **终止条件：**梯度范数 $\|\text{grad} f(X_k)\| < 10^{-6}$ 或达到最大迭代次数 1000。
- **线搜索：**采用 Zhang-Hager 非单调线搜索 ($\rho = 0.5$)。
- **实验环境：**AMD Ryzen 5 5500U, 16GB RAM, Python 3.11。

实验分为两组，分别考察矩阵规模 n 和流形列数 p 对算法性能的影响。

3.5.2 结果分析

1. 列数 p 变化的影响 (固定 $n = 2000$)

表 6 展示了当 n 固定为 2000，列数 p 从 10 增加到 200 时算法的表现。

表 6: 固定 $n = 2000$，不同 p 值下的算法性能对比

维度 (n, p)	方法	总时间 (s)	迭代次数	单步时间 (s)	最终 $\|\nabla f\|$
$(2000, 10)$	L-BFGS	8.0898	1000	0.0081	8.77e-06
	Damped	8.8346	741	0.0119	1.04e-06
$(2000, 50)$	L-BFGS	30.7671	717	0.0429	8.41e-06
	Damped	27.3751	545	0.0502	1.88e-05
$(2000, 100)$	L-BFGS	49.0515	404	0.1214	1.20e-05
	Damped	39.4684	467	0.0845	3.31e-05
$(2000, 200)$	L-BFGS	136.8057	717	0.1908	4.11e-05
	Damped	115.5013	692	0.1669	1.04e-05

从表 6 可以观察到：

(1) 计算成本随 p 增加：随着 p 的增大，单步迭代时间显著增加。这是因为流形上的收缩映射（SVD 分解）以及梯度计算的复杂度主要与 np^2 或 n^2p 相关。

(2) 阻尼策略的优势：在 $p = 50, 100, 200$ 的较大规模问题中，阻尼 L-BFGS 的总运行时间均优于标准 L-BFGS。特别是在 $(2000, 200)$ 的情况下，阻尼方法节省了约 15% 的时间。

(3) 鲁棒性：在 $p = 10$ 时，标准 L-BFGS 达到最大迭代次数 1000 仍未完全收敛至 10^{-6} 精度（最终梯度 8.77e-6），而阻尼方法在 741 步收敛。这表明在非凸的 Stiefel 流形上，阻尼更新通过强制保证曲率条件 $s_k^T y_k > 0$，有效地避免了拟牛顿矩阵的不正定问题，从而生成了质量更高的搜索方向。

2. 维度 n 变化的影响 (固定 $p = 10$)

表 7 展示了当 p 固定为 10，行数 n 从 1000 增加到 6000 时算法的表现。

表 7: 固定 $p = 10$，不同 n 值下的算法性能对比

维度 (n, p)	方法	总时间 (s)	迭代次数	单步时间 (s)	最终 $\|\nabla f\|$
$(1000, 10)$	L-BFGS	3.2421	1000	0.0032	2.97e-06
	Damped	2.2308	666	0.0034	4.77e-06
$(2000, 10)$	L-BFGS	6.9859	1000	0.0070	8.77e-06
	Damped	8.5661	741	0.0116	1.04e-06
$(4000, 10)$	L-BFGS	25.2620	1000	0.0253	1.07e-05
	Damped	30.9647	837	0.0370	8.23e-06
$(6000, 10)$	L-BFGS	94.7619	791	0.1198	3.13e-06
	Damped	52.9006	470	0.1126	1.92e-05

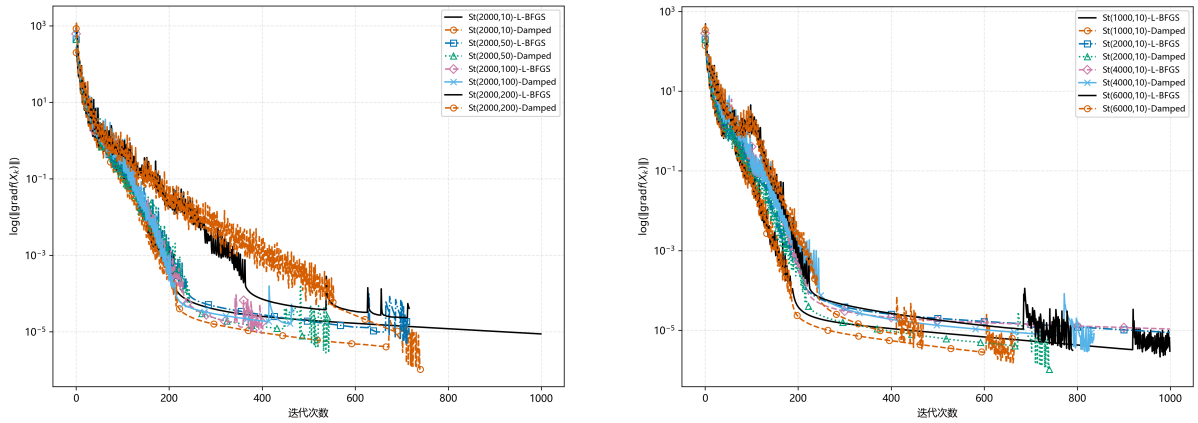
从表 7 可以观察到：

(1) 在 $(6000, 10)$ 组别中，阻尼 L-BFGS 的总时间（52.9s）显著低于标准 L-BFGS（94.8s）。这说明随着问题规模 n 的增大，目标函数的曲率变化可能更加复杂，标准 L-BFGS 容易因为曲率信息不准确而导致步长过小或方向偏差，而阻尼技术有效地修正了这一问题。

(2) 虽然阻尼更新引入了额外的向量内积和标量计算（见算法 4 中的 θ 计算），导致单步时间略微增加（例如 $n = 2000$ 时，0.0116s vs 0.0070s），但其带来的收敛步数的减少足以抵消这一开销，最终在总时间上获得优势。

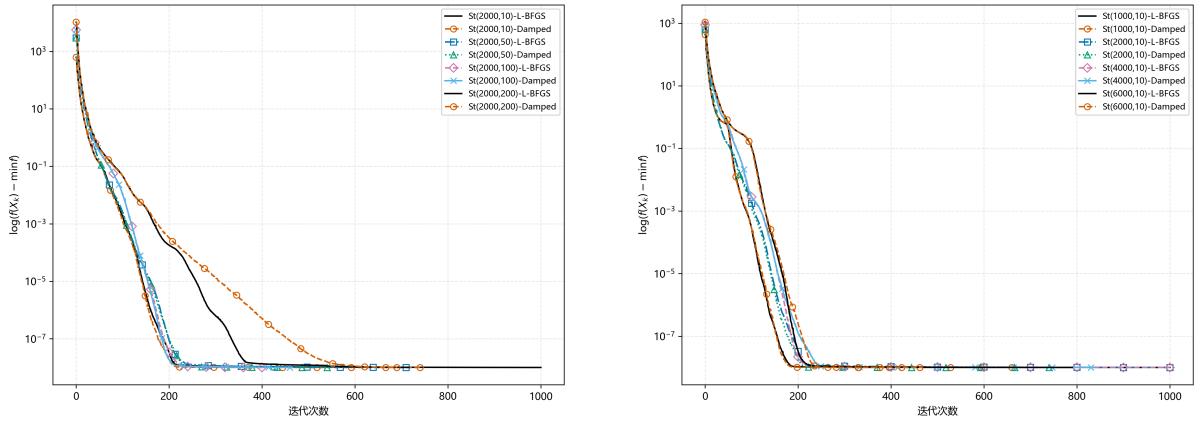
3. 收敛曲线分析

图 1 展示了梯度范数随迭代次数下降的曲线, 图 2 展示了目标函数值随迭代次数下降的曲线.



(a) 固定 $n = 2000$，不同 p 值　　(b) 固定 $p = 10$，不同 n 值

图 1: 标准 L-BFGS 与阻尼 L-BFGS 的梯度范数收敛曲线对比



(a) 固定 $n = 2000$，不同 p 值　　(b) 固定 $p = 10$，不同 n 值

图 2: 标准 L-BFGS 与阻尼 L-BFGS 的函数值收敛曲线对比

从收敛曲线可以看出，阻尼 L-BFGS（虚线/点线）通常表现出更快的下降速率，尤其是在迭代初期和中期。标准 L-BFGS 在某些阶段会出现“震荡”或下降停滞的现象（平台期），这往往是由于当前的拟牛顿矩阵 H_k 未能很好地捕获目标函数的局部几何性质。阻尼项 $(1-\theta)\delta s_k$ 的引入，相当于在拟牛顿更新中混合了一定比例的单位阵（或初始矩阵），起到了正则化的作用，使得算法运行更加平稳。

3.6 结论

本部分的数值实验验证了流形优化算法在处理 Stiefel 流形上二次规划问题的有效性。实验结果表明：

1. L-BFGS 的有效性：利用向量传输（或其隐式近似）将 L-BFGS 推广至流形是成功的，能够处理数千维的优化问题。

2. 阻尼技术的必要性：在非凸优化问题中，阻尼 L-BFGS 相比标准 L-BFGS 表现出更强的鲁棒性和更高的效率。它通过修正梯度差向量 y_k，确保了拟牛顿矩阵的正定性，从而在大规模 $(n = 6000)$ 或高列数 $(p = 200)$ 的困难问题上显著减少了迭代次数和运行时间。