第二次作业报告

本次作业主要完成的是最速下降法和牛顿迭代法，及其分别基于回溯法和Wolfe条件的迭代步长选择实现。

最优化算法设计

由于我们希望最速下降法和牛顿迭代法这两种最优化算法能接收不同的目标函数以及步长选择准则，所以我设计的两个函数定义如下：

```
% 最速下降法
function [new_x, fx, count] = SteepestDescent(f, sl_f, x0, tol)

% 牛顿法
function [new_x, fx, count] = Newton(f, sl_f, x0, tol)
```

其中输入f 为目标函数，本例中即Rosenbrock函数，sl-f为步长选择方式，本例中可选回溯法和基于Wolfe条件的方法，x0为初始点，即从何处开始迭代，tol为迭代门限，即最大迭代次数。输出new_x为最终找到的最优函数值处自变量取值，fx为最优函数值，count为实际迭代次数。

由于最速下降法迭代需要求解梯度，而牛顿法更进一步还需要Hessian矩阵，所以设计Rosenbrock函数如下：

```
% 目标函数
function [y, df, ddf] = Rosenbrock(x)
```

分别返回在点x处的函数值，梯度值以及Hessian矩阵，由于此处并没有批量计算的需求，所以函数输入仅为一个$m*1$的向量，即仅计算一个点。

牛顿法和最速下降法可以统一表示为如下形式：

$$\text{更新下降方向：}\ p_k = -B_k^{-1}\nabla f_k$$

$$\text{更新位置：}\ x_{k+1} = x_k + \alpha_k p_k$$

区别在于最速下降法中$B_k = I$，而牛顿法中$B_k = \nabla^2 f(x_k)$，所以可以看到我实现的两个优化算法除了迭代方向的计算略微有区别之外，基本流程一致，具体实现请见附件。

还有一部分很重要的是关于步长选择函数的实现，即sl_f。此处实现了回溯法和Wolfe条件两种，函数定义分别如下：

```
% 回溯法
function alpha = BackTracking(x, p, f)
```

输入x为当前位置，p为迭代方向，f为目标函数，返回值alpha即为满足条件的步长。

参考教材算法3.1，有如下实现：

```
[~, df, ~] = f(x);
while f(x + alpha * p) > f(x) + c * alpha * df' * p
    alpha = alpha * rho;
end
```

参数α值设置为1，c设置为1e-4，ρ设置为0.5。

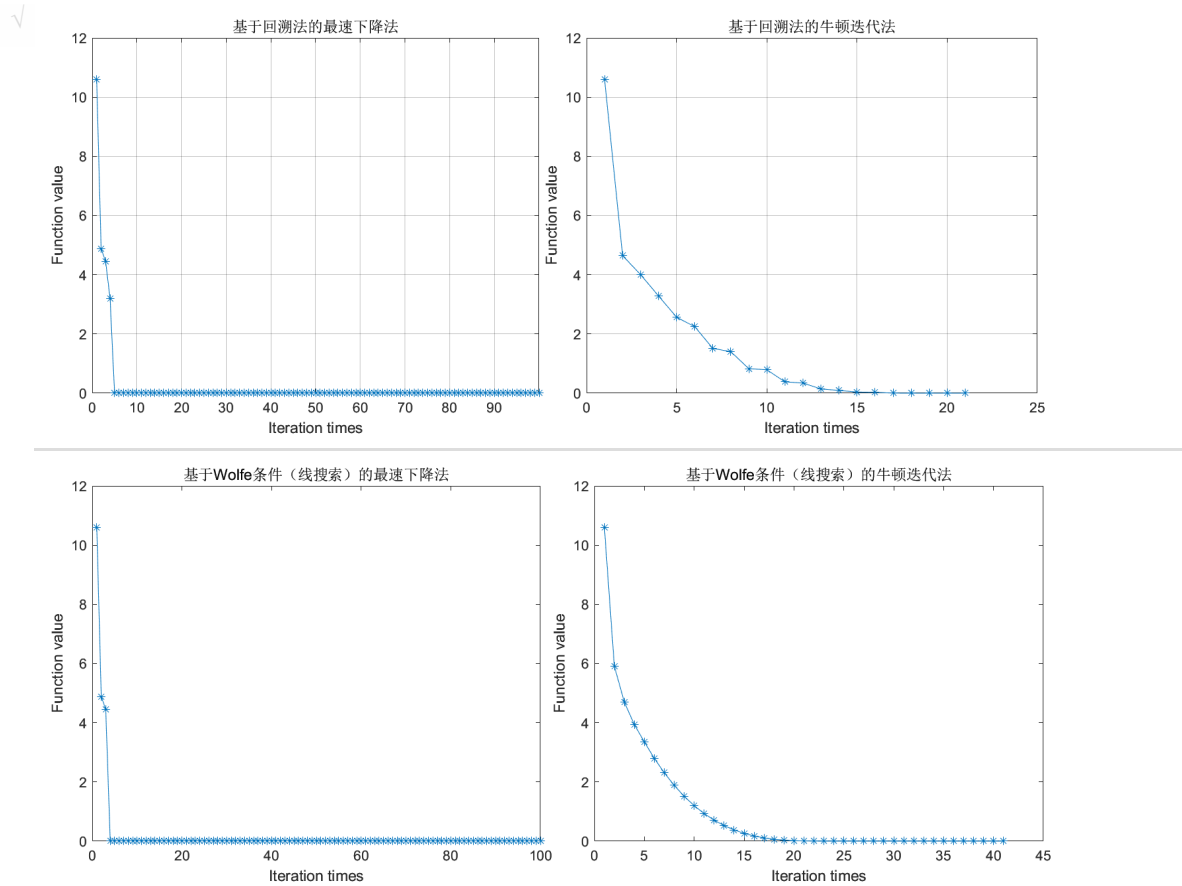
Wolfe条件的实现略微复杂，此处略去，参考书上算法3.5和3.6，定义函数如下：

```
% Wolfe条件
function alpha = Wolfe(x, p, f)
```

输入输出定义同回溯法。主要参数设置有$c_1 = 1e-4, c_2 = 0.5, \alpha_0 = 0, \alpha_{max} = 1$。

测试对比

如上所述，总共实现了两种最优化算法和两种步长选择方式，可以任意搭配。为了方便比较，测试过程中选址2元Rosenbrock函数，易得全局最小值位于(1, 1)处，函数值为0，设计迭代初值为(-1.2, 1.2)，最大迭代次数为100，并且当前后两次迭代求得函数值之差小于10*eps时认为已找到最优值，可跳出迭代。测试结果如下图所示：



从测试结果可以看到牛顿法总是比最速下降法更快更好地收敛到期望的最优函数值，这也与我们的预期结果一致。另外基于Wolfe条件（线搜索）步长选择的最优化算法看起来比基于回溯法的收敛更慢一些，虽然收敛过程更加平稳。我推测是由于回溯法总是倾向于选择较大的迭代步长导致的，也有可能与我实现代码中的参数选择有关系。代码中也实现了两种优化算法分别基于回溯法Wolfe条件的结果，收敛效率与普通回溯法，即只包含有效下降条件的结果基本一致。

上述测试结果在main.mlx脚本中同样给出，有也有其他一些更详细的测试结果。