COMPGV19 Assignment 5: Exercise 3

Table of Contents

Marta Betcke and Kiko Rullan

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

Function, gradient and Hessian


```
% % Different Functions to minimize
% % For computation define as function of 1 vector variable
% F.f = @(x) 100.*(x(2) - x(1)^2).^2 + (1 - x(1)).^2;
% F.df = @(x) [-400*(x(2) - x(1)^2)*x(1) - 2*(1 - x(1));
%              200*(x(2) - x(1)^2)];
% F.d2f = @(x) [-400*(x(2) - 3*x(1)^2) + 2, -400*x(1); -400*x(1),
 200];
%
% % For visualisation proposes define as a function of 2 variables
 (x,y)
% FV.f = @(x,y) 100.*(y - x.^2).^2 + (1 - x).^2;
% FV.dfx = @(x,y) -400.*(y - x.^2).*x - 2.*(1 - x);
% FV.dfy = @(x,y) 200.*(y - x.^2);
% FV.d2fxx = @(x,y) -400.*(y - 3*x.^2) + 2;
% FV.d2fxy = @(x,y) -400.*x;
% FV.d2fyx = @(x,y) -400.*x;
% FV.d2fyy = @(x,y) 200;
%
% % 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;

% For computation define as function of 1 vector variable
F.f = @(x) (x(1) - 3*x(2)).^2 + x(1).^4;
F.df = @(x) [2*(x(1) - 3*x(2)) + 4*x(1).^3; -6*(x(1) - 3*x(2))];
F.d2f = @(x) [2 + 12*x(1).^2, -6; -6, 18];

% For visualisation proposes define as a function of 2 variables (x,y)
FV.f = @(x,y) (x-3*y).^2 + x.^4;
FV.dfx = @(x,y) 2*(x - 3*y) + 4*x.^3;
FV.dfy = @(x,y)  -6*(x - 3*y);
FV.d2fxx = @(x,y) 2+12*x.^2;
FV.d2fxy = @(x,y) -6;
FV.d2fyx = @(x,y) -6;
FV.d2fyy = @(x,y) 18;

% Starting point
x0 = [10; 10];

% Parameters
maxIter = 200;
tol = 1e-4; % Stopping tolerance for relative step length between
 iterations
debug = 0; % Debugging parameter will switch on step by step
 visualisation of quadratic model and various step options

% Visualisation setup

% Define grid for visualisation
n = 1000;
x = linspace(-5, 10,n+1);
y = x;
[X,Y] = meshgrid(x,y);
% Compute log(f) to better highlight the function behaviour
Z = log(max(FV.f(X,Y), 1e-3)); % take max for better distribution of
 the contours (the log as a very steep deep at (0,0)
% % Compute f for true contours
% Z = FV.f(X,Y);
```


Minimise f using BFGS

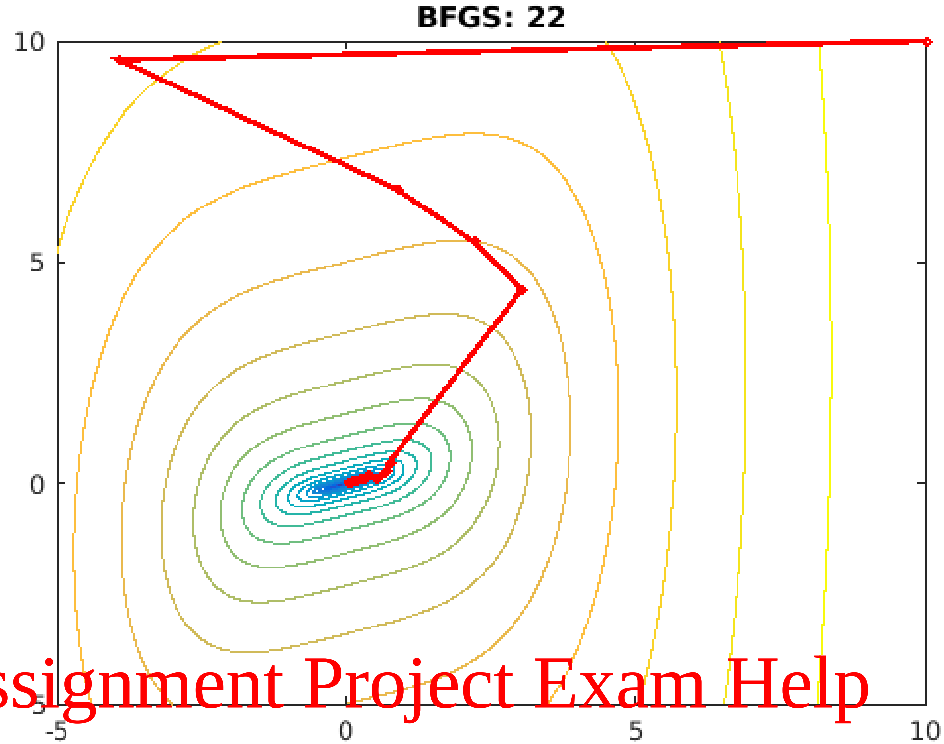
```
%==========================================================================
% Line search parameters
alpha0 = 1;
c1 = 1e-4;
% Backtracking (only for comparison purposes)
lsOpts.rho = 0.9;
lsOpts.c1 = c1;
lsFunB = @(x_k, p_k, alpha0) backtracking(F, x_k, p_k, alpha0,
 lsOpts);
% Strong Wolfe LS
lsOpts_LS.c1 = c1;
```

```
lsOpts_LS.c2 = 0.5; % 0.1 Good for Newton, 0.9 - good for steepest
 descent, 0.5 compromise.
lsFunS = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOpts_LS);
lsFun = lsFunS;

% Minimisation with Newton, Steepest descent and BFGS line search
 methods
%[xLS_BFGS, fLS_BFGS, nIterLS_BFGS, infoLS_BFGS] =
 descentLineSearch(F, 'steepest', lsFun, alpha0, x0, tol, maxIter)
[xLS_BFGS, fLS_BFGS, nIterLS_BFGS, infoLS_BFGS] =
 descentLineSearch(F, 'bfgs', lsFun, alpha0, x0, tol, maxIter)
```

```
Rescaling H0 with 0.0032602

xLS_BFGS =

    0.0237
    0.0079


fLS_BFGS =

   3.1629e-07


nIterLS_BFGS =

    21


infoLS_BFGS =

        xs: [2x22 double]
    alphas: [1x22 double]
         H: {1x21 cell}
```

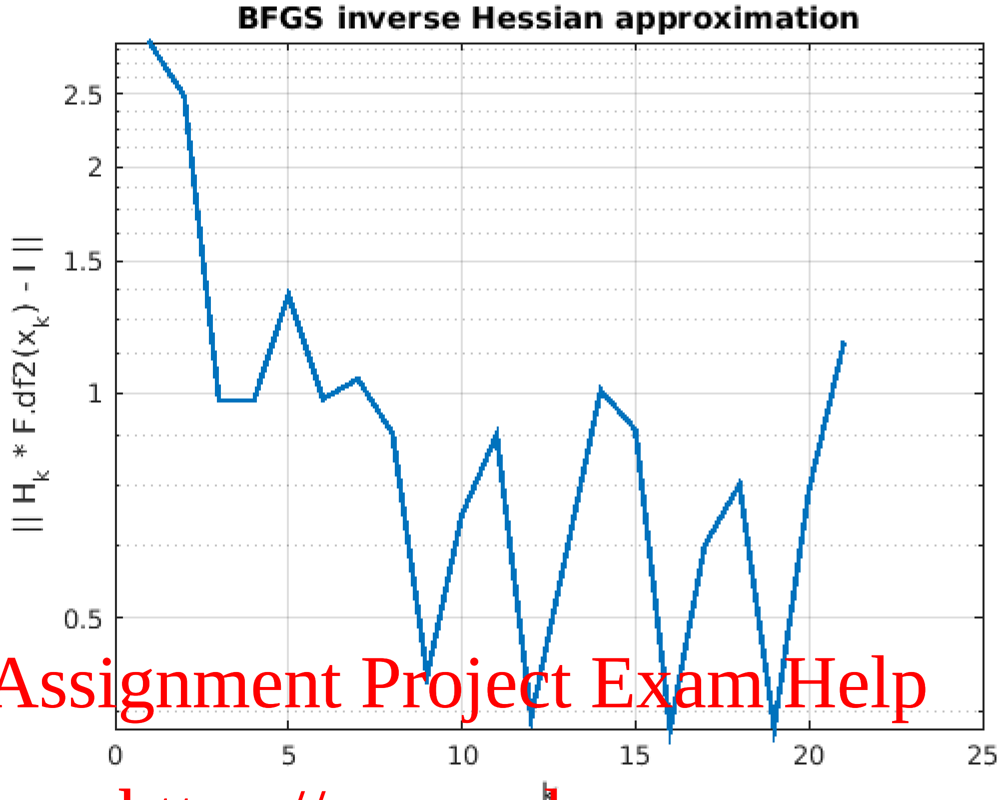

Trajectory: line search

```
visualizeConvergence(infoLS_BFGS,X,Y,Z,'final');
box on;
title(['BFGS: ' num2str(size(infoLS_BFGS.xs,2))]);
saveas(gcf, '../figs/01_BFGS_trajectory', 'png');
```



H_k * F.df2(x_k) - I


```
difNorm = [];
for n = 1:nIterLS_BFGS
    B = F.d2f(infoLS_BFGS.xs(:, n))
    H_BFGS = infoLS_BFGS.H{n}(eye(length(x0)));
    difNorm = [difNorm norm(eye(length(x0)) - B*H_BFGS)];
end
figure;
semilogy(difNorm, 'LineWidth', 2);
title('BFGS inverse Hessian approximation') ;
ylabel(' || H_k * F.df2(x_k) - I || ');
xlabel('k');
grid on;
saveas(gcf, '../figs/02_BFGS_hessian', 'png');
```



Minimise f using trust region methods

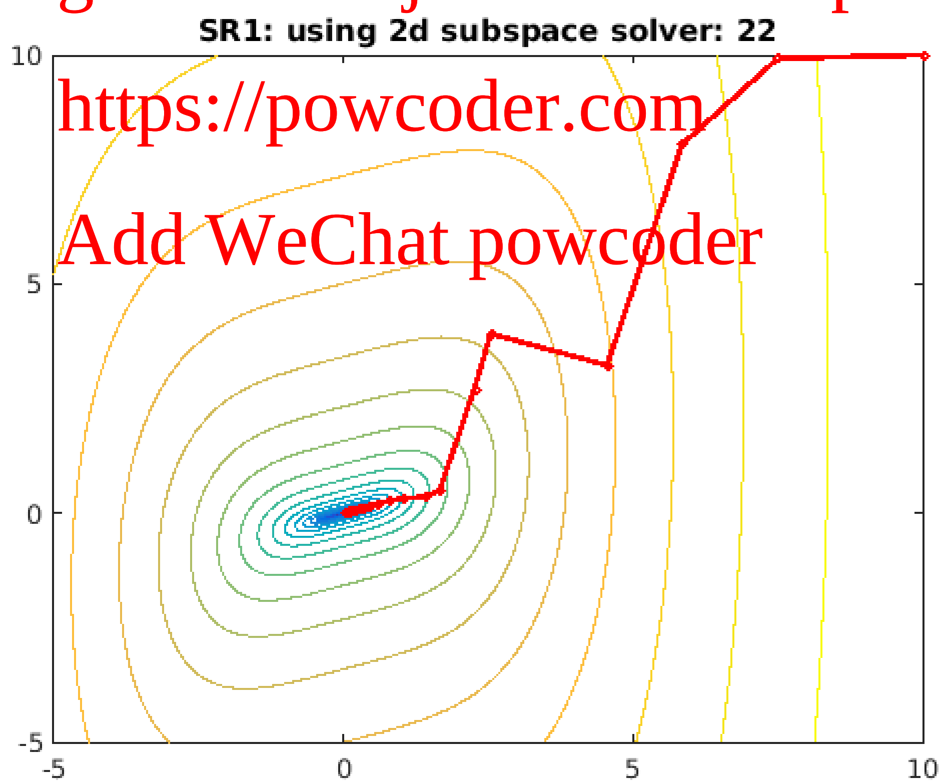

```
%===============================================================================
% Trust region parameters
maxIter = 200;
eta = 0.2;  % Step acceptance relative progress threshold
Delta = 5; % Trust region radius

% Minimisation with 2d subspace and dogleg trust region methods
Fsr1 = rmfield(F,'d2f');
[xTR_SR1, fTR_SR1, nIterTR_SR1, infoTR_SR1] = trustRegion(Fsr1, x0,
 @solverCM2dSubspaceExt, Delta, eta, tol, maxIter, debug)
```

```
xTR_SR1 =

    0.0268
    0.0089


fTR_SR1 =

   5.1313e-07


nIterTR_SR1 =
```

```
    21


infoTR_SR1 =

          xs: [2x22 double]
        xind: [1 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
 21]
           B: {1x21 cell}
        rhos: [1x21 double]
      Deltas: [1x21 double]
    stopCond: 1
```

Trajectory: trust region

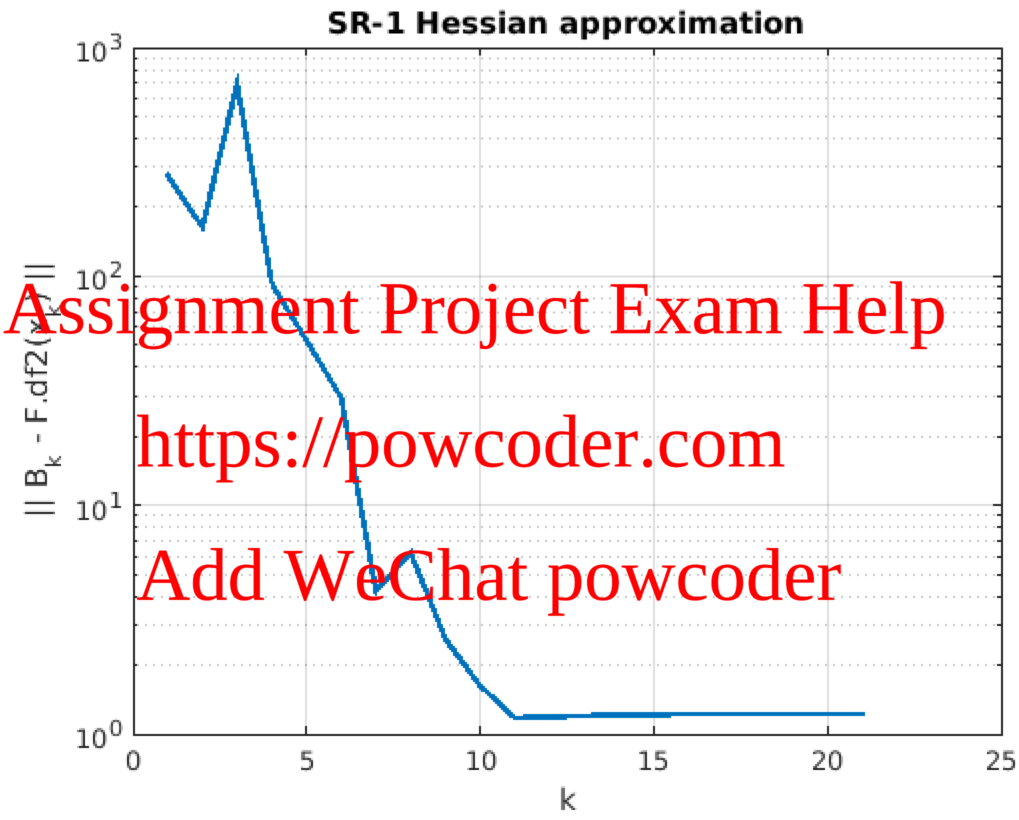
```
visualizeConvergence(infoTR_SR1,X,Y,Z,'final');
box on;
title(['SR1: using 2d subspace solver: '
 num2str(size(infoTR_SR1.xs,2))])
saveas(gcf, '../figs/03_TR_trajectory', 'png');
```

B_k - F.df2(x_k)

```
difNorm = [];
for n = 1:min(length(infoTR_SR1.B), length(infoTR_SR1.xs))
```

```
    B = F.d2f(infoTR_SR1.xs(:, n));
    B_SR1 = infoTR_SR1.B{n}(eye(length(x0)));
    difNorm = [difNorm norm(B-B_SR1)];
end
figure;
semilogy(difNorm, 'LineWidth', 2);
title('SR-1 Hessian approximation');
ylabel(' || B_k - F.df2(x_k) || ');
xlabel('k');
grid on;
saveas(gcf, '../figs/04_TR_hessian', 'png');
```



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

descentLineSearch.m

Descent line search

```
function [xMin, fMin, nIter, info] = descentLineSearch(F, descent, ls,
 alpha0, x0, tol, maxIter)
% DESCENTLINESEARCH Wrapper function executing  descent with line
 search
% [xMin, fMin, nIter, info] = descentLineSearch(F, descent, ls,
 alpha0, x0, tol, maxIter)
%
```

```
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
%   - d2f: Hessian handler
% descent: specifies descent direction {'steepest', 'newton', 'bfgs'}
% alpha0: initial step length
% ls: line search algorithm
% x0: initial iterate
% tol: stopping condition on minimal allowed step
%      norm(x_k - x_k_1)/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  Marta M. Betcke, Kiko Rullan

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

% Extract inverse Hessian approximation handler
extractH = 1;

% Initialization
nIter = 0;
x_k = x0;
info.xs = x0;
info.alphas = alpha0;
stopCond = false;

switch lower(descent)
  case 'bfgs'
    H_k = @(y) y;
    % Store H matrix in columns
    info.H = [];
end

% Loop until convergence or maximum number of iterations
while (~stopCond && nIter <= maxIter)

  % Increment iterations
    nIter = nIter + 1;

    % Compute descent direction
    switch lower(descent)
      case 'steepest'
        p_k = -F.df(x_k); % steepest descent direction
      case 'newton'
```

```
      p_k = -F.d2f(x_k)\F.df(x_k); % Newton direction
      if p_k'*F.df(x_k) > 0 % force to be descent direction (only
active if F.d2f(x_k) not pos.def.)
        p_k = -p_k;
      end
    case 'bfgs'
      p_k = -H_k(F.df(x_k));
  end

  % Call line search given by handle ls for computing step length
  alpha_k = ls(x_k, p_k, alpha0);

  % Update x_k and f_k
  x_k_1 = x_k;
  x_k = x_k + alpha_k*p_k;

  switch lower(descent)
    case 'bfgs'
      % Compute s_k and y_k
      s_k = x_k - x_k_1;
      y_k = F.df(x_k) - F.df(x_k_1);
      % Verify s_k^T*y_k > 0
      if s_k'*y_k <= 0
        error('Positivity condition: <s_k,y_k> > 0 not satisfied.
Check if LS satisfied Wolfe conditions.');
      end

      if nIter == 1
        % Update initial guess H_0. Note that initial p_0 = -
F.df(x_0) and x_1 = x_0 + alpha * p_0.
        disp(['Rescaling H0 with ' num2str((s_k'*y_k)/(y_k'*y_k)) ])
        H_k = @(x) (s_k'*y_k)/(y_k'*y_k) * x;
      end

      % Efficient update H
      rho_k = 1/(s_k'*y_k);
      projL = @(x) (x - rho_k*s_k*(y_k'*x));
      projR = @(x) (x - rho_k*y_k*(s_k'*x));
      H_k = @(x) projL( H_k( projR(x) ) ) + rho_k*s_k*(s_k'*x);

      if extractH
          % Extraction of H_k as handler
          info.H{length(info.H)+1} = H_k;
      end
  end


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

  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(F.df(x_k), 'inf') < tol*(1 + abs(F.f(x_k))));
    end

end

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

trustRegion.m

Trust region

```
function [x_k, f_k, k, info] = trustRegion(F, x0, solverCM, Delta,
 eta, tol, maxIter, debug, F2)
% TRUSTREGION Trust region iteration
% [x_k, f_k, k, info] = trustRegion(F, x0, solverCM, Delta, eta, tol,
 maxIter, debug, F2)
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
%   optional, if missing quasi Newton (sr1) is used
%   - d2f: Hessian handler
% x_k: current iterate
% solverCM: handle to solver to quadratic constraint trust region
 problem
% Delta: upper limit on trust region radius
% eta: step acceptance relative progress threshold
% tol: parameter in stopping condition (specified with stopType
%      parameter inside the function)
%      on minimal allowed step
%        norm(x_k - x_k_1)/norm(x_k) < tol
%      or norm of the gradient
%        norm(F.df(x_k), 'inf') < tol*(1 + abs(F.f(x_k)))
% maxIter: maximum number of iterations
% debug: debugging parameter switches on visualization of quadratic
 model
%        and various step options. Only works for functions in R^2
% F2: needed if debug == 1. F2 is equivalent of F but formulated as
 function of (x,y)
%     to enable meshgrid evaluation
% OUTPUT
% x_k: minimum
% f_k: objective function value at minimum
% k: number of iterations
% info: structure containing iteration history
```

```
%   - xs: taken steps
%   - xind: iterations at which steps were taken
%   - stopCond: shows if stopping criterium was satisfied, otherwsise
 k = maxIter
%
% Reference: Algorithms 4.1, 6.2 in Nocedal Wright
%
% Copyright (C) 2017 Marta M. Betcke, Kiko Rullan

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

% Extract Hessian approximation handler
extractB = 1;

% Initialisation
Delta_k = 0.5*Delta;
r = 1e-8; % skipping tolerance for SR-1 update


stopCond = false;
k = 0;
x_k = x0;
nTaken = 0;

info.xs = zeros(length(x0), maxIter);
info.xs(:,1) = x0;
info.xind = zeros(1,maxIter);
info.xind(1) = 1;

% Determine if Hessian is available, if not quasi Newton method SR-1
 is used.
sr1 = ~isfield(F, 'd2f');
if sr1
  % Initialise with B_0
  F.d2f = @(x) eye(length(x0)); % B_0 = Identity matrix.
  info.B = [];
end

while ~stopCond && (k < maxIter)
  k = k+1;

  % Construct and solve quadratic model
  Mk.m = @(p) F.f(x_k) + F.df(x_k)'*p + 0.5*p'*F.d2f(x_k)*p;
  Mk.dm = @(p) F.df(x_k) + F.d2f(x_k)*p;
  Mk.d2m = @(p) F.d2f(x_k);

  p = solverCM(F, x_k, Delta_k); % for SR-1 s_k = p_k
  if sr1
    % Compute y_k. Note, that B update happens even if the step is not
 taken.
    y_k = F.df(x_k + p) - F.df(x_k);
  end
```

```
  if debug
    % Visualise quadratic model and various steps
    figure(1); clf;
    plotTaylor(F2, x_k, [x_k - 4*Delta_k, x_k + 4*Delta_k], Delta_k,
 p);
    hold on,
    g = -F.df(x_k);
    gu = -F.d2f(x_k)\g;
    plot(x_k(1) + g(1)*Delta_k/norm(g), x_k(2) + g(2)*Delta_k/
norm(g), 'rs')
    plot(x_k(1) + gu(1)*Delta_k/norm(gu), x_k(2) + gu(2)*Delta_k/
norm(gu), 'bo')
    pause
  end

  % Evaluate actual to predicted reduction ratio
  rho_k = (F.f(x_k) - F.f(x_k + p)) / (Mk.m(0*p) - Mk.m(p)) ;
  if (Mk.m(0*p) < Mk.m(p))
    disp(strcat('Ascent - iter ', num2str(k)))
  end
  % Record iteration information
  info.rhos(k) = rho_k;
  info.Deltas(k) = Delta_k;

  if rho_k < 0.25
    % Shrink trust region
    Delta_k = 0.25*Delta_k;
  else
    if rho_k > 0.75 && abs(p'*p - Delta_k^2) < 1e-12
      % Expand trust region
      Delta_k = min(2*Delta_k, Delta);
    end
  end

  % Accept step if rho_k > eta
  x_k_1 = x_k; % if step is not accepted x_k_1 = x_k
  if rho_k > eta
  %  x_k_1 = x_k;
    x_k = x_k + p;

    % Record all taken steps including iteration index
    nTaken = nTaken + 1;
    info.xs(:,nTaken+1) = x_k;
    info.xind(nTaken+1) = k;

    % Evaluate stopping condition:
    switch stopType
      case 'step'
        % relative step length
        stopCond = (norm(x_k - x_k_1)/norm(x_k_1) < tol);
      case 'grad'
        % gradient norm
        %stopCond = (norm(F.df(x_k)) < tol);
```

```
        stopCond = (norm(F.df(x_k), 'inf') < tol*(1 + abs(F.f(x_k))));
      end
    elseif Delta_k < 1e-6*Delta
      % Stop iteration if Delta_k shrank below 1e-6*Delta. Otherwise, if
 the model
      % does not improve inspite of shinking, the algorithm would shrink
 Delta_k indefinitely.
      warning('Region of interest is to small. Terminating iteration.')
      break;
    end

    % SR1 quasi Netwon: Hessian update
    % Note: this implementation constructs Hessian matrices explicitely
 hence is not suitable for large scale.
    % Efficient large scale implementation either needs to use iterative
 solvers to invert the Hessian inside solverCM
    % or needs to update the Hessian and its inverse at the same time.
    % Then they both can be implemented as their action on a vector
 H_k(x) = H_k*x, B_k(x) = B_k*x as for H_k in BFGS.
    if sr1
      % Residual vector of the secant equation.
      % B_k_1 = F.d2f(x_k_1). This is legacy notation as F.d2f =
 F.d2f(x_k_1) is actually a matrix and hence does not depend on x.
      rSec = y_k - F.d2f(x_k_1)*p; % recall notation s_k = p_k,

      % Update Hessian approximation if condition holds, otherwise skip
 update.
      if (abs(rSec'*p) >= r*norm(p)*norm(rSec))
        F.d2f = @(x) F.d2f(x_k_1) + (rSec*rSec')/(rSec'*p); % this
 update is only valid at x_k, F.d2f = F.d2f(x_k) is a matrix.
      end

      if extractB
        % Extraction of B_k function handler
        info.B{length(info.B)+1} = F.d2f;
      end
    end
  end

  f_k = F.f(x_k);
  info.stopCond = stopCond;
  info.xs(:,nTaken+2:end) = [];
  info.xind(nTaken+2:end) = [];
  info.rhos(k+1:end) = [];
  info.Deltas(k+1:end) = [];
```


Published with MATLAB® R2015a