DFP算法

DFP（Davidon-Fletcher-Powell）算法是一种用于多元函数优化的迭代方法，属于拟牛顿法的一种。该算法通过构建目标函数的二次模型来近似Hessian矩阵的逆，以此找到函数的局部最小值。

DFP算法的基本步骤如下：

1. 选择一个初始点 x_0 和一个初始矩阵 H_0 ，通常 H_0 是单位矩阵。
2. 对于第 k 次迭代：
    - 计算梯度 $g_k = \nabla f(x_k)$ 。
    - 如果 g_k 足够小，则停止迭代。
    - 计算搜索方向 $p_k = -H_k g_k$ 。
    - 使用线搜索找到使 $f(x_k + \alpha p_k)$ 最小化的 α 。
    - 更新 x 的值：$x_{k+1} = x_k + \alpha_k p_k$。
    - 计算 $g_{k+1} = \nabla f(x_{k+1})$并计算 $y_k = g_{k+1} - g_k$。
    - 计算 H 的更新值：$H_{k+1} = H_k + \frac{\Delta x_k \Delta x_k^T}{\Delta x_k^T y_k} - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k}$，其中 $\Delta x_k = x_{k+1} - x_k$。
3. 重复步骤2直到满足停止准则。

编程实现如下：

```
import numpy as np
# DFP Algorithm
def dfp(f, grad_f, x0, max_iter=100, epsilon=1e-5):
    n = len(x0)
    Hk = np.eye(n)
    xk = np.array(x0)

    for _ in range(max_iter):
        # Calculate gradient
        gk = grad_f(xk)

        # Stopping condition
```

```
        if np.linalg.norm(gk) < epsilon:
            break

        # Search direction
        pk = -np.dot(Hk, gk)

        # Line search (using backtracking here)
        alpha = 1
        while f(xk + alpha * pk) > f(xk) + 0.1 * alpha * np.dot(gk, 
pk):
            alpha *= 0.5

        # Update x
        xk_new = xk + alpha * pk

        # Update Hk
        sk = xk_new - xk
        yk = grad_f(xk_new) - gk
        rho = 1.0 / (yk @ sk)
        Hk_new = (np.eye(n) - rho * np.outer(sk, yk)) @ Hk @ 
(np.eye(n) - rho * np.outer(yk, sk)) + rho * np.outer(sk, sk)

        # Update for next iteration
        xk = xk_new
        Hk = Hk_new

    return xk
```

BFGS算法

BFGS 算法（Broyden-Fletcher-Goldfarb-Shanno）是拟牛顿法中的一种，它用于解决无约束的非线性优化问题。BFGS 算法通过迭代更新近似 Hessian 矩阵的逆，以此来寻找多元函数的局部最小值。

BFGS 算法的步骤与 DFP 算法类似，但是更新 Hessian 矩阵的逆的公式不同：

1. 选择一个初始点 x_0 和一个初始近似矩阵 B_0，通常 B_0 是单位矩阵。
2. 对于第 k 次迭代：

- 计算梯度 $g_k = \nabla f(x_k)$。
- 如果 g_k 足够小，则停止迭代。
- 计算搜索方向 $p_k = -B_k^{-1} g_k$。
- 使用线搜索找到使 $f(x_k + \alpha p_k)$ 最小化的 α。
- 更新 x 的值：$x_{k+1} = x_k + \alpha_k p_k$。
- 计算 $g_{k+1} = \nabla f(x_{k+1})$ 并计算 $y_k = g_{k+1} - g_k$。
- 计算 $s_k = x_{k+1} - x_k$。
- 使用 BFGS 公式更新 B 的逆的近似：B_{k+1}^{-1}。

3. 重复步骤2直到满足停止准则。

BFGS 迭代公式如下：

$$B_{k+1}^{-1} = (I - \rho_k s_k y_k^T) B_k^{-1} (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T$$

其中 $\rho_k = \frac{1}{y_k^T s_k}$。

编程实现如下：

```
import numpy as np
# BFGS Algorithm
def bfgs(f, grad_f, x0, max_iter=100, epsilon=1e-5):
    n = len(x0)
    Bk_inv = np.eye(n)
    xk = np.array(x0)

    for _ in range(max_iter):
        # Calculate gradient
        gk = grad_f(xk)

        # Stopping condition
        if np.linalg.norm(gk) < epsilon:
            break

        # Search direction
        pk = -Bk_inv @ gk
```

```
        # Line search (using backtracking here, but other methods can be used)
        alpha = 1
        while f(xk + alpha * pk) > f(xk) + 0.1 * alpha * np.dot(gk, pk):
            alpha *= 0.5

        # Update x
        xk_new = xk + alpha * pk

        # Compute s_k and y_k
        sk = xk_new - xk
        yk = grad_f(xk_new) - gk
        rho_k = 1.0 / (yk @ sk)

        # Update Bk_inv using BFGS formula
        I = np.eye(n)
        Bk_inv = (I - rho_k * np.outer(sk, yk)) @ Bk_inv @ (I - rho_k * np.outer(yk, sk)) + rho_k * np.outer(sk, sk)

        # Update for next iteration
        xk = xk_new

    return xk
```

FR算法

FR 算法是共轭梯度法中的 Fletcher-Reeves 算法。它是一种用于求解大规模线性和非线性问题的迭代优化算法。FR 算法对于无约束优化问题特别有效，特别是当目标函数为二次型时。

Fletcher-Reeves 算法的基本步骤如下：

1. 选择一个初始点 x_0 并计算 $g_0 = \nabla f(x_0)$。
2. 设定 $p_0 = -g_0$（第一个搜索方向是梯度的负方向）。
3. 对于 $k = 0, 1, 2, \ldots$，直到满足停止准则：
    - 使用线搜索确定步长 α_k 使得 $f(x_k + \alpha_k p_k)$ 最小化。

    - 更新 x 的值：$x_{k+1} = x_k + \alpha_k p_k$。
    - 计算 $g_{k+1} = \nabla f(x_{k+1})$。
    - 如果 g_{k+1} 足够小，则停止迭代。
    - 使用 FR 公式更新搜索方向：$p_{k+1} = -g_{k+1} + \beta_k p_k$，其中 $\beta_k = \frac{\|g_{k+1}\|^2}{\|g_k\|^2}$。
4. 重复步骤3直到满足停止准则。

编程实现如下：

```python
import numpy as np
# Fletcher-Reeves (FR) Conjugate Gradient Algorithm
def fr_conjugate_gradient(f, grad_f, x0, max_iter=1000, epsilon=1e-5):
    xk = np.array(x0)
    gk = grad_f(xk)
    pk = -gk
    for _ in range(max_iter):
        # Line search to find alpha_k
        alpha_k = line_search(f, grad_f, xk, pk)

        # Update xk
        xk_new = xk + alpha_k * pk

        # Calculate new gradient
        gk_new = grad_f(xk_new)

        # Check convergence
        if np.linalg.norm(gk_new) < epsilon:
            break

        # Fletcher-Reeves update
        beta_k = np.dot(gk_new, gk_new) / np.dot(gk, gk)

        # Update pk
        pk = -gk_new + beta_k * pk

        # Update for next iteration
        xk = xk_new
```

```
        gk = gk_new

    return xk

# Line search function to find an appropriate alpha using
backtracking
def line_search(f, grad_f, xk, pk, alpha=1, rho=0.5, c=1e-4):
    while f(xk + alpha * pk) > f(xk) + c * alpha *
np.dot(grad_f(xk), pk):
        alpha *= rho
    return alpha
```

具体算例

使用Rosenbrock函数来测试上述三个算法，这是一个非凸函数，通常用于测试优化算法的性能。

Rosenbrock函数定义为：

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

其中常数 a 和 b 通常设为 $a = 1$ 和 $b = 100$，函数的全局最小值在 $(x, y) = (a, a^2)$。

使用上述三个算法，测试了 Rosenbrock 函数不同 a 值（从 1 到 10）的最小值点。以下是每个 a 值对应的最小值点 (x, y)：

- $a = 1$: $(1, 1)$
- $a = 2$: $(2, 4)$
- $a = 3$: $(3, 9)$
- $a = 4$: $(4, 16)$
- $a = 5$: $(5, 25)$
- $a = 6$: $(6, 36)$
- $a = 7$: $(7, 49)$
- $a = 8$: $(8, 64)$
- $a = 9$: $(9, 81)$
- $a = 10$: $(10, 100)$

对于DFP算法，设计如下测试：

```
# Define the Rosenbrock function
def rosenbrock(x, a=1, b=100):
    return (a - x[0])**2 + b*(x[1] - x[0]**2)**2

# Define the gradient of the Rosenbrock function
def grad_rosenbrock(x, a=1, b=100):
    return np.array([-2*(a - x[0]) - 4*b*x[0]*(x[1] - x[0]**2),
                     2*b*(x[1] - x[0]**2)])
# Initial guess
x0 = np.array([0.0, 0.0])
rosenbrock_results = {}
for a in range(1, 11):
    result = dfp(lambda x: rosenbrock(x, a=a), lambda x:
grad_rosenbrock(x, a=a), x0)
    rosenbrock_results[a] = result

rosenbrock_results
```

运行结果：

```
{1: array([1., 1.]),
 2: array([2., 4.]),
 3: array([3.        , 9.00000002]),
 4: array([ 3.99999989, 15.99999915]),
 5: array([ 5., 25.]),
 6: array([ 5.99999999, 35.99999992]),
 7: array([ 7.        , 49.00000003]),
 8: array([ 7.99999988, 63.99999815]),
 9: array([ 9., 81.]),
 10: array([ 10., 100.])}
```

对于每个 a 的值，算法成功找到了对应的全局最小值点 (a, a^2)。这些结果验证了DFP算法在这个问题上的有效性。

对于BFGS算法，设计如下测试：

```
# Define the Rosenbrock function
def rosenbrock(x, a=1, b=100):
    return (a - x[0])**2 + b*(x[1] - x[0]**2)**2
```

```python
# Define the gradient of the Rosenbrock function
def grad_rosenbrock(x, a=1, b=100):
    return np.array([-2*(a - x[0]) - 4*b*x[0]*(x[1] - x[0]**2),
                     2*b*(x[1] - x[0]**2)])
# Initial guess
x0 = np.array([0.0, 0.0])
# Test the BFGS algorithm on Rosenbrock function with different
values of a (from 1 to 10)
bfgs_rosenbrock_results = {}
for a in range(1, 11):
    result = bfgs(lambda x: rosenbrock(x, a=a), lambda x:
grad_rosenbrock(x, a=a), x0)
    bfgs_rosenbrock_results[a] = result

bfgs_rosenbrock_results
```

运行结果：

```
{1: array([1., 1.]),
 2: array([2., 4.]),
 3: array([3.        , 9.00000002]),
 4: array([ 3.99999989, 15.99999915]),
 5: array([ 5., 25.]),
 6: array([ 5.99999999, 35.99999992]),
 7: array([ 7.        , 49.00000003]),
 8: array([ 7.99999988, 63.99999815]),
 9: array([ 9., 81.]),
 10: array([ 10., 100.])}
```

对于每个 a 的值，算法成功找到了对应的全局最小值点 (a, a^2)。这些结果验证了BFGS算法在这个问题上的有效性。

对于FR算法，设计如下测试：

```python
# Define the Rosenbrock function
def rosenbrock(x, a=1, b=100):
```

```
    return (a - x[0])**2 + b*(x[1] - x[0]**2)**2

# Define the gradient of the Rosenbrock function
def grad_rosenbrock(x, a=1, b=100):
    return np.array([-2*(a - x[0]) - 4*b*x[0]*(x[1] - x[0]**2),
                     2*b*(x[1] - x[0]**2)])
# Initial guess
x0 = np.array([-1.2, 1.0])
# Test the FR conjugate gradient algorithm on Rosenbrock function
with different values of a (from 1 to 10)

fr_rosenbrock_results = {}
for a in range(1, 11):
    result = fr_conjugate_gradient(lambda x: rosenbrock(x, a=a),
lambda x: grad_rosenbrock(x, a=a), x0)
    fr_rosenbrock_results[a] = result

fr_rosenbrock_results
```

运行结果：

```
{1: array([0.9999977 , 0.99999537]),
 2: array([2.00000354, 4.00001386]),
 3: array([3.00000034, 9.00000199]),
 4: array([ 4.00003997, 16.00031977]),
 5: array([ 4.99996809, 24.99968092]),
 6: array([ 5.9999788 , 35.99974561]),
 7: array([ 7.00006119, 49.00085667]),
 8: array([ 7.99994706, 63.99915291]),
 9: array([ 9.00001326, 81.00023843]),
 10: array([ 9.99992641, 99.99852818])}
```

对于每个 a 的值，算法均找到了接近全局最小值点 (a, a^2)的结果。这些轻微的偏差可能是由于线搜索精度、迭代次数限制或浮点数精度限制造成的。这些结果显示，FR 算法能够有效地解决不同参数下的优化问题。