# 梯度下降与牛顿迭代

## 问题描述

对函数$f(x) = \sum_{i=1}^{5}\left[(1-x_i)^2 + 100(x_{i+1} - x_i^2)^2\right]$，初始点取为$x_0 = (0,0,\ldots,0) \in \mathbb{R}^6$，分别用最速下降法和牛顿编程迭代10次，结果记录：

| 迭代次数$k$ | 最速下降法 | 牛顿法 |
|---|---|---|
| 1 | 4.94211464 | 4.94211464 |
| 2 | 4.89815621 | 4.63350489 |
| 3 | 4.81260383 | 4.52764763 |
| 4 | 4.63223753 | 4.52086354 |
| 5 | 4.48031619 | 4.46103915 |
| 6 | 4.36349505 | 4.45437617 |
| 7 | 4.30909567 | 4.09120261 |
| 8 | 4.26579133 | 4.00735888 |
| 9 | 4.22601659 | 3.85641375 |
| 10 | 4.16691051 | 3.85707386 |
| 10 次迭代两个方法 | | |
| 所花时间 | 1.449毫秒 | 2.005毫秒 |

## 分析

根据提供的计算结果，我们可以分析最速下降法和牛顿法在求解优化问题时的性能差异：

## 收敛性能

- **最速下降法** 的函数值从初始的 4.94211464 逐渐降低到 4.16691051。这显示了最速下降法能够稳定地减少函数值，但下降速度较慢，这与最速下降法依赖于梯度方

向作为搜索方向的特性一致。由于梯度方向并不总是指向最陡的下降方向，特别是在复杂的多维空间中，这可能导致路径出现“之字形”并减慢收敛速度。
- **牛顿法** 的函数值从初始的 4.94211464 快速下降至 3.85707386。牛顿法在前几次迭代中迅速减少函数值，显示出比最速下降法更快的收敛速度。这是因为牛顿法考虑了函数的二阶导数信息（即Hessian矩阵），能够更准确地估计函数的局部曲率，从而在每一步选择更优的下降方向和步长。然而，值得注意的是，在第9次和第10次迭代中，函数值略有上升，这可能是由于Hessian矩阵的逆计算不准确或者步长选择不当导致的。

## 时间开销

- **最速下降法** 花费了 1.449 毫秒完成 10 次迭代，而 **牛顿法** 花费了 2.005 毫秒。尽管牛顿法在每次迭代中要计算更多的信息（如Hessian矩阵及其逆），这自然导致了更高的计算成本，但从总体收敛速度来看，牛顿法仍然是更高效的选择。这表明牛顿法在求解此类优化问题时，尽管单次迭代的时间成本更高，但由于迭代次数少，总体上可能更有效率。

## 结论

- **收敛速度**：牛顿法明显快于最速下降法，这反映了利用二阶导数信息能够更有效地指导搜索过程的优势。
- **稳定性**：牛顿法在迭代过程中可能会遇到稳定性问题，如函数值的小幅回升，这需要通过适当的步长控制和Hessian矩阵的正定性调整来解决。
- **计算成本**：虽然牛顿法的单次迭代成本更高，但其较快的收敛速度可能在许多情况下抵消了这一点，尤其是在需要快速收敛到高精度解的应用中。

## 建议

- 对于复杂或高维的优化问题，可以考虑使用**拟牛顿法**（如BFGS或L-BFGS），这些方法试图平衡牛顿法的快速收敛性和计算Hessian矩阵的高成本。
- 在使用牛顿法时，应当注意Hessian矩阵的正定性和逆的准确性，必要时可以采用正则化或修改Hessian矩阵的方法来提高稳定性。
- 动态调整步长（如通过线搜索）对于加速收敛和保证算法稳定性是非常重要的，尤其是在牛顿法中。

# 运行源代码

```python
import numpy as np
import time
from scipy.optimize import line_search


# 定义目标函数
def f(x):
    return sum((1 - x[i]) ** 2 + 100 * (x[i + 1] - x[i] ** 2) ** 2 for i in range(5))


# 定义梯度
def grad_f(x):
    grad = np.zeros_like(x)
    for i in range(5):
        if i < 4:
            grad[i] += -2 * (1 - x[i]) - 400 * x[i] * (x[i + 1] - x[i] ** 2)
            grad[i + 1] += 200 * (x[i + 1] - x[i] ** 2)
        else:
            grad[i] += -2 * (1 - x[i])
    return grad


# 定义Hessian
def hessian_f(x):
    hess = np.zeros((6, 6))
    for i in range(5):
        hess[i][i] += 2 + 1200 * x[i] ** 2 - 400 * x[i + 1]
        if i < 4:
            hess[i][i + 1] = -400 * x[i]
            hess[i + 1][i] = -400 * x[i]
            hess[i + 1][i + 1] += 200
    return hess


# 最速下降法
def steepest_descent(x0, max_iter=10):
    x = x0.copy()
    f_values = []
    times = []
```

```
    start_time = time.time()

    for k in range(max_iter):
        grad = grad_f(x)
        alpha = line_search(f, grad_f, x, -grad)[0]  # 使用线搜索确定步长
        if alpha is None:
            alpha = 0.001  # 如果线搜索失败，使用默认步长
        x -= alpha * grad
        f_values.append(f(x))
        times.append(time.time() - start_time)

    return f_values, times


# 牛顿法
def newton_method(x0, max_iter=10):
    x = x0.copy()
    f_values = []
    times = []
    start_time = time.time()
    B = np.eye(len(x0))  # 初始化近似Hessian矩阵

    for k in range(max_iter):
        grad = grad_f(x)
        p = -np.linalg.solve(B, grad)  # 使用近似Hessian矩阵求解方向
        alpha = line_search(f, grad_f, x, p)[0]  # 使用线搜索确定步长
        if alpha is None:
            alpha = 0.001  # 如果线搜索失败，使用默认步长
        s = alpha * p  # 计算步长
        x_new = x + s
        y = grad_f(x_new) - grad  # 计算梯度差
        rho = 1 / (y @ s)
        B += rho * np.outer(y, y) - rho * B @ np.outer(s, s) @ B  # 更新近似Hessian矩阵
        x = x_new
        f_values.append(f(x))
        times.append(time.time() - start_time)

    return f_values, times
```

```python
# 初始化
x0 = np.zeros(6)
# 最速下降法
start_time_sd = time.time()
f_values_sd, times_sd = steepest_descent(x0)
time_sd = time.time() - start_time_sd

# 牛顿法
start_time_nm = time.time()
f_values_nm, times_nm = newton_method(x0)
time_nm = time.time() - start_time_nm

# 打印结果表格
print("迭代次数 k\t最速下降法\t牛顿法\t\t最速下降法时间\t牛顿法时间")
for k in range(10):
    print(f"{k + 1}\t\t{f_values_sd[k]:.8f}"
          f"\t{f_values_nm[k]:.8f}"
          f"\t{times_sd[k]:.4f}\t\t{times_nm[k]:.4f}")

# 打印总耗时
print(f"\n最速下降法总耗时: {time_sd:.6f} 秒")
print(f"牛顿法总耗时: {time_nm:.6f} 秒")
```