COMPGV19: Tutorial 4

Table of Contents

Marta Betcke and Kiko Rul#lan

Exercise 2

Implement the Fletcher-Reeves method and the Polak-Ribiere method using the strong Wolfe conditions for the line search.

```
clear all, close all;
```

Rosenbrock function

```
% For computation define as function of 1 vector variable
F.f = @(x) x(1)^2 + 5*x(1)^4 + 10*x(2)^2;
F.df = @(x) [2*x(1) + 20*x(1)^3; 20*x(2)];
F.d2f = @(x) [2 + 60*x(1)^2, 0; 0, 20];

% For visualisation proposes define as a function of 2 variables (x,y)
FV.f = @(x,y) x.^2 + 5*x.^4 + 10.*y.^2;
FV.dfx = @(x,y) 2*x + 20*x.^3;
FV.dfy = @(x,y) 20*y;
FV.d2fxx = @(x,y) 2 + 60*x.^2;
FV.d2fxy = @(x,y) 0;
FV.d2fyx = @(x,y) 0;
FV.d2fyy = @(x,y) 20;
rosenbrock = FV.f;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
tol = 1e-12;
maxIter = 500;

%============================
% Points x0 = [1.2; 1.2], -[1.2,1.2]
%============================
x0 = [-1; -1];
```

Conjugate gradient with backtracking

```
lsOptsCG_BT.rho = 0.1;
```

```
lsOptsCG_BT.c1 = c1;
lsFun = @(x_k, p_k, alpha0) backtracking(F, x_k, p_k, alpha0,
 lsOptsCG_BT);
[xCG_FR_BT, fCG_FR_BT, nIterCG_FR_BT, infoCG_FR_BT] =
 nonlinearConjugateGradient(F, lsFun, 'FR', alpha0, x0, tol, maxIter)
[xCG_PR_BT, fCG_PR_BT, nIterCG_PR_BT, infoCG_PR_BT] =
 nonlinearConjugateGradient(F, lsFun, 'PR', alpha0, x0, tol, maxIter)
```

```
xCG_FR_BT =

   1.0e-13 *

    0.6125
    0.4365


fCG_FR_BT =

   2.2804e-26


nIterCG_FR_BT =

   467


infoCG_FR_BT =

        xs: [2x468 double]
    alphas: [1x468 double]
     betas: [1x467 double]


xCG_PR_BT =

   1.0e-12 *

    0.4240
   -0.0199


fCG_PR_BT =

   1.8370e-25


nIterCG_PR_BT =

    93


infoCG_PR_BT =
```

```
        xs: [2x94 double]
    alphas: [1x94 double]
     betas: [1x93 double]
```

Conjugate gradient with line search satisfying strong Wolfe condition

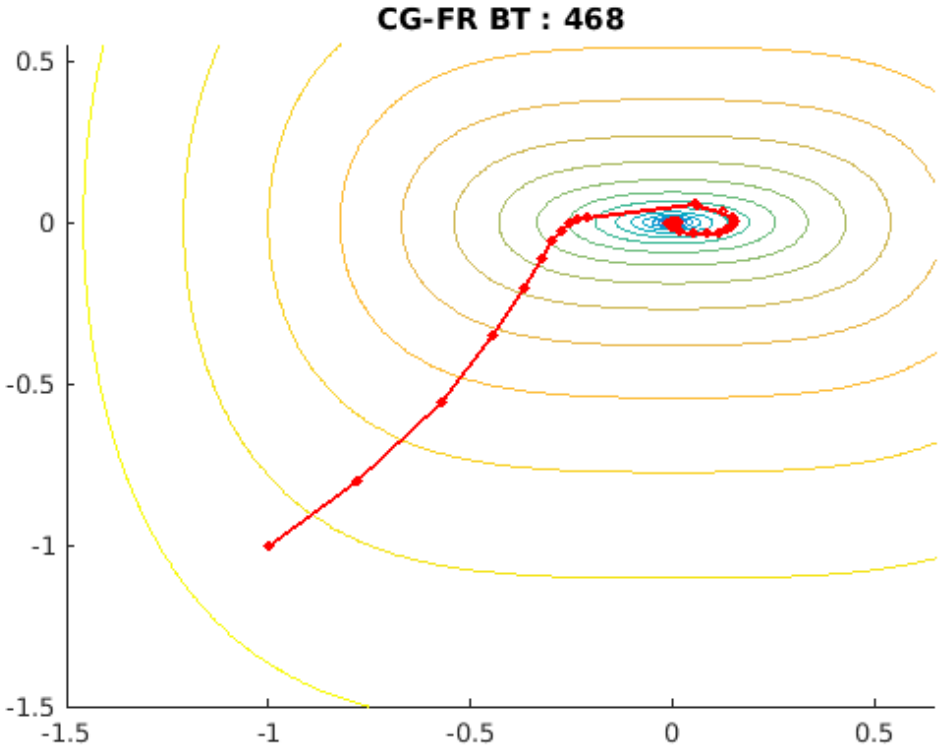
```
lsOptsCG_LS.c1 = c1;
lsOptsCG_LS.c2 = 0.1;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsCG_LS);
[xCG_FR_LS, fCG_FR_LS, nIterCG_FR_LS, infoCG_FR_LS] =
 nonlinearConjugateGradient(F, lsFun, 'FR', alpha0, x0, tol, maxIter)
[xCG_PR_LS, fCG_PR_LS, nIterCG_PR_LS, infoCG_PR_LS] =
 nonlinearConjugateGradient(F, lsFun, 'PR', alpha0, x0, tol, maxIter)


xCG_FR_LS =

   1.0e-12 *

   -0.1387
   -0.0497


fCG_FR_LS =

   4.3900e-26


nIterCG_FR_LS =

    23


infoCG_FR_LS =

        xs: [2x24 double]
    alphas: [1x24 double]
     betas: [1x23 double]


xCG_PR_LS =

   1.0e-13 *

   -0.7900
    0.2064


fCG_PR_LS =
```

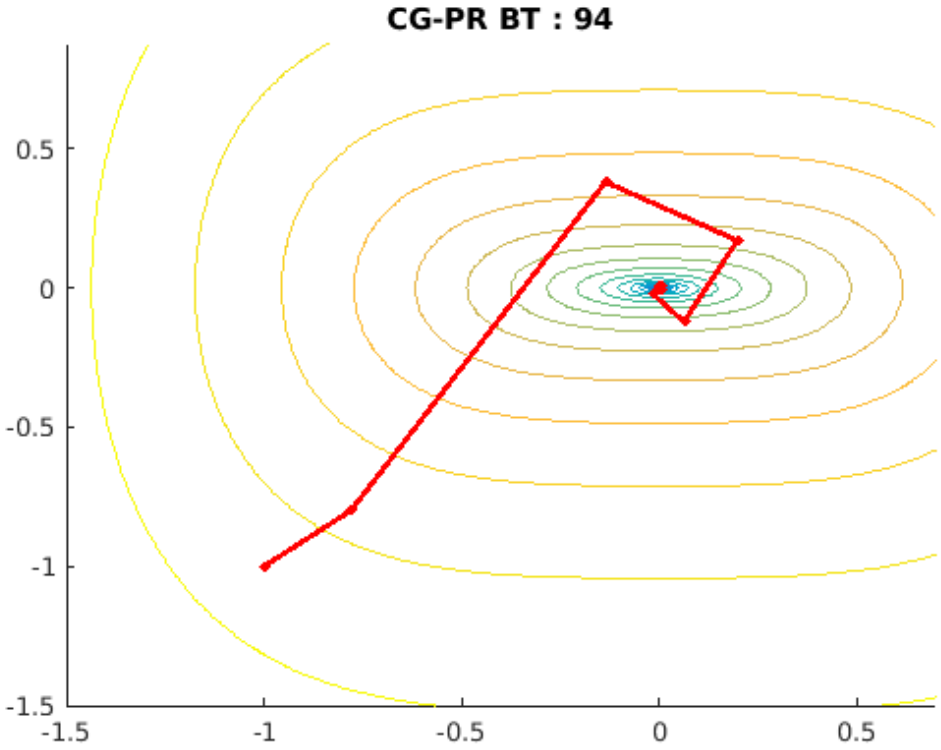
```
   1.0500e-26


nIterCG_PR_LS =

    23


infoCG_PR_LS =

        xs: [2x24 double]
    alphas: [1x24 double]
     betas: [1x23 double]


%==============================
```

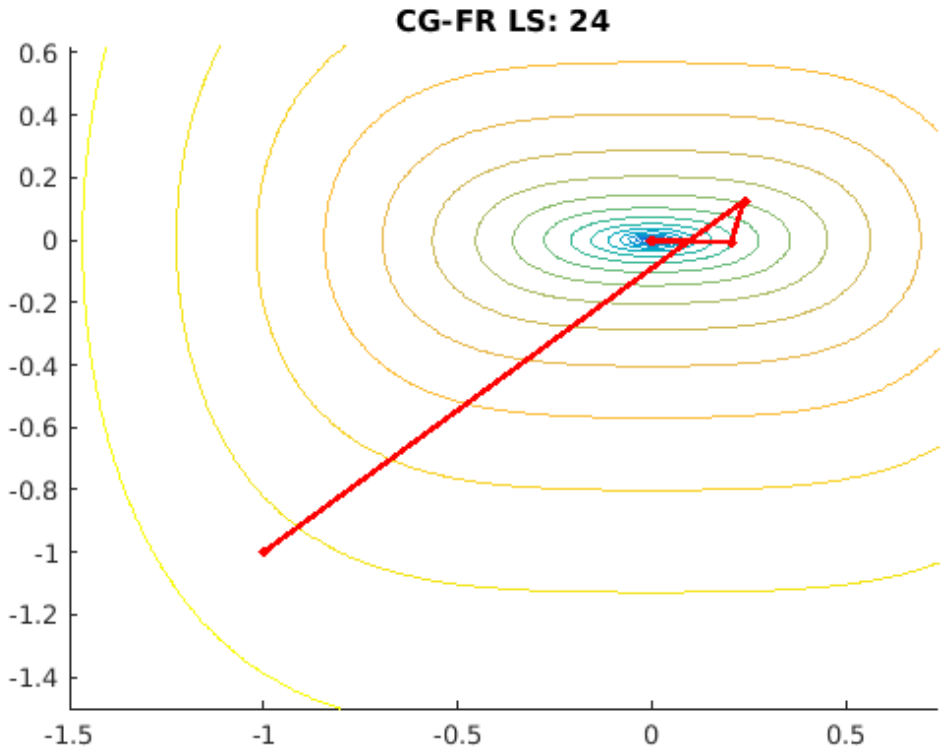
Visualisation

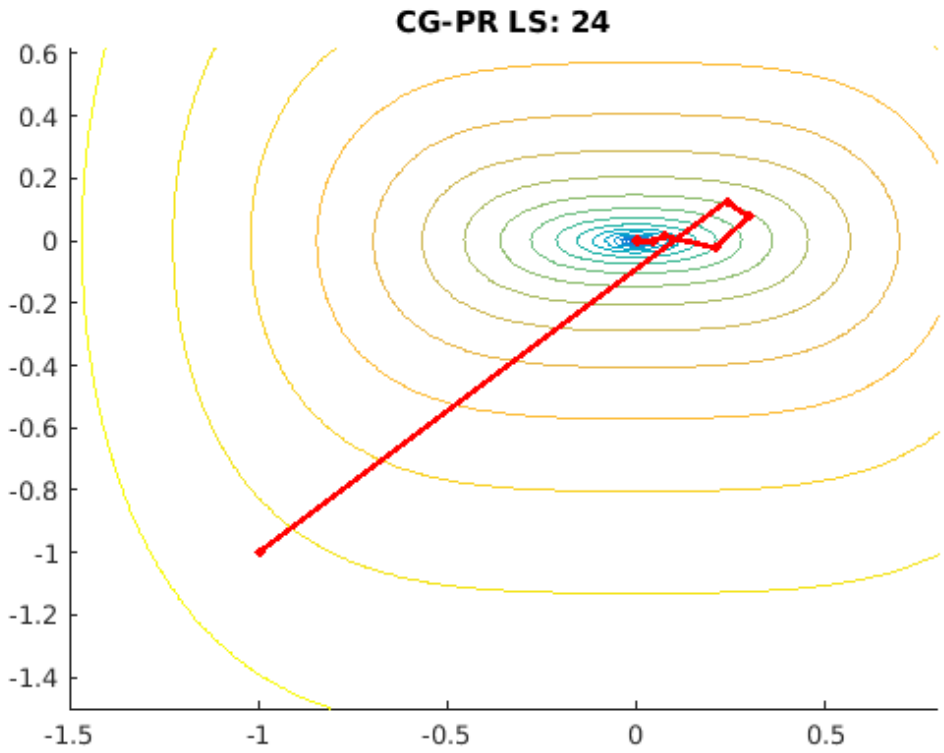
```
%==============================
% Backtracking - FR
n = 300;
x =
 linspace(min(infoCG_FR_BT.xs(1,:))-0.5,max(infoCG_FR_BT.xs(1,:))+0.5,n
+1);
y =
 linspace(min(infoCG_FR_BT.xs(2,:))-0.5,max(infoCG_FR_BT.xs(2,:))+0.5,n
+1);
[X,Y] = meshgrid(x,y);
Z = rosenbrock(X,Y);
% Iterate plot
visualizeConvergence(infoCG_FR_BT,X,Y,log(Z),'final'); title(['CG-FR
 BT : ' num2str(size(infoCG_FR_BT.xs,2))])
saveas(gcf, '../figs/02_01_CG-FR-BT', 'png');

% Backtracking - PR
n = 300;
x =
 linspace(min(infoCG_PR_BT.xs(1,:))-0.5,max(infoCG_PR_BT.xs(1,:))+0.5,n
+1);
y =
 linspace(min(infoCG_PR_BT.xs(2,:))-0.5,max(infoCG_PR_BT.xs(2,:))+0.5,n
+1);
[X,Y] = meshgrid(x,y);
Z = FV.f(X,Y);
% Iterate plot
visualizeConvergence(infoCG_PR_BT,X,Y,log(Z),'final'); title(['CG-PR
 BT : ' num2str(size(infoCG_PR_BT.xs,2))])
saveas(gcf, '../figs/02_02_CG-PR-BT', 'png');

% Line search satisfying strong Wolfe condition - FR
n = 300;
```

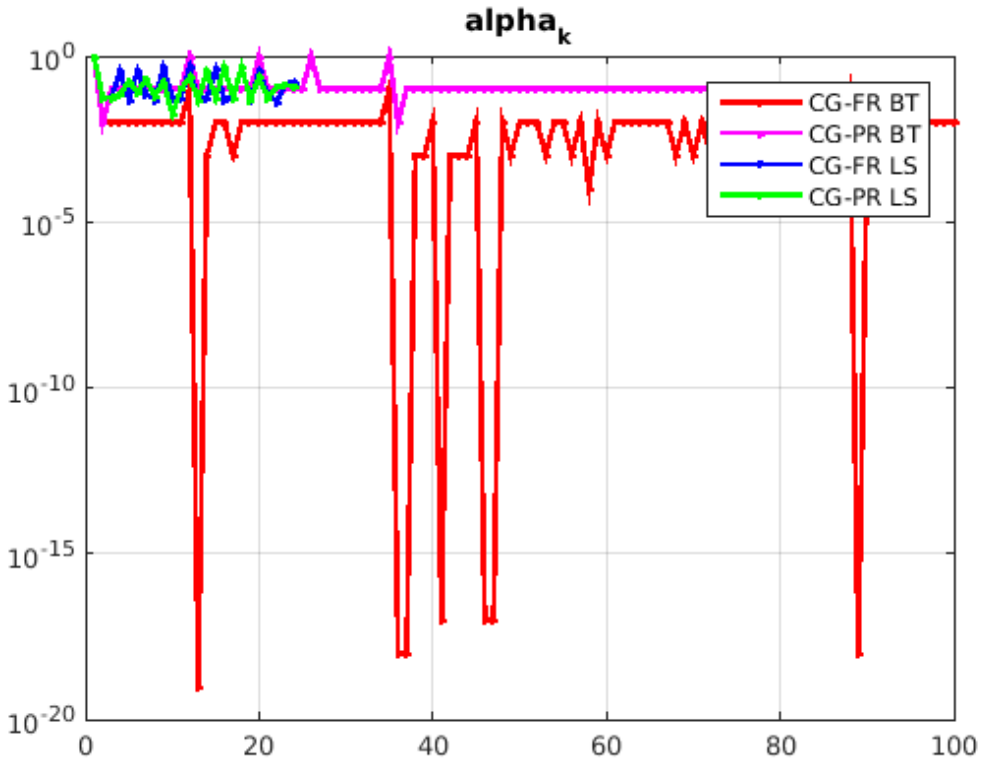
```
x =
 linspace(min(infoCG_FR_LS.xs(1,:))-0.5,max(infoCG_FR_LS.xs(1,:))+0.5,n
+1);
y =
 linspace(min(infoCG_FR_LS.xs(2,:))-0.5,max(infoCG_FR_LS.xs(2,:))+0.5,n
+1);
[X,Y] = meshgrid(x,y);
Z = FV.f(X,Y);
% Iterate plot
visualizeConvergence(infoCG_FR_LS,X,Y,log(Z),'final'); title(['CG-FR
 LS: ' num2str(size(infoCG_FR_LS.xs,2))])
saveas(gcf, '../figs/02_03_CG-FR-LS', 'png');

% Line search satisfying strong Wolfe condition - PR
n = 300;
x =
 linspace(min(infoCG_PR_LS.xs(1,:))-0.5,max(infoCG_PR_LS.xs(1,:))+0.5,n
+1);
y =
 linspace(min(infoCG_PR_LS.xs(2,:))-0.5,max(infoCG_PR_LS.xs(2,:))+0.5,n
+1);
[X,Y] = meshgrid(x,y);
Z = FV.f(X,Y);
% Iterate plot
visualizeConvergence(infoCG_PR_LS,X,Y,log(Z),'final'); title(['CG-PR
 LS: ' num2str(size(infoCG_PR_LS.xs,2))])
saveas(gcf, '../figs/02_04_CG-PR-LS', 'png');

% Step length plot
figure;
semilogy(infoCG_FR_BT.alphas(1:100), '-or', 'LineWidth',
 2, 'MarkerSize', 2); hold on;
semilogy(infoCG_PR_BT.alphas, '-om', 'LineWidth', 2, 'MarkerSize', 2);
semilogy(infoCG_FR_LS.alphas, '-ob', 'LineWidth', 2, 'MarkerSize', 2);
semilogy(infoCG_PR_LS.alphas, '-og', 'LineWidth', 2, 'MarkerSize', 2);
saveas(gcf, '../figs/02_05_Steplength', 'png');

grid on;
title('alpha_k');
legend('CG-FR BT', 'CG-PR BT', 'CG-FR LS', 'CG-PR LS');
```


CG-FR BT : 468
0.5
0
-0.5
-1
-1.5
-1.5
-1
-0.5
0
0.5


CG-PR BT : 94
0.5
0
-0.5
-1
-1.5
-1.5
-1
-0.5
0
0.5


CG-FR LS: 24
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1.2
-1.4
-1.5
-1
-0.5
0
0.5


CG-PR LS: 24
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1.2
-1.4
-1.5
-1
-0.5
0
0.5



============================== Subfunctions ==============================

nonlinearNonjugateGradient.m

Framework for nonlinear conjugate gradient methods.

```
function [xMin, fMin, nIter, info] = nonlinearConjugateGradient(F, ls,
 type, alpha0, x0, tol, maxIter)
% NONLINEARCONJUGATEGRADIENT Wrapper function executing conjugate
 gradient with Fletcher Reeves algorithm
% [xMin, fMin, nIter, info] = nonlinearConjugateGradient(F, ls,
 'type', alpha0, x0, tol, maxIter)
%
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
%   - d2f: Hessian handler
% ls: handle to linear search function
% type: beta update type {'FR', 'PR', 'PR+'}
% alpha0: initial step length
% rho: in (0,1) backtraking step length reduction factor
% c1: constant in sufficient decrease condition f(x_k + alpha_k*p_k) >
 f_k + c1*alpha_k*(df_k')*p_k)
%     Typically chosen small, (default 1e-4).
% x0: initial iterate
```

```
% tol: stopping condition on relative error norm tolerance
%      norm(x_Prev - x_k)/norm(x_k) < tol;
% maxIter: maximum number of iterations
%
% OUTPUTS
% xMin, fMin: minimum and value of f at the minimum
% nIter: number of iterations
% info: structure with information about the iteration
%   - xs: iterate history
%   - alphas: step lengths history
%
% Copyright (C) 2017  Kiko Rullan, Marta M. Betcke

% Parameters
% Stopping condition {'step', 'grad'}
stopType = 'grad';

% Initialization
nIter = 0;
normError = 1;
x_k = x0;
df_k = F.df(x_k);
p_k = -df_k;
info.xs = x0;
info.alphas = alpha0;
info.betas = [];
stopCond = 0;

% Loop until convergence or maximum number of iterations
while (~stopCond && nIter <= maxIter)
    % Call line search given by handle ls for computing step length
    alpha_k = ls(x_k, p_k, alpha0);

    % Update x_k and df_k
    x_k_1 = x_k;
    x_k = x_k + alpha_k*p_k;
    df_k_1 = df_k;
    df_k = F.df(x_k);

    % Compute descent direction
    switch upper(type)
      case 'FR'
        beta_k = (df_k'*df_k)/(df_k_1'*df_k_1);
      case 'PR'
        beta_k = (df_k'*(df_k - df_k_1))/(df_k_1'*df_k_1);
        % Safe guard in case p_k is not a descent direction
        if df_k'*(-df_k + beta_k*p_k) >= 0
          % Take FR update
          beta_k = (df_k'*df_k)/(df_k_1'*df_k_1);
        end
      case 'PR+' % requires modified strong Wolfe conditions
        beta_k = max((df_k'*(df_k - df_k_1))/(df_k_1'*df_k_1), 0);
      case 'FR-PR'
        beta_k_FR = (df_k'*df_k)/(df_k_1'*df_k_1); % >= 0
```

```
        beta_k = (df_k'*(df_k - df_k_1))/(df_k_1'*df_k_1); %PR if |
beta_k| <= beta_k_FR
        if beta_k > beta_k_FR
          beta_k = beta_k_FR;
        elseif beta_k < -beta_k_FR
          beta_k = -beta_k_FR;
        end
    end
    p_k = -df_k + beta_k*p_k;

    % Store iteration info
    info.xs = [info.xs x_k];
    info.alphas = [info.alphas alpha_k];
    info.betas = [info.betas beta_k];

    % Increment iterations
    nIter = nIter + 1;

    switch stopType
      case 'step'
        % Compute relative step length
        normStep = norm(x_k - x_k_1)/norm(x_k_1);
        stopCond = (normStep < tol);
      case 'grad'
        stopCond = (norm(df_k, 'inf') < tol*(1 + abs(F.f(x_k))));
    end
end

% Assign output values
xMin = x_k;
fMin = F.f(x_k);
```

Published with MATLAB® R2015a