最优化理论与算法 作业 2

姓名：华浩宇 学号：23020211153893

要求：

(1) 在文档中说明解题思路、方法实现、求解结果等，必要时进行分析和讨论。

(2) 将源代码作为附录粘贴于作业文档末尾，并同时作为提交的附件与作业文档 一起放在压缩包内备查。

(3) 推荐编程语言为 MATLAB，可选用其他熟悉的编程语言。

(4) 独立完成，严禁抄袭。

1. 编写程序实现共轭梯度法，要求具有通用性。一维搜索过程采用割线法（可使用作业 1 中实现的割线法函数）。以 Rosenbrock 函数

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

作为测试函数，初始点为$x^{(0)} = [-2,2]^{\mathrm{T}}$。在程序中，采用不同的公式来计算$\beta_k$，根据计算过程和结果比较各自的性能。要求每开展 6 次迭代，就将搜索方向重置为梯度的负方向。

共轭梯度法（Conjugate Gradient）是介于最速下降法与牛顿法之间的一个方法，它仅需利用一阶导数信息，但克服了最速下降法收敛慢的缺点，又避免了牛顿法需要存储和计算 Hesse 矩阵并求逆的缺点，共轭梯度法不仅是解决大型线性方程组最有用的方法之一，也是解大型非线性最优化最有效的算法之一。

实验代码

首先，我们实现共轭梯度法：

```
function [fv, bestx, iter_num] = conjungate_gradient(f, x, x0, epsilon, show_detail)

syms lambdas

n = length(x);

nf = cell(1, n);
for i = 1 : n
    nf{i} = diff(f, x{i});
end

nfv = subs(nf, x, x0);

nfv_pre = nfv;
count = 0;
k = 0;
xv = x0;
```

```
d = - nfv;

if show_detail
    fprintf('Initial:\n');
    fprintf('f = %s, x0 = %s, epsilon = %f\n\n', char(f), num2str(x0),
epsilon);
end

while (norm(nfv) > epsilon)
    xv = xv+lambdas*d;
    phi = subs(f, x, xv);
    nphi = diff(phi);
    lambda = solve(nphi);
    lambda = double(lambda);
    if length(lambda) > 1
        lambda = lambda(abs(imag(lambda)) < 1e-5);
        lambda = lambda(lambda > 0);
        lambda = lambda(1);
    end
    if lambda < 1e-5
        break;
    end

    xv = subs(xv, lambdas, lambda);
    xv = double(xv);
    nfv = subs(nf, x, xv);
    count = count + 1;
    k = k + 1;
    alpha = sum(nfv(:).*nfv(:)) / sum(nfv_pre(:).*nfv_pre(:));

    if show_detail
        fprintf('epoch: %d\n', count);
        fprintf('x(%d) = %s, lambda = %f\n', count, num2str(xv),
lambda);
        fprintf('nf(x) = %s, norm(nf) = %f\n', num2str(double(nfv)),
norm(double(nfv)));
        fprintf('d = %s, alpha = %f\n', num2str(double(d)),
double(alpha));
        fprintf('\n');
    end

    d = -nfv + alpha .* d;
    nfv_pre = nfv;
    if k >= 6
```

```
        k = 0;
        d = - nfv;
    end
end % while

fv = double(subs(f, x, xv));
bestx = double(xv);
iter_num = count;

end
```

对实现的共轭梯度法进行测试：

```
syms x1 x2;
f = 100 * (x2-x1.^2).^2 + (1 - x1).^2;
x = {x1, x2};

% initial value
x0 = [-2 2];
% tolerance
epsilon = 1e-5;
%% call conjungate gradient method
show_detail = true;
[bestf, bestx, count] = conjungate_gradient(f, x, x0, epsilon,
show_detail);
% print result
fprintf('bestx = %s, bestf = %f, count = %d\n', num2str(bestx), bestf,
count);
```

实验结果

```
bestx = 1, bestf = 0.000000
```

2. 编写程序，实现拟牛顿法，可以求解目标函数为一般形式非线性函数时的优化问题。利用割线法开展一维搜索（可使用作业 1 中实现的割线法函数）。以 Rosenbrock 函数

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

作为测试函数，初始点为$x^{(0)} = [-2, 2]^{\mathrm{T}}$。对$H_k$不同的更新公式进行评测。在程序中，要求每 6 次迭代就将搜索方向更新为梯度负方向。

拟牛顿法和最速下降法(Steepest Descent Methods)一样只要求每一步迭代时知道目标函数的梯度。通过测量梯度的变化，构造一个目标函数的模型使之足以产生超线性收敛性。这类方法大大优于最速下降法，尤其对于困难的问题。

实验代码

BFGS 算法，即使用 BFGS 矩阵作为拟牛顿法中的对称正定迭代矩阵的方法。

```
function [f, xk, k] = BFGS(x0, fun, grid, eps, kmax)
  k = 0;
  n = length(x0);
  H0 = eye(n); % 初始选取单位阵作为 Hessen 矩阵的逆的近似阵
  Hk = H0;
  xk = x0;
  gk = feval(grid, xk);
  while k <= kmax
      if norm(gk) < eps
          break;
      end
      dk = -Hk * gk; % 拟牛顿下降方向
      alpha = Armjio(fun, grid, xk, dk);
      x_ = xk; % x_ 保存上一个点坐标
      xk = x_ + alpha * dk; % 更新 xk
      gk_ = gk; % gk_ 保存上一个点的梯度值
      gk = feval(grid, xk); % 更新 gk
      sk = xk - x_; % 记 xk - x_ 为 sk
      yk = gk - gk_; % 记 gk - gk_ 为 yk
      if sk' * yk > 0
          v = yk' * sk;
          % BFGS 公式
          Hk = Hk + (1 + (yk' * Hk * yk) / v) * (sk * sk') / v - (sk *
yk' * Hk + Hk * yk * sk') / v;
      end
      k = k + 1;
  end
  f = feval(fun, xk);
end
```

实验结果

初始点	极小点	目标函数值	迭代次数	运行时间
[-8.491293, -9.339932]	[1.691594, 2.861921]	0.478320	3001	0.113080
[-6.787352, -7.577401]	[1.008268, 1.021558]	0.002522	3001	0.109828
[-7.431325, -3.922270]	[0.957160, 0.914076]	0.002267	3001	0.104231
[-6.554779, -1.711867]	[1.018073, 1.041407]	0.002761	3001	0.104697
[-7.060461, -0.318328]	[0.975235, 0.955384]	0.002464	3001	0.105013
[-2.769230, -0.461714]	[0.999686, 0.999282]	0.000001	3001	0.103167
[-0.971318, -8.234578]	[0.990664, 0.982330]	0.000171	3001	0.103379
[-6.948286, -3.170995]	[1.047682, 1.097101]	0.002302	3001	0.103382
[-9.502220, -0.344461]	[4.042285, 16.307197]	9.363567	3001	0.104598
[-4.387444, -3.815585]	[1.012093, 1.026443]	0.000591	3001	0.102904

DFP 算法：

算法　(DFP 算法)

步 0　给定参数 $\delta \in (0,1)$, $\sigma \in (0,0.5)$, 初始点 $x_0 \in \mathbb{R}^n$, 终止误差 $0 \le \varepsilon \ll 1$. 初始对称正定阵 H_0 (通常取为 $G(x_0)^{-1}$ 或单位阵 I_n). 令 $k := 0$.

步 1　计算 $g_k = \nabla f(x_k)$. 若 $\|g_k\| \le \varepsilon$, 停算, 输出 x_k 作为近似极小点.

步 2　计算搜索方向:

$$d_k = -H_k g_k.$$

步 3　设 m_k 是满足下列不等式的最小非负整数 m:

$$f(x_k + \delta^m d_k) \le f(x_k) + \sigma \delta^m g_k^T d_k.$$

令 $\alpha_k = \delta^{m_k}$, $x_{k+1} = x_k + \alpha_k d_k$.

步 4　由校正公式　　确定 H_{k+1}.

步 5　令 $k := k+1$, 转步 1.

```
function [f, xk, k] = DFP(x0, fun, grid, eps, kmax)
  k = 0;
  n = length(x0);
  H0 = eye(n); % 初始选取单位阵作为 Hessen 矩阵的逆的近似阵
  Hk = H0;
  xk = x0;
  gk = feval(grid, xk);
  while k <= kmax
      if norm(gk) < eps
          break;
      end
      dk = -Hk * gk; % 拟牛顿下降方向
      alpha = Armjio(fun, grid, xk, dk);
      x_ = xk; % x_ 保存上一个点坐标
      xk = x_ + alpha * dk; % 更新 xk
      gk_ = gk; % gk_ 保存上一个点的梯度值
      gk = feval(grid, xk); % 更新 gk
      sk = xk - x_; % 记 xk - x_ 为 sk
      yk = gk - gk_; % 记 gk - gk_ 为 yk
      if sk' * yk > 0
          v = Hk * yk;
          % DFP 公式
          Hk = Hk + (sk * sk') / (sk' * yk) - (v * v') / (yk' * v);
      end
      k = k + 1;
  end
  f = feval(fun, xk);
end
```

实验结果

初始点	极小点	目标函数值	迭代次数	运行时间
[6.340544, 0.888908]	[1.000009, 1.000018]	0.000000	19	0.029119
[6.393631, 4.426515]	[1.000000, 1.000000]	0.000000	27	0.002961
[0.682783, 1.949488]	[1.000000, 1.000000]	0.000000	16	0.001046
[3.828171, 6.702548]	[1.000000, 1.000000]	0.000000	53	0.002609
[6.754220, 1.103292]	[1.000000, 1.000000]	0.000000	26	0.003102
[6.794149, 6.700169]	[1.000000, 1.000000]	0.000000	20	0.001095
[3.397630, 5.601963]	[1.000000, 1.000000]	0.000000	27	0.001299
[0.993204, 2.952329]	[1.000000, 1.000000]	0.000000	24	0.001167
[6.410149, 5.545451]	[1.000000, 1.000000]	0.000000	26	0.001455
[6.716447, 4.590185]	[1.000000, 1.000000]	0.000000	23	0.001132