Conjugate gradient method for finding minimizer

Line search procedure

The method is adapted directly from *Numerical Optimization* by Jorge Nocedal and Stephen J. Wright, called the *strong backtracking*. The procedure will be given with later explanation.

1. Set $a = 1$. Set $a_{old} = 0$ then goto 2.
2. If $f(x + as) > f(x) + \mu as^T \nabla f(x)$ or $f(x + as) > f(x + a_{old})$, then set $a_{low} = a_{old}$ and $a_{high} = a$ and go to step 6. Else, go to step 3.
3. If $|s^T \nabla f(x + as)| \leq -\eta s^T \nabla f(x)$, then $\lambda = a$ and Exit. Else, go to step 4.
4. If $s^T \nabla f(x + as) \geq 0$ then set $a_{low} = a_{old}$ and $a_{high} = a$ and go to step 6. Else, go to step 5.
5. Let $a_{old} = a$ and $a = 2a$. Back to step 2.
6. Set $f_{low} = f(x + a_{low}s)$. goto step 7. Note that this value will be fixed regardless of changing a_{low}.
7. Use binary search or golden section search to find suitable a between a_{low} and a_{high}.
   (a) For binary search, let $a = (a_{low} + a_{high})/2$
   (b) For golden section search, let $c = a_{low} + (a_{high} - a_{low})/\phi$ and $d = a_{high} - (a_{high} - a_{low})/\phi$, where ϕ is a golden ratio. If $f(x + cs) < f(x + ds)$, let $a = c$. Otherwise, let $a = d$.

   Then goto step 8.
8. If $f(x + as) > f(x) + \mu as^T \nabla f(x)$ or $f(x + as) > f_{low}$, then let $a_{high} = a$. Then go back to step 7. Else, goto step 9.
9. If $|s^T \nabla f(x + as)| \leq -\eta s^T \nabla f(x)$, then $\lambda = a$ and Exit. Else, go to step 10.
10. If $(s^T \nabla f(x + as))(a_{high} - a_{low}) \geq 0$, then let $a_{high} = a_{low}$ and goto step 11.
11. Let $a_{low} = a$ and back to step 7.

The step-by-step explanation is as follow:

1. a is representing λ. Let it be 1 first.
2. If $f(x + as) > f(x) + \mu as^T \nabla f(x)$, then it is violating first Wolfe's condition. It is thus ensuring that the bracket (a_{old}, a) will contain range that is not violating that. (The slope at $s^T \nabla f(x + a_{old}s)$ is always negative according to step 4, so that (a_{old}, a) must contain local minimum, that is, containing second Wolfe's point.) Also, $f(x+as) > f(x+a_{old})$ is indicating that the function is going to increase, thus (a_{old}, a) must also contain local minimum (that is second Wolfe's point, too.) Note that a_{high} and a_{low} can swap regardless of their values.
3. The first Wolfe's condition is already checked in step 2. Thus, if the point also satisfies second Wolfe's condition, then let it be λ and exit.
4. If $s^T \nabla f(x + as) \geq 0$ then it is indicating that the function is going to increase, thus (a_{old}, a) must contain local minimum (that is second Wolfe's point.) (It is cleared because $s^T \nabla f(x + a_{old}s)$ is always negative.) Also, note that a_{high} and a_{low} can swap regardless of their values.
5. If the range (a_{old}, a) is not containing second Wolfe's point, slide it to the immediate right and expand it two times.
6. This evaluated value will be used in the step 8.
7. Finding a between a_{high} and a_{low}. a will converge to range which satisfies second Wolfe's condition.

8. If $f(x+as) > f(x) + \mu as^T \nabla f(x)$ or $f(x+as) > f_{low}$, then $f(x+as)$ value is to big. Thus, let a_{high} be it to lower the upper bound.

9. The first Wolfe's condition is already checked in step 8. Thus, if the point also satisfies second Wolfe's condition, then let it be λ and exit.

10. $(s^T \nabla f(x+as))(a_{high} - a_{low}) \geq 0$, together with step 11, ensures us that the range between a_{high} and a_{low} always contain local minimum(s). To see that, let $a_{high} < a_{low}$, so $a_{high} - a_{low} < 0$. Moreover, the slope at a_{high} must be negative and the slope at a_{low} must be positive (from step 2, step 4, and recursive characteristics that happens here.) $a_{high} < a < a_{low}$ must be true, and in step 11 a_{low} becomes a, so the slope at a must be positive. If the slope at a is negative, then, a_{high} must be swap with a_{low} to make the slope of a_{high} positive. Then between a_{high} and a will be ensured to have local minimum. The same is applied when $a_{high} > a_{low}$. Just swap the pair accordingly and we will get the same proof.

11. Let $a_{low} = a$ to shorten the length.

From the book *Numerical Optimization*, step 2 to step 5 are called *Bracket phase*, and step 7 to step 11 are called *Zoom phase*.

Conjugate gradient method implementation

First is an implementation of `StrongBacktrack.m`. I choose to use the binary search because it usually give less steps than the golden section search.

```
function [lambda,nF,nG] = StrongBacktrack(FcnName, x0, s, a, mu, eta)

% -------------- Function inputs --------------
%
% FcnName: function to return the value, the gradient, and the Hessian
% of the particular function.
%   Mode 1: return only f.
%   Mode 2: return f and gradient.
%
% x0: starting point of searching.
%
% s: search direction.
%
% a: size of initial lambda.
%
% mu, eta: the parameters used in the stopping criterion for line search.

% -------------- Function outputs --------------
%
% lambda: returned the lambda value that satisfies strong Wolfe's.
%
% nF, nG: numbers of f and gradient calculations.

nF = 0; % number of f calculations.
nG = 0; % number of gradient calculations.

[f0, g0] = FcnName(x0, 2); nF = nF + 1; nG = nG + 1;
% first evaluation of f and gradient.
fprev = f0; aprev = 0;
% fprev: previous value of f at (x0 + a*s).
% aprev: previous value of a (that is the value going to represent lambda.)
alo = NaN; ahi = NaN; % the bracket that contain strong Wolfe's.
% alo represent lower value of f, ahi represent higher value of f.
% However, the value of alo and ahi can be swap without ruining the algorithm.

gr = (sqrt(5) + 1) / 2; % golden ratio.

% finding suitable bracket.
while 1
```

```
    [f1, g1] = FcnName(x0 + a*s, 2); nF = nF + 1; nG = nG + 1;
    % evaluate function at (x0 + a*s).
    if f1 > f0 + mu*a*dot(s,g0) || f1 >= fprev
        % If the function value of the right point is more than Armijo's rule or
        % more than that of previous value,
        % it is sure that the lowest point is there in the bracket, and
        % there is strong Wolfe's point in that bracket. (Since it is ensured that
        % at aprev, the slope is negative.)
        alo = aprev; ahi = a;
        break
    elseif abs(dot(g1,s)) <= -eta*dot(g0,s)
        % Checking the second strong Wolfe's condition.
        lambda = a;
        return
    elseif dot(g1,s) >= 0
        % If the slope on the right point is positive, it is sure that the
        % lowest point is there in the bracket, and
        % there is strong Wolfe's point in that bracket.
        % (Since it is ensured that at aprev, the slope is negative.)
        alo = aprev; ahi = a;
        break
    end
    % slide the bracket to the right and expand it by multiple of 2.
    fprev = f1; aprev = a; a = 2*a;
end

[flo,~] = FcnName(x0 + alo*s, 1); nF = nF + 1; % f value at (x0 + alo*s).
while 1

    % if want to use golden search, use these lines.
    % ----Golden search to find a between alo and ahi.
    % ----f1 is f at (x0 + a*s), g1 is gradient at (x0 + a*s).
    %     c = ahi - (ahi - alo) / gr;
    %     d = alo + (ahi - alo) / gr;
    %
    %     [fc,gc] = FcnName(x0 + c*s, 2);
    %     [fd,gd] = FcnName(x0 + d*s, 2);
    %
    %     if fc < fd % f(c) > f(d) to find the maximum.
    %         f1 = fc; g1 = gc; a = c;
    %     else
    %         f1 = fd; g1 = gd; a = d;
    %     end
    % ----end of Golden search

    % if want to use binary search, use this line.
    a = (ahi+alo)/2; [f1,g1] = FcnName(x0 + a*s, 2);
    nF = nF + 1; nG = nG + 1;

    if f1 > f0 + mu*a*dot(s,g0) || f1 > flo
        % if violating Armijo's rule or if f1 still higher than flo, then,
        % set new hi to decrease f at ahi.
        ahi = a;
    else
        if abs(dot(g1,s)) <= -eta*dot(g0,s)
            % Checking the second strong Wolfe's condition.
            lambda = a;
            return
        elseif dot(g1,s)*(ahi - alo) >= 0
            % This condition ensures that ahi and alo always bracket the
            % lowest point. That is, there is strong Wolfe's point between
            % alo and ahi.
            ahi = alo;
        end
        alo = a;
    end

end
```

```
end
```

Implementation of function `CG.m` is given here. The `epsilon` value is used to give a threshold for norm of $x(k+1) - x(k)$. You can choose either Fletcher-Reeves or Polak-Ribiere method by indicating option (1 or 2 respectively.)

```
function  [xmin,fmin,Xk,Fk,Gk,Lk,nF,nG,IFLAG,nReset] = CG(FcnName,x0,epsilon,mu,eta,itmax,
    option)

% -------------- Function inputs --------------
%
% FcnName: function to return the value, the gradient, and the Hessian
% of the particular function.
%   Mode 1: return only f.
%   Mode 2: return f and gradient.
%
% x0: starting point of searching.
%
% epsilon: stoping criterion of the minimum search. (norm(x1-x0) < epsilon.)
%
% mu, eta: the parameters used in the stopping criterion for line search.
%
% itmax: max allowed number of iterations.
%
% option: 1 = Fletcher-Reeves, 2 = Polak-Ribiere.

% -------------- Function outputs --------------
%
% xmin, fmin: returned minimum function argument and value, respectively.
%
% Xk ,Fk, Gk, Lk: arrays to keep x, f, gradient and lambda along the search steps.
%
% nF, nG: numbers of f and gradient calculations in each iteration.
%
% IFLAG: indicate the success. 0 if success, -999 otherwise.
%
% nReset: reset condition in each iteration
% 0 = no reset, 1 = reset because too large angle between s and -g
% 2 = reset because s does not have a descent property.

Xk = []; % list to store x_k.
Fk = []; % list to store f_k.
Gk = []; % list to store g_k.
Lk = []; % list to store l_k.

nF_val = 0; % number of f calculations in each iteration.
nG_val = 0; % number of gradient calculations in each iteration.

nF = []; % array to store nF_val of each iteration.
nG = []; % array to store nG_val of each iteration.
nReset = []; % array to store reset condition of each iteration.

IFLAG = -999; % IFLAG: indicate the success.

[f0, g0] = FcnName(x0, 2); nF_val = nF_val + 1; nG_val = nG_val + 1;
s = -g0; % set first line search direction.

for i = 1:itmax

    % strong backtracking.
    a = 1; % first value of lambda.
    [lambda,nFnew,nGnew] = StrongBacktrack(FcnName, x0, s, a, mu, eta);
    % finding lambda that satisfied strong Wolfe's.
    nF_val = nF_val + nFnew; nG_val = nG_val + nGnew;

    % update values.
    x1 = x0 + lambda*s;
```

```
    [f1,g1] = FcnName(x1, 2); nF_val = nF_val + 1; nG_val = nG_val + 1;
    if option == 1
        beta = norm(g1)/norm(g0); % Fletcher-Reeves
    elseif option == 2
        beta = dot(g1,g1-g0)/norm(g0); % Polak-Ribiere
    else
        disp('invalid option.');
        break
    end
    s = -g1 + beta*s; % new line search direction.

    % Reset if angle between s and g1 is too large (> 85 degree.)
    cos_angle = dot(s,-g1)/(norm(s)*norm(-g1));
    if cos_angle < cosd(85) && cos_angle > 0
        s = -g1; nReset(i) = 1; % the angle is too large.
    elseif cos_angle <= 0
        s = -g1; nReset(i) = 2; % s does not have a descent property.
    else
        nReset(i) = 0; % no reset.
    end

    % store values.
    Xk(:,i) = x0; Fk(i) = f0; Gk(:,i) = g0; Lk(i) = lambda;
    nF(i) = nF_val; nG(i) = nG_val;

    % terminate
    if norm(x1-x0) < epsilon % at local minimum, gradient converges to 0.
        xmin = x1; fmin = f1; IFLAG = 0;
        disp('search successful.');
        break
    end

    % update values.
    x0 = x1; f0 = f1; g0 = g1; nF_val = 0; nG_val = 0;
end

if IFLAG == -999
    xmin = 0; fmin = 0; disp('search unsuccessful.');
end

end
```

Implementation of `Rosenbrock.m` is here.

```
function [f,gradient] = Rosenbrock(x,options)

    % Declare the functions.
    f_fun = @(x) 100*(x(2) - x(1)^2)^2 + (1-x(1))^2;
    gradient_fun = @(x)[400*x(1)*(x(1)^2 - x(2)) - 2*(1-x(1)); 200*(x(2)-x(1)^2)];

    % Evaluate numerical values.
    switch options
        case 1 % calculate only f.
            f = f_fun(x);
            gradient = 0;
        case 2 % calculate f and gradient.
            f = f_fun(x);
            gradient = gradient_fun(x);
        otherwise % invalid option.
            disp('invalid option.')
            f = 0; gradient = 0;
    end

end
```

This is a script used to test the code.

```
%% First, use the Fletcher-Reeves's
[xmin,fmin,Xk,Fk,Gk,Lk,nF,nG,IFLAG,nReset] = CG(@Rosenbrock,[-1.2;1],5e-9,1e-4,0.1,10000,1);
```

```
% print out the result.
disp("Fletcher-Reeves:")
fprintf('% 4s % 10s % 10s % 10s % 10s % 10s % 10s % 3s % 3s % 1s\n', 'Iter', 'x_1', 'x_2', '
    f', 'gradient_1', 'gradient_2', 'lambda', 'nF', 'nG', 'nReset');
for i = 1:length(Xk)-1
    fprintf('% 4.i % 10.5f % 10.5f % 10.4f % 10.4f % 10.4f % 10.4f % 3.f % 3.f % 1.f \n', i,
     Xk(1,i+1), Xk(2,i+1), Fk(i+1), Gk(1,i+1), Gk(2,i+1), Lk(i), nF(i), nG(i), nReset(i));
end

fprintf("Number of f calculations:           %i \n", sum(nF))
fprintf("Number of gradient calculations:    %i \n", sum(nG))
fprintf("Number of resets:                   %i \n", sum(nReset ~= 0))

%% Then, use the Polak-Ribiere's
[xmin,fmin,Xk,Fk,Gk,Lk,nF,nG,IFLAG,nReset] = CG(@Rosenbrock,[-1.2;1],5e-9,1e-4,0.1,10000,2);
% print out the result.
disp("Polak-Ribiere:")
fprintf('% 4s % 10s % 10s % 10s % 10s % 10s % 10s % 3s % 3s % 1s\n', 'Iter', 'x_1', 'x_2', '
    f', 'gradient_1', 'gradient_2', 'lambda', 'nF', 'nG', 'nReset');
for i = 1:length(Xk)-1
    fprintf('% 4.i % 10.5f % 10.5f % 10.4f % 10.4f % 10.4f % 10.4f % 3.f % 3.f % 1.f \n', i,
     Xk(1,i+1), Xk(2,i+1), Fk(i+1), Gk(1,i+1), Gk(2,i+1), Lk(i), nF(i), nG(i), nReset(i));
end

fprintf("Number of f calculations:           %i \n", sum(nF))
fprintf("Number of gradient calculations:    %i \n", sum(nG))
fprintf("Number of resets:                   %i \n", sum(nReset ~= 0))
```

The reported tabular is given here when using Fletcher-Reeves method.

```
search successful.
Fletcher-Reeves:
Iter        x_1         x_2           f gradient_1 gradient_2     lambda  nF  nG nReset
   1   -1.04209     1.06445     4.2163   -13.0454    -4.2996     0.0007  17  16  0
   2   -1.02951     1.06909     4.1274    -0.2724     1.8390     0.0005  15  14  0
   3    0.88709     0.78637     0.0128    -0.0253    -0.1130     0.5098  18  17  0
   4    0.88722     0.78641     0.0128     0.0412    -0.1503     0.0005  15  14  0
   5    0.88767     0.78678     0.0128     0.1934    -0.2355     0.0015  15  14  0
   6    0.95153     0.90177     0.0037     1.2888    -0.7282     0.1562   9   8  0
   7    0.95238     0.90702     0.0023    -0.0926    -0.0014     0.0012  16  15  0
   8    1.00345     1.00849     0.0003    -0.6305     0.3176     0.3750   7   6  0
   9    1.00508     1.01020     0.0000     0.0024     0.0039     0.0010  14  13  2
  10    1.00478     1.00971     0.0000    -0.0408     0.0250     0.1250   7   6  0
  11    1.00480     1.00963     0.0000     0.0074     0.0011     0.0012  16  15  0
  12    1.00434     1.00856     0.0000     0.0657    -0.0284     0.0938   9   8  0
  13    1.00421     1.00847     0.0000    -0.0058     0.0071     0.0012  16  15  0
  14    0.99991     0.99976     0.0000     0.0245    -0.0123     0.5000   5   4  0
  15    0.99986     0.99972     0.0000     0.0007    -0.0005     0.0010  14  13  0
  16    0.99986     0.99972     0.0000     0.0001    -0.0002     0.0004  18  17  0
  17    0.99986     0.99972     0.0000    -0.0000    -0.0001     0.0001  20  19  0
  18    0.99986     0.99972     0.0000    -0.0000    -0.0001     0.0001  20  19  0
  19    0.99986     0.99972     0.0000    -0.0001    -0.0001     0.0001  19  18  0
  20    0.99986     0.99972     0.0000    -0.0001    -0.0001     0.0001  20  19  0
  21    0.99986     0.99972     0.0000    -0.0002    -0.0001     0.0002  19  18  0
  22    0.99986     0.99972     0.0000    -0.0003    -0.0000     0.0003  18  17  0
  23    0.99986     0.99972     0.0000    -0.0005     0.0001     0.0010  14  13  0
  24    0.99997     0.99994     0.0000    -0.0028     0.0014     0.1250   7   6  0
  25    0.99997     0.99995     0.0000    -0.0002     0.0001     0.0010  14  13  0
  26    0.99997     0.99995     0.0000     0.0000    -0.0000     0.0006  17  16  0
  27    0.99997     0.99995     0.0000     0.0001    -0.0001     0.0012  16  15  0
  28    0.99999     0.99997     0.0000     0.0005    -0.0003     0.0469  10   9  0
  29    0.99999     0.99997     0.0000    -0.0001     0.0000     0.0012  16  15  0
  30    1.00000     0.99999     0.0000     0.0002    -0.0001     0.0625   8   7  0
  31    1.00000     0.99999     0.0000    -0.0000     0.0000     0.0012  16  15  0
  32    1.00000     1.00000     0.0000     0.0001    -0.0000     0.0469  10   9  0
  33    1.00000     1.00000     0.0000    -0.0000     0.0000     0.0012  16  15  1
  34    1.00000     1.00000     0.0000    -0.0000    -0.0000     0.0010  14  13  0
Number of f calculations:           501
Number of gradient calculations:    466
```

```
Number of resets:                   2
```

The reported tabular is given here when using Polak-Ribière method.

```
 Iter        x_1         x_2           f gradient_1 gradient_2     lambda  nF  nG nReset
...
 8410    1.00000     1.00000      0.0000     0.0000     0.0000     0.0017  16  15  0
 8411    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0024  15  14  0
 8412    1.00000     1.00000      0.0000     0.0000     0.0000     0.0015  15  14  0
 8413    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0039  12  11  0
 8414    1.00000     1.00000      0.0000     0.0000     0.0000     0.0012  16  15  0
 8415    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0078  11  10  0
 8416    1.00000     1.00000      0.0000     0.0000     0.0000     0.0012  16  15  0
 8417    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0039  12  11  0
 8418    1.00000     1.00000      0.0000     0.0000     0.0000     0.0015  15  14  0
 8419    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0024  15  14  0
 8420    1.00000     1.00000      0.0000     0.0000     0.0000     0.0017  16  15  0
 8421    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0024  15  14  0
 8422    1.00000     1.00000      0.0000     0.0000     0.0000     0.0017  16  15  0
 8423    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0024  15  14  0
 8424    1.00000     1.00000      0.0000     0.0000     0.0000     0.0017  16  15  0
 8425    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0024  15  14  0
 8426    1.00000     1.00000      0.0000     0.0000     0.0000     0.0015  15  14  0
 8427    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0039  12  11  0
 8428    1.00000     1.00000      0.0000     0.0000     0.0000     0.0012  16  15  0
 8429    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0078  11  10  0
 8430    1.00000     1.00000      0.0000     0.0000     0.0000     0.0012  16  15  0
 8431    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0039  12  11  0
 8432    1.00000     1.00000      0.0000     0.0000     0.0000     0.0015  15  14  0
 8433    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0024  15  14  0
 8434    1.00000     1.00000      0.0000     0.0000     0.0000     0.0017  16  15  0
 8435    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0024  15  14  0
 8436    1.00000     1.00000      0.0000     0.0000     0.0000     0.0017  16  15  0
 8437    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0024  15  14  0
 8438    1.00000     1.00000      0.0000     0.0000     0.0000     0.0017  16  15  0
 8439    1.00000     1.00000      0.0000    -0.0000     0.0000     0.0024  15  14  0
Number of f calculations:           122962
Number of gradient calculations:    114522
Number of resets:                   5
```

The result indicates that Fletcher-Reeves method give a very-much faster convergence than Polak-Ribière. Polak-Ribière's uses many iterations to converge, so that many calculation of function and its gradient are needed. Furthermore, when examine the `nReset` array, the 5 resets of Polak-Ribière's experimented in the table are only occured during the first 9 iterations of a calculation. That is, the number of resets of Polak-Ribière's is greater that that of Fletcher-Reeves's not because the number of iterations of Polak-Ribière's is much more greater, but because it is the intrinsic nature during the first few iterations of both methods themselves.

From my speculation, the reason that Polak-Ribière method converges very slow is that the minusing term, $g(k+1) - g(k)$, can be oscillated when the value of both gradients are so low that the computer precision can not handle.