STEEPEST GRADIENT DESCENT

1 问题描述

用线搜索方法求解 n 维 Rosenbrock 函数的最小值点。

$$f(x)=f(x_1,x_2,\cdots,x_n)=\sum_{i=1}^{\frac{N}{2}}\left[100(x_{2i-1}-x_{2i})^2+(x_{2i}-1)^2\right]$$

2 问题分析

2.1 线搜索方法

线搜索方法的即给定当前迭代点 x^k，以函数在当前点梯度的负方向作为下降方向 $d_k=-\nabla f(x^k)$，按照一定的步长 α_k 确定下一个迭代点。

$$x^{k+1}=x^k+\alpha_k d^k$$

线搜索的目标是选择合适的 α_k，使得新迭代点处的函数值 $\phi(\alpha)$ 尽可能减小，即使函数在迭代过程充分下降。

$$\phi(\alpha)=f(x^k+\alpha_k d^k)$$

所以此时的问题就变成了寻找函数 $\phi(\alpha)$ 的极小值点，获得最佳步长：

$$\alpha_k=\arg\min_{\alpha>0}\phi(\alpha)$$

2.2 Armijo 准则

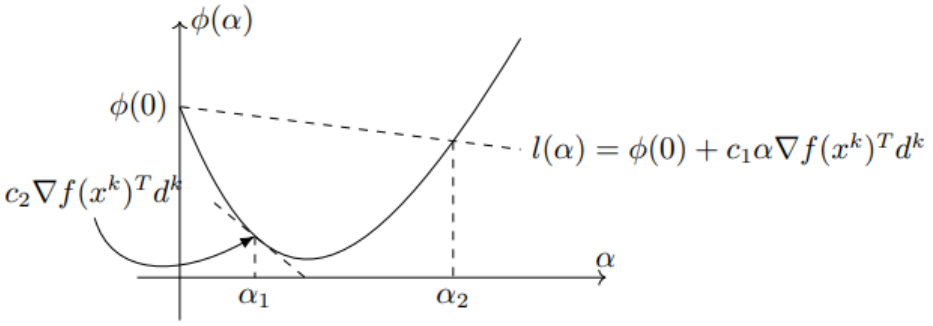
然而求解最佳步长的过程费时费力，虽然可以令迭代过程可以充分下降，但每一步迭代都变成了一个一维的优化问题，使得每次迭代会消耗大量时间。为了权衡迭代次数与每次迭代的计算量，常使用非精确线搜索方法，利用 Armijo 准则以较小的计算量则确定一个次优的迭代步长。

$$f(x^k)-f(x^k-\alpha_k d^k)\geq -c\cdot\alpha_k\nabla f(x^k)^T d^k$$
$$c\in(0,1)$$

Armijo 准则也成为充分下降条件，其几何意义为：

- 沿着搜索方向 d 将多维函数截出一个一维函数，显然，只要 $\phi(\alpha)<\phi(0)$ 就可以实现梯度下降，但为了让函数尽可能充分的下降，需要规定一个更严苛的取值区间，将 $\phi(0)$ 处的斜率乘以一个系数 $c\in(0,1)$ 松弛一下，得到直线 $l(\alpha)$，只要新的 $\phi(\alpha)$ 处在直线下方，即认为步长 α 满足要求。

- 但也并非区间内的所有步长都是我们想要的，我们希望步长尽可能靠近最低点的右侧，避免步长过小使得下降速度缓慢。



具体的计算步骤为：

1. 设置初始步长 $\tau = 1$
2. 若步长不满足 Armijo Condition，则 $\tau = \tau/2$，直到 τ 满足 Armijo Condition 为止
3. 更新步长 $\alpha^k = \tau$

3 代码实现

3.1 二维的 Rosenbrock 函数

为了方便可视化，也为了验证代码的可行性，先拿二维的 Rosenbrock 函数开刀，完成最基本的任务要求：

$$f(x_0, x_1) = 100(x_0^2 - x_1)^2 + (x_0 - 1)^2$$

$$\nabla f(x_0, x_1) = \begin{bmatrix} 200(x_0^2 - x_1) \times 2x_0 + 2(x_0 - 1) \\ -200(x_0^2 - x_1) \end{bmatrix}$$

在代码中，则可以用数组来表示矩阵，完整的代码如下：

PYTHON

```python
import numpy as np
import matplotlib.pyplot as plt

'''
计算 Rosenbrock 函数的值
'''
def Rosenbrock(x):
    return 100*(x[0]**2.0 - x[1])**2.0 + (x[0] - 1)**2

'''
计算 Rosenbrock 函数的梯度
'''
```

```python
def RosenbrockGradient(x):
    gradX1 = 400 * x[0] * (x[0]**2 - x[1]) + 2*(x[0] - 1)
    gradX2 = -200 * (x[0]**2 - x[1])
    grad = np.array([gradX1, gradX2])
    return grad

'''
用 Armijo Condition 计算步长
'''
def Armijo(x, grad):
    c = 0.1                          # 松弛因子
    tau = 1                          # 初始步长
    x1 = x[0] - tau * grad[0]
    x2 = x[1] - tau * grad[1]
    nextX = np.array([x1, x2])       # 初始步长对应的迭代点

    while Rosenbrock(nextX) > Rosenbrock(x) + (c * tau) * np.dot(grad,
grad):
        tau *= 0.5                   # 步长减半
        x1 = x[0] - tau * grad[0]
        x2 = x[1] - tau * grad[1]
        nextX = np.array([x1, x2]) # 更新迭代点

    alpha = tau
    return alpha

'''
使用线搜索梯度下降法求解函数极小值
'''
def LineSearch(x0):
    pointList = x0   # 用于记录迭代点

    iter = 1
    maxIter = 5000   # 最大迭代次数

    x = x0           # 初始迭代点
    error = 10       # 初始误差
    tolerance = 0.01 # 误差允许范围

    while (iter < maxIter) and (error > tolerance):
        grad = RosenbrockGradient(x)
        error = np.linalg.norm(grad)
```

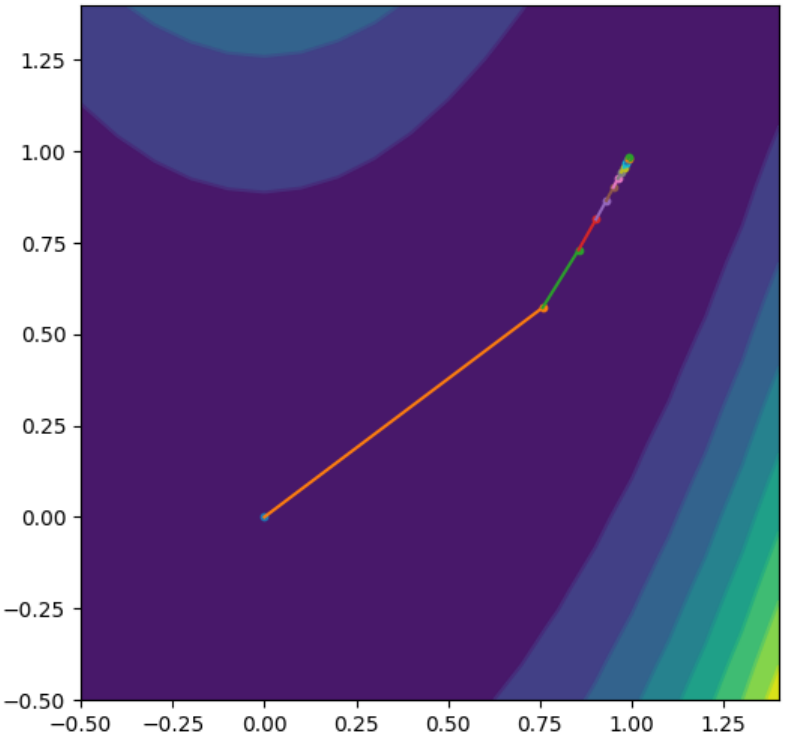
```
        alpha = Armijo(x, grad)

        X0 = x[0] - alpha * grad[0]
        X1 = x[1] - alpha * grad[1]
        x = np.array([X0, X1])                    # 更新迭代点

        iter += 1
        pointList = np.row_stack((pointList, x)) # 记录历史迭代点

        print("Iteration: ", iter, ", Error", error, ", Local Minimum:
", x)

    return x, pointList

if __name__ == '__main__':
    x0 = np.array([0, 0])
    globalMinimum, pointList = LineSearch(x0)

    # 开始画图
    x = np.arange(-0.5, 1.5, 0.1)
    y = np.arange(-0.5, 1.5, 0.1)
    X, Y = np.meshgrid(x, y)
    Z = 100*(X**2.0 - Y)**2.0 + (X - 1)**2

    plt.figure(figsize=(6, 6))
    plt.contourf(X, Y, Z)
    plt.contour(X, Y, Z)

    lastI = 0
    for i in range(pointList.shape[0] - 1):

        if i % 300 == 0:
            plt.scatter(pointList[i, 0], pointList[i, 1], s=10)

            xAxis = np.array([pointList[lastI,0], pointList[i,0]])
            yAxis = np.array([pointList[lastI,1], pointList[i,1]])
            lastI = i

            plt.plot(xAxis, yAxis)
            plt.pause(0.1)
```

```
    plt.show()
```

代码中设置初始迭代点为 $[0, 0]$，初始步长为 1，通过 Armijo Condition 对步长进行二分，得到次优的下降步长，经过 3661 次迭代得到了精度小于 0.01 的全局最小值：

```
Iteration:  1 , Error 2 , Local Minimum:  [0.25 0.]
...
...
Iteration:  3659 , Error 0.01062708552340646 , Local Minimum:
[0.99260635 0.98525505]
Iteration:  3660 , Error 0.010198070976353916 , Local Minimum:
[0.99262568 0.98525986]
Iteration:  3661 , Error 0.00980877219568692 , Local Minimum:
[0.99261214 0.9852957 ]
```

函数的运行结果如下，彩色的线段表征了函数迭代的方向。由图可知，Armijo Condition 提供了行之有效的搜索步长，结果喜人。



3.2 N 维的 Rosenbrock 函数

有了一维的经验，N 维的 Rosenbrock 就很好搞定了。唯一的区别就在于矩阵的维度更大，并且需要引入 `for` 循环来计算矩阵中每一个元素的值：

$$f(x) = f(x_1, x_2, \cdots, x_n) = \sum_{i=1}^{\frac{N}{2}} \left[100(x_{2i-1} - x_{2i})^2 + (x_{2i} - 1)^2\right]$$

$$\nabla f(x) = \begin{bmatrix} 200(x_1^2 - x_2) \times 2x_1 + 2(x_1 - 1) \\ \vdots \\ -200(x_{2i-1}^2 - x_{2i}) + 200(x_{2i}^2 - x_{2i+1}) \times 2x_{2i-1} + 2(x_{2i-1} - 1) \\ \vdots \\ -200(x_{N-1}^2 - x_N) \end{bmatrix}$$

```
// TODO:
// 周末用 C++ 改写上面的 Python 代码，用 Matplotlib-cpp 画图
// 再写一个求解 n 维 Rosenbrock 的类，可以自定义 n 的那种
// 再补充一个阻尼牛顿法的代码
```