COMPGV19: Tutorial 3

Table of Contents

Marta Betcke and Kiko Rul#lan

Exercise 1

```
Derive the 2d subspace trust region method. Note that
  - when p is constraint to a subspace V, it can be expressed as a linea
      of basis vectors p = V*a. You can use any basis, here orthonormal
  - use the result in Theorem 4.1 to obtain optimal p. Observe that comp
    condition (2nd equation) results in two cases;
  - use the 1st equation to obtain an explicit expression for each coeff
    and plug them into the remaining condition;
```

Exercise 2

```
Implement the 2d subspace trust region method.
```

Exercise 3

```
Implement a trust region function based on Algorithm 4.1 in Nocedal Wrig
Let this function take a handle to a solver for the constraint quadratic
as an argument. This will allow us to plug in different solvers to obtai
trust region methods.
```

Exercise 4

```
Apply the 2d subspace trust region method to rosenbrock function with a
point (1.2, 1.2) and a farther away point (-1.2, 1). Pay attention to th
region radius in each case.
```

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

Rosenbrock function

```
% 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)
F2.f = @(x,y) 100.*(y - x.^2).^2 + (1 - x).^2;
F2.dfx = @(x,y) -400.*(y - x.^2).*x - 2.*(1 - x);
F2.dfy = @(x,y) 200.*(y - x.^2);
F2.d2fxx = @(x,y) -400.*(y - 3*x.^2) + 2;
F2.d2fxy = @(x,y) -400.*x;
F2.d2fyx = @(x,y) -400.*x;
F2.d2fyy = @(x,y) 200;
```

Parameters

```
% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 100;
% Stopping tolerance on relative step length between iterations
tol = 1e-6;
% Debugging paramter will switch on step by step visualisation of
 quadratic model and various step options
debug = 0;
```

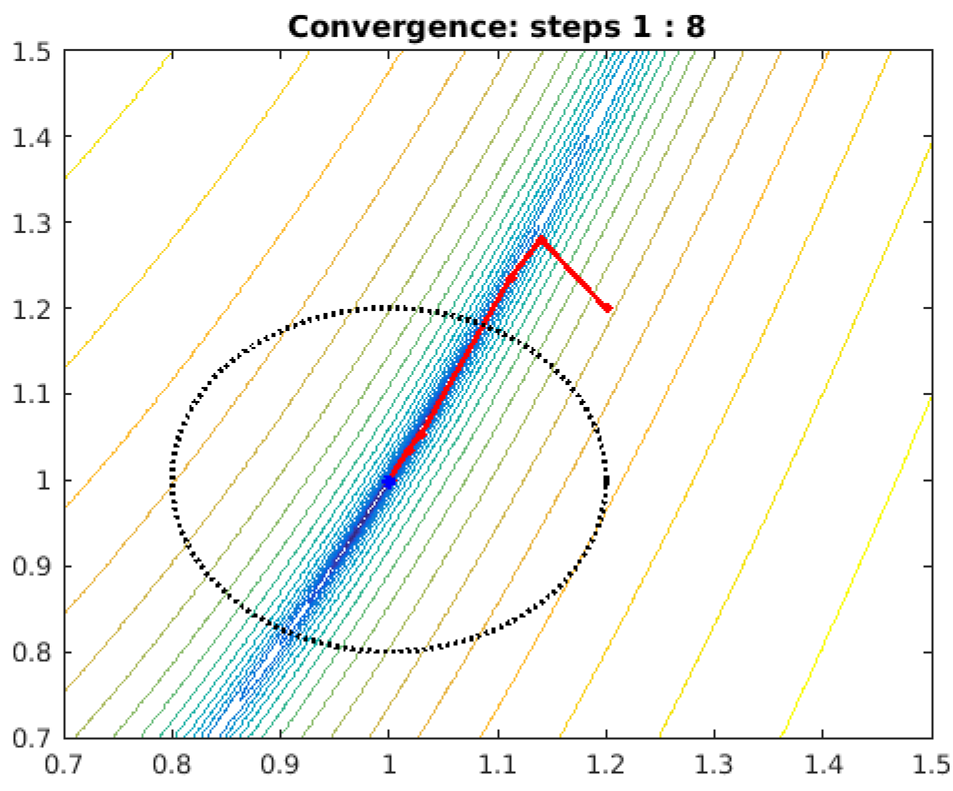
Trust region with 2d subspace, $x_0 = (1.2, 1.2)^T$

```
x0 = [1.2; 1.2];
% Trust region radius
Delta = 0.2; %[0.2, 1) work well, below many iterations.

[xTR, fTR, nIterTR, infoTR] = trustRegion(F, x0,
 @solverCM2dSubspaceExt, Delta, eta, tol, maxIter, debug, F2)


xTR =

    1.0000
    1.0000


fTR =

   6.7549e-17


nIterTR =

     7
```

```
infoTR =

          xs: [2x8 double]
        xind: [1 1 2 3 4 5 6 7]
        rhos: [1.0283 1.0700 0.6784 1.0443 1.0049 1.0018 1.0001]
      Deltas: [0.1000 0.2000 0.2000 0.2000 0.2000 0.2000 0.2000]
    stopCond: 1
```

Visualize

```
% Define grid for visualisation
n = 300;
x = linspace(0.7,1.5,n+1);
y = x;
[X,Y] = meshgrid(x,y);
Z = log(max(F2.f(X,Y), 1e-3));

% Iterate plot one by one to see the order in which step are taken
visualizeConvergence(infoTR,X,Y,Z,'iterative')
```



Trust region with 2d subspace, $x_0 = (-1.2, 1)^T$

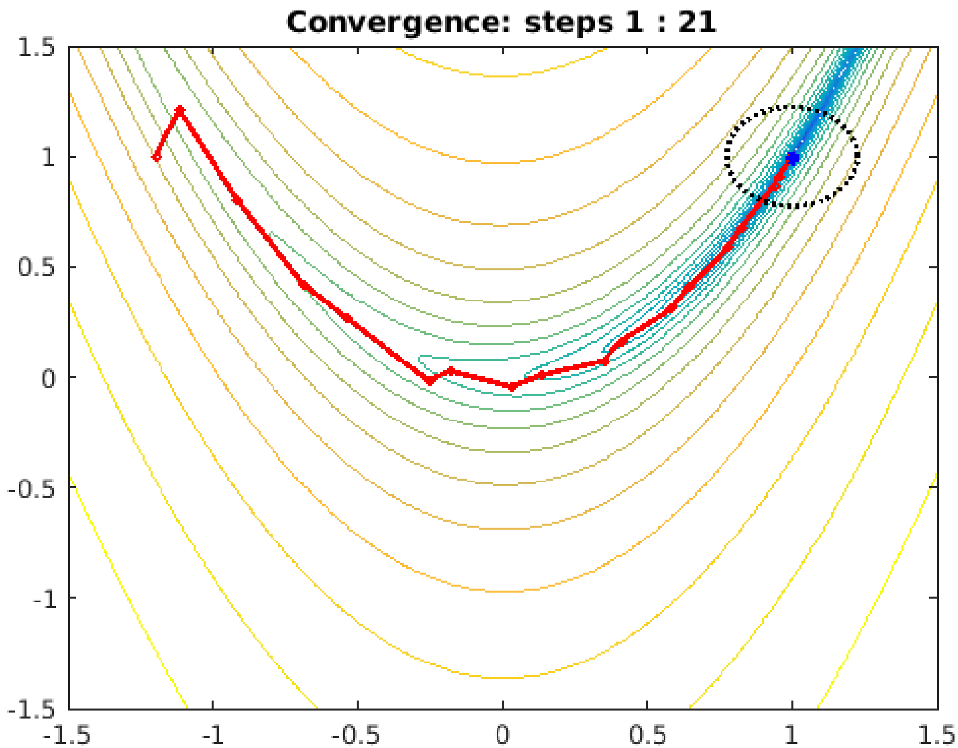
```
x0 = [-1.2; 1];
```

```
maxIter = 500;
Delta = 0.45; %[0.3, 0.5] works with exception 0.4?, otherwise to many
 iterations.

[xTR, fTR, nIterTR, infoTR] = trustRegion(F, x0,
 @solverCM2dSubspaceExt, Delta, eta, tol, maxIter, debug, F2)
```

```
xTR =

    1.0000
    1.0000


fTR =

   8.3049e-22


nIterTR =

    21


infoTR =

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

Visualize

```
% Define grid for visualisation
n = 300;
x = linspace(-1.5,1.5,n+1);
y = x;
[X,Y] = meshgrid(x,y);
Z = log(max(F2.f(X,Y), 1e-3));

% Iterate plot one by one to see the order in which step are taken
visualizeConvergence(infoTR,X,Y,Z,'iterative')
```



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

trustRegion.m

Wrapper function executing trust region iteration taking handle to a solver for the constraint model problem

```
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
%   - 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: stopping condition on minimal allowed step
%      norm(x_k - x_k_1)/norm(x_k) < tol;
% 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: Algorithm 4.1 in Nocedal Wright
%
% Copyright (C) 2017 Marta M. Betcke, Kiko Rullan

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

% Initialisation
Delta_k = 0.5*Delta;

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;


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);

  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
    % Shink 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 = max(2*Delta_k, Delta);
    end
  end

  % Accept step if rho_k > eta
  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
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) = [];
```

solverCM2dSubspaceExt.m

2d subspace solver for the quadratic constraint model problem

```
function p = solverCM2dSubspaceExt(F, x_k, Delta)
% SOLVERCM2DSUBSPACEEXT Solves quadratic constraint trust region
 problem via 2d subspace
% p = solverCM2dSubspace(F, x_k, Delta)
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
%   - d2f: Hessian handler
% x_k: current iterate
% Delta: trust region radius
% OUTPUT
% p: step (direction times lenght)
%
% Copyright (C) 2017 Marta M. Betcke, Kiko Rullan

% Compute gradient and Hessian
g = F.df(x_k);
B = F.d2f(x_k);

% Eigenvalues of Hessian. If B is large, Lanczos - eigs - should be
 used
lambdasB = eig(B);
lambdaB1 = min(lambdasB); %smallest eigenvalue


% Special cases if B has
% 1) negative eigenvalues
% 2) zero eigenvalues
if min(abs(lambdasB)) < eps % zero eigenvalue(s)
  % Take Cauchy point step
  gTBg = g'*(B*g);
  if gTBg <= 0
    tau = 1;
  else
    tau = min(norm(g)^3/(Delta*gTBg), 1);
  end
  p = -tau*Delta/norm(g)*g;
  return;
```

```
elseif lambdaB1 < 0 % negative eigenvalue(s)
  alpha = -1.5*lambdaB1; % shift ensuring that B + alpha*I is p.d.

  B = (B + alpha*eye(length(x_k)));
  pNewt = B\g; %2nd order direction
  if norm(pNewt) <= Delta
    npNewt = pNewt/norm(pNewt);
    v = randn(size(x_k));
    v = v/norm(v);
    v = -0.1*npNewt + 0.1*(v - npNewt*npNewt'*v); % v: v'*pNewt <= 0

    p = -pNewt + v;  % ensure ||p|| >= ||pNewt||
    %p = -1.1*pNewt;  % ensure ||p|| >= ||pNewt||
    %p = -npNewt*Delta;  % ensure ||p|| >= ||pNewt||
    return;

  else
    % Orthonormalize the 2D projection subspace
    V = orth([g, pNewt]);
  end
else %positive eigenvalues

% Orthonormalize the 2D projection subspace
  V = orth([g, B\g]);
end

% Check if gradient and Newton steps are collinear. If so return
 Cauchy point.
if size(V,2) == 1
  % Calculate Cauchy point
  gTBg = g'*(B*g);
  if gTBg <= 0
    tau = 1;
  else
    tau = min(norm(g)^3/(Delta*gTBg), 1);
  end
  p = -tau*Delta/norm(g)*g;
  return;
end

% To constraint the optimisation to subspace span([g, B\g]),
% we express the solution i.e. the direction a linear combination
% p = V*a with 'a' being a vector of two coefficients.
% Substituting p = V*a into the quadratic model
%
%     m(p) = f(x_k) + g'*p + 0.5*p'*B*p with g = df(x_k), B =
 d2f(x_k)
%     s.t. p'*p <= Delta^2
%
% we obtain the projected model, which is a quadratic model for 'a'
%
%     mv(a) = f(x_k) + gv'*a + 0.5*a'*Bv*a
%     s.t. a'*a <= Delta^2      (due to V'*V = I)
```

```
%
% Furthermore, as long as V has a full rank (g and B\g
% are not collinear), if B is s.p.d. so is Bv.
% Note, that if g = c*B\g the problem becomes 1D.

% Project on V
Bv = V'*(B*V);
gv = V'*g;

% To solve the projected model mv subject to p'*p <= Delta^2
% we make use of Theorem 4.1 Nocedal Wiright.
% From this theorem for mv we have that 'a'
% minimizes mv s.t. a'*a <= Delta^2 iff
%
%     (Bv + lambda*I) * a = -gv,     lambda >= 0
%     lambda * (Delta^2 - a'*a) = 0
%     (Bv + lambda * I) is s.p.d.
%
% This gives two cases:
% (1) lambda = 0  &  a'*a < Delta^2 (the unconstraint solution
%     is inside the trust region).
%     Then the first equation becomes Bv * a = -gv i.e. a = -Bv\gv;
% (2) lambda>= 0  &  a'*a = Delta^2 (the constraint is active)
%     Then we can solve the first equation
%     (E1)    a = -(Bv + lambda*I) \ gv
%     The additional equation is provided by the constraint
%     (E2)    a'*a = Delta^2
%     To solve this system we make use or eigendecomposition
%     of Bv = Q*Lambdas*Q' with Q orthonormal
%         Q'*a = - inv(Lambdas + lambda*I) * Q'*gv
%     and realise that (Q'*a)'*(Q'*a) = a'*Q*Q'*a = a'*a.
%     We denote Qa = Q'*a and Qg = Q'*gv.
%     For ith element on Qa,
%         Qa(i) = - 1/(lambdas(i) + lambda) * Qg(i),
%     with lambdas(i) = Lambdas(i,i).
%     Substituting Qa into Qa'*Qa = Qa(1)^2 + Qa(2)^2 = Delta^2 we
 obtain
%         Qg(1)^2/(lambdas(1) + lambda)^2 + Qg(2)^2/(lambdas(2) +
 lambda)^2 = Delta^2
%     which we transform to 4th degree polynomial in lambda
%     (assuming that lambdas(i) + lambda > 0)
%         r(1) lambda^4 + r(2) lambda^3 + r(3) lambda^2 + r(4) lambda
 + r(5) = 0


% Case (1)
%if lambdaB1 > 0
% Compute unconstrained solution and check if it lies in the trust
 region
a = -Bv\gv;
if a'*a < Delta^2
  % Compute the solution p
  p = V*a;
  return;
```

```
end
%end

% Case (2)
[Q, Lambdas] = eig(Bv);
lambdas = diag(Lambdas);
Qg = Q'*gv;

r(5) = Delta^2*lambdas(1)^2*lambdas(2)^2 - Qg(1)^2*lambdas(2)^2 -
 Qg(2)^2*lambdas(1)^2;

r(4) = 2*Delta^2*lambdas(1)^2*lambdas(2) +
 2*Delta^2*lambdas(1)*lambdas(2)^2 ...
      -2*Qg(1)^2*lambdas(2) - 2*Qg(2)^2*lambdas(1);

r(3) = Delta^2*lambdas(1)^2 + 4*Delta^2*lambdas(1)*lambdas(2) +
 Delta^2*lambdas(2)^2 ...
      -Qg(1)^2- Qg(2)^2;

r(2) = 2*Delta^2*lambdas(1) + 2*Delta^2*lambdas(2);

r(1) = Delta^2;

% Compute roots of the polynomial and select positive one
rootsR = roots(r);
%rootsR = rootsR(rootsR >= 0);
lambda = min(rootsR(rootsR + min(lambdas) > 0));
%lambda = min(rootsR);


% Compute a from   Qa(i) = - 1/(lambdas(i) + lambda) * Qg(i)
a = Q* ( (-1./(lambdas(:) + lambda)) .* Qg);
% Compute the solution p
p = V*a;
% Renormalize to ||p|| = Delta, because the condition number of the
 polynomial root finder is high
p = Delta/norm(p)*p;
```

visualizeConvergence.m

Visualization function: plots iterates over the countour plot

```
function visualizeConvergence(info,X,Y,Z,mode)
% VISUALIZECONVERGENCE Convergence plot of iterates
% visualizeConvergence(info,X,Y,Z,mode)
% INPUTS
% 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
%   - Deltas: trust region radii
%   - rhos: relative progress
% X,Y: grid as returned by meshgrid
% Z: objective function evaluated on the grid
% mode: choose from {'final', 'iterative'}
%   'final': plot all iterates at once
%   'iterative': plot the iterates one by on to see the order in which
 steps are taken
%
% Copyright (C) 2017 Marta M. Betcke, Kiko Rullan

figure;
hold on;
% Plot contours of Z - function evaluated on grid
contour(X, Y, Z, 20);

switch mode
  case 'final'
    % Plot all iterations
    plot(info.xs(1, :), info.xs(2, :), '-or', 'LineWidth',
 2, 'MarkerSize', 3);
    title('Convergence')

  case 'iterative'
    % Plot the iterates one by one to see the order in which steps are
 taken
    nIter = size(info.xs,2);

    for j = 1:nIter,
      hold off; contour(X, Y, Z, 20); hold on
      plot(info.xs(1, 1:j), info.xs(2, 1:j), '-or', 'LineWidth',
 2, 'MarkerSize', 3);
      plot(info.xs(1, j), info.xs(2, j), '-*b', 'LineWidth',
 2, 'MarkerSize', 5);

      if isfield(info, 'Deltas') && j > 2
        plot(info.xs(1,
 j-1)+cos(0:0.05:2*pi)*info.Deltas(info.xind(j)), ...
             info.xs(2,
 j-1)+sin(0:0.05:2*pi)*info.Deltas(info.xind(j)), ...
             ':k', 'LineWidth', 2);
      end

      title(['Convergence: steps 1 : ' num2str(j)])
      pause(1);
    end

end
```

plotTaylor.m

Visualization function: plots quadratic model and various step options. To activate this level of visualization, set debug = 1

```
function plotTaylor(F2, x_k, xlim, Delta, p)
% PLOTTAYLOR Contour plot of 2nd order Taylor polynomial at x_k
% plotTaylor(F2, x_k, xlim, Delta, p)
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
%   - d2f: Hessian handler
% xlim: plotting region
% Delta: upper limit on trust region radius
% p: computed direction (with correct length)
%
% Copyright (C) 2017 Marta M. Betcke, Kiko Rullan

% % Quadratic model as a function of (x,y)
% m = @(x,y) F2.f(x,y) + F2.dfx(x,y).*p(1) + F2.dfy(x,y).*p(2) ...
%            + 0.5 .* ( F2.d2fxx(x,y).*p(1).^2 +
 F2.d2fxy(x,y).*p(1).*p(2) ...
%                    +  F2.d2fyx(x,y).*p(1).*p(2) +
 F2.d2fyy(x,y).*p(2).^2 );

% Quadratic model as a function of (px,py) = (x_k+1 - x_k, y_k+1 -
 y_k) ->  (x,y):=(x_k+1, y_k+1) = (x_k, y_k) + (px, py)
m = @(x,y) F2.f(x_k(1),x_k(2)) + F2.dfx(x_k(1),x_k(2)).*(x-x_k(1)) +
 F2.dfy(x_k(1),x_k(2)).*(y-x_k(2)) ...
           + 0.5 .* ( F2.d2fxx(x_k(1),x_k(2)).*(x-x_k(1)).^2
 + F2.d2fxy(x_k(1),x_k(2)).*(x-x_k(1)).*(y-x_k(2)) ...
                   +  F2.d2fyx(x_k(1),x_k(2)).*(x-x_k(1)).*(y-x_k(2))
 + F2.d2fyy(x_k(1),x_k(2)).*(y-x_k(2)).^2 );



% Define grid for visualisation
n = 300;
x = linspace(xlim(1,1),xlim(1,2),n+1);
y = linspace(xlim(2,1),xlim(2,2),n+1);
[X,Y] = meshgrid(x,y);

% Evaluate and plot model
Z = m(X,Y);
contour(X, Y, Z, 20); hold on;
% Plot current iterate x_k
plot(x_k(1), x_k(2), '-xk', 'LineWidth', 2, 'MarkerSize', 5);
% Plot trust region around x_k
if nargin  > 3
  plot(x_k(1)+cos(0:0.01:2*pi)*Delta,
 x_k(2)+sin(0:0.01:2*pi)*Delta, ':k', 'LineWidth', 2);
```

```
end
% Plot the new iterate x_k + p
if nargin  > 4
  plot(x_k(1)+p(1), x_k(2)+p(2), '-xb', 'LineWidth', 2, 'MarkerSize',
 5);
end
legend('m_k', 'x_k', 'trust region', 'x_k+p')
```

Published with MATLAB® R2015a