# COMPGV19: Coursework 1 - Theoretical Questions and Commentary

Antonio Remiro Azócar, MSc Machine Learning

27 February 2017

**Note:** This part of the hand-in provides answers to the theoretical questions (**1a-c**, **2a**, **2c**) and an extended explanation of the MATLAB implementation questions (**1d-f**,**2b**,**2d-e**). These explanations are supplemented by the second part of the hand-in (code).

## Exercise 1

**(1a)** The Hessian $\mathbf{H}$ of $f$ is computed as follows:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}.$$

We have,

$$\frac{\partial f}{\partial x} = 2x + 20x^3,$$
$$\frac{\partial f}{\partial y} = 20y.$$

Then,

$$\mathbf{H} = \begin{pmatrix} 2 + 60x^2 & 0 \\ 0 & 20 \end{pmatrix}.$$

$\mathbf{H}$ is a diagonal matrix. The entries on its diagonal are $h_{11} = 2 + 60x^2$ and $h_{22} = 20$. Both $h_{11} > 0$ and $h_{22} > 0$ for all $(x, y) \in \mathbb{R}^2$. Hence the matrix is positive defined for all $(x, y) \in \mathbb{R}^2$ ($\langle \mathbf{v}, \mathbf{Hv} \rangle > 0$ for all $\mathbf{v} \in \mathbb{R}^2$ ) .

**(1b)** This problem brings together the following conditions. We have the gradient descent sequence:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \gamma \nabla F(\mathbf{x}_t), \qquad t \geq 0,$$

where $\gamma$ is a step size, $F$ is our function and $\mathbf{x}_1, ..., \mathbf{x}_T$ is our sequence of points. In this case, we require the steepest descent method to converge in one step. We have $F = x^2 + 5x^4 + 10y^2$. $F$ has a minimum at (0,0). Therefore for $t \geq 1$, we do not expect $\mathbf{x}_t$ to change, with:

$$\mathbf{x}_t = \begin{pmatrix} x_t \\ y_t \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad t \geq 1.$$

At $t = 1$, we have $\mathbf{x}_1 = \mathbf{0}$. Then using the definition of steepest descent,

$$\mathbf{0} = \mathbf{x}_0 - \gamma \nabla F(\mathbf{x}_0) = \begin{pmatrix} x_0 - 2\gamma x_0 - 20\gamma x_0^3 \\ y_0 - 20\gamma y_0 \end{pmatrix}.$$

Firstly, we encounter the trivial solution,

$$x_0 = 0, y_0 = 0 \qquad \gamma \in \mathbb{R}$$

We also have $y_0 = 20\gamma y_0$. Substituting $\gamma = \frac{1}{20}$ into the $x_0$ equation gives:

$$0 = x_0 - \frac{1}{10}x_0 - x_0^3 \rightarrow 9x_0 = 10x_0^3 \rightarrow x_0^2 = \frac{9}{10}.$$

Then, the steepest descent method converges in one iteration for starting points $x_0 = \pm\frac{3}{\sqrt{10}}$, $y_0 \in \mathbb{R}$, given $\gamma = \frac{1}{20}$. Note also that given, $y_0 = 0$, we have the more general solution:

$$0 = x_0 - 2\gamma x_0 - 20\gamma x_0^3 \rightarrow (1 - 2\gamma) = 20x_0^2 \rightarrow x_0 = \pm\sqrt{\frac{1 - 2\gamma}{20\gamma}}, \qquad \gamma \leq \frac{1}{2}, \gamma \neq 0.$$

In conclusion, we have a solution $x_0 = \pm\sqrt{\frac{1-2\gamma}{20\gamma}}$ for $\gamma \leq \frac{1}{2}, \gamma \neq 0$. This is accompanied by $y_0 \in \mathbb{R}$ in the $\gamma = \frac{1}{20}$ case (subbing this value in the expression above yields $x_0 = \pm\frac{3}{\sqrt{10}}$). Otherwise we have a $y_0 = 0$ restriction. We have also a trivial solution $x_0, y_0 = 0$ (has already converged).

**(1c)** The true Hessian is denoted as $B$. We denote the true gradient (parallel to steepest descent) as $g$. We encounter the eigenvalue problem,

$$\lambda g = B^{-1} g. \tag{1}$$

From **1a**, for this particular problem:

$$B = \begin{pmatrix} 2 + 60x^2 & 0 \\ 0 & 20 \end{pmatrix}.$$

Then,

$$B^{-1} = \frac{1}{20(2 + 60x^2)} \begin{pmatrix} 20 & 0 \\ 0 & 2 + 60x^2 \end{pmatrix} = \begin{pmatrix} \frac{1}{2+60x^2} & 0 \\ 0 & \frac{1}{20} \end{pmatrix}.$$

Substituting this result into (1),

$$\lambda g = \begin{pmatrix} \frac{1}{2+60x^2} & 0 \\ 0 & \frac{1}{20} \end{pmatrix} g.$$

The gradient $g$ is given by,

$$g = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} 2x + 20x^3 \\ 20y. \end{pmatrix}$$

Then, (1) becomes:

$$\lambda \begin{pmatrix} 2x + 20x^3 \\ 20y \end{pmatrix} = \begin{pmatrix} \frac{1}{2+60x^2} & 0 \\ 0 & \frac{1}{20} \end{pmatrix} \cdot \begin{pmatrix} 2x + 20x^3 \\ 20y \end{pmatrix} = \begin{pmatrix} \frac{x+10x^3}{1+30x^2} \\ y \end{pmatrix}.$$

If $x \neq 0$, we require the eigenvalue $\lambda = \frac{1}{2(1+30x^2)}$ for the subspace method to become degenerate. Similarly, for $y \neq 0$, we require the eigenvalue $\lambda = \frac{1}{20}$ for a degenerate subspace method. We solve for $y \neq 0$, ($\lambda = \frac{1}{20}$). We now have:

$$\frac{2x + 20x^3}{20} = \frac{x + 10x^3}{1 + 30x^2}.$$

Cancelling out $(x + 10x^3)$ ,

$$\frac{1}{10} = \frac{1}{1 + 30x^2},$$

Rearranging,

$$30x^2 = 9 \to x = \pm\frac{\sqrt{9}}{\sqrt{30}} = \pm\frac{\sqrt{3}\sqrt{3}}{\sqrt{3}\sqrt{10}} = \pm\sqrt{\frac{3}{10}}$$

The subspace method becomes degenerate for $x = \pm\sqrt{\frac{3}{10}}$, $y \neq 0$. The same result is obtained solving for $x \neq 0$.

**Note for 1d-1f**: The minimum of $f$ is [0,0].

**(1d)** We utilise the provided `descentLineSearch.m` file to perform Steepest Descent and Newton methods. For the BFGS implementation, refer to `BFGS.m` (see Appendix). We have utilised parameter values $c_1 = 10^{-4}$ $c_2 = 0.9$ for all methods, as suggested in the lecture notes. For BFGS, the Hessian is set initially as the Identity matrix. Step lengths are plotted over the number of iterations in the visualisations presented (in the Publish Output section). We can observe that Newton only requires two steps to attain a stable, appropriate, direction. In addition, Newton appears to require a lesser number of steps than BFGS/Steepest Descent to reach convergence. All implementations work correctly, converging to [0,0] as expected.

The differences between all three methods are as follows. Firstly, despite taking a greater number of steps to converge, computation times for BFGS and Steepest Descent are lower. This is because they do not require computing the Hessian and hence, are computationally cheaper with respect to computing search directions. Whereas BFGS and Steepest Descent have similar initial trajectories (caused by using the same Hessian initialisation), these end up diverging. This occurs as a result of BFGS' secant equation (Nocedal and Wright p. 24). This condition makes BFGS select more accurate Hessian approximations which mimic the properties of the true Hessian. For this reason, BFGS manages to attain a stable, horizontal direction (like Newton) in later iterations. This trade-off between accuracy (wrt Steepest Descent) and a cheaper computational cost (than Newton) makes BFGS our preferred method. Note also that whereas gradient descent guarantees global convergence, the global convergence of Newton/BFGS requires Wolfe conditions being fulfilled and positive definite Hessian approximations. In addition, Newton has the additional drawback of giving ascent directions when the current iterate differs strongly from the solution. Note that convergence is slowest for steepest descent (linear and requiring over 500 iterations to dip below the tolerance level); BFGS and Newton provide faster convergence.

**(1e)** We utilise the provided `solverCM2dSubspace.m` file to perform the 2d-subspace method. For our dogleg implementation, refer to `dogleg.m` (see Appendix). We set the delta (radius of the trust region) parameter to 0.2 as suggested in the tutorials. The dogleg method supposedly works better when the true Hessian is positive definite. The method works as follows. The dogleg method finds an approximate solution by substituting the curved trajectory for the solution with a path of two line segments. If the minimiser is found in segment number one, the Cauchy point is selected. Alternatively, the algorithm finds an 'improvement' of the Cauchy point (if the minimiser is found in the second segment).

Positive features of the dogleg method are that it is cheap computationally, converges fast (superlinear rate providing knowledge about the curvature of $f$) and that it provides better approximations than the Cauchy point. The 2d-subspace method extends the dogleg method, giving improved performance for non-positive definite Hessians and broadening the search space. It appears to converge slightly slower than dogleg, requiring two more steps. Path trajectories for dogleg and 2d-subspace look similar, 2d-subspace perhaps with a smoother trajectory. Dogleg's less smooth trajectory is caused by the two line-segment approximation suggested by its name. Note that during the first iterations, since the target is outside the trust region, both dogleg and 2d-subspace take the direction of steepest descent. Also notice how 2-d subspace attains a straight-line horizontal path when reaching the degeneracy point in **1b** ($x = \sqrt{\frac{3}{10}}$).

**(1f)** The Polak-Ribière method is implemented as suggested by Nocedal and Wright (p. 123). They propose a hybrid Polak-Ribière-Fletcher-Reeves procedure. This procedure has been implemented utilising parameter values $c_1 = 10^{-4}$, $c_2 = 0.12$ and an initial step $\alpha_0 = 1$; as suggested in the tutorials. For the Polak-Ribière (hybrid) implementation, refer to `FRPR.m`. For Fletcher-Reeves, refer to `FR.m`. By looking at convergence trajectories/step length sequences (see Publish Output), both methods seem to converge to the minimum in very similar fashion. Both implementations work correctly, converging to [0,0] as expected. Both appear to work better than the implemented line search methods. Despite not being that fast (as for example Newton, and in this case even slower than steepest descent), they are computationally cheaper as no matrix storage is involved. For a vector computation, only the previous vector is required. The differences between Polak-Ribiere and Fletcher-Reeves are as follows. Whereas Polak-Ribière provides greater numerical stability and is more robust, applying strong Wolfe conditions may result in an ascent direction. To counter this effect, Nocedal and Wright suggest choosing the maximum of (0, $\beta$). This presents another problem; the recurring presence of a zero may result in perpetual iterations. In order to correct this, Nocedal and Wright suggest utilising a Polak-Ribiére-Fletcher-Reeves hybrid. Fletcher-Reeves always converges globally. The hybrid makes use of this property. Again, by observing the convergence trajectories, we can see that both are similar. We note that Polak-Ribière (hybrid)'s convergence trajectory appears to be more stable/robust, but converges more slowly in later iterations (at first it chooses to perform Fletcher-Reeves steps and converges at the same rate).

## Exercise 2

**(2a)** We have the function,

$$f(x, y) = (1-x)^2 + (1-y)^2 + 10(x^2 + y^2 - 2)^2.$$

Using Mathematica, the following critical points ($\frac{\partial f}{\partial x} = 0$, $\frac{\partial f}{\partial y} = 0$) are found:

1. $(x, y)$=$(1, 1)$.
2. $(x, y) = (\frac{1}{20}(-10 - 3\sqrt{10}), \frac{1}{20}(-10 - 3\sqrt{10})) \approx (-0.974, -0.974)$.
3. $(x, y) = (\frac{1}{20}(-10 + 3\sqrt{10}), \frac{1}{20}(-10 + 3\sqrt{10})) \approx (-0.026, -0.026)$.

We have,

$$\frac{\partial f}{\partial x} = -2(1-x) + 40x(x^2 + y^2 - 2), \qquad \frac{\partial f}{\partial y} = -2(1-y) + 40y(x^2 + y^2 - 2),$$

$$\frac{\partial^2 f}{\partial x^2} = 120x^2 + 40y^2 - 78, \qquad \frac{\partial^2 f}{\partial x \partial y} = 80xy,$$

$$\frac{\partial^2 f}{\partial y \partial x} = 80xy, \qquad \frac{\partial^2 f}{\partial y^2} = 120y^2 + 40x^2 - 78,$$

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix} = \begin{pmatrix} 120x^2 + 40y^2 - 78 & 80xy \\ 80xy & 120y^2 + 40x^2 - 78 \end{pmatrix}.$$

Substituting our critical point values into the expressions above, we note:

1. $(x, y) = (1, 1)$ is a **local minimum** since $\det(\mathbf{H}) = 82^2 - 80^2 = 324 > 0$ and $\frac{\partial^2 f}{\partial x^2} = 82 > 0$.
2. $(x, y) = (\frac{1}{20}(-10 - 3\sqrt{10}), \frac{1}{20}(-10 - 3\sqrt{10}))$ is a **saddle point** of $f$ since $\det(\mathbf{H}) = -307.6 < 0$.
3. $(x, y) = (\frac{1}{20}(-10+3\sqrt{10}), \frac{1}{20}(-10+3\sqrt{10}))$ is a **local maximum** since $\det(\mathbf{H}) \approx 6068 > 0$ and $\frac{\partial^2 f}{\partial x^2} = -77.9 < 0$.

There are three critical points and only one of them is a local minimum. Hence, such local minimum $(x, y) = (1, 1)$, is the unique global minimum of $f$.

**(2b)** The code utilised to plot the absolute gradient is presented in the second part of the assignment. We can observe the gradient dipping to zero in three occasions: these correspond to the critical points derived in **2a**.

**(2c)** We have,

$$\mathbf{H} = \begin{pmatrix} 120x^2 + 40y^2 - 78 & 80xy \\ 80xy & 120y^2 + 40x^2 - 78 \end{pmatrix}.$$

Then we compute the characteristic polynomial,

$$\det(\mathbf{H} - \lambda \mathbf{I}) = 0,$$

which gives,

$$(120x^2 + 40y^2 - 78 - \lambda)(120y^2 + 40x^2 - 78 - \lambda) - (80xy)^2 = 0.$$

Letting,

$$\phi = 120x^2 + 40y^2 - 78,$$
$$\theta = 120y^2 + 40x^2 - 78,$$

the previous expression can be reformatted as:

$$(\phi - \lambda)(\theta - \lambda) - 6400x^2y^2 = 0,$$
$$\lambda^2 - \phi\lambda - \theta\lambda + \phi\theta - 6400x^2y^2 = \lambda^2 - \lambda(\phi + \theta) + (\phi\theta - 6400x^2y^2) = 0.$$

We can now utilise the quadratic formula,

$$\lambda = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

with

$$a = 1,$$
$$b = -\phi - \theta = 156 - 160x^2 - 160y^2,$$

$$c = \phi\theta - 6400x^2y^2 = (120x^2 + 40y^2 - 78)(120y^2 + 40x^2 - 78) - 6400x^2y^2 =$$
$$4800x^4 + 4800y^4 + 9600x^2y^2 - 12480x^2 - 12480y^2 + 6084 = 4800(x^2 + y^2)^2 - 12480(x^2 + y^2) + 6084.$$

We obtain the roots,

$$\lambda_1 = 40x^2 + 40y^2 - 78,$$
$$\lambda_2 = 120x^2 + 120y^2 - 78.$$

For positive definiteness we require $\lambda_1, \lambda_2 > 0$. In terms of $t = \frac{x^2+y^2}{2}$, we require:

$$78 < 40x^2 + 40y^2 \rightarrow \frac{39}{40} < t^2,$$

The condition triggered by the second eigenvalue ($\frac{39}{120} < t^2$) is included by that above. For indefiniteness we require $\lambda_1 \leq 0$, $\lambda_2 \geq 0$

$$40x^2 + 40y^2 \leq 78 \rightarrow t^2 \leq \frac{39}{40},$$

$$120x^2 + 120y^2 \geq 78 \rightarrow t^2 \geq \frac{13}{40} = \frac{13}{40},$$

These conditions result in

$$\frac{39}{120} \leq t^2 \leq \frac{39}{40}.$$

For negative definiteness we require $\lambda_1 < 0, \lambda_2 < 0$,

$$120x^2 + 120y^2 \leq 78 \rightarrow t^2 < \frac{13}{40},$$

where the condition triggered by the first eigenvalue is included by that above ($t^2 \leq \frac{39}{40}$).

**(2d)** For the starting point $\mathbf{x}_0 = [-0.5, -0.5], tol = 10^{-6}$, and using the specified parameters in Exercise **1** ($c_1 = 10^{-4}, c_2 = 0.9$, delta radius $= 0.2$) steepest descent and both trust region methods do not converge to the global minimum. This is pictured in the second part (Publish Output). They converge to local minimum [-0.974, -0.974], corresponding to a saddle point as seen in **2a**. Both dogleg and 2d-subspace methods follow the direction of steepest descent for this particular problem. Altering the delta radius to a higher value e.g. 0.8, triggers the global convergence of dogleg (to global minimum [1,1]). It results more complicated for 2d-subspace to converge globally for this problem - this appears to be a recurring issue in **2e**. We know all methods should provide global convergence given appropriate settings (all compute the Cauchy point). Using a delta radius of 0.2, dogleg, steepest descent and 2d-subspace converge in 10,8,6 steps respectively.

**(2e)** For [0.5, 0.5]: Dogleg, steepest descent and 2d-subspace all converge globally with dogleg taking a greater number of iterations. All procedures pursue the direction of steepest descent until reaching the global minimum. For [0.75, 0.75]: dogleg, steepest descent and 2d-subspace all converge globally with dogleg taking a greater number of iterations. All procedures pursue the direction of steepest descent until reaching the global minimum. For starting point [-0.75, -0.75]: dogleg, steepest descent and 2d-subspace all converge locally to the saddle point [-0.974, -0.974]. This is perhaps due to saddle point degeneracy. Dogleg takes three more iterations to converge than the other two methods. It is entertaining to experiment with starting point [-2.0, -2.0]; varying the step size $\alpha_0$ produces different convergences. Steepest descent with $\alpha_0 = 1$ passes over the local minimum (saddle point) in the first step, only to return to it and converge (again saddle point degeneracy). Regulating $\alpha_0$ to lower values makes the trajectory bounce back to the global minimum after the first step. [2.0, 2.0]; the behaviour is similar as for [-2.0, -2.0] but in the reverse direction. With greater values of $\alpha_0$ it is more likely to converge to the local minimum (saddle point).

Next page...

# COMPGV19: Coursework 1 - MATLAB Publish Output

**Note**: The main code and output for exercises **1d-f**, exercises **2c,d**; some of **2e** are presented in the next section - with helper functions at the end. Extended commentary is on Section 1.

## Table of Contents

Exercise 1d ..... 1
Exercise 1e ..... 4
Exercise 1f ..... 7
Exercise 2b ..... 10
Exercise 2di ..... 10
Exercise 2dii ..... 13
Exercise 2diii ..... 15
Exercise 2ei ..... 17
Exercise 2eii ..... 28
Exercise 2eiii ..... 33
Appendix ..... 38
BFGS.m ..... 38
descentLineSearch.m ..... 39
dogleg.m (dogleg method) ..... 41
FR.m (Fletcher-Reeves) ..... 42
FRPR.m Fletcher-Reeves/Polak-Ribiere (Hybrid) ..... 43
linesearch.m ..... 44
SolverCM2dSubspace.m ..... 45
trustRegion.m ..... 50
visualizeConvergence.m ..... 52
zoom.m ..... 53

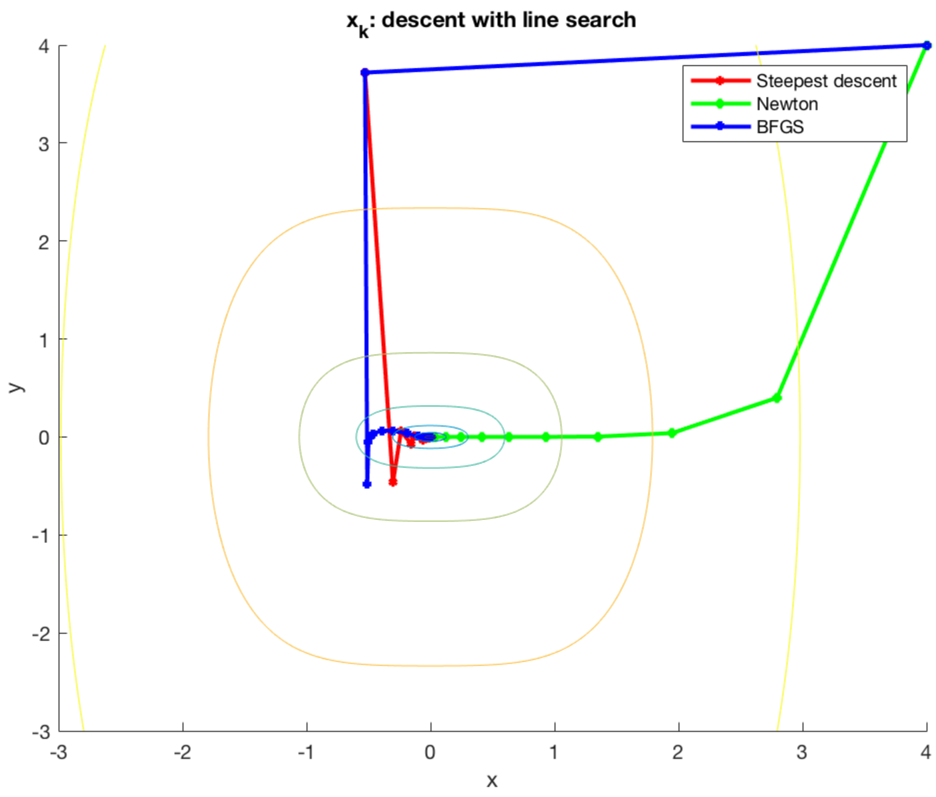
## Exercise 1d

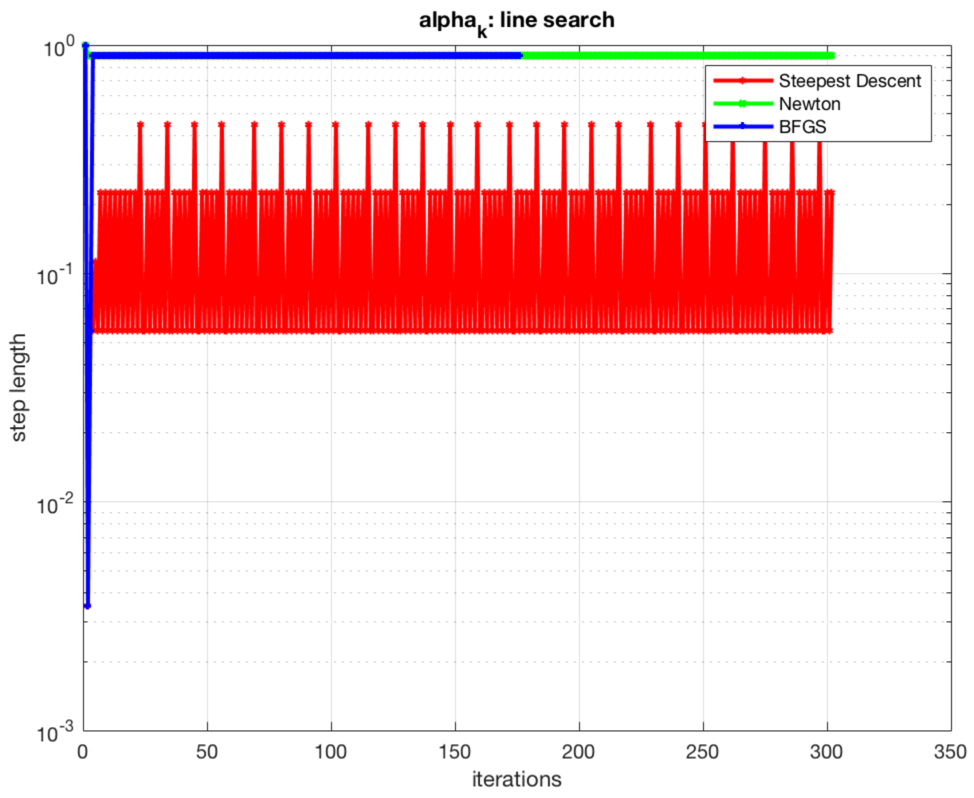
Main code for 1d. It features the functions lineSearch.m, zoom.m, visualizeConvergence.m, BFGS.m, descentLineSearch.m Refer to Part I of the hand-in for extended commentary (helper functions are in the appendix).

```
clear all, close all;

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

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) x.^2 + 5*x.^4 + 10*y.^2;
F2.dfx = @(x,y) 2*x + 20*x.^3;
F2.dfy = @(x,y) 20*y;
F2.d2fxx = @(x,y) 2+60*x^2;
F2.d2fxy = @(x,y) 0;
F2.d2fyx = @(x,y) 0;
F2.d2fyy = @(x,y) 20;

% Initialisation
alpha0 = 1;
```

```
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4], initial tolerance
x0 = [4;4];
tolerance = 1e-6;

% Steepest descent backtracking line search
lsOptsSteep.c1 = 1e-4;
lsOptsSteep.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F,'steepest',lsFun, ...

 alpha0, ...
                                                            x0,
 tolerance, maxIter);
% Newton backtracking line search
lsOptsNewton.c1 = 1e-4;
lsOptsNewton.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsNewton);
[xNewton, fNewton, nIterNewton, infoNewton] =
 descentLineSearch(F, 'newton', ...
                                                 lsFun, alpha0, x0,
 tolerance, maxIter);
%BFGS implementation w/ Wolfe
lsOptsBFGSWolfe.c1 = 1e-4;
lsOptsBFGSWolfe.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsBFGSWolfe);
[xBFGSWolfe, fBFGSWolfe, nIterBFGSWolfe, infoBFGSWolfe] = BFGS(F, ...
                                                lsFun, alpha0, x0,
 tolerance, maxIter);

%visualisation
%
n=300;
x=linspace(-3,4,n);
y=linspace(-3,4,n);
[X,Y]=meshgrid(x,y);

% Iterate plot one by one to see the order in which step are taken
figure;
hold on;
plot(infoSteep.xs(1,:), infoSteep.xs(2,:), '-or', 'LineWidth',
 2, 'MarkerSize', 3);
plot(infoNewton.xs(1,:), infoNewton.xs(2,:), '-*g', 'LineWidth',
 2, 'MarkerSize', 3);
plot(infoBFGSWolfe.xs(1,:), infoBFGSWolfe.xs(2,:), '-sb', 'LineWidth',
 2, ...

  'MarkerSize', 3)
```

```
contour(X,Y, log(max(F2.f(X,Y), 1e-3)));
title('x_k: descent with line search')
xlabel('x')
ylabel('y')
legend('Steepest descent', 'Newton', 'BFGS')


% Step length plot
figure
semilogy(infoSteep.alphas, '-or', 'LineWidth', 2, 'MarkerSize', 2);
hold on;
semilogy(infoNewton.alphas, '-*g', 'LineWidth', 2, 'MarkerSize', 2);
hold on;
semilogy(infoBFGSWolfe.alphas, '-sb', 'LineWidth', 2, 'MarkerSize',
 2);
grid on;
title('alpha_k: line search')
legend('Steepest Descent', 'Newton', 'BFGS')
xlabel('iterations')
ylabel('step length')
```



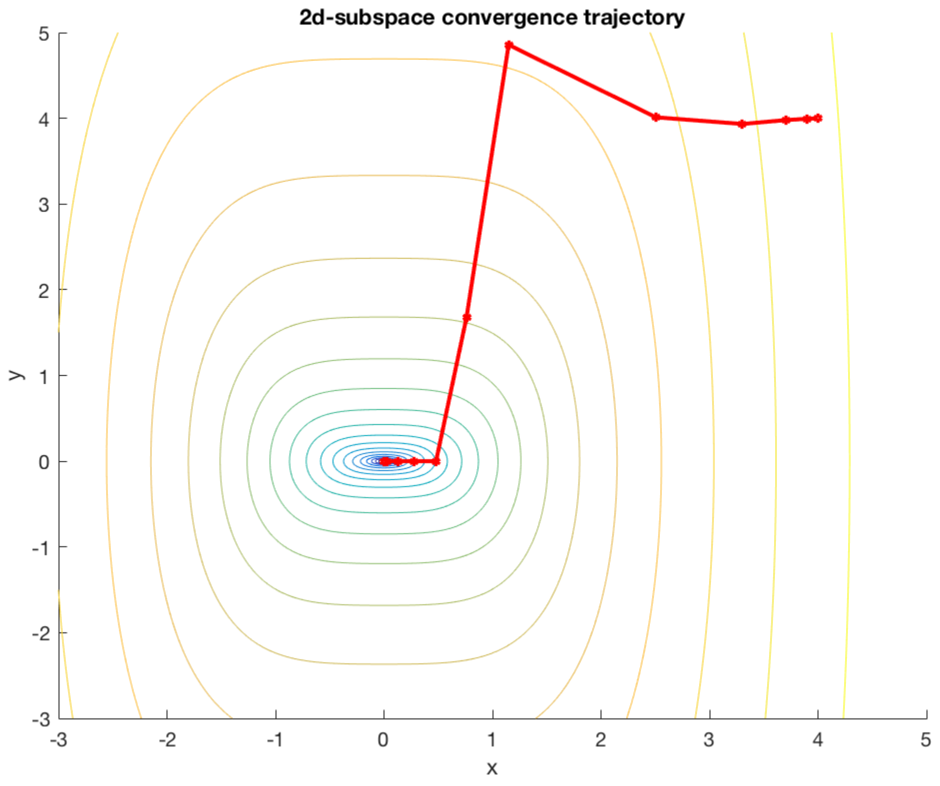


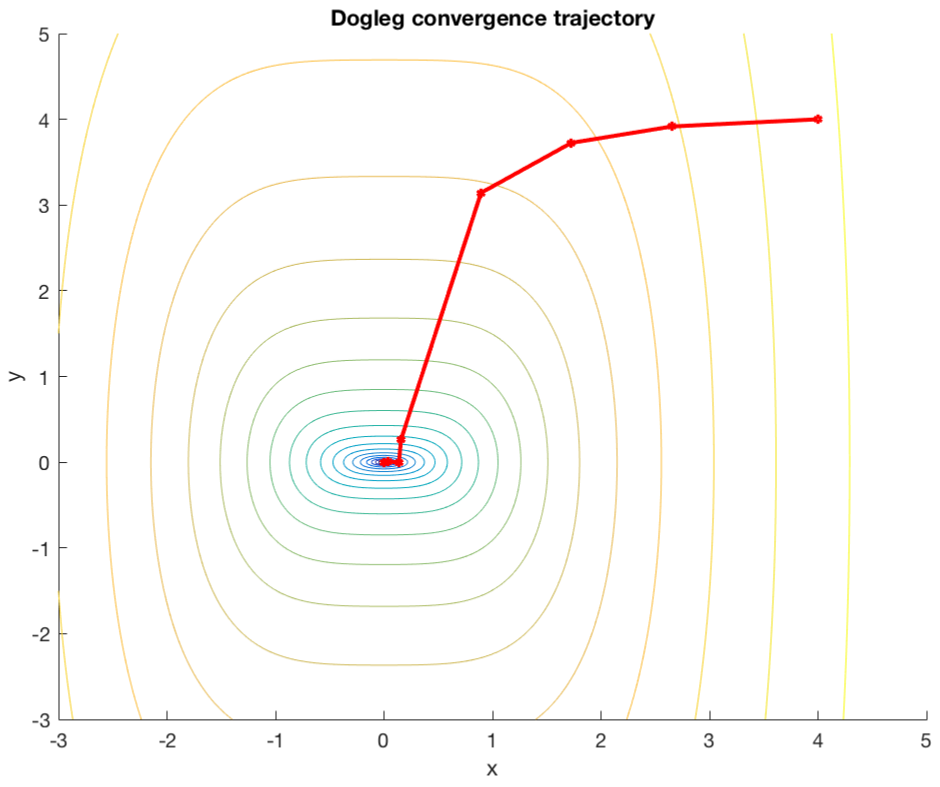
## Exercise 1e

Main code for 1e. It features the functions trustRegion.m, dogleg.m (dogleg implementation), solverCM2dSubspace.m (2d subspace method) Refer to Part I of the hand-in for extended commentary (helper functions are in the appendix).

```
clear all
close all
%
% For computation define as function of 1 vector variable
F.f = @(x) x(1)^2 + 5.*x(1)^4 + 10.*x(2)^2; % function handler, 2-dim
 vector
F.df = @(x) [2.*x(1) + 20.*x(1)^3; 20.*x(2)]; % gradient handler, 2-
dim vector
F.d2f = @(x) [2+60.*x(1)^2, 0; 0, 20]; % hessian handler, 2-dim vector
%
% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) x.^2 + 5*x.^4 + 10*y.^2;
F2.dfx = @(x,y) 2*x + 20*x.^3;
F2.dfy = @(x,y) 20*y;
F2.d2fxx = @(x,y) 2+60*x^2;
F2.d2fxy = @(x,y) 0;
F2.d2fyx = @(x,y) 0;
F2.d2fyy = @(x,y) 20;

% Initialisation
```

```
alpha0 = 1;
c1 = 1e-4;

% Point x0 = [4; 4]
x_0 = [4;4]; % starting point
tol = 1e-6; % tolerance


% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% 2d-subspace
[xTR1, fTR1, nIterTR1, infoTR1] = trustRegion(F, x_0,
 @solverCM2dSubspace, Delta, ...
                                                    eta, tol, maxIter,
 debug, F2);

% Dogleg
[xTR2, fTR2, nIterTR2, infoTR2] = trustRegion(F, x_0, @dogleg, Delta,
 eta, tol, ...,
                                                    maxIter, debug, F2);

% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-3,5,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(infoTR1,X,Y,Z,'final')
title('2d-subspace convergence trajectory')
xlabel('x')
ylabel('y')

visualizeConvergence(infoTR2,X,Y,Z,'final')
title('Dogleg convergence trajectory')
xlabel('x')
ylabel('y')
```

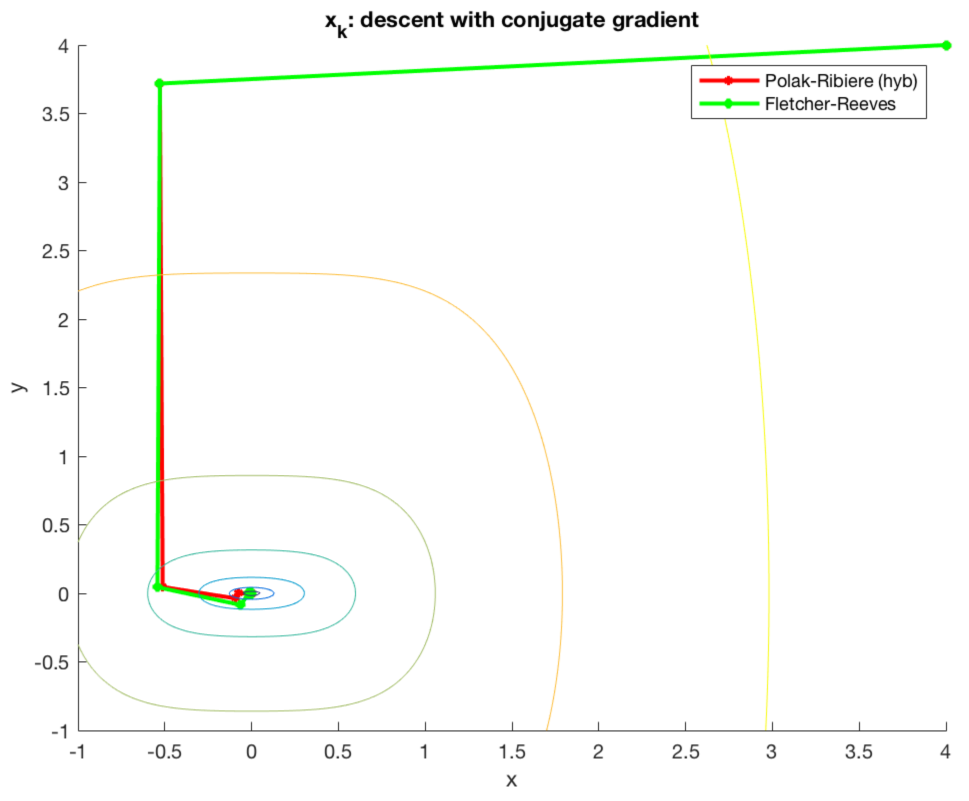

2d-subspace convergence trajectory
y
x

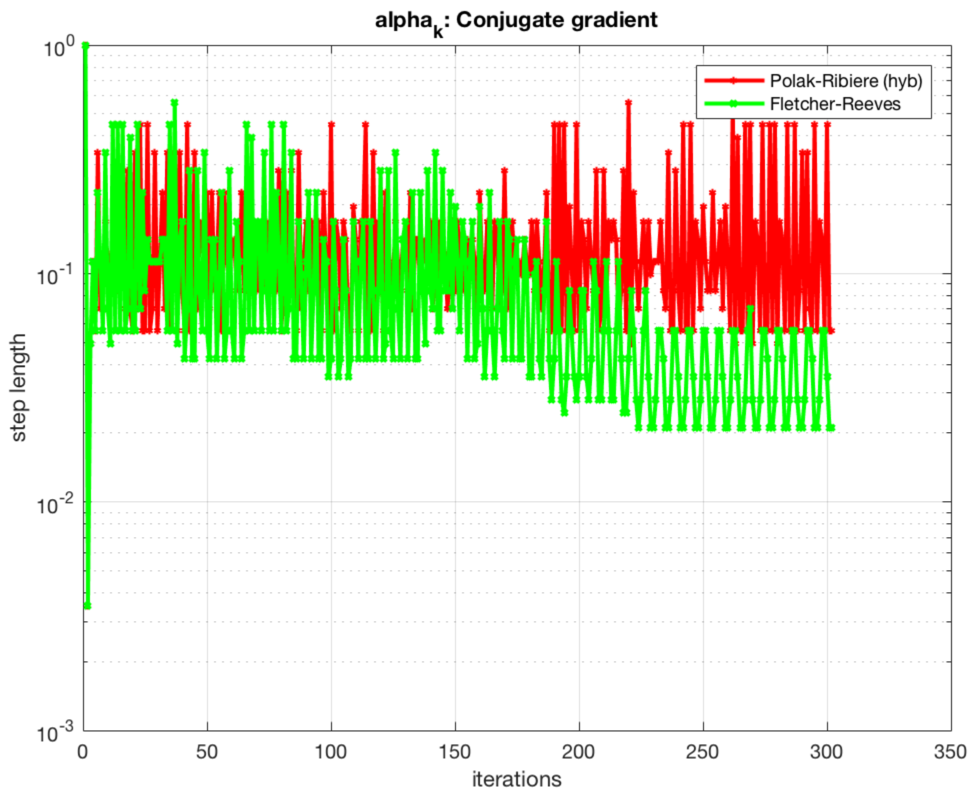

Dogleg convergence trajectory
y
x

# Exercise 1f

Main code for 1f. It features the functions FR.m (Fletcher-Reeves), FRPR.m (Hybrid Fletcher-Reeves-Polak-Ribiere), lineSearch.m, zoom.m, visualizeConvergence.m. Refer to Part I of the hand-in for extended commentary (helper functions are in the appendix).

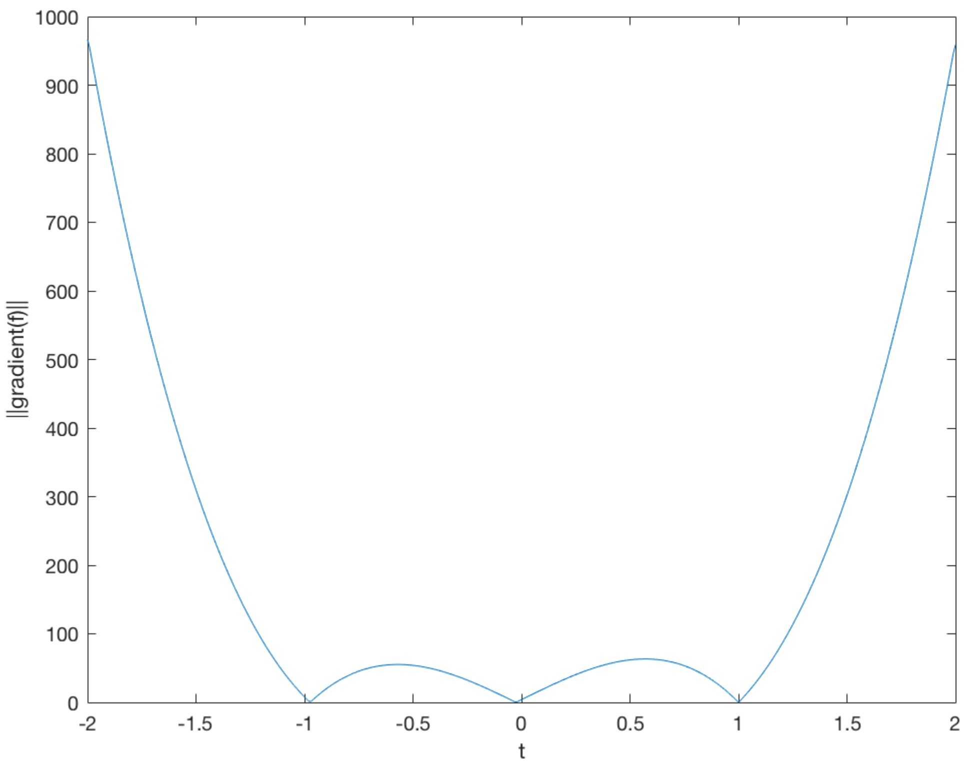
```
clear all, close all;

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

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) x.^2 + 5*x.^4 + 10*y.^2;
F2.dfx = @(x,y) 2*x + 20*x.^3;
F2.dfy = @(x,y) 20*y;
F2.d2fxx = @(x,y) 2+60*x^2;
F2.d2fxy = @(x,y) 0;
F2.d2fyx = @(x,y) 0;
F2.d2fyy = @(x,y) 20;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4]
x0 = [4;4];
tolerance = 1e-6;

% Conjugate gradient - Fletcher Reeves, Polak Ribiere.
lsOptsCG_Polak.c1 = c1;
lsOptsCG_Fletcher.c1 = 1e-4;
lsOptsCG_Polak.c2 = 0.12;
lsOptsCG_Fletcher.c2 = 0.12;
lsFunPolak = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsCG_Polak);
lsFunFletcher = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsCG_Fletcher);
% Optimisation
% Polak-Riviere (hybrid) algorithm
[xCG_Polak, fCG_Polak, nIterCG_Polak, infoCG_Polak] = FRPR(F,
 lsFunPolak, alpha0,...
                                                              x0,
 tolerance, maxIter);
% Fletcher Reeves algorithm
[xCG_Fletcher, fCG_Fletcher, nIterCG_Fletcher, infoCG_Fletcher] =
 FR(F,...

 lsFunFletcher, ...
 alpha0, x0, ...

 tolerance, maxIter);
% Visualisation
%%%%%%%%%%%%%%%

n=300;
x=linspace(-1,4,n);
y=linspace(-1,4,n);
[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
figure;
hold on;
plot(infoCG_Polak.xs(1,:), infoCG_Polak.xs(2,:), '-or', 'LineWidth',
 2, ...
                                                 'MarkerSize', 3);
plot(infoCG_Fletcher.xs(1,:), infoCG_Fletcher.xs(2,:), '-
*g', 'LineWidth', 2, ...
                                                     'MarkerSize', 3);
contour(X,Y, log(max(F2.f(X,Y), 1e-3)));
title('x_k: descent with conjugate gradient')
xlabel('x')
ylabel('y')
legend('Polak-Ribiere (hyb)', 'Fletcher-Reeves')

% Step length plot
figure
semilogy(infoCG_Polak.alphas, '-or', 'LineWidth', 2, 'MarkerSize', 2);
hold on;
semilogy(infoCG_Fletcher.alphas, '-*g', 'LineWidth', 2, 'MarkerSize',
 2);
grid on;
title('alpha_k: Conjugate gradient')
legend('Polak-Ribiere (hyb)', 'Fletcher-Reeves')
xlabel('iterations')
ylabel('step length')
```


$x_k$: descent with conjugate gradient
Polak-Ribiere (hyb)
Fletcher-Reeves
y
x
4
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4


$\text{alpha}_k$: Conjugate gradient
Polak-Ribiere (hyb)
Fletcher-Reeves
step length
iterations
$10^0$
$10^{-1}$
$10^{-2}$
$10^{-3}$
0
50
100
150
200
250
300
350

## Exercise 2b

```
% For visualisation proposes define as a function of 2 variables (x,y)
func = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;
x = linspace(-2,2,401);
y = linspace(-2,2,401);
[X,Y] = meshgrid(x,y); alpha0 = 1;

% plot
[gradt] = gradient(func(alpha0*X,alpha0*Y), alpha0*0.005);
figure
% make the plot
tnew = [abs(diag(gradt)), (-2:0.01:2)'];
plot(tnew(:,2), tnew(:,1))
xlabel('t')
ylabel('||gradient(f)||')
```



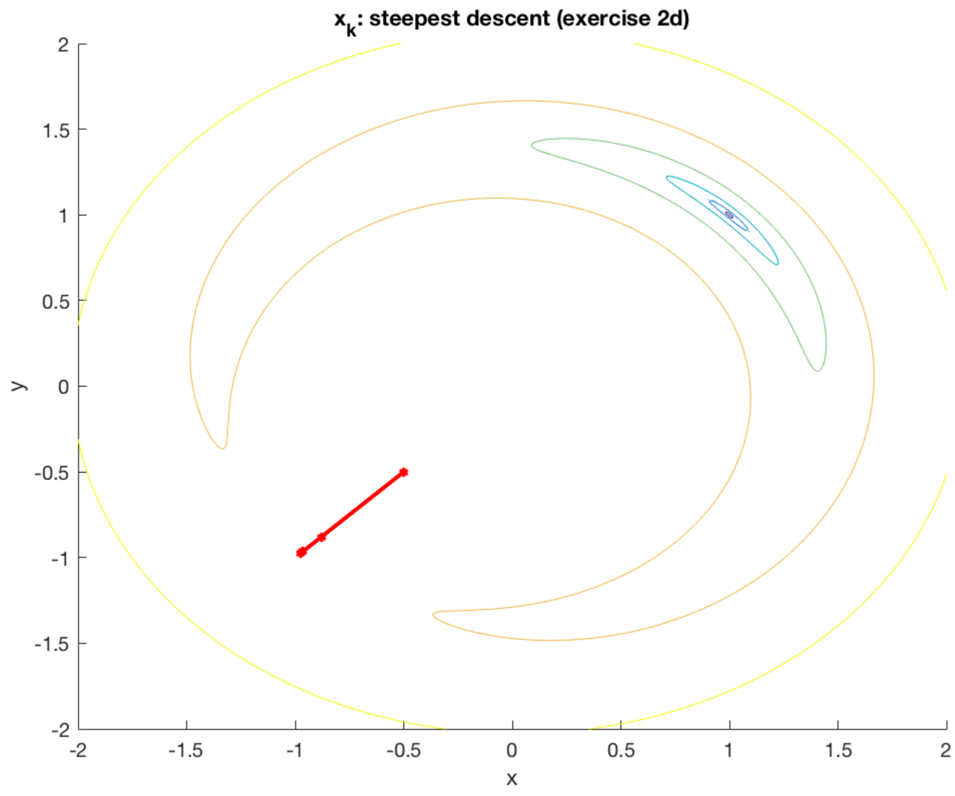
## Exercise 2di

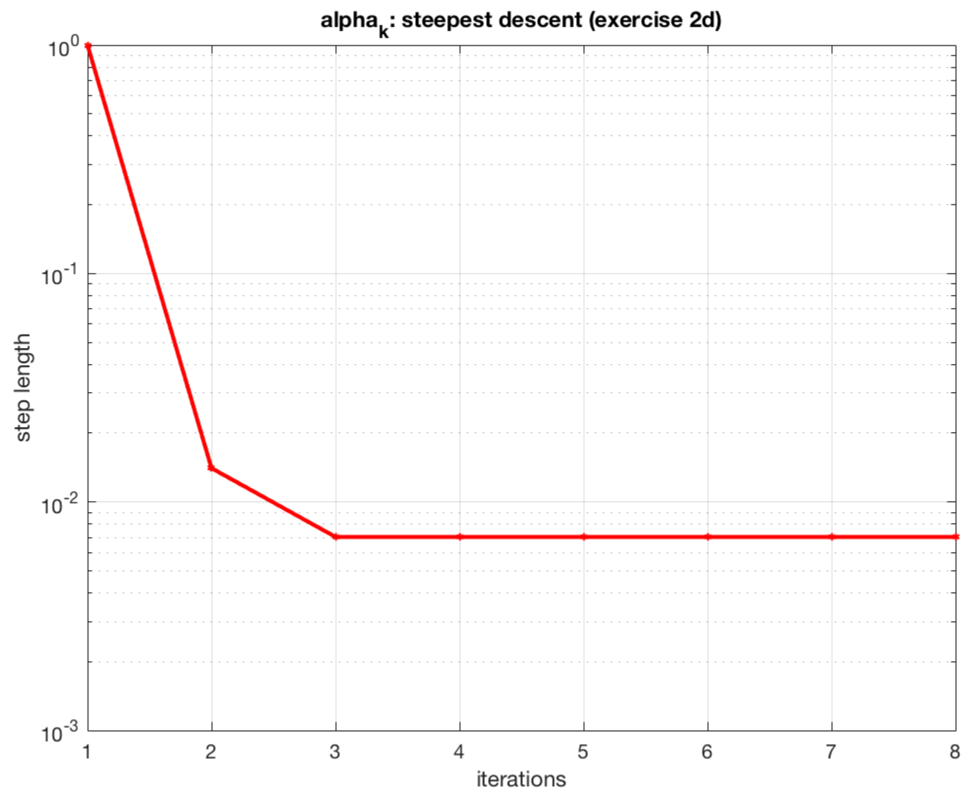
Main code for 2di. It features the functions lineSearch.m, zoom.m, visualizeConvergence.m, descent-LineSearch.m Refer to Part I of the hand-in for extended commentary (helper functions are in the appendix).

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

```
% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4], initial tolerance
x0 = [-0.5;-0.5];
tolerance = 1e-6;

% Steepest descent backtracking line search
lsOptsSteep.c1 = 1e-4;
lsOptsSteep.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F,'steepest',lsFun, ...

 alpha0, ...
                                                            x0,

 tolerance, maxIter);

% Visualisation
%
n=300;
x=linspace(-2,2,n);
y=linspace(-2,2,n);
[X,Y]=meshgrid(x,y);

% Iterate plot one by one to see the order in which step are taken
figure;
hold on;
plot(infoSteep.xs(1,:), infoSteep.xs(2,:), '-or', 'LineWidth',
 2, 'MarkerSize', 3);
contour(X,Y, log(max(F2.f(X,Y), 1e-3)));
title('x_k: steepest descent (exercise 2d)')
xlabel('x')
ylabel('y')

% Step length plot
figure
semilogy(infoSteep.alphas, '-or', 'LineWidth', 2, 'MarkerSize', 2);
```

```
hold on;
grid on;
title('alpha_k: steepest descent (exercise 2d)')
xlabel('iterations')
ylabel('step length')
```



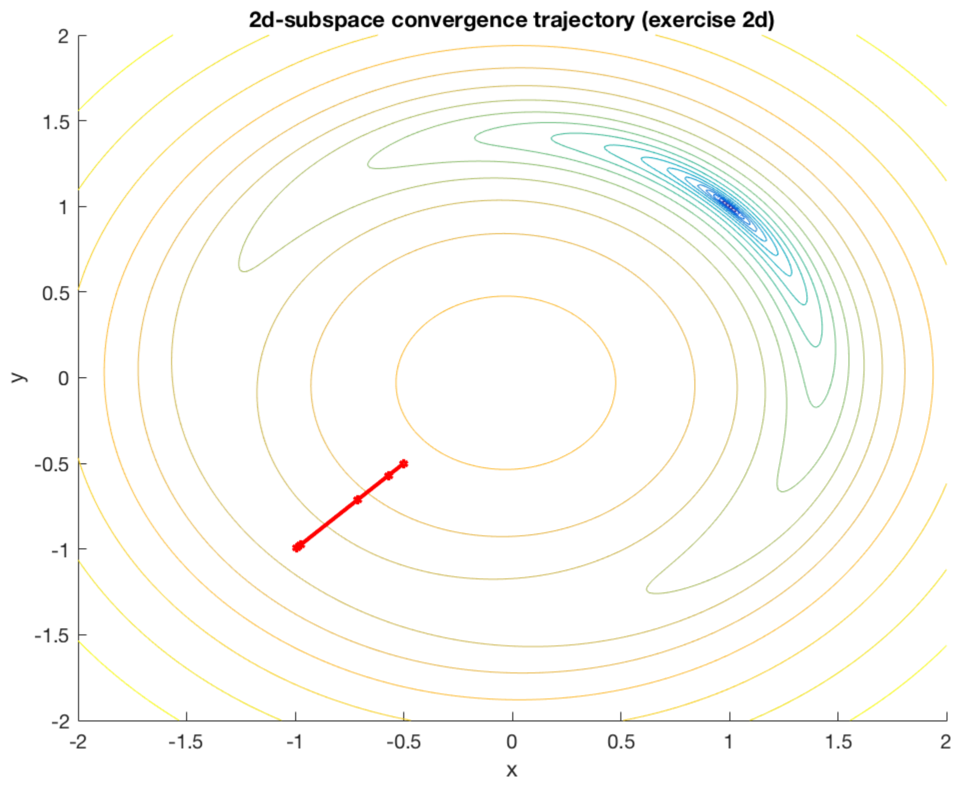


## Exercise 2dii

Main code for 2dii. It features the functions trustRegion.m, solverCM2dSubspace.m (2d subspace method) Refer to Part I of the hand-in for extended commentary (helper functions are in the appendix).

```
clear all, close all;

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4], initial tolerance
```

```
x0 = [-0.5;-0.5];
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% 2d-subspace
[xTR1, fTR1, nIterTR1, infoTR1] = trustRegion(F, x0,
 @solverCM2dSubspace, Delta, ...
                                              eta, tolerance, maxIter,
 debug, F2);

% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR1,X,Y,Z,'final')
title('2d-subspace convergence trajectory (exercise 2d)')
xlabel('x')
ylabel('y')
```

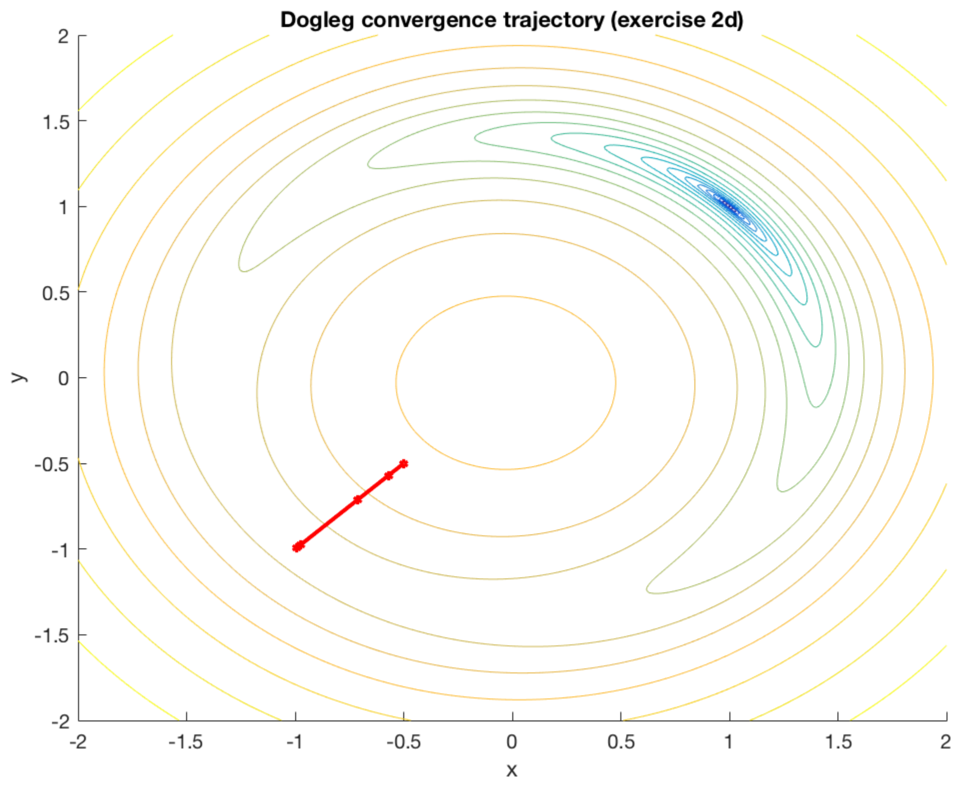


## Exercise 2diii

Main code for 2dii. It features the functions trustRegion.m, solverCM2dSubspace.m (2d subspace method) Refer to Part I of the hand-in for extended commentary (helper functions are in the appendix).

```
clear all, close all;

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;

% Point x0 = [4; 4], initial tolerance
x0 = [-0.5;-0.5];
```

```
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% Dogleg
[xTR2, fTR2, nIterTR2, infoTR2] = trustRegion(F, x0, @dogleg2, Delta,
 eta, ...
                                                    tolerance, maxIter,
 debug, F2);
% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR2,X,Y,Z,'final')
title('Dogleg convergence trajectory (exercise 2d)')
xlabel('x')
ylabel('y')
```



## Exercise 2ei

Main code for 2ei. It features the functions lineSearch.m, zoom.m, visualizeConvergence.m, descentLineSearch.m Refer to Part I of the hand-in for extended commentary (helper functions are in the appendix).

```
clear all, close all;

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;
```

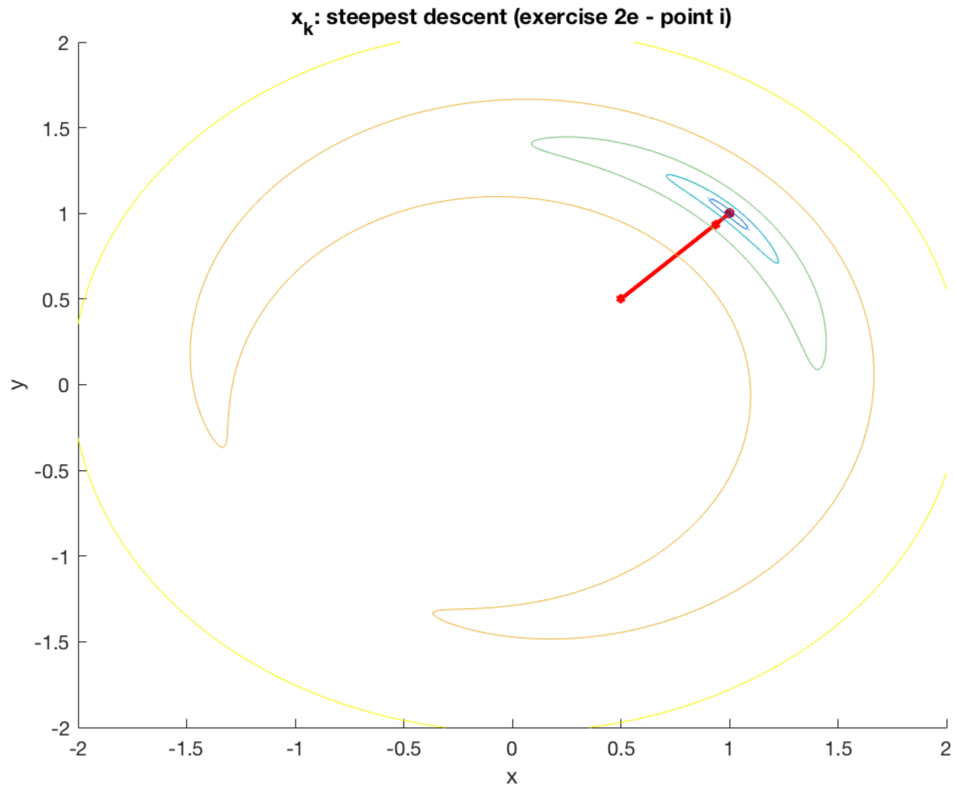
```
% Point x0 = [4; 4], initial tolerance
x0 = [0.5;0.5];
tolerance = 1e-6;

% Steepest descent backtracking line search
lsOptsSteep.c1 = 1e-4;
lsOptsSteep.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F,'steepest',lsFun, ...

 alpha0, ...
                                                            x0,

 tolerance, maxIter);

% Visualisation
%
n=300;
x=linspace(-2,2,n);
y=linspace(-2,2,n);
[X,Y]=meshgrid(x,y);

% Iterate plot one by one to see the order in which step are taken
figure;
hold on;
plot(infoSteep.xs(1,:), infoSteep.xs(2,:), '-or', 'LineWidth',
 2, 'MarkerSize', 3);
contour(X,Y, log(max(F2.f(X,Y), 1e-3)));
title('x_k: steepest descent (exercise 2e - point i)')
xlabel('x')
ylabel('y')

% Step length plot
figure
semilogy(infoSteep.alphas, '-or', 'LineWidth', 2, 'MarkerSize', 2);
hold on;
grid on;
title('alpha_k: steepest descent (exercise 2e - point i)')
xlabel('iterations')
ylabel('step length')

clear all

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];
```

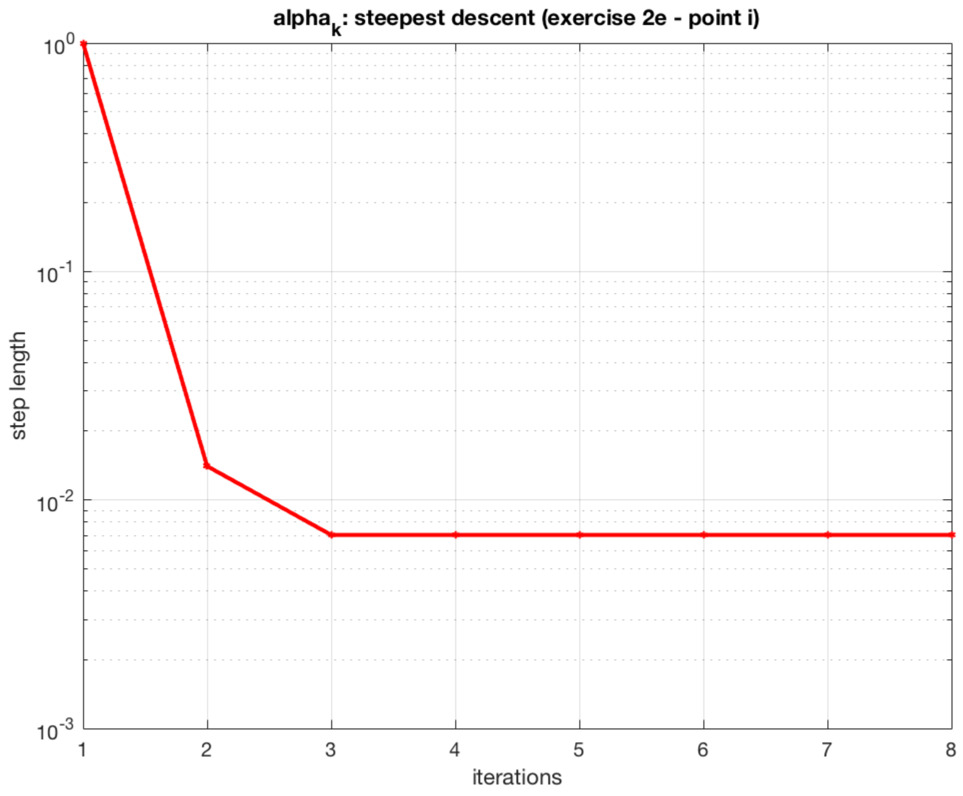
```
% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4], initial tolerance
x0 = [-0.75;-0.75];
tolerance = 1e-6;

% Steepest descent backtracking line search
lsOptsSteep.c1 = 1e-4;
lsOptsSteep.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F,'steepest',lsFun, ...

 alpha0, ...
                                                            x0,

 tolerance, maxIter);

% Visualisation
%
n=300;
x=linspace(-2,2,n);
y=linspace(-2,2,n);
[X,Y]=meshgrid(x,y);

% Iterate plot one by one to see the order in which step are taken
figure;
hold on;
plot(infoSteep.xs(1,:), infoSteep.xs(2,:), '-or', 'LineWidth',
 2, 'MarkerSize', 3);
contour(X,Y, log(max(F2.f(X,Y), 1e-3)));
title('x_k: steepest descent (exercise 2e - point ii )')
xlabel('x')
ylabel('y')

% Step length plot
figure
semilogy(infoSteep.alphas, '-or', 'LineWidth', 2, 'MarkerSize', 2);
hold on;
grid on;
title('alpha_k: steepest descent (exercise 2e- point ii)')
xlabel('iterations')
ylabel('step length')

clear all

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
```

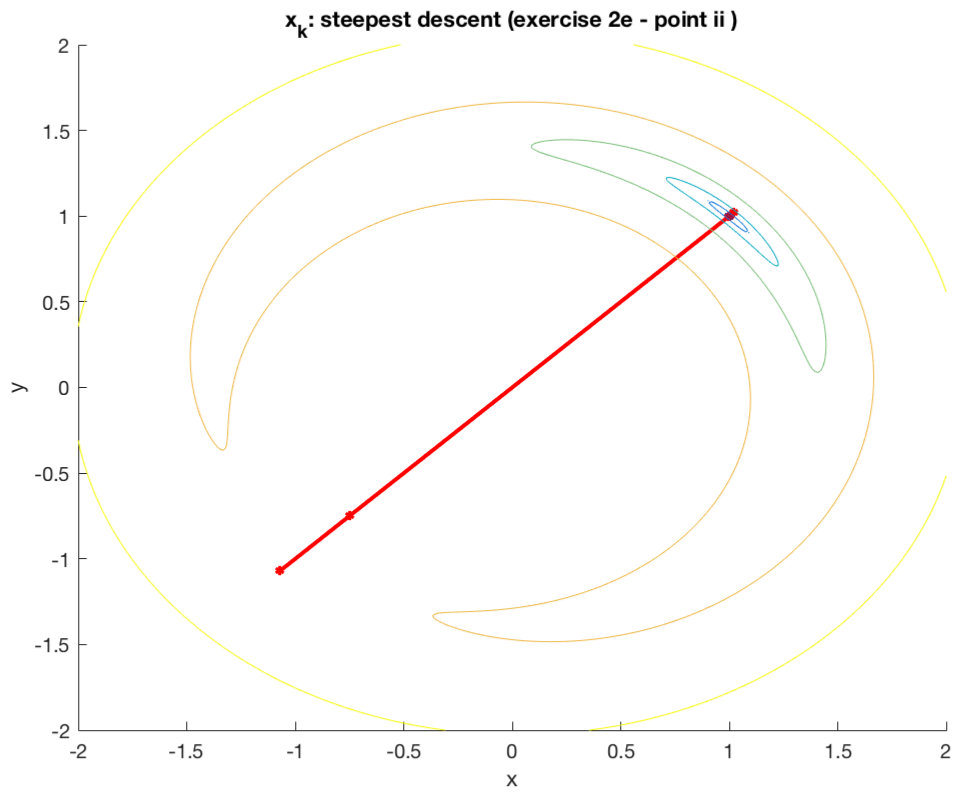
```
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4], initial tolerance
x0 = [0.75;0.75];
tolerance = 1e-6;

% Steepest descent backtracking line search
lsOptsSteep.c1 = 1e-4;
lsOptsSteep.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F,'steepest',lsFun, ...

 alpha0, ...
                                                             x0,

 tolerance, maxIter);

% Visualisation
%
n=300;
x=linspace(-2,2,n);
y=linspace(-2,2,n);
[X,Y]=meshgrid(x,y);

% Iterate plot one by one to see the order in which step are taken
figure;
hold on;
plot(infoSteep.xs(1,:), infoSteep.xs(2,:), '-or', 'LineWidth',
 2, 'MarkerSize', 3);
contour(X,Y, log(max(F2.f(X,Y), 1e-3)));
title('x_k: steepest descent (exercise 2e - point iii)')
xlabel('x')
ylabel('y')

% Step length plot
figure
semilogy(infoSteep.alphas, '-or', 'LineWidth', 2, 'MarkerSize', 2);
hold on;
grid on;
```

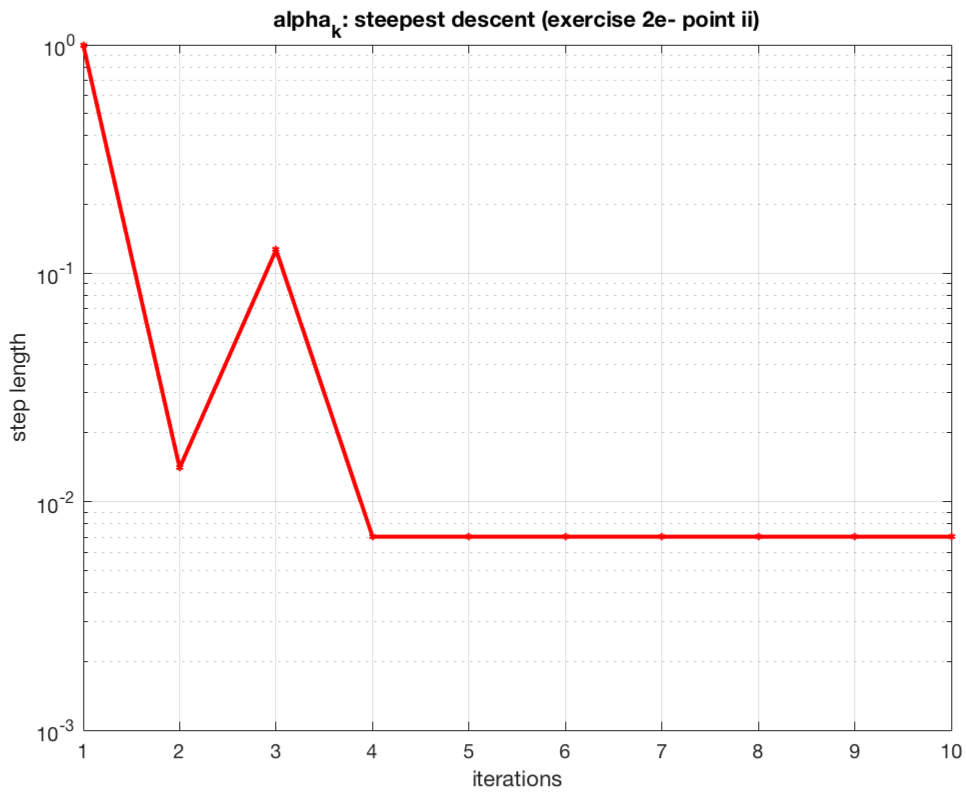
```
title('alpha_k: steepest descent (exercise 2e - point iii)')
xlabel('iterations')
ylabel('step length')

% Point x0 = [4; 4], initial tolerance
x0 = [-2; -2];
tolerance = 1e-6;

% Steepest descent backtracking line search
lsOptsSteep.c1 = 1e-4;
lsOptsSteep.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F,'steepest',lsFun, ...

 alpha0, ...
                                                            x0,
 tolerance, maxIter);

% Visualisation
%
n=300;
x=linspace(-2,2,n);
y=linspace(-2,2,n);
[X,Y]=meshgrid(x,y);

% Iterate plot one by one to see the order in which step are taken
figure;
hold on;
plot(infoSteep.xs(1,:), infoSteep.xs(2,:), '-or', 'LineWidth',
 2, 'MarkerSize', 3);
contour(X,Y, log(max(F2.f(X,Y), 1e-3)));
title('x_k: steepest descent (exercise 2e - point iv)')
xlabel('x')
ylabel('y')

% Step length plot
figure
semilogy(infoSteep.alphas, '-or', 'LineWidth', 2, 'MarkerSize', 2);
hold on;
grid on;
title('alpha_k: steepest descent (exercise 2e - point iv)')
xlabel('iterations')
ylabel('step length')

% Point x0 = [4; 4], initial tolerance
x0 = [2; 2];
tolerance = 1e-6;

% Steepest descent backtracking line search
lsOptsSteep.c1 = 1e-4;
lsOptsSteep.c2 = 0.9;
```

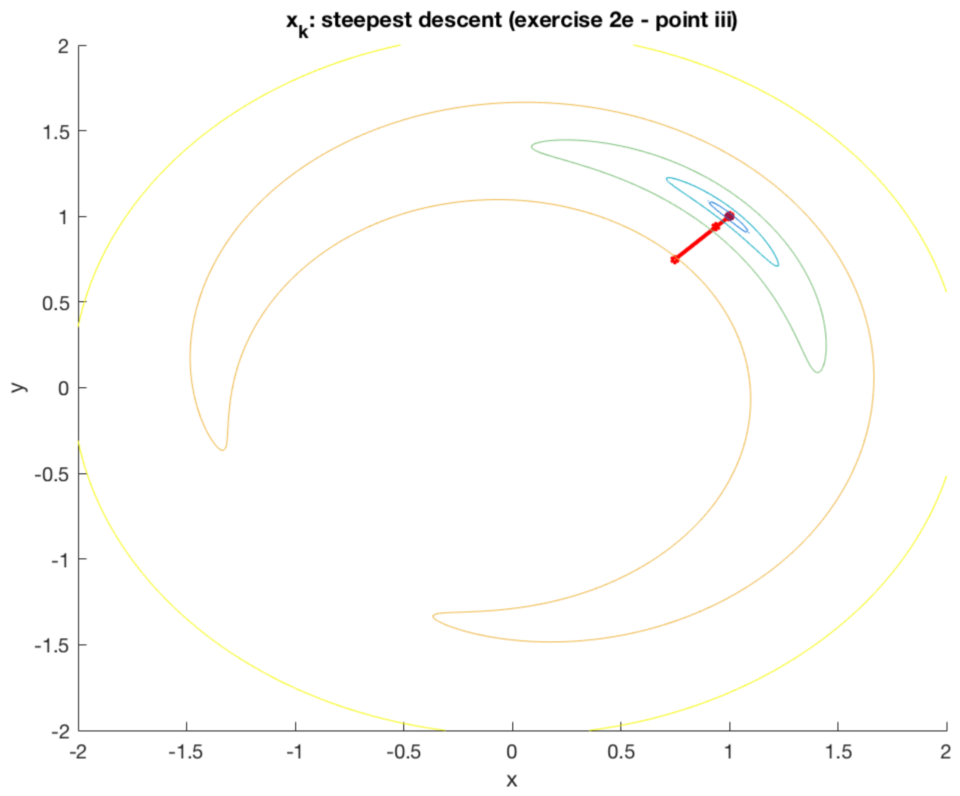
```
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F,'steepest',lsFun, ...

 alpha0, ...
                                                             x0,
 tolerance, maxIter);

% Visualisation
%
n=300;
x=linspace(-2,2,n);
y=linspace(-2,2,n);
[X,Y]=meshgrid(x,y);

% Iterate plot one by one to see the order in which step are taken
figure;
hold on;
plot(infoSteep.xs(1,:), infoSteep.xs(2,:), '-or', 'LineWidth',
 2, 'MarkerSize', 3);
contour(X,Y, log(max(F2.f(X,Y), 1e-3)));
title('x_k: steepest descent (exercise 2e - point v)')
xlabel('x')
ylabel('y')

% Step length plot
figure
semilogy(infoSteep.alphas, '-or', 'LineWidth', 2, 'MarkerSize', 2);
hold on;
grid on;
title('alpha_k: steepest descent (exercise 2e - point v)')
xlabel('iterations')
ylabel('step length')
```

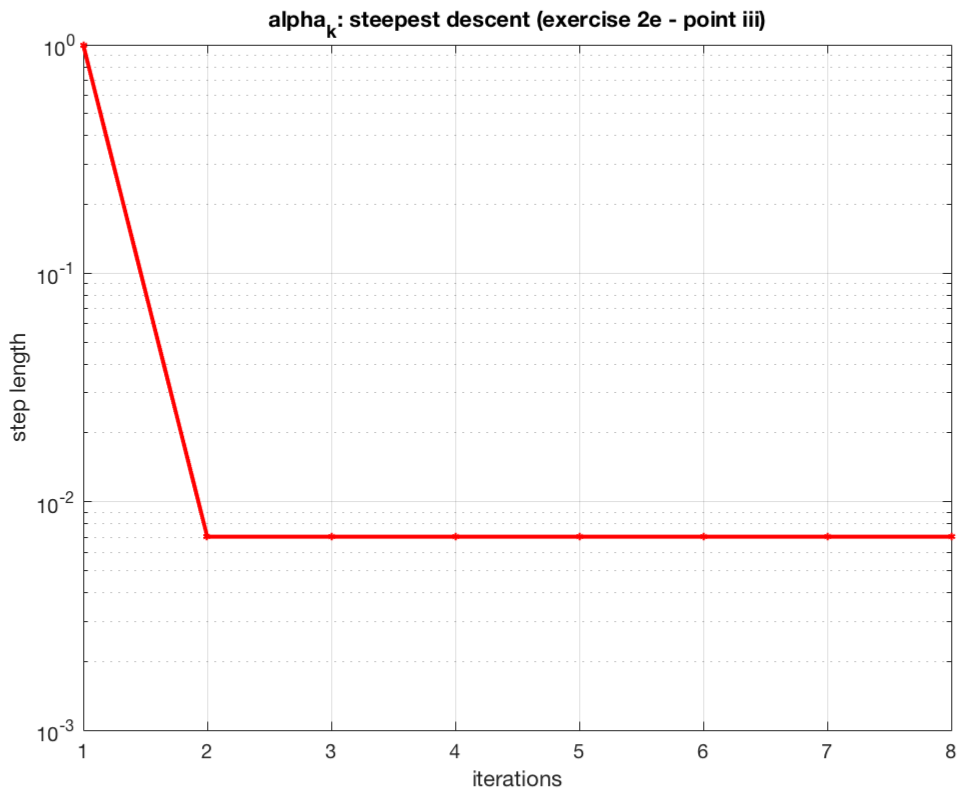

$x_k$: steepest descent (exercise 2e - point i)
x
y

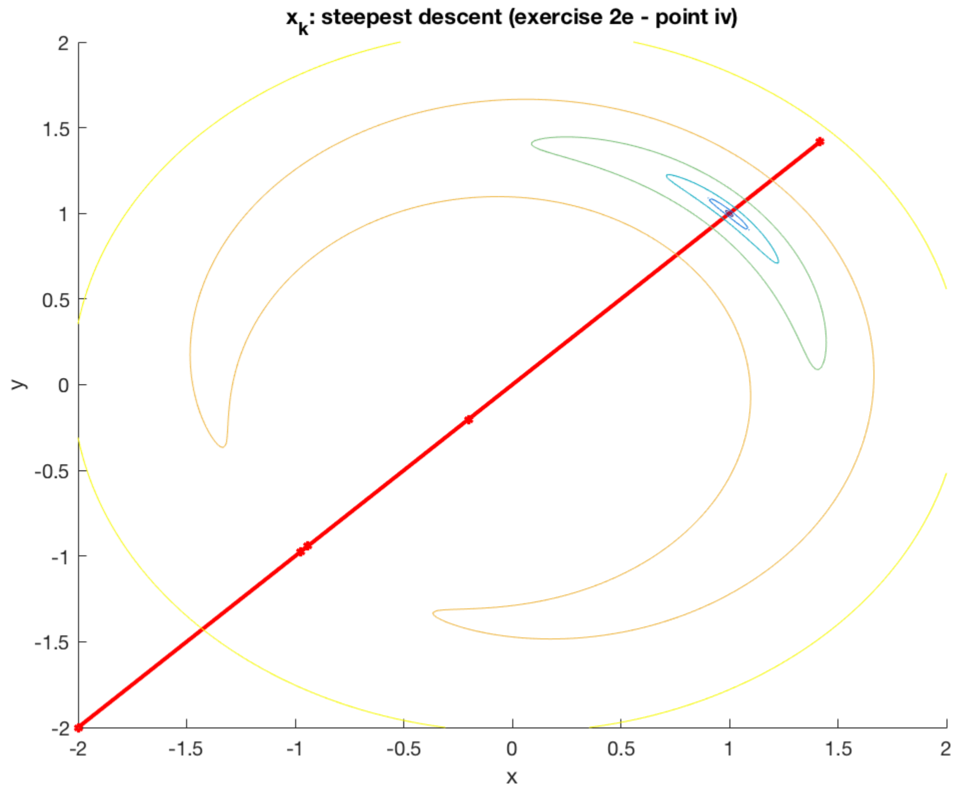

$\text{alpha}_k$: steepest descent (exercise 2e - point i)
step length
iterations


$x_k$: steepest descent (exercise 2e - point ii )
x
y
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

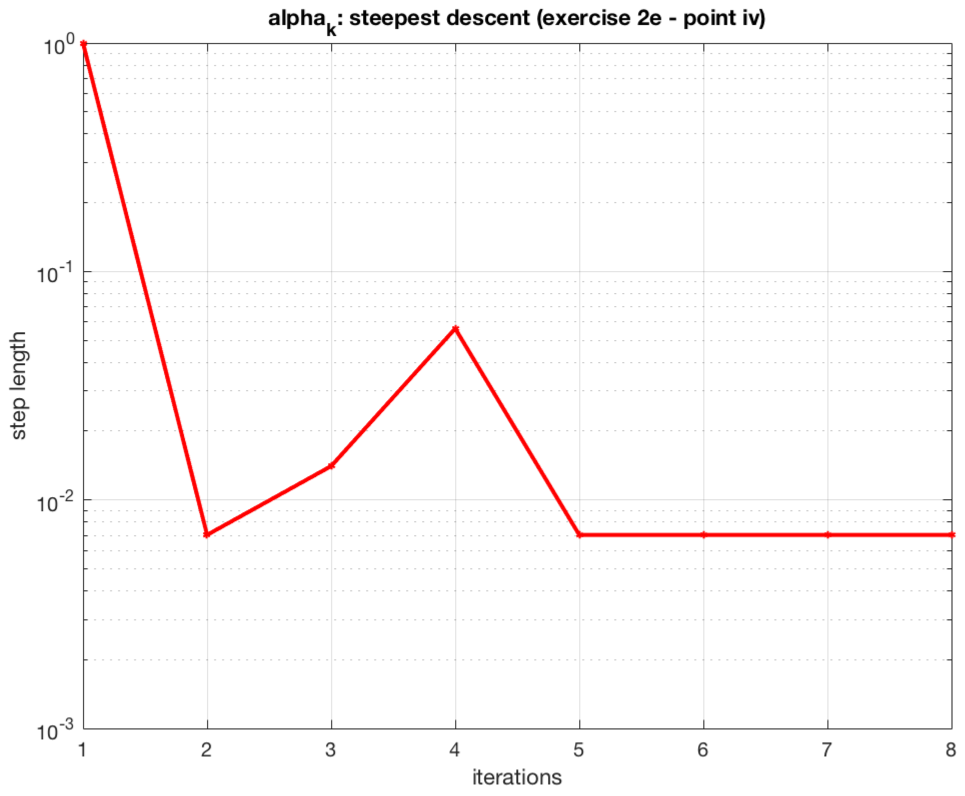

$alpha_k$: steepest descent (exercise 2e- point ii)
step length
iterations
$10^0$
$10^{-1}$
$10^{-2}$
$10^{-3}$
1
2
3
4
5
6
7
8
9
10


$x_k$: steepest descent (exercise 2e - point iii)
x
y

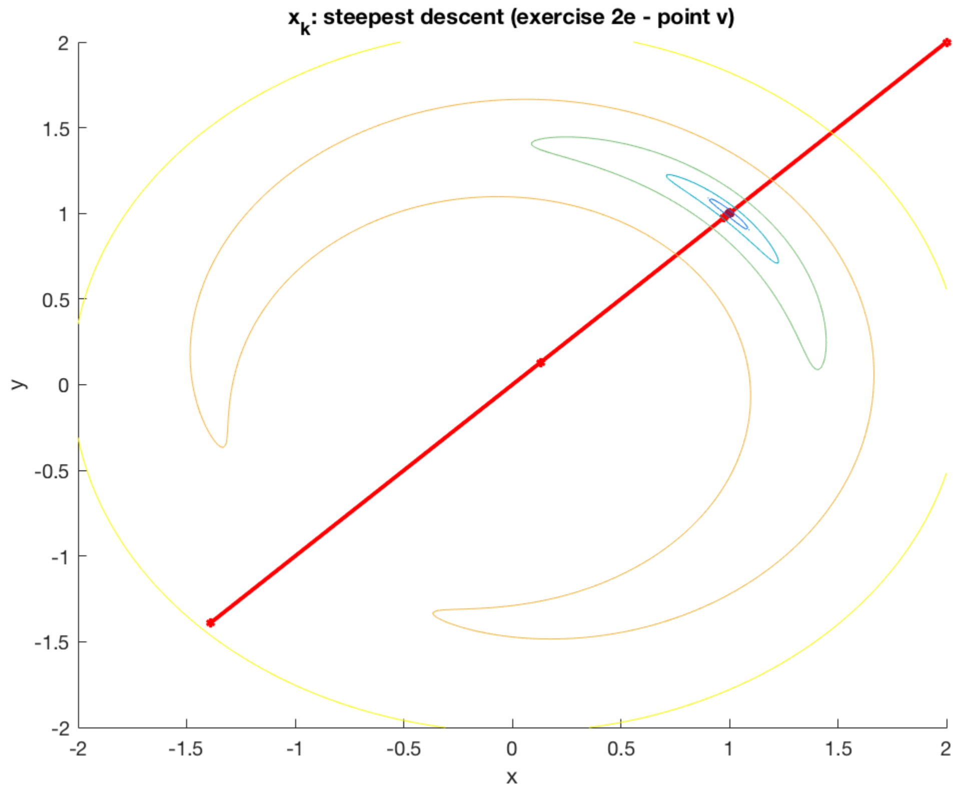

$alpha_k$: steepest descent (exercise 2e - point iii)
step length
iterations

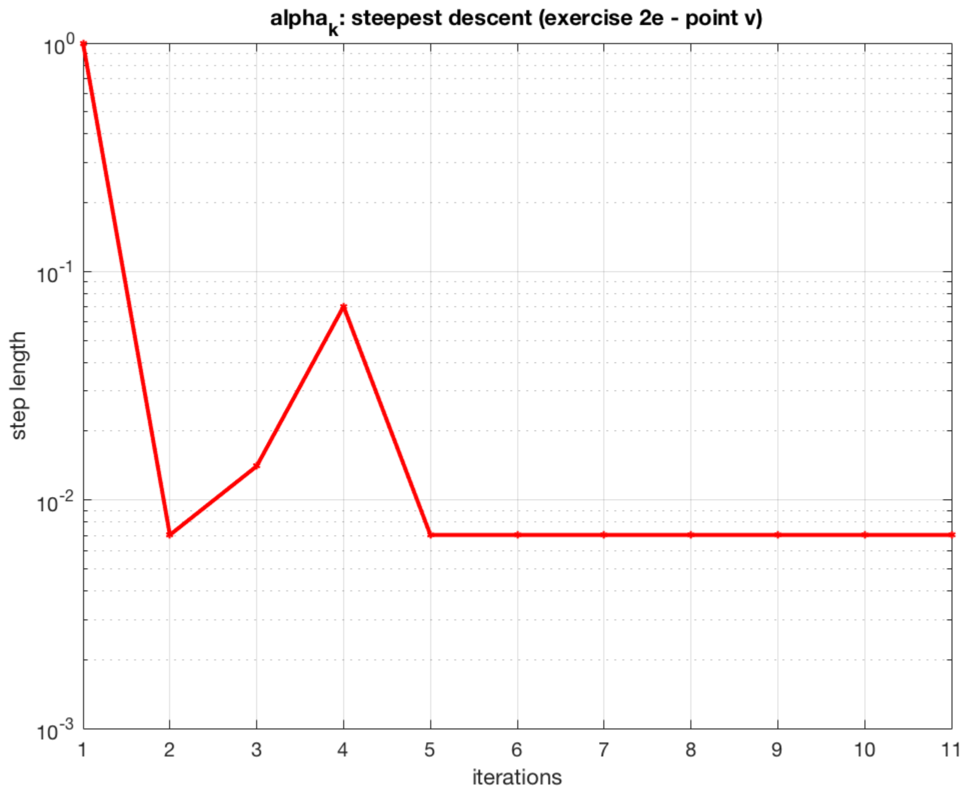


$x_k$: steepest descent (exercise 2e - point iv)
y
x



alpha$_k$: steepest descent (exercise 2e - point iv)
step length
iterations


x_k: steepest descent (exercise 2e - point v)
x
y
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2


alpha_k: steepest descent (exercise 2e - point v)
step length
iterations
10^0
10^-1
10^-2
10^-3
1
2
3
4
5
6
7
8
9
10
11

# Exercise 2eii

Main code for 2eii. It features the functions lineSearch.m, zoom.m, visualizeConvergence.m, descentLineSearch.m Refer to Part I of the hand-in for extended commentary (helper functions are in the appendix).

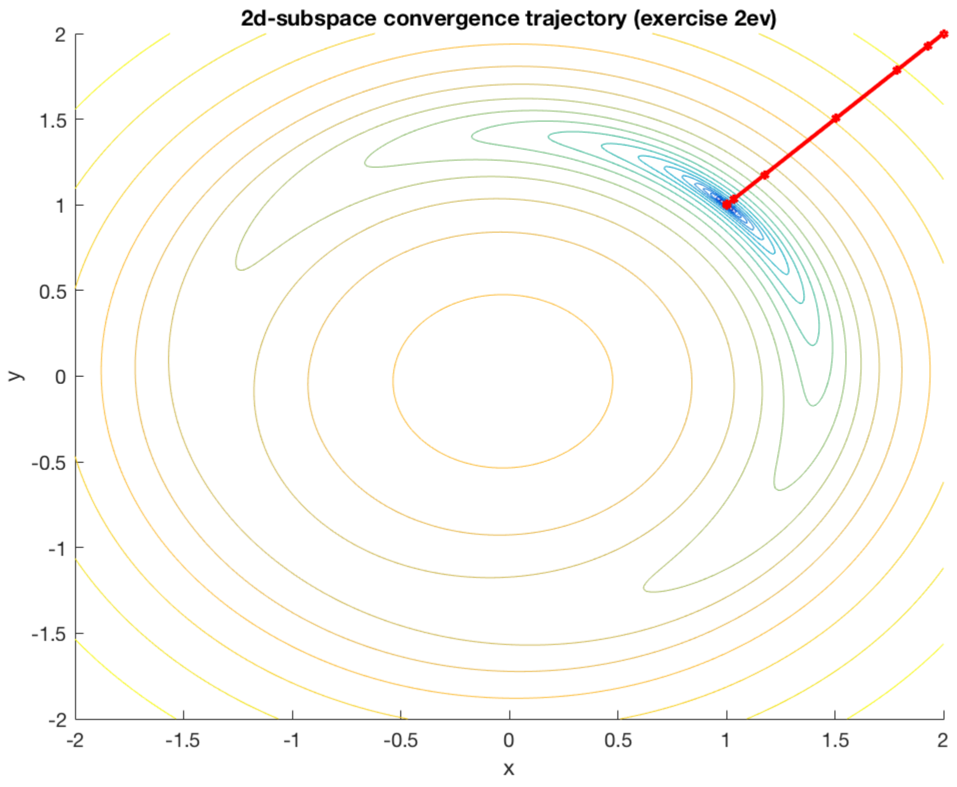
```
clear all, close all;

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4], initial tolerance
x0 = [0.5;0.5];
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% 2d-subspace
[xTR1, fTR1, nIterTR1, infoTR1] = trustRegion(F, x0,
 @solverCM2dSubspace, Delta, ...
                                                 eta, tolerance, maxIter,
 debug, F2);

% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR1,X,Y,Z,'final')
title('2d-subspace convergence trajectory (exercise 2ei)')
xlabel('x')
ylabel('y')

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4], initial tolerance
x0 = [-0.75;-0.75];
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% 2d-subspace
[xTR1, fTR1, nIterTR1, infoTR1] = trustRegion(F, x0,
 @solverCM2dSubspace, Delta, ...
                                                 eta, tolerance, maxIter,
 debug, F2);

% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR1,X,Y,Z,'final')
title('2d-subspace convergence trajectory (exercise 2eii)')
xlabel('x')
ylabel('y')

clear all

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4], initial tolerance
x0 = [0.75;0.75];
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% 2d-subspace
[xTR1, fTR1, nIterTR1, infoTR1] = trustRegion(F, x0,
 @solverCM2dSubspace, Delta, ...
                                              eta, tolerance, maxIter,
 debug, F2);

% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR1,X,Y,Z,'final')
title('2d-subspace convergence trajectory (exercise 2eiii)')
xlabel('x')
ylabel('y')

% % For computation define as function of 1 vector variable
% % function handler, 2-dim vector
% F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% % gradient handler, 2-dim vector
% F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
%              2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% % hessian handler, 2-dim vector
% F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
%               80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];
%
% % For visualisation proposes define as a function of 2 variables
 (x,y)
% F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;
%
% % Initialisation
% alpha0 = 1;
% c1 = 1e-4;
% maxIter = 300;
%
% % Point x0 = [4; 4], initial tolerance
% x0 = [-2,-2];
% tolerance = 1e-6;
%
% % Step acceptance relative progress threshold
% eta = 0.1;
% maxIter = 300;
% % Debugging paramter will switch on step by step visualisation of
 quadratic model
% % and various step options
% debug = 0;
% % Trust region radius
% Delta = 0.2;
%
% % 2d-subspace
% [xTR1, fTR1, nIterTR1, infoTR1] = trustRegion(F, x0,
 @solverCM2dSubspace, Delta, ...
%                                               eta, tolerance,
 maxIter, debug, F2);
%
% % Visualisation
% %%%%%%%%%%%%%%%
%
% % Define grid for visualisation
% n = 300;
% x = linspace(-2,2,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(infoTR1,X,Y,Z,'final')
% title('2d-subspace convergence trajectory (exercise 2eiv)')
% xlabel('x')
% ylabel('y')

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
maxIter = 300;

% Point x0 = [4; 4], initial tolerance
x0 = [2;2];
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% 2d-subspace
[xTR1, fTR1, nIterTR1, infoTR1] = trustRegion(F, x0,
 @solverCM2dSubspace, Delta, ...
                                              eta, tolerance, maxIter,
 debug, F2);

% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR1,X,Y,Z,'final')
title('2d-subspace convergence trajectory (exercise 2ev)')
xlabel('x')
ylabel('y')
```



## Exercise 2eiii

Main code for 2eiii. It features the functions lineSearch.m, zoom.m, visualizeConvergence.m, descent-LineSearch.m Refer to Part I of the hand-in for extended commentary (helper functions are in the appendix).

```
clear all, close all;

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
```

```
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;

% Point x0 = [4; 4], initial tolerance
x0 = [0.5;0.5];
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% Dogleg
[xTR2, fTR2, nIterTR2, infoTR2] = trustRegion(F, x0, @dogleg2, Delta,
 eta, ...
                                               tolerance, maxIter,
 debug, F2);
% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR2,X,Y,Z,'final')
title('Dogleg convergence trajectory (exercise 2ei)')
xlabel('x')
ylabel('y')

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
```

```
alpha0 = 1;
c1 = 1e-4;

% Point x0 = [4; 4], initial tolerance
x0 = [-0.75;-0.75];
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% Dogleg
[xTR2, fTR2, nIterTR2, infoTR2] = trustRegion(F, x0, @dogleg2, Delta,
 eta, ...
                                                 tolerance, maxIter,
 debug, F2);
% Visualisation
%%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR2,X,Y,Z,'final')
title('Dogleg convergence trajectory (exercise 2eii)')
xlabel('x')
ylabel('y')

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;
```

```
% Point x0 = [4; 4], initial tolerance
x0 = [0.75;0.75];
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% Dogleg
[xTR2, fTR2, nIterTR2, infoTR2] = trustRegion(F, x0, @dogleg2, Delta,
 eta, ...
                                             tolerance, maxIter,
 debug, F2);
% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR2,X,Y,Z,'final')
title('Dogleg convergence trajectory (exercise 2eiii)')
xlabel('x')
ylabel('y')

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;

% Point x0 = [4; 4], initial tolerance
x0 = [-2;-2];
tolerance = 1e-6;
```

```
% Step acceptance relative progress threshold
eta = 0.1;
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% Dogleg
[xTR2, fTR2, nIterTR2, infoTR2] = trustRegion(F, x0, @dogleg2, Delta,
 eta, ...
                                                 tolerance, maxIter,
 debug, F2);
% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR2,X,Y,Z,'final')
title('Dogleg convergence trajectory (exercise 2eiv)')
xlabel('x')
ylabel('y')

% For computation define as function of 1 vector variable
% function handler, 2-dim vector
F.f = @(x) (1-x(1)).^2 + (1-x(2)).^2 + 10*(x(1).^2+x(2).^2-2).^2;
% gradient handler, 2-dim vector
F.df = @(x) [2*(x(1)-1)+40*x(1).*(x(1).^2+x(2).^2-2);
             2*(x(2)-1)+40*x(2).*(x(1).^2+x(2).^2-2)];
% hessian handler, 2-dim vector
F.d2f = @(x) [120*x(1).^2+40*x(2).^2-78,80*x(1).*x(2);
              80*x(1).*x(2), 120*x(2).^2+40*x(1).^2-78];

% For visualisation proposes define as a function of 2 variables (x,y)
F2.f = @(x,y) (1-x).^2 + (1-y).^2 + 10*(x.^2 + y.^2 -2).^2;

% Initialisation
alpha0 = 1;
c1 = 1e-4;

% Point x0 = [4; 4], initial tolerance
x0 = [2;2];
tolerance = 1e-6;

% Step acceptance relative progress threshold
eta = 0.1;
```

```
maxIter = 300;
% Debugging paramter will switch on step by step visualisation of
 quadratic model
% and various step options
debug = 0;
% Trust region radius
Delta = 0.2;

% Dogleg
[xTR2, fTR2, nIterTR2, infoTR2] = trustRegion(F, x0, @dogleg2, Delta,
 eta, ...
                                             tolerance, maxIter,
 debug, F2);
% Visualisation
%%%%%%%%%%%%%%%

% Define grid for visualisation
n = 300;
x = linspace(-2,2,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(infoTR2,X,Y,Z,'final')
title('Dogleg convergence trajectory (exercise 2ev)')
xlabel('x')
ylabel('y')
```

# Appendix

## BFGS.m

```
function [xMin, fMin, nIter, info] = BFGS(F, ls, alpha0, x0, ...
                                         tol, maxIter)
% BFGS.m - implementation of descent with BFGS.
%
% it takes as INPUTS F, (a structure with fields containing a function
% handler f, a gradient handler df and a Hessian handler d2f), ls
 (handle to linear
% search function), alpha0 (initial step length), x0 (initial
 iterate),
% maxIter (max number of iterations),
% tol (stopping condition on relative err norm tolerance).
%
% it OUTPUTS xmin, fmin (minimum and value of f at the minimum), nIter
% (number of iterations), info (structure with information about of
 the
% iteration ).
% -xs: iterate history, alphas: step lengths history).
%
    % Initialisation
    nIter = 0;
    normError = 1;
    x_k = x0;
    info.xs = x0;
    info.alphas = alpha0;
    % specify initial Hessian
    Hess = eye(size(x0,1));
    % loop until convergence or max number of iterations
    while (normError >= tol && nIter <= maxIter)
        % Increment iterations
        nIter = nIter + 1;
        % specify direction (algorithm 8.1 Nocedal and Wright)
        p_k = -Hess*F.df(x_k);
        % call line search given by handle ls for step length
 computation
        alpha_k = ls(x_k, p_k, alpha0);
        % step updates  'displacement - s_k in Nocedal, Wright'
        displacement = alpha_k*p_k;
        x_k1 = x_k;
        x_k = x_k + displacement;
        y_k = F.df(x_k) - F.df(x_k1);
        % compute Hessian via equation 8.16
        Hess = ((eye(size(x0,1))-((displacement*y_k')/
(y_k'*displacement)))*Hess...
                *(eye(size(x0,1))-((y_k*displacement')/
(y_k'*displacement))))+...
                ((displacement*displacement')/(y_k'*displacement));
        % Compute relative error norm
        normError = norm(x_k - x_k1)/norm(x_k1);
        % store iteration info
        info.xs = [info.xs x_k];
        info.alphas = [info.alphas alpha_k];
    end
    % assign output values
    xMin = x_k;
    fMin = F.f(x_k);
end
```

## descentLineSearch.m

```
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
%   - f2f: Hessian handler
% descent: specifies descent direction {'steepese', 'newton'}
% 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  Marta M. Betcke, Kiko Rullan

% Initialization
nIter = 0;
normError = 1;
x_k = x0;
info.xs = x0;
info.alphas = alpha0;

% Loop until convergence or maximum number of iterations
while (normError >= tol && 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
    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;

    % Compute relative error norm
    normError = norm(x_k - x_k_1)/norm(x_k_1);

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

end
```

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

end
```

## dogleg.m (dogleg method)

```
function step = dogleg(F, xk, delt)
% dogleg.m - implementation of the dogleg method: used to solve a
 quadratic
% constraint trust region problem.
%
% it takes as INPUTS F, (a structure with fields containing a function
% handler f, a gradient handler df and a Hessian handler d2f) xk (the
 current
% iterate) and delt (the trust region radius).
%
% it OUTPUTS the 'step' (direction x length).
%
% Compute gradient and Hessian
    H = F.d2f(xk); % Hessian computation
    grad = F.df(xk); % gradient computation
    % cauchy point computation
    gTBg = grad'*(H*grad);
    if gTBg <= 0
        tau = 1;
    else
    % NOCEDAL and WRIGHT for computing tau (degenerate case)
        tau = min((norm(grad)^3)/(delt*gTBg), 1);
    end
    step = -(tau*delt)/(norm(grad)*grad);
    % follows NOCEDAL AND WRIGHT Ch. 4
    % steepest descent step for cauchy point - when delta small
    pb = -(H\grad);
    if delt >= norm(pb)
        step = pb;
    else
        bottom = grad'*H*grad;
        if bottom <= 0
            tau = 1;
        else
            tau = min(norm(grad)^3/(delt*bottom),1);
        end
        pu = -((grad'*grad)/bottom*grad);
        if tau <= 1 && tau >= 0
            step = tau*pu;
        elseif tau <= 2
            step = pu + (tau - 1) * (pb - pu);
        end
    end
end
```

## FR.m (Fletcher-Reeves)

```
function [xmin, fmin, nIter, info] = FR(F, ls, alpha0, x0, tol,
 maxIter)
% PR.m - implementation of conjugate gradient using the Fletcher
 Reeves algorithm.
%
% it takes as INPUTS F, (a structure with fields containing a function
% handler f, a gradient handler df and a Hessian handler d2f) ls
 (handle to linear
% search function), alpha0 (initial step length), x0 (initial
 iterate), tol
% (stopping condition on relative error norm tolerance).
%
% it OUTPUTS xmin, fmin (minimum and value of f at the minimum), nIter
% (number of iterations), info (structure with information about of
 the iteration ).
% -xs: iterate history, alphas: step lengths history).
%
    % Initialization
    nIter = 0;
    normError = 1;
    x_k = x0;
    df_k = F.df(x_k);
    p_k = -df_k;
    info.xs = x0;
    info.alphas = alpha0;
    % Loop until convergence or maximum number of iterations
    while (normError >= tol && 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
        kbeta = ((df_k)'*df_k)/(df_k_1'*df_k_1);
        p_k = -df_k + kbeta*p_k;
        % Store iteration info
        info.xs = [info.xs x_k];
        info.alphas = [info.alphas alpha_k];
        % Compute relative error norm
        normError = norm(x_k - x_k_1)/norm(x_k_1);
        % Increment iterations
        nIter = nIter + 1;
    end
    % Assign output values
    xmin = x_k;
    fmin = F.f(x_k);
end
```

## FRPR.m Fletcher-Reeves/Polak-Ribiere (Hybrid)

```
function [xmin, fmin, nIter, info] = FRPR(F, ls, alpha0, x0, tol,
 maxIter)
% FRPR.m - implementation of conjugate gradient using an interpolation
 of the
% Fletcher Reeves and the Polak Ribiere algorithms.
%
% it takes as INPUTS F, (a structure with fields containing a function
% handler f, a gradient handler df and a Hessian handler d2f) ls
 (handle to linear
% search function), alpha0 (initial step length), x0 (initial
 iterate), tol
% (stopping condition on relative error norm tolerance).
%
% it OUTPUTS xmin, fmin (minimum and value of f at the minimum), nIter
% (number of iterations), info (structure with information about of
 the iteration ).
% -xs: iterate history, alphas: step lengths history).
%
    % Initialization
    nIter = 0;
    normError = 1;
    x_k = x0;
    df_k = F.df(x_k);
    p_k = -df_k;
    info.xs = x0;
    info.alphas = alpha0;
    % Loop until convergence or maximum number of iterations
    while (normError >= tol && 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
        polakbeta = (df_k'*(df_k -df_k_1))/(df_k_1'*df_k_1);
        fletcherbeta = ((df_k)'*df_k)/(df_k_1'*df_k_1);
        if polakbeta > fletcherbeta
            kbeta = fletcherbeta;
        elseif polakbeta < -fletcherbeta
            kbeta = -fletcherbeta;
        else
            kbeta = polakbeta;
        end
        p_k = -df_k + kbeta*p_k;
        % Store iteration info
        info.xs = [info.xs x_k];
        info.alphas = [info.alphas alpha_k];
        % Compute relative error norm
        normError = norm(x_k - x_k_1)/norm(x_k_1);
        % Increment iterations
        nIter = nIter + 1;
    end
    % Assign output values
    xmin = x_k;
    fmin = F.f(x_k);
end
```

## linesearch.m

```
function [alpha_s, info] = lineSearch(F, x_k, p_k, alpha_max, opts)
% LINESEARCH Line Search algorithm satisfying strong Wolfe conditions
% alpha_s = lineSearch(F, x_k, p_k, alpha_max, opts)
%
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
% x_k: current iterate
% p_k: descent direction
% alpha_max: maximum step length
% opts: line search specific option structure with fields
%   - c1: constant in sufficient decrease condition
%         f(x_k + alpha_k*p_k) > f(x_k) + c1*alpha_k*(df_k'*p_k)
%         Typically chosen small, (default 1e-4)
%   - c2: constant in strong curvature condition
%         |df(x_k + alpha_k*p_k)'*p_k| <= c2*|df(x_k)'*p_k|
%
% OUTPUT
% alpha_s: step length
% info: structure containing alpha_j history
%
% Reference: Algorithm 3.5 from Nocedal, Numerical Optimization
%
% It generates a monotonically increasing sequence of step lenghts
 alpha_j.
% Uses the fact that interval (alpha_j_1, alpha_j) contains step
 lengths satisfying strong Wolfe conditions
% if one of the conditions below is satisfied:
% (C1) alpha_j violates the sufficient decrease condition
% (C2) phi(alpha_j) >= phi(alpha_j_1)
% (C3) dphi(alpha_j) >= 0
%
% Copyright (C) 2017 Kiko Rullan, Marta M. Betcke

% Paramters
% Multiple of alpha_j used to generate alpha_{j+1}
FACT = 10;

% Calculate handle to function phi(alpha) = f(x_k + alpha*p_k)
% Phi: function structure with fields
% - phi: function handler
% - dphi: derivative handler
Phi.phi = @(alpha) F.f(x_k + alpha*p_k);
Phi.dphi = @(alpha) (F.df(x_k + alpha*p_k)')*p_k;

% Initialization
alpha(1) = 0;
phi_i(1) = Phi.phi(0);
dphi_i(1) = Phi.dphi(0);
alpha(2) = 0.9*alpha_max; %0.5*alpha_max;
alpha_s = 0;
n = 2;
maxIter = 10;
stop = false;

while (n < maxIter && stop == false)
    phi_i(n) = Phi.phi(alpha(n));
    dphi_i(n) = Phi.dphi(alpha(n));
    if(phi_i(n) > phi_i(1) + opts.c1*alpha(n)*dphi_i(1) || (phi_i(n)
 >= phi_i(n-1) && n > 2))
        alpha_s = zoom(Phi, alpha(n-1), alpha(n), opts.c1, opts.c2);
        stop = true;
    elseif(abs(dphi_i(n)) <= -opts.c2*dphi_i(1))
        alpha_s = alpha(n);
        stop = true;
    elseif(dphi_i(n) >= 0)
        alpha_s = zoom(Phi, alpha(n), alpha(n-1), opts.c1, opts.c2);
        stop = true;
    end;
    %alpha(n+1) = 0.5*(alpha(n)+alpha_max);
    alpha(n+1) = max(FACT*alpha(n), alpha_max);
    n = n + 1;
end

info.alphas = alpha;
end
```

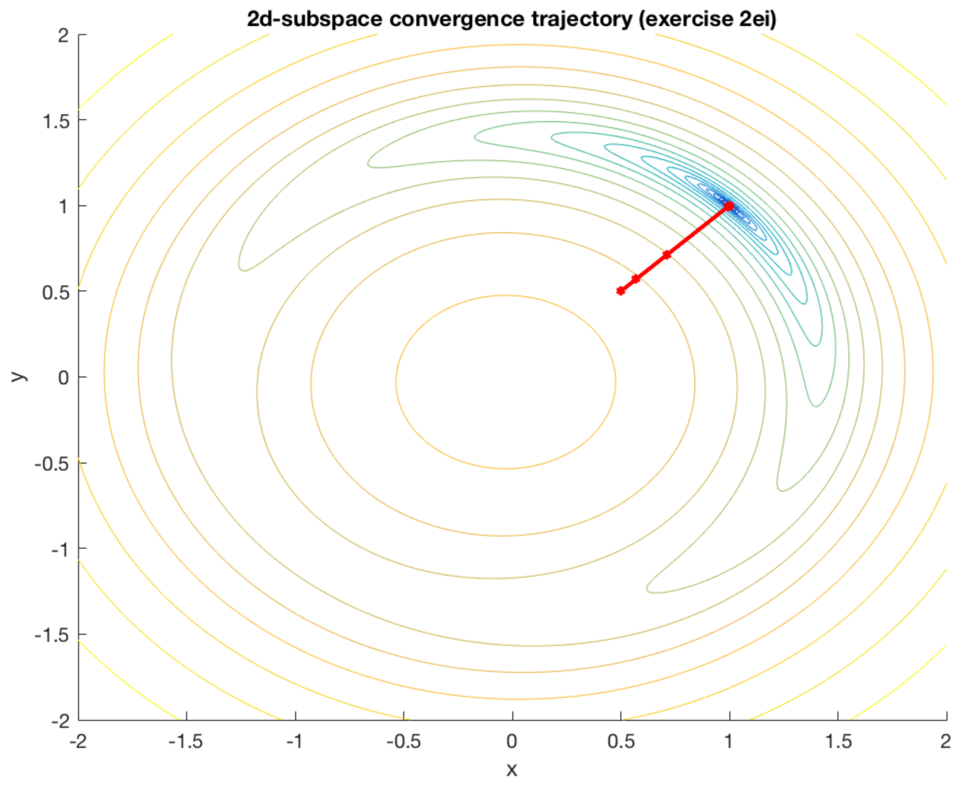
## SolverCM2dSubspace.m

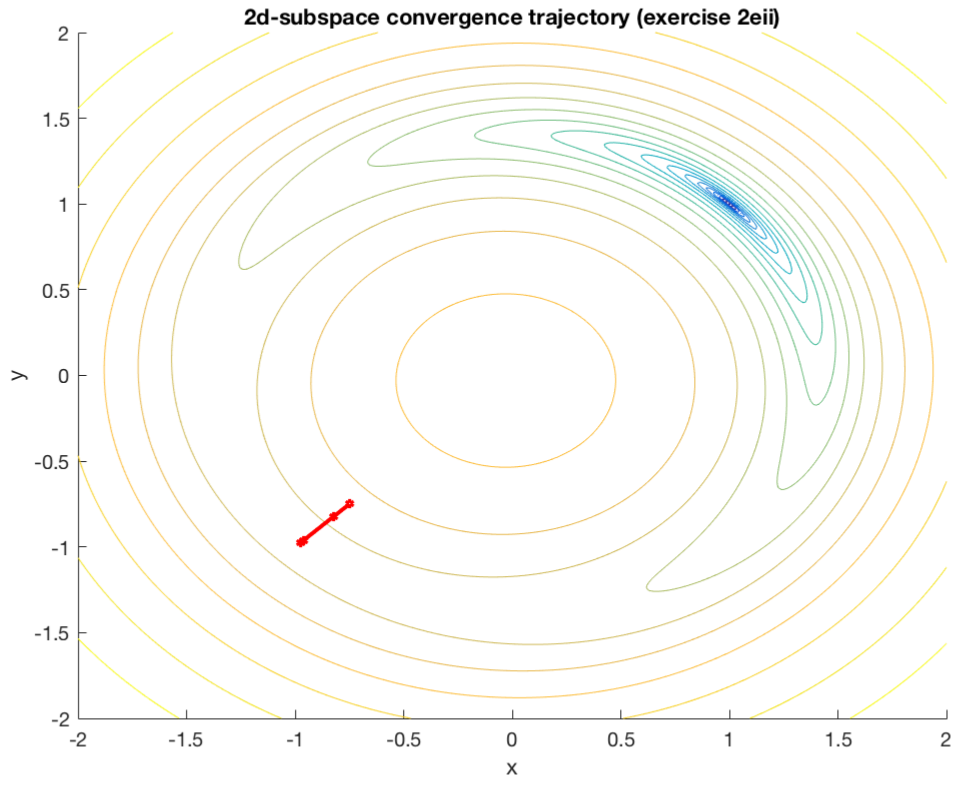
```
function p = solverCM2dSubspace(F, x_k, Delta)
% SOLVERCM2DSUBSPACE Solves quadratic constraint trust region problem
 via 2d subspace
% p = solverCM2dSubspace(F, x_k, Delta)
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
%   - d2f: Hessian handler
% x_k: current iterate
% Delta: trust region radius
% OUTPUT
% p: step (direction times lenght)
```

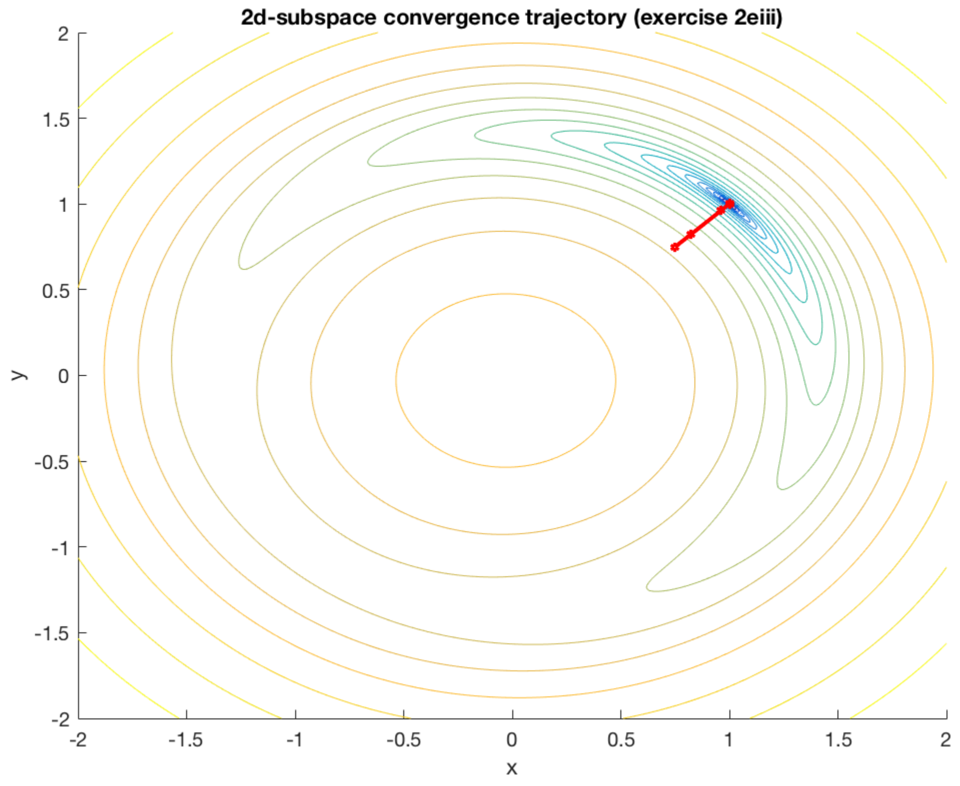
```
%
% Copyright (C) 2017 Marta M. Betcke, Kiko Rullan


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

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

% 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)
% 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

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

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

% 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;

end
```




2d-subspace convergence trajectory (exercise 2eii)
x
y


2d-subspace convergence trajectory (exercise 2eiii)
x
y

## trustRegion.m

```
function [x_k, f_k, k, info] = trustRegion(F, x0, solverCM, Delta,
 eta, tol, maxIter, debug, F2)
% TRUSTREGION Trust region iteration
% p = solverCM2dSubspace(F, x_k, Delta)
% 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 {'error', 'grad'}
stopType = 'error';

% 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)) ;

  % 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 'error'
            % relative error norm
            stopCond = (norm(x_k - x_k_1)/norm(x_k_1) < tol);
          case 'grad'
            % gradient norm
            stopCond = (norm(F.df(x_k)) < tol);
        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) = [];

    end
```

## visualizeConvergence.m

```
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(0:0.01:2*pi)*info.Deltas(info.xind(j)), ...
             info.xs(2,
j-1)+sin(0:0.01:2*pi)*info.Deltas(info.xind(j)), ...
             ':k', 'LineWidth', 2);
      end

      title(['Convergence: steps 1 : ' num2str(j)])
      pause(0.5);
    end

end

end
```

## zoom.m

```
function [alpha, info] = zoom(Phi, alpha_l, alpha_h, c1, c2)
% ZOOM Zoom algorithm for line search with strong Wolfe conditions
% alpha = zoom(Phi, alpha_l, alpha_h, c1, c2)
%
% INPUTS
% Phi: structure for function of step length phi(alpha) = f(x_k +
 alpha*p_k) with fields
%   - phi: function handler
%   - dphi: derivative handler
% alpha_l: lower boundary of the trial interval
% alpha_h: upper boundary of the trial interval
% c1 & c2: constants for Wolfe conditions (see lineSearch.m)
%
% OUTPUT
% alpha: step length
% info: structures containing iteration history
%
% Reference: Algorithm 3.5 from Nocedal, Numerical Optimization
%
```

```
% Properties ensured at each iteration
% (P1) Interval (alpha_l, alpha_h) contains step lengths satisfying
 strong Wolfe conditions.
% (P2) Among the step lengths generated so far satisfying the
 sufficient decrease condition
%      alpha_l is the one with smallest phi value
% (P3) alpha_h is chose such that dphi(alpha_l)*(alpha_h - alpha_l) <
 0
%
% Copyright (C) 2017 Kiko Rullan, Marta M. Betcke

% Parameters
% Trial step in {'bisection', 'interp2'}
TRIALSTEP = 'bisection';

% Structure containing information about the iteration
info.alpha_ls = [];
info.alpha_hs = [];
info.alpha_js = [];
info.phi_js = [];
info.dphi_js = [];

n = 1;
stop = false;
maxIter = 10;
while (n < maxIter && stop == false)
    % Find trial step length alpha_j in [alpha_l, aplha_h]
    switch TRIALSTEP
      case 'bisection'
        alpha_j = 0.5*(alpha_h + alpha_l);
      case 'interp2'
    end
    phi_j = Phi.phi(alpha_j);

    % Update info
    info.alpha_ls = [info.alpha_ls alpha_l];
    info.alpha_hs = [info.alpha_hs alpha_h];
    info.alpha_js = [info.alpha_js alpha_j];
    info.phi_js = [info.phi_js phi_j];

    if (phi_j > Phi.phi(0) + c1*alpha_j*Phi.dphi(0) ||
 Phi.phi(alpha_j) >= Phi.phi(alpha_l))
      % alpha_j does not satisfy sufficient decrease condition -> look
 for alpha < alpha_j
      % or phi(alpha_j) >= phi(alpha_l)
      % -> [alpha_l, alpha_j]
      alpha_h = alpha_j;

      % Update info
      info.dphi_js = [info.dphi_js NaN];

    else
       % alpha_j satisfies sufficient decrease condition
        dphi_j = Phi.dphi(alpha_j);
        % Update info
        info.dphi_js = [info.dphi_js dphi_j];

        if (abs(dphi_j) <= -c2*Phi.dphi(0))
          % alpha_j satisfies strong curvature condition
            alpha = alpha_j;
            stop = true;
        elseif (dphi_j*(alpha_h - alpha_l) >= 0)
          % alpha_h : dphi(alpha_l)*(alpha_h - alpha_l) < 0
          % alpha_j violates this condition but swapping alpha_l <->
 alpha_h will reestabish it
          % -> [alpha_j, alpha_l]
            alpha_h = alpha_l;
        end
        alpha_l = alpha_j;
    end
end
end
```

*Published with MATLAB® R2016b*