流形优化期末报告

GallyYoko

2024 年 10 月 9 日

1 Question 1

编程实现 Algorithm 10 和 Algorithm 11. 下载文再文老师的代码. 运行文老师的代码, 比较简化版与 ARNT, 指出文老师方法的优势.

1.1 数值实验比较

首先, 我们使用文再文老师原论文中的非线性特征值问题 (Single Nonlinear Eigenvalue Problems) 为例, 从简化版算法与 ARNT 在这个优化问题的表现上出发, 比较两个算法的计算效果.

考虑优化问题

$$\min_{X\in\mathbb{R}^{n\times p}} \frac{1}{2}\mathrm{tr}(X^TLX)+\frac{\alpha}{4}\rho(X)^TL^{-1}\rho(X),\quad \text{subject to } X^TX=I_k.$$

其中 L 为一维 Laplace 算子离散化, 主对角线上全为 2, 次对角线上全为 1. $\rho(X)$ 是一个向量, 定义为 $\rho(X)=\mathrm{diag}(XX^T)$. α 为一个常数. 我们使用 Algorithm 11 和 ARNT 对这个问题进行求解. 在参数选择上, 我们使用表 1 中的参数值.

参数	具体数值	参数描述
σ	10	初始正则参数
γ	0.2	回退幅度
c	0.001	Armijo 条件的系数
η_1	0.01	正则化调整下限参数
η_2	0.09	正则化调整上限参数
γ_1	0.2	正则化缩小幅度
γ_2	10	正则化增大幅度
κ	0.01	修正共轭梯度法偏差阈值
method	Cayley 变换	收缩映射方式
ε	10^{-5}	梯度阈值

表 1: 算法参数

对于参数 n, p 和 α, 我们固定其中两个参数, 改变第三个参数, 进行数值实验, 得到结果分别见表 2, 3 和 4, 其中的迭代数指正则化牛顿法的迭代次数, 总迭代数指修正共轭梯度法的总迭代次数.

	Algorithm 11				ARNT			
n	迭代数	总迭代数	梯度范数	时间	迭代数	总迭代数	梯度范数	时间
2000	8	146	1.18e-05	1.23	3	95	1.21e-05	0.47
3000	8	154	6.23e-06	2.58	3	95	2.08e-05	0.84
5000	8	157	4.96e-06	5.92	4	121	2.47e-05	1.57
8000	8	157	5.06e-06	12.37	3	100	1.29e-05	2.60
10000	8	167	2.77e-06	18.95	3	117	3.15e-06	3.43

表 2: $(p, \alpha) = (30, 10)$ 时的数值实验结果

	Algorithm 11				ARNT			
p	迭代数	总迭代数	梯度范数	时间	迭代数	总迭代数	梯度范数	时间
10	8	43	6.01e-05	3.51	3	32	3.85e-07	0.21
20	8	106	4.73e-06	3.97	3	66	3.70e-06	0.70
30	8	157	4.96e-06	5.80	4	121	2.47e-05	1.60
50	8	248	2.22e-05	9.60	3	180	3.28e-05	4.30

表 3: $(n, \alpha) = (5000, 10)$ 时的数值实验结果

	Algorithm 11				ARNT			
α	迭代数	总迭代数	梯度范数	时间	迭代数	总迭代数	梯度范数	时间
1	10	107	8.00e-05	6.73	3	73	1.11e-06	1.87
10	8	157	4.96e-06	5.61	4	121	2.47e-05	1.61
100	8	170	1.88e-06	5.96	4	108	7.00e-06	2.02

表 4: $(n, p) = (5000, 30)$ 时的数值实验结果

从这些结果中可以看出, 对于实验中的这些例子, Algorithm 11 的迭代数几乎都是 8, ARNT 的迭代数几乎都是 3. 不过, 这些结果仍然揭示了一些信息:

- 当参数 n 增大时, 共轭梯度法的迭代次数没有明显差异, 但算法花费的时间增大了.
- 当参数 p 增大时, 共轭梯度法的迭代次数和算法花费的时间都增大了.
- 当参数 α 增大时, 算法花费的时间没有明显的变化, 共轭梯度法的迭代次数的在 α 较小时上升, 而在 α 较大时保持基本不变.
- Algorithm 11 的表现在各方面都要差于 ARNT.

1.2 修正共轭梯度法比较

本小节中, 我们对简化版 ARNT 中使用的修正共轭梯度法 Algorithm 10 与 ARNT 中使用的修正共轭梯度法 RNewton 进行比较, 说明 ARNT 相比简化版的优点所在.

首先, 我们将两个算法同时给出, 见算法 1 和算法 2, 对于其中不同的部分, 我们使用不同的颜色加以区分, 其中绿色表示仅在这个算法中出现的部分, 红色表示在两个算法中都有出现, 但有所区别的部分. 为了更直观地展现两个算法的不同之处, 下面的伪代码将与书上和论文中的伪代码略有不同.

算法 1: Algorithm 10

输入: 黎曼梯度方向 g, 黎曼海森算子 B, 偏差阈值 κ

输出: 方程的解 d, 其中方程为 $Bd = -g$

$\varepsilon \leftarrow \min\left(0.5, \sqrt{\|g\|}\right)\|g\|$; $z_0 \leftarrow 0$; $r_0 \leftarrow -g$; $p_0 \leftarrow -g$; // 初始设置

for $j = 0, 1, 2, \cdots$ **do**

  if $\langle p_j, Bp_j\rangle \le 0$ **then**

    若 $j = 0$ 则 $d \leftarrow -g$, 否则$d \leftarrow z_j$, 终止循环; // 前进方向为零或负曲率方向

  end

  $\alpha_j \leftarrow \frac{\langle r_j, r_j\rangle}{\langle p_j, Bp_j\rangle}$; $z_{j+1} \leftarrow z_j + \alpha_j p_j$; $r_{j+1} \leftarrow r_j - \alpha_j Bp_j$; // 方程近似解更新

  if $\langle z_{j+1}, -g\rangle \le \kappa\|g\|\|z_{j+1}\|$ **then**

    $d \leftarrow z_j$, 终止循环; // 方程近似解偏离负梯度方向

  end

  if $\|r_{j+1}\| \le \varepsilon$ **then**

    $d \leftarrow z_{j+1}$, 终止循环; // 残差很小

  end

  $\beta_j \leftarrow \frac{\langle r_{j+1}, r_{j+1}\rangle}{\langle r_j, r_j\rangle}$; $p_{j+1} \leftarrow r_{j+1} + \beta_j p_j$; // 解的前进方向更新

end

从算法 1 和算法 2 中可以直观地看出, 这两个算法的方程近似解更新和前进方向更新阶段是完全一样的, 不同点在于输入值、初始设置和跳出循环的方式, 而其中输入值和初始设置的不同是由于二者跳出循环的方式不同而导致的. 因此, 我们主要探讨二者跳出循环方式的差别:

- 算法 1 和算法 2 都会在前进方向为零曲率或负曲率后跳出循环. 不同的是, 在算法 1 中使用的是数值 0 来判断, 并且当前进方向为零曲率或负曲率方向时, 算法都是同样的输出 $d \leftarrow z_j$; 在算法 2 中使用的是输入的小参数 ϵ 来判断, 可以略微降低计算误差带来的影响, 并且对于不同的前进方向, 算法给出了不同的输出 $d \leftarrow z_j + \tau p_j$(其中 τ 在零曲率时为 0, 在负曲率时不为 0), 相比算法 1 更精细.
- 算法 1 会在迭代的解偏离负梯度方向过远的时候跳出循环, 而算法 2 中并没有这种停止循环方式.

算法 2: RNewton

输入: 黎曼梯度方向 g, 黎曼海森算子 B, 参数 T, θ, ϵ

输出: 方程的解 d, 其中方程为 $Bd = -g$

$z_0 \leftarrow 0$; $r_0 \leftarrow -g$; $p_0 \leftarrow -g$; // 初始设置

for $j = 0, 1, 2, \cdots$ **do**

  if $-\epsilon\langle p_j, p_j\rangle \le \langle p_j, Bp_j\rangle \le \epsilon\langle p_j, p_j\rangle$ **then**

    若 $j = 0$ 则 $d \leftarrow -g$, 否则 $d \leftarrow z_j$, 终止循环; // 前进方向为零曲率方向

  end

  if $\langle p_j, Bp_j\rangle \le -\epsilon\langle p_j, p_j\rangle$ **then**

    若 $j = 0$ 则 $d \leftarrow -g$, 否则$d \leftarrow z_j + \frac{\langle p_j, g\rangle}{\langle p_j, Bp_j\rangle} p_j$, 终止循环; // 前进方向为负曲率方向

  end

  $\alpha_j \leftarrow \frac{\langle r_j, r_j\rangle}{\langle p_j, Bp_j\rangle}$; $z_{j+1} \leftarrow z_j + \alpha_j p_j$; $r_{j+1} \leftarrow r_j - \alpha_j Bp_j$; // 方程近似解更新

  if $\|r_{j+1}\| \le \min(\|r_0\|^\theta, T)$ **then**

    $d \leftarrow z_{j+1}$, 终止循环; // 残差很小

  end

  $\beta_j \leftarrow \frac{\langle r_{j+1}, r_{j+1}\rangle}{\langle r_j, r_j\rangle}$; $p_{j+1} \leftarrow r_{j+1} + \beta_j p_j$; // 解的前进方向更新

end

- 算法 1 和算法 2 都会在残差过小的时候跳出循环, 但二者的判别阈值并不完全相同, 算法 1 使用的是阈值 ε 来判断, 算法 2 使用的是阈值 $\min(\|r_0\|^\theta, T)$ 来判断. 值得注意的是, 当切向量 g 的范数 $\|g\| \le 0.25$ 时, 有 $\varepsilon = \|g\|^{1.5} = \|r_0\|^{1.5}$, 因此算法 2 的判别阈值实际上是比算法 1 更精细的.

1.3 正则化牛顿法比较

在上一节中我们已经比较过了 Algorithm 10 与 RNewton 算法之间的不同点. 在本节中, 我们用同样的的方式比较 Algorithm 11 与 ARNT 算法之间的不同点, 两个算法的伪代码实现见算法 3 和算法 4, 其中的 f 为待优化函数, m_k 为 f 在 x_k 点二阶展开后正则化的近似模型

$$m_k(d) = f(x_k) + \langle \mathrm{grad} f(x_k), d\rangle + \frac{1}{2}\langle \mathrm{Hess} f(x_k)d, d\rangle + \frac{\sigma_k}{2}\|d\|^2.$$

从算法 3 和算法 4 可以看出, Algorithm 11 与 ARNT 的区别主要在两点:

- Algorithm 11 使用的是关于 f 的回退法线搜索, 而 ARNT 使用的是关于 m_k 的回退法线搜索. 事实上, m_k 是 f 在 x_k 点二阶展开后的近似模型, 因此在计算量上会比直接计算关于 f 的回退法线搜索要略低一点.

- 二者的正则化参数更新过程不同, Algorithm 11 设定了正则化参数更新的具体方式, 而 ARNT 仅给出了正则化参数的更新范围, 因此 ARNT 在正则化参数更新上要更精细一些.

算法 3: Algorithm 11

输入: 初始点 x_0, 初始参数 σ_0, 系数 c, 回退幅度 γ, 正则阈值 η_1, η_2, 正则幅度 γ_1, γ_2

$k \leftarrow 0$;
while 终止条件不成立 **do**
  使用 Algorithm 10得到步长 d_k;
  初始步长为 1, 使用关于 f的回退法线搜索得到步长 t_k 和中间点 $y_k = \mathbf{R}_{x_k}(t_k d_k)$;
  计算预测比 $\rho_k \leftarrow \frac{f(y_k)-f(x_k)}{m_k(t_k d_k)-f(x_k)}$;
  更新正则参数: 若 $\rho_k \geq \eta_2$, 则$\sigma_{k+1} \leftarrow \gamma_1 \sigma_k$; 若 $\eta_2 \geq \rho_k \geq \eta_1$, 则$\sigma_{k+1} \leftarrow \sigma_k$; 若 $\rho_k \leq \eta_1$, 则$\sigma_{k+1} \leftarrow \gamma_2 \sigma_k$;
  更新迭代点: 若 $\rho_k \geq \eta_1$, 则 $x_{k+1} \leftarrow y_k$; 若 $\rho_k \leq \eta_1$, 则 $x_{k+1} \leftarrow x_k$;
  $k \leftarrow k + 1$;
end

算法 4: ARNT

输入: 初始点 x_0, 初始参数 σ_0, 系数 c, 回退幅度 γ, 正则阈值 η_1, η_2, 正则幅度 $\gamma_0, \gamma_1, \gamma_2$

$k \leftarrow 0$;
while 终止条件不成立 **do**
  使用 RNewton得到步长 d_k;
  初始步长为 1, 使用关于 m_k的回退法线搜索得到步长 t_k 和中间点 $y_k = \mathbf{R}_{x_k}(t_k d_k)$;
  计算预测比 $\rho_k \leftarrow \frac{f(y_k)-f(x_k)}{m_k(t_k d_k)-f(x_k)}$;
  更新正则参数: 若 $\rho_k \geq \eta_2$, 则$\sigma_{k+1} \in (0, \gamma_0 \sigma_k]$; 若 $\eta_2 \geq \rho_k \geq \eta_1$, 则$\sigma_{k+1} \in [\gamma_0 \sigma_k, \gamma_1 \sigma_k]$; 若 $\rho_k \leq \eta_1$, 则$\sigma_{k+1} \in [\gamma_1 \sigma_k, \gamma_2 \sigma_k]$;
  更新迭代点: 若 $\rho_k \geq \eta_1$, 则 $x_{k+1} \leftarrow y_k$; 若 $\rho_k \leq \eta_1$, 则 $x_{k+1} \leftarrow x_k$;
  $k \leftarrow k + 1$;
end

事实上, 相较于 Algorithm 11, 在文老师的具体代码实现中, 当预测比 $\rho_k < 0$ 时, 正则化参数 σ_k 的增大幅度会比 $0 \leq \rho_k \leq \eta_1$ 更大, 这带来了更好的计算结果.

2 Question 2

编程实现教材上的 Algorithm 3 和 Algorithm 4, 随机生成 Stiefel 流形的一个二次函数, 用 Algorithm 4 极小化这个二次函数, 比较两个算法的计算效果, 探索非单调的作用, 或者交替使用 BB 步长两个公式的作用.

2.1 随机二次函数的生成

首先考虑实现二次函数的随机生成问题. 对于整数 $n \geq p > 0$, 欧氏空间 $\mathbb{R}^{n\times p}$ 上的 $Stiefel$ 流形为

$$\mathrm{St}(n,p) = \{X \in \mathbb{R}^{n\times p} | X^T X = I_p\}.$$

对于矩阵 $B \in \mathbb{R}^{n\times p}$ 和对称矩阵 $A \in \mathbb{R}^{n\times n}$, $St(n,p)$ 上的二次函数定义为

$$f(X; A, B) = \mathrm{tr}(X^T AX + 2X^T B).$$

故二次函数 f 的生成问题等价于矩阵 A, B 的生成问题. 在 MATLAB 中, 可以使用 **randn** 函数随机生成矩阵, 因此, 可以取

$$B \leftarrow \mathrm{randn}(n,p).$$

但 MATLAB 中并没有直接生成对称矩阵的函数. 不过, 我们可以注意到, 对于任意的方阵 $\hat{A}$, 矩阵 $\hat{A} + \hat{A}^T$ 是一个对称矩阵. 基于这一结论, 我们使用吴钢老师论文中的随机对称矩阵生成方法:

$$\hat{A} \leftarrow \mathrm{sprand}(n,n,0.5), \quad A \leftarrow \hat{A} + \hat{A}^T.$$

在下文中, 我们考虑 $n = 500$, $p = 50$ 时各算法的表现, 取 MATLAB 中随机函数的种子为 99, 生成对应的矩阵对 (A, B).

2.2 梯度下降算法与 BB 算法的比较

本小节中, 我们使用基于单调线搜索的梯度下降算法与基于非单调线搜索的 BB 算法对二次函数 $f(X) = \mathrm{tr}(X^T AX + 2X^T B)$ 进行优化. 在参数选择上, 我们使用表 5 中的参数值.

参数	具体数值	参数描述
t_0	0.01	回退法的初始步长
α_0	0.01	BB 算法的初始步长
ρ	0.5	回退幅度
c	0.001	Armijo 条件的系数
M	5	非单调线搜索的前项个数
$\alpha_{\max}$	1000	BB 步长的上界
$\alpha_{\min}$	0.001	BB 步长的下界
method	QR 分解	收缩映射算法
ε	10^{-10}	梯度范数的阈值

表 5: 算法参数

我们先了解一下两个算法的收敛速度, 随机选取初始点 X_0, 观察函数值 $f(X_k)$ 与点列 X_k 的误差变化情况, 以及黎曼梯度 $\mathrm{grad} f(X_k)$ 的范数变化情况. 实验结果见图 1, 其中左边是自变量点列 X_k 关于最小值点 X^* 的误差随迭代次数的变化情况, 其度量为两点作为 $\mathbb{R}^{n\times p}$ 矩阵时, 差

值 $X_k - X^*$ 的 Frobenius 范数; 中间是函数值 $f(X_k)$ 关于最小值 $f(X^*)$ 的相对误差变化情况; 右边是黎曼梯度 $\mathrm{grad} f(X_k)$ 的范数变化情况.

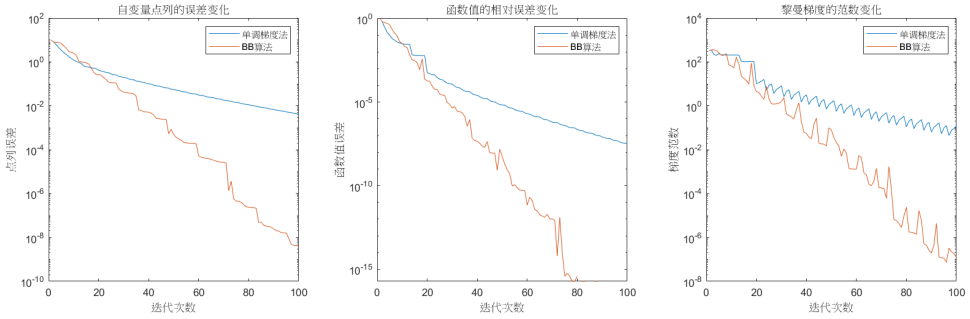

图 1: 单调梯度法与 BB 算法的比较

从图 1 中可以看出两个算法的表现情况:

- BB 算法在 80 步内使函数值误差达到了机器精度, 自变量 X_k 的误差降到了 10^{-8} 以内, 黎曼梯度的范数降到了 10^{-7} 以内, 效果非常好.
- 单调梯度法在 100 步内将函数值误差降到了 10^{-7} 以内, 自变量 X_k 的误差仅在 10^{-2} 内, 黎曼梯度的范数更是有 0.1 的误差, 远不如 BB 算法.
- 梯度下降算法使点列和函数值的误差单调下降, BB 算法却有时会导致二者误差上升, 但 BB 算法的误差从总体上看是下降的, 这一点是由非单调线搜索的性质产生的.
- 函数值的收敛速度比自变量点列的收敛速度更快, 这一点是由二次函数的性质产生的.

接下来, 对于不同的收缩映射算法, 我们对比一下算法的收敛速度, 结果见图 2, 其中左边为自变量点列的误差变化, 中间为函数值的相对误差变化, 右边为黎曼梯度的范数变化, 第一行为使用单调梯度法得到的变化图, 第二行为使用 BB 算法得到的变化图.

从图 2 中可以看到, 不同的收缩映射在同一个优化算法下的表现是几乎完全相同的. 唯一有所不同的是使用极分解的 BB 算法. 从图 2 的第二行可以看出, 当迭代次数超过 70 次后, 极分解的表现要略微差于其他收缩映射.

值得特别一提的是, 在理论上, 基于 SVD 分解的收缩映射和基于极分解的收缩映射应当是同一个, 而这两个算法在单调梯度法中的表现也的确是完全一致的. 然而, 这两种不同的收缩映射在 BB 算法下的表现是不同的, 极分解在迭代后期要明显差于 SVD 分解, 这可能是由计算误差所导致的: 在实现基于 SVD 分解的收缩映射时, 我们直接使用了 MATLAB 内置的 **svd** 函数, 这一函数是使用商业库 MKL 构建的, 因此会进行专业的优化, 达到比极分解更好的表现.

2.3 非单调线搜索的作用

本小节中, 我们通过使用不同的前项个数 M 对二次函数 $f(X)$ 进行优化, 探索非单调线搜索的作用. 在正式讨论前, 先观察一个现象: 取初始步长 $t_0 = 0.005$ 和 0.05, 前项个数 $M = 10$,

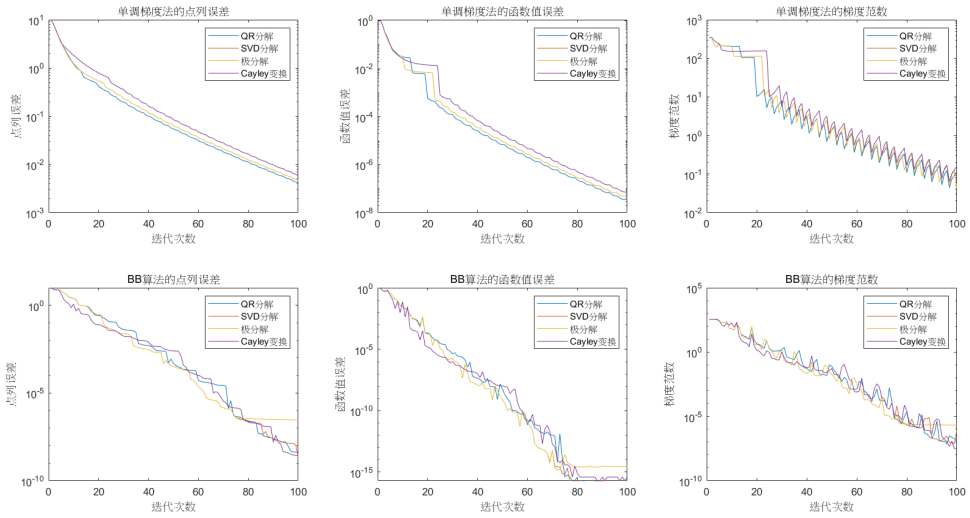

图 2: 不同收缩映射算法的表现

其他参数同表 5, 查看单调线搜索与非单调线搜索的表现. 其结果见图 3, 其中左图为初始步长 $t_0 = 0.005$ 时的表现, 右图为初始步长 $t_0 = 0.05$ 时的表现, 图像为函数值的相对误差随迭代次数的变化情况, Armijo 条件指单调线搜索, Grippo 条件指 Grippo 提出的非单调线搜索, 凸组合条件指 H. Zhang 与 W. W. Hager 提出的非单调线搜索, 其系数 $\varrho = 0.5$.

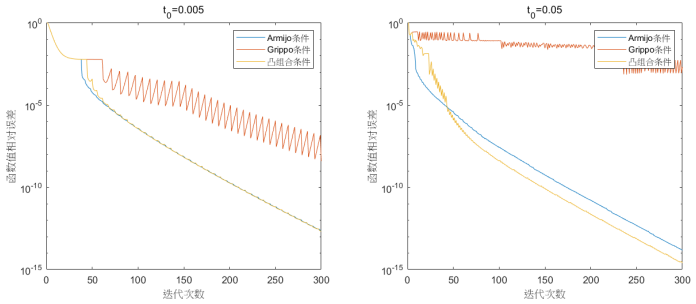

图 3: 步长对函数值误差的影响

从图 3 中可以看到, 当初始步长 $t_0 = 0.005$ 时, 单调线搜索误差与非单调线搜索误差都在持续地下降, 但当初始步长 t_0 提高 10 倍, 变成 0.05 后, 单调线搜索和凸组合线搜索的收敛速度都提高了, 但 Grippo 线搜索的收敛速度变得非常缓慢. 一种可能的解释是, Grippo 条件比 Armijo 条件和凸组合条件更宽松, 这导致算法在接近最小值点、初始步长很大时容易越过最小值点而跳到更远的地方, 导致下降缓慢.

现在我们正式考察不同参数 M 对梯度下降算法的影响, 取初始步长 $t_0 = 0.01$, $M = 1, 5, 10, 15$, 其余参数同表 5, 结果见图 4, 其中左边为自变量点列的误差变化, 中间为函数值的相对误差变化, 右边为黎曼梯度的范数变化.

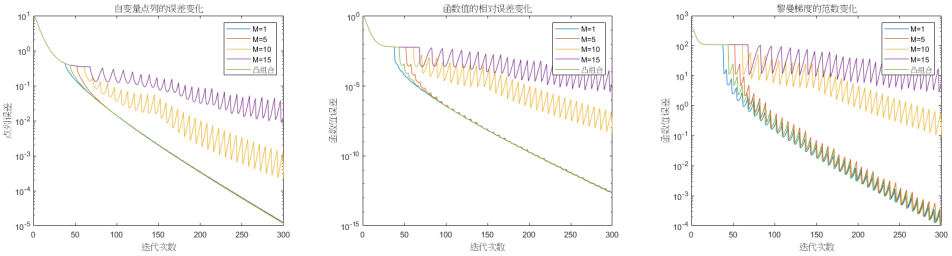

图 4: 不同参数 M 下梯度下降法的表现

从图 4 中观察到的情况如下:

- 参数 $M = 1, 5$ 以及凸组合线搜索的表现是几乎完全一致的.
- 随着 M 的继续增大, 算法的表现也越来越差.
- 随着 M 的增大, 自变量点列和函数值的误差波动也越来越剧烈.

对于第一点, 这有可能是因为本问题中的矩阵 A 是一个稀疏对称矩阵, 本身性质很好, 使得单调梯度法和小参数非单调梯度法的效果都很好, 使用非单调线搜索没有优势, 因此这些算法并没有明显的区分; 对于第二点, 这可能是因为 M 的增大导致了线搜索判别条件的加宽, 得不到想要的结果, 反而不利于算法的运行; 对于第三点, 这是符合直观的, 因为单调线搜索可以保证函数值单调下降, 但非单调线搜索有可能使得函数值短暂上升, 并且 M 越大, 函数值的波动就越剧烈, 不过其总体上是下降的.

2.4 交替使用 BB 步长的作用

本小节中, 我们探索一下交替使用 BB 步长的作用. 选取参数同表 5, 我们直接考察使用交替步长、仅使用短 BB 步长、仅使用长 BB 步长这三种情况下 BB 算法的表现情况, 结果见图 5, 左边为自变量点列的误差变化, 中间为函数值的相对误差变化, 右边为 BB 步长随迭代次数的变化.

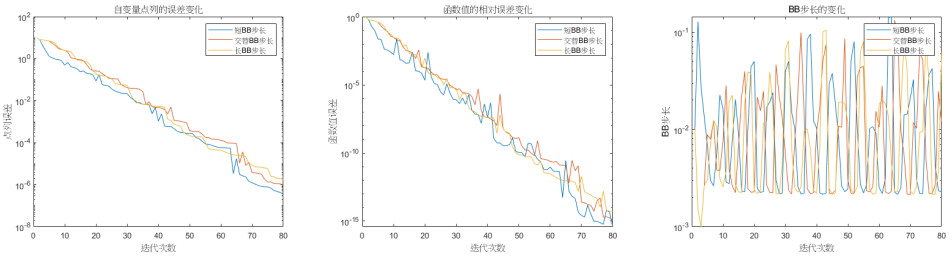

图 5: 不同 BB 步长的表现

从图 5 中可以看出, 在本问题中, 选择不同的 BB 步长并不会带来很大不同, 三种算法无论是在函数值 $f(X)$ 还是在自变量 X 的收敛速度几乎一样, 甚至连 BB 步长变化都是类似的波动.