逐步二次规划 SQP 算法

姓名：陈泽豪　学号：SA22001009

2023 年 4 月 30 日

摘要

本报告将会针对逐步二次规划 SQP 算法，给出其具体实现的理论描述以及实验结果展示，最后给出本次 SQP 算法实验的收获以及总结。

一、SQP 算法介绍

1.1 对 SQP 算法的简要流程描述

SQP（sequential quadratic programming）算法，即逐步二次规划算法，是一类典型的用来处理非线性约束最优化问题的方法。我们可以从等式约束出发推广到不等式约束，对于如下的非线性等式约束最优化问题：

$$\begin{aligned} &min \quad f(x) \\ &s.t. \quad c(x) = 0 \end{aligned} \tag{1}$$

其中 $c(x) = (c_1(x), c_2(x), ..., c_m(x))^T$ 为约束函数向量。那么写出它的 (K-T) 条件如下所示：

$$G(x) = \begin{cases} L(x,\lambda) = \nabla f(x) - A(x)^T\lambda = 0 \\ c(x) = 0 \\ A(x) = (\nabla c(x)) = (\nabla c_1(x), ..., \nabla c_m(x))^T \in \mathbf{R}^{m\times n} \end{cases} \tag{2}$$

其中 $\lambda \in \mathbf{R}^m$. 那么利用 $Newton - Raphson$ 迭代求解 (2) 对应的等式组零点便有:

$$\begin{aligned} J(G(x)) \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} &= -G(x) \\ \begin{pmatrix} W(x,\lambda) & -A(x)^T \\ -A(x) & 0 \end{pmatrix} \begin{pmatrix} \delta_x \\ \delta_\lambda \end{pmatrix} &= - \begin{pmatrix} \nabla f(x) - A(x)^T\lambda \\ -c(x) \end{pmatrix} \end{aligned} \tag{3}$$

其中有 $W(x,\lambda) = \nabla^2 L(x,\lambda) = \nabla^2 f(x) - \sum_{i=1}^m \lambda_i \nabla^2 c_i(x)$。在求解出 δ_x, δ_λ 之后更新给的初值 x, λ，并重新进行上述方程组的计算，最终利用价值函数 $\psi = \|\nabla f(x) - A(x)^T\lambda\|^2 + \|c(x)\|^2$ 作为终止判定条件。另一种思路：我们将 (3) 中的矩阵等式进行改写，可以变成如下的形式：

$$\begin{cases} W(x,\lambda)\delta_x + \nabla f(x) = A(x)^T(\lambda + \delta_\lambda) \\ c(x) + A(x)^T\delta_x = 0 \end{cases} \tag{4}$$

而我们可以进一步发现有此时的 δ_{x_k} 就是如下二次规划问题的 (K-T) 点：

$$\begin{aligned} \min_d \quad & \tfrac{1}{2} d^T W(x^{(k)}, \lambda^{(k)}) d + \nabla f(x^{(k)})^T d \\ s.t. \quad & c(x^{(k)}) + A(x^{(k)})^T d = 0 \end{aligned} \tag{5}$$

求解这个 QP 问题 dual problem，便有解 $d^{(k)}$ 满足 $d^{(k)} = \delta_{x^{(k)}}$，求解出来的 lagrange multipliers $\overline{\lambda}^{(k)}$ 满足 $\overline{\lambda}^{(k)} = \lambda^{(k)} + \delta_{\lambda^{(k)}}$。因此与上面类似的流程，每次都去更新 x, λ，并用一个价值函数进行终止判定以及步长调整。

在有了上面的等式约束 SQP 问题的铺垫之后，对于下面的不等式约束优化问题：

$$\begin{aligned} &min \quad f(x) \\ &s.t. \quad c_i(x) = 0, i \in \epsilon = \{1, ..., m_e\}, \\ &\qquad c_i(x) \geq 0, i \in I = \{m_e + 1, ..., m\}. \end{aligned} \tag{6}$$

便也是同样的处理思路，它的 lagrange 函数如下：

$$L(x,\lambda)=f(x)-\Big(\lambda_1,...,\lambda_{m_e},\lambda_{m_e+1},...,\lambda_m\Big)\begin{pmatrix}c_1(x)\\...\\c_{m_e}(x)\\c_{m_e+1}(x)\\...\\c_m(x)\end{pmatrix}=f(x)-\lambda^Tc(x) \tag{7}$$

此时矩阵等式 (4) 由于不等式约束的加入会发生变化，如下所示：

$$\begin{cases}W(x^{(k)},\lambda^{(k)})d^{(k)}+\nabla f(x^{(k)})=A(x^{(k)})^T\overline{\lambda}^{(k)}.\\ \overline{\lambda}_i^{(k)}\geq 0,i\in I.\\ c(x^{(k)})+A(x^{(k)})^Td^{(k)}=0.\\ d^{(k)}=\delta_{x^{(k)}}.\\ \overline{\lambda}^{(k)}=\lambda^{(k)}+\delta_{\lambda^{(k)}}.\end{cases} \tag{8}$$

而它就相当于求解如下的二次规划问题：

$$\begin{aligned}\min_d\quad & \tfrac{1}{2}d^TW(x^{(k)},\lambda^{(k)})d+\nabla f(x^{(k)})^Td\\ s.t.\quad & c_i(x^{(k)})+a_i(x^{(k)})d=0,i\in\epsilon\\ & c_i(x^{(k)})+a_i(x^{(k)})d\geq 0,i\in I\end{aligned} \tag{9}$$

其中有 $A(x)=(\nabla c_1(x),...,\nabla c_m(x))^T=(a_1(x),...,a_m(x))^T$，同样的，(9) 中解出来的解 $d^{(k)}$ 满足 $d^{(k)}=\delta_{x^{(k)}}$，求解出来的 lagrange multipliers $\overline{\lambda}^{(k)}$ 满足 $\overline{\lambda}^{(k)}=\lambda^{(k)}+\delta_{\lambda^{(k)}}$。

另一个与等式约束 SQP 不同的地方是，此时我们使用罚函数 $P(x,\sigma)$ 进行步长的调整，对于不等式约束问题，我们不希望下一步的 x^{k+1} 落在可行域外，因此需要使用罚项，当落在可行域外时使罚项变得很大，以此迫使 $\underset{\alpha}{argmin}P(x^{(k)}+\alpha d^{(k)},\sigma)$ 解出来的 α 满足 $x^{(k+1)}=x^{(k)}+\alpha d^{(k)}$ 落在可行域内。$P(x,\sigma)$ 如下所示定义：

$$\begin{aligned}&P(x,\sigma)=f(x)+\sigma\textstyle\sum_{i=1}^m|c_i(x)\_|\\ &c_i(x)\_=\begin{cases}c_i(x),i\in\epsilon\\ min(0,c_i(x)),i\in I\end{cases}\end{aligned} \tag{10}$$

整个算法流程如下所示：

1. 给定 $x^{(0)},\lambda^{(0)},\sigma>0,\rho\in(0,1),\epsilon\geq 0$，令 $k=0$。
2. 利用 $x^{(k)},\lambda^{(k)}$ 求出 $W(x^{(k)},\lambda^{(k)}),\nabla f(x^{(k)}),c(x^{(k)}),A(x^{(k)})$。求解子问题 (9) 得到解 $d^{(k)}$ 以及 lagrange multipliers $\overline{\lambda}^{(k)}$，如果此时 $\|d^{(k)}\|\leq\epsilon$，则停止整个过程，否则进行一维搜索:

$$\alpha_k=\underset{\alpha}{argmin}P(x^{(k)}+\alpha d^{(k)},\sigma) \tag{11}$$

3. 令 $x^{(k+1)}=x^{(k)}+\alpha_kd^{(k)},\lambda^{(k+1)}=\lambda^{(k)}+\alpha_k(\overline{\lambda}^{(k)}-\lambda^{(k)}),k=k+1$，转到 2 继续进行算法。

二、代码处理与收敛性分析

2.1 代码处理

首先需要注意的是在整个编程过程中设计到了计算 Jacobian 矩阵以及 Hessian 矩阵，在代码中采取直接使用 sympy 库的方式计算一个函数的符号梯度以及符号 Hessian 阵，并在需要使用的时候转化为数值形式方便使用，具体的代码片段处理如下所示（详细见技术文档）：

```
# 获取雅克比矩阵
def jacobi(self):
self.jac_m = self.funcs.jacobian(self.vars)
self.jac_state = True
return self.jac_m

# 返回numerical jacobi
def n_jacobi(self, x):
if not self.jac_m:
self.jacobi()
return (sy.lambdify(self.vars, self.jac_m, 'numpy'))(x[0], x[1])

# 获取海塞矩阵
def hessian(self):
self.His_m = sy.hessian(self.funcs, self.vars)
self.hes_state = True
return self.His_m

# 返回numerical hessian
def n_hessian(self, x):
if not self.hes_state:
self.hessian()
return (sy.lambdify(self.vars, self.His_m, 'numpy'))(x[0], x[1])
```

对二次规划问题直接调用 qpsolvers 的 qp_dual_solver 函数，在代码中为了使用的方便重新封装了一个 MyQPSolver 类对输入数据进行处理并求解 (9) 中所示的二次规划问题。同时需要注意的是，输入二次规划问题中的 W 矩阵应该为一个正定对称矩阵，因此在实际编码过程中需要对 W 矩阵做正定矫正，否则二次规划调用可能出现问题，不是 qpsolvers 无法运行，而是解出来的解的模会很大，最终返回 None 值。在代码中也给出了一个具体的例子，当对如下的约束问题进行处理时发现如果不加入正定矫正会使代码直接无法运行：

$$\begin{cases} f(x,y) = x^3 - y^3 + xy + 2x^2 \\ x^2 + y^2 \leq 6 \\ xy = 2 \end{cases} \tag{12}$$

取初值在 $(0.5, 0)$ 处进行实验时就会出现错误。因此需要进行正定矫正，这里我采用的方法也非常粗暴简单，因为 W 作为 Hessian 矩阵本身已经是对称阵了，因此对于不正定的 W 只需要进行如下操作：

$$W = W + (Abs(Min(Eigenvalue(W))) + \epsilon)I \tag{13}$$

其中 $\epsilon > 0$ 为一个小量，但是这样处理之后对 W 的特性改变较大，不一定还能使得解出来的 $d^{(k)}$ 为一个下降方向，所以对收敛性也许会有一定程度的影响，单就这个例子来说确实解决了无法收敛的问题并且得到了一个好的结果。对于正定的矩阵就不做操作直接使用即可。

对于一维搜索，由于上面所取的罚函数是一个不可求导函数，因此我们这里用的搜索方法也较为简单，只用到了罚函数的值进行搜索，用的是 0.618 法，具体的步骤如下所示：

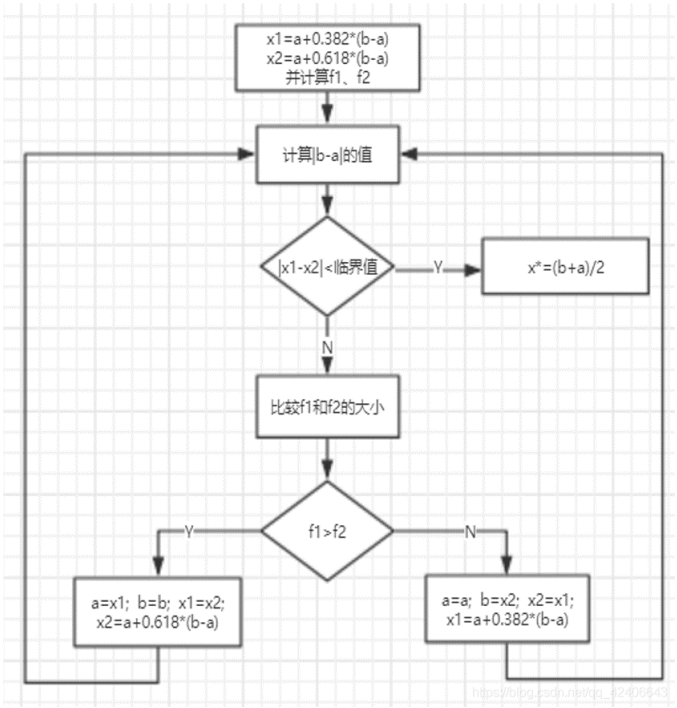

图 1: 黄金分割法示意图

2.2 收敛性分析

首先对于 SQP 算法本身,收敛性取决于初值的选取,这是因为 SQP 算法它就是对 $Newton-Raphson$ 迭代的重新定义，因此收敛性与收敛阶均与牛顿迭代法相似，具有局部收敛性而且是二阶收敛。但是需要注意的是，由于引入了对二次规划问题的调用，而且本身做了正定矫正，因此最终的收敛性并不全受算法本身的影响，也有粗糙正定矫正的影响。对于整个代码，针对无法收敛的情况直接指定了最大的循环 epoch 防止无法跳出迭代。

三、实验结果展示

在上述准备工作都完成之后，代码主体部分就放在 MySQP.my_sqp 函数中，实现如下所示：

```
# sqp算法，得到函数的极小值，这里的W直接取的lagrange函数的hessian矩阵并作一次正定矫正
def my_sqp(self, x0):
k = 0
x_k = x0.copy()
self.mysqp_intermedium_result = []   # 将迭代中间数据x_k,lambda_k均进行记录
self.mysqp_intermedium_result.append(x_k.copy())
lambda_k = np.zeros(len(self.cons_with_bounds))

# 规定循环次数
epoch = 200
for k in range(epoch):
W_k = self.Wk_lagrange_hessian(x_k, lambda_k)
g_k = self.g_k(x_k)
A_k = self.A_k(x_k)
c_k = self.c_k(x_k)

# 进行二次规划求解
W_k = self.Wk_ortho_correct(W_k)   # 做一次正定判定与矫正
qp_sol = MyQPSolver(W_k, g_k, cons=self.cons_with_bounds, cons_func_jac=A_k,
 cons_func_val=c_k)
d_k, lagrange_multiplier_k = qp_sol.qp_dual_solver()

# 如果有d_k二范数小于epi，则退出循环
if np.linalg.norm(d_k, ord=2) <= self.epi:
break

# 做一次一维搜索并更新x_k,lambda_k
alpha = self.my_linesearch(x_k, d_k)
x_k += alpha * d_k
self.mysqp_intermedium_result.append(x_k.copy())
lambda_k += alpha * (lagrange_multiplier_k - lambda_k)

# 输出
return x_k, lambda_k
```

下面进行结果展示，首先对于如下的 Rosenbrock 约束问题进行实验：

$$\begin{aligned} \underset{x_0,x_1}{min} \quad & 100(x_1 - x_0^2)^2 + (1 - x_0)^2 \\ & x_0 + 2x_1 \leq 1 \\ & x_0^2 + x_1 \leq 1 \\ & x_0^2 - x_1 \leq 1 \\ & 2x_0 + x_1 = 1 \\ & 0 \leq x_0 \leq 1 \\ & -0.5 \leq x_1 \leq 2.0 \end{aligned} \tag{14}$$

设定罚函数中的 $\sigma = 100$，整个系统的 $\epsilon = 10^{-5}$，罚函数的一维搜索查找限定范围 $\rho = 1$，如果取初值为 $x_0 = [0.5, 0]$，那么最终的迭代基本上两步就接近了精确值，如下所示：

迭代次数 k	0	1	2	3
x_k	[0.5,0]	[0.42166301 0.15667398]	[0.4149917 0.1700166]	[0.41494432 0.17011136]

表 1: Rosenbrock 实验中 SQP 每次迭代后 x_k 变化表格

此时使用 minimize 进行实验的结果为 [0.41494475 0.1701105]，具体的可视化等高线图如下所示：

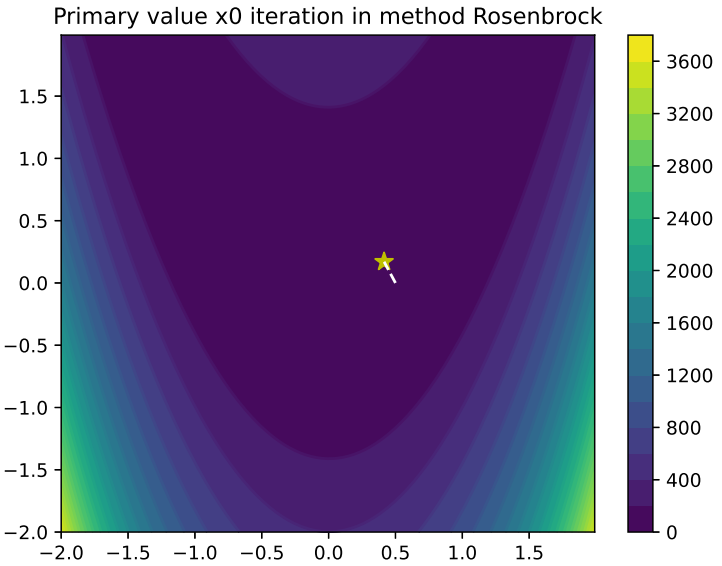

图 2: 初始点为 [0.5,0] 时的迭代结果

图上的星形表示 minimize 的结果，白线就是我的程序从 [0.5,0] 出发接近最终结果的过程。

当然只给出这一个初值结果比较没有说服力，下面又给出了多个初值的迭代结果，取了如下几个初值：$x_0 = [-1.5, -1.], [-1., 1.], [1.0, 0.5], [0.5, -1.5], [1.5, -1.]$，并在图上一一画出了迭代结果，如下图 3 所示：

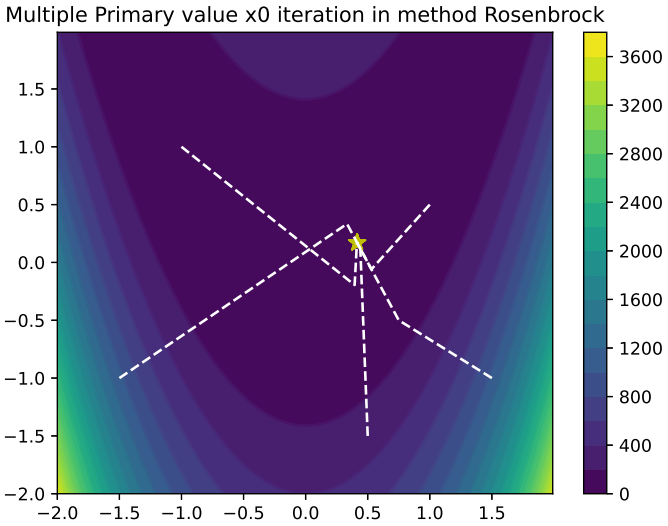

图 3: 多个初始点的迭代结果

下面再给出另一个例子的测试结果，与 Rosenbrock 的结果不同，这一个例子会因为小范围内初值的选取不同而迭代到不同的点出，例子便是上文 (12) 提到的：

$$\begin{cases} f(x,y) = x^3 - y^3 + xy + 2x^2 \\ x^2 + y^2 \leq 6 \\ xy = 2 \end{cases} \tag{15}$$

它的结果应当是在 [0.87403205 , 2.28824561] 处，取值为-7.78584，但是即使是 minimize 函数，在我们取迭代初始值为 [2,-1.5] 时，它会直接收敛到另一个点 [-2.28824561 , -0.87403205] 处，取值为 1.15843，因此初值的选取会极大程度影响最终能否收敛到极小点处。首先给出我的代码在初值为 [0.5,0] 时收敛图：

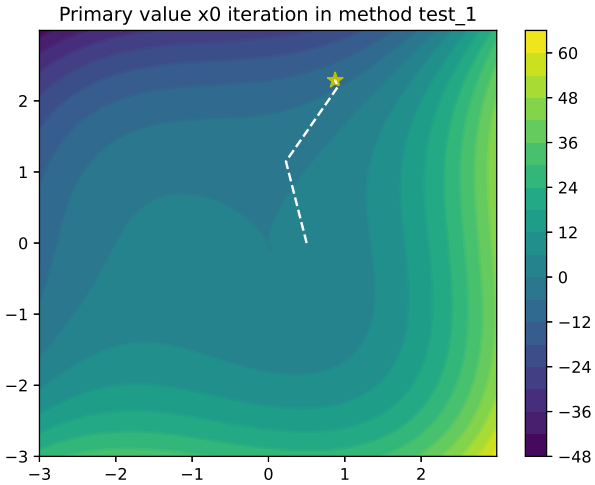

图 4: 初始点为 [0.5,0] 的迭代结果

具体的迭代表如下所示：

迭代次数 k	0	1	2	3	4
x_k	[0.5,0]	[0.22876 1.14781]	[0.9097 2.19847]	[0.8723 2.28891]	[0.874031 2.288248]

表 2: test_1 func 实验中 SQP 每次迭代后 x_k 变化表格

同时也可以针对不同的初值给出 minimize 的结果以及我的 sqp 函数迭代过程，初值依然选取：$x_0 = [-1.5, -1.], [-1., 1.], [1.0, 0.5], [0.5, -1.5], [1.5, -1.]$，则迭代结果如下所示：

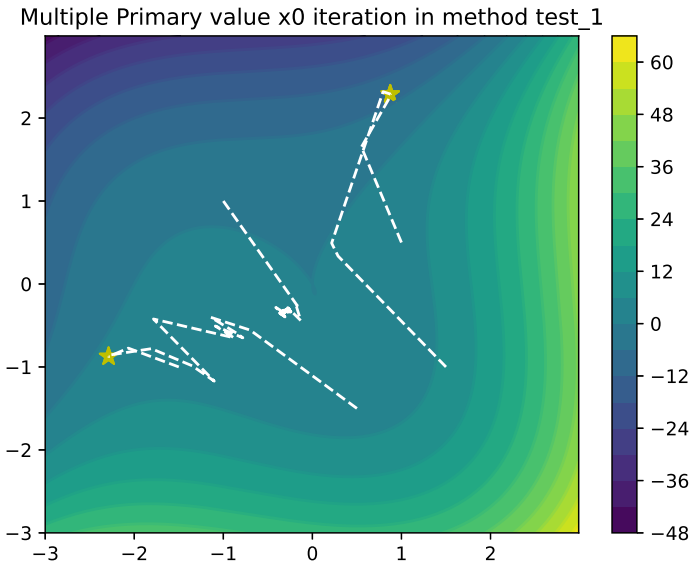

图 5: 多个初始点的迭代结果

可以发现对我的算法来说，存在 [-1,1] 是无法收敛成功的初值，这可能与我对正定矫正的粗糙处理有关，而 [1.5,-1],[1,0.5] 均可以收敛至 [0.87403205 , 2.28824561] 处，[0.5,-1.5],[-1.5,-1] 的初值则会与 minimize 函数一样收敛至 [-2.28824561 , -0.87403205] 处。

四、实验总结

本次实验尝试编写了 SQP 算法，对非线性约束最优化问题进行了处理，对如何使用 (K-T) 条件有了更深刻的认识，并且了解了整个优化问题大概的一个编写思路，收获很大。在矫正时也想过 BFGS 矫正，但是感觉我了解的 BFGS 是利用前一次的 hessian 近似矩阵 W 去推出这一次的 W，但是本次实验 W 是通过求解出每次迭代步骤的 $x^{(k)}$ 与 $\lambda^{(k)}$ 后求出的，不会用到上一次的 W 信息，因此只用了最暴力的方法进行正定矫正；同时针对一维搜索的非光滑罚函数最小值寻找也可以做的再好一些。