COMPGV19: Tutorial 2

Table of Contents

Marta Betcke and Kiko Rul#lan

Exercise 1

Program the steepest descent and Newton algorithms using the backtracking line search, Algorithm 3.1. Use them to minimize the Rosenbrock function (2.22). Set the initial step length `alpha_0 = 1` and print the step length used by each method at each iteration. First try the initial point `x_0 = (1.2, 1.2)^T` and then use the more difficult starting point `x_0 = (-1.2, 1)^T`.


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

Rosenbrock function

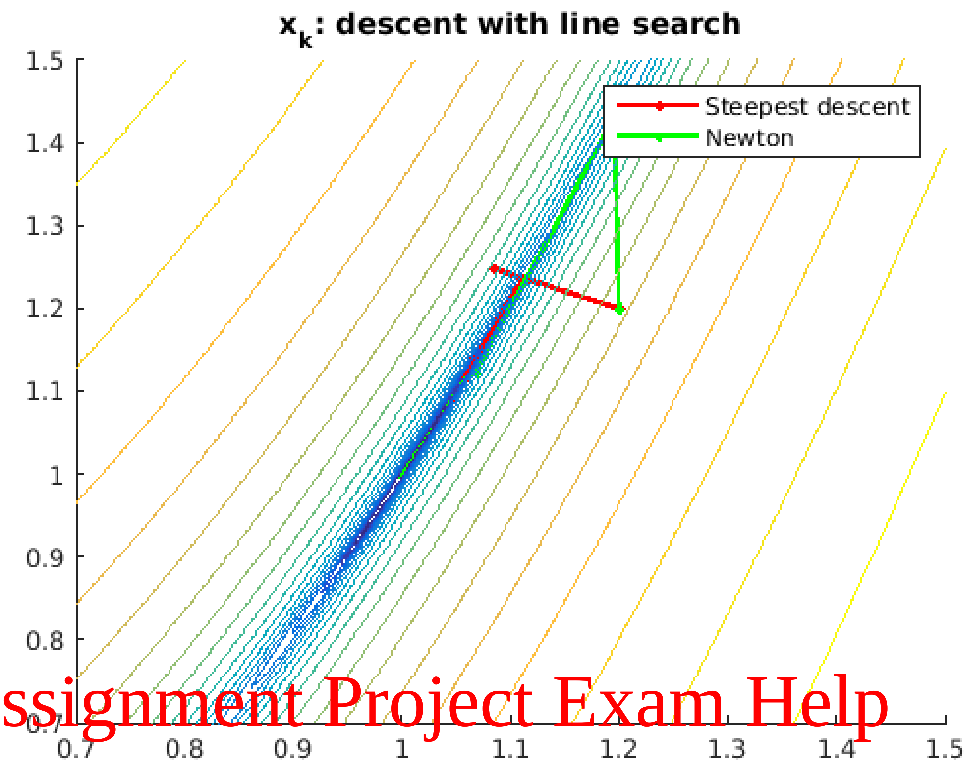
For computation define as function of 1 vector variable

```
F.f = @(x) 100.*(x(2) - x(1)^2).^2 + (1 - x(1)).^2; % function
 handler, 2-dim vector
F.df = @(x) [-400*(x(2) - x(1)^2)*x(1) - 2*(1 - x(1));
              200*(x(2) - x(1)^2)];  % gradient handler, 2-dim vector
F.d2f = @(x) [-400*(x(2) - 3*x(1)^2) + 2, -400*x(1); -400*x(1),
 200]; % hessian handler, 2-dim vector
% For visualisation proposes define as function of 2 variables
rosenbrock = @(x,y) 100.*(y - x.^2).^2 + (1 - x).^2;
```

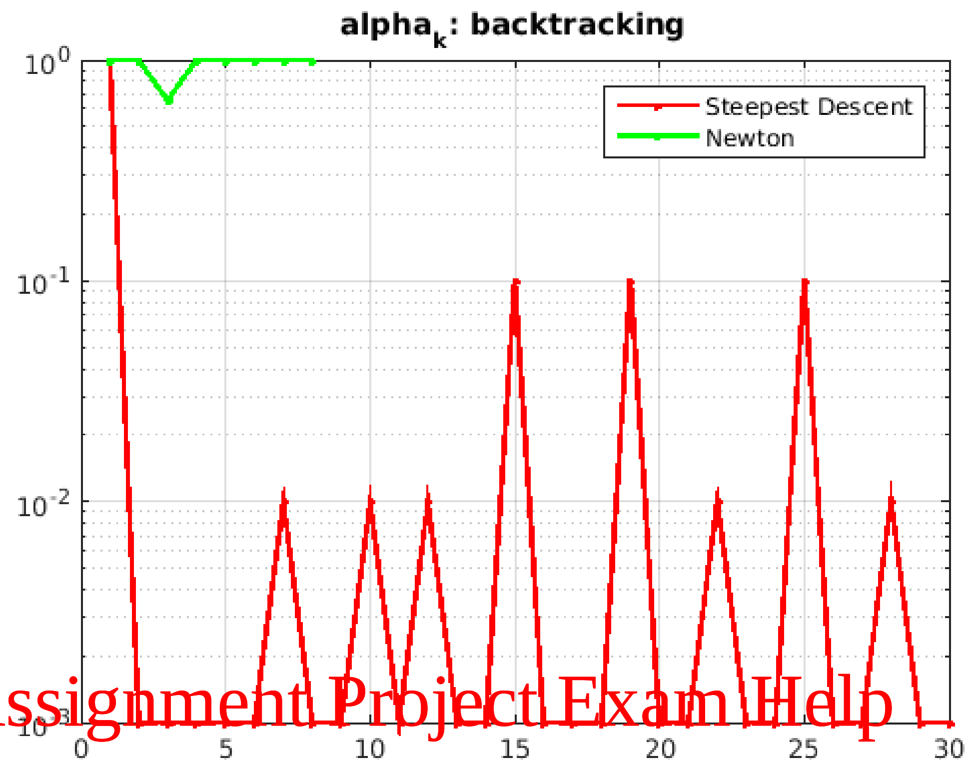
Backtracking line search

```
% Initialisation
alpha0 = 1;
c1 = 1e-4;
tol = 1e-6;
maxIter = 300;

%==============================
% Point x0 = [1.2; 1.2]
%==============================
```

```
x0 = [1.2; 1.2];

% Steepest descent backtracking line search
lsOptsSteep.rho = 0.1;
lsOptsSteep.c1 = c1;
lsFun = @(x_k, p_k, alpha0) backtracking(F, x_k, p_k, alpha0,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F, 'steepest', lsFun, alpha0, x0, tol, maxIter);

% Newton backtracking line search
lsOptsNewton.rho = 0.9;
lsOptsNewton.c1 = c1;
lsFun = @(x_k, p_k, alpha0) backtracking(F, x_k, p_k, alpha0,
 lsOptsNewton);
[xNewton, fNewton, nIterNewton, infoNewton] =
 descentLineSearch(F, 'newton', lsFun, alpha0, x0, tol, maxIter);

% Define grid for visualisation
n = 300;
x = linspace(0.7,1.5,n+1);
y = x;
[X, Y] = meshgrid(x,y);

% Iterate plot
figure;
hold on;
plot(infoSteep.xs(1, :), infoSteep.xs(2, :), '-or', 'LineWidth',
 2, 'MarkerSize', 3); % only first 10 iterations
plot(infoNewton.xs(1, :), infoNewton.xs(2, :), '-*g', 'LineWidth',
 2, 'MarkerSize', 3); % only first 10 iterations
contour(X, Y, log(max(rosenbrock(X,Y), 1e-3)), 20);
title('x_k: descent with line search')
legend('Steepest descent', 'Newton');
```

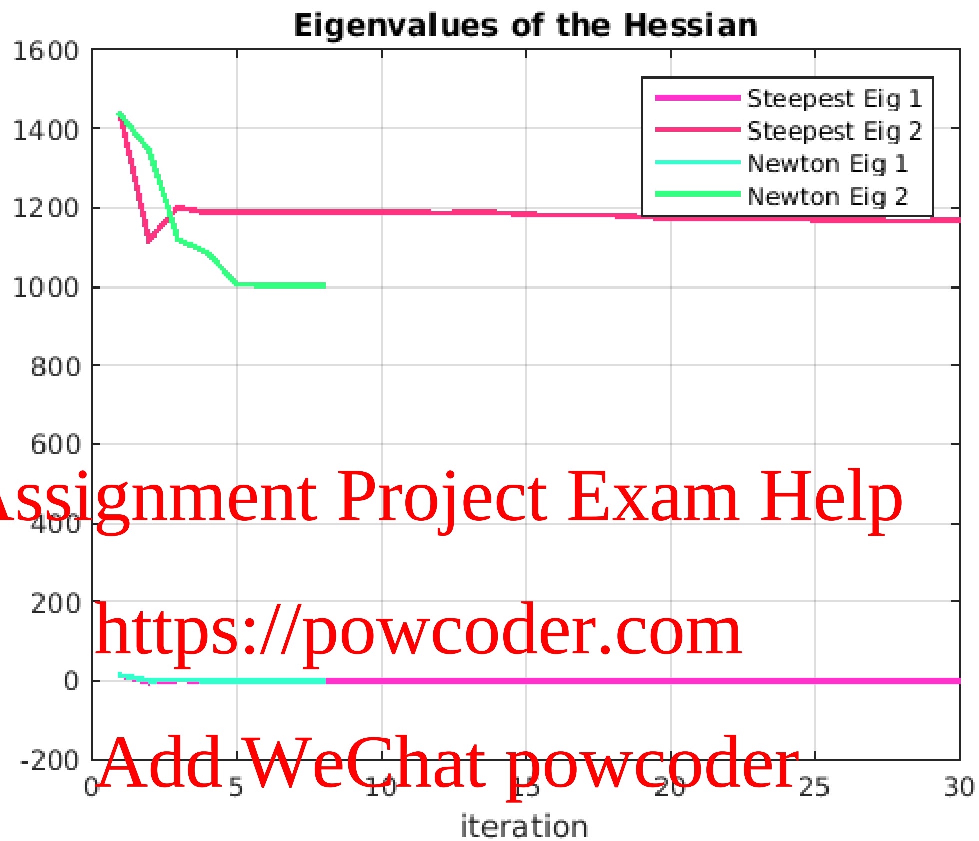


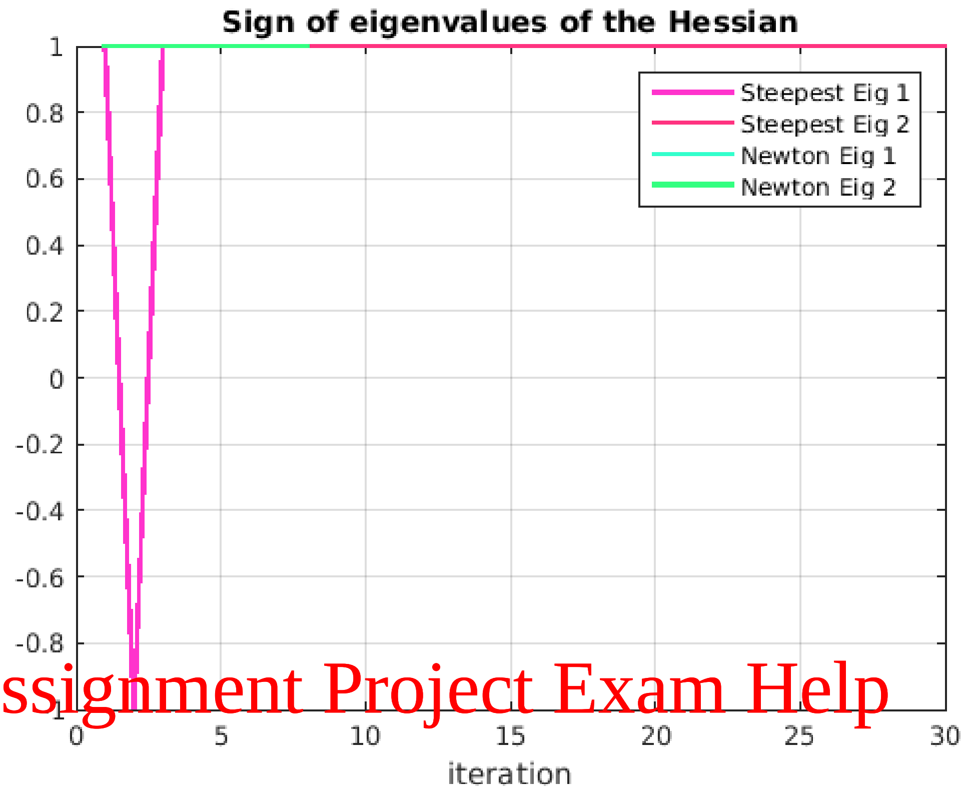

```
% Step length plot
figure;
semilogy(infoSteep.alphas(1:30), '-or', 'LineWidth', 2, 'MarkerSize',
 2); hold on;
semilogy(infoNewton.alphas(1:end), '-*g', 'LineWidth',
 2, 'MarkerSize', 2);
grid on;
title('alpha_k: backtracking');
legend('Steepest Descent', 'Newton');
```




```
% Plot eigenvalues
eigSteep = [];
eigNewton = [];
for i = 1:size(infoSteep.xs, 2)
    eigSteep = [eigSteep eig(F.d2f(infoSteep.xs(:, i)))];
end
for i = 1:size(infoNewton.xs, 2)
    eigNewton = [eigNewton eig(F.d2f(infoNewton.xs(:, i)))];
end
figure;
plot(eigSteep(1, 1:30), 'Color', [1 0.2 0.8], 'LineWidth', 2);
 hold on;
plot(eigSteep(2, 1:30), 'Color', [1 0.2 0.5], 'LineWidth', 2);
plot(eigNewton(1, :), 'Color', [0.2 1 0.8], 'LineWidth', 2);
plot(eigNewton(2, :), 'Color', [0.2 1 0.5], 'LineWidth', 2);
legend('Steepest Eig 1', 'Steepest Eig 2', 'Newton Eig 1', 'Newton Eig
 2');
title('Eigenvalues of the Hessian');
grid on;
xlabel('iteration');

figure;
plot(sign(eigSteep(1, 1:30)), 'Color', [1 0.2 0.8], 'LineWidth', 2);
 hold on;
plot(sign(eigSteep(2, 1:30)), 'Color', [1 0.2 0.5], 'LineWidth', 2);
plot(sign(eigNewton(1, :)), 'Color', [0.2 1 0.8], 'LineWidth', 2);
plot(sign(eigNewton(2, :)), 'Color', [0.2 1 0.5], 'LineWidth', 2);
```

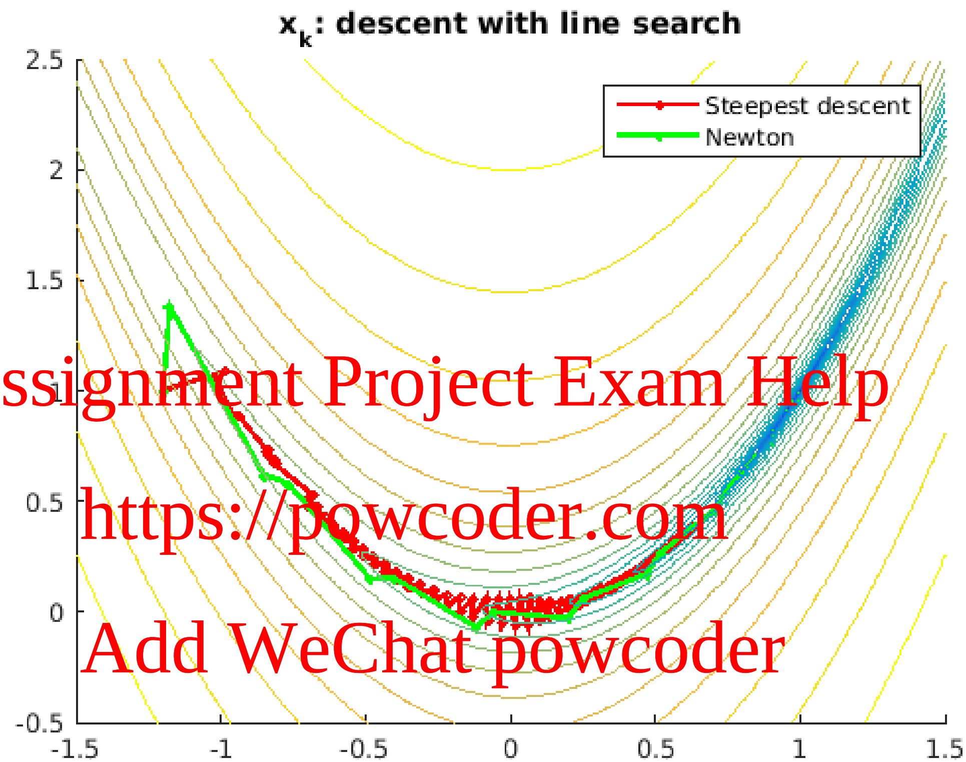
```
legend('Steepest Eig 1', 'Steepest Eig 2', 'Newton Eig 1', 'Newton Eig
 2');
title('Sign of eigenvalues of the Hessian');
grid on;
xlabel('iteration');
```



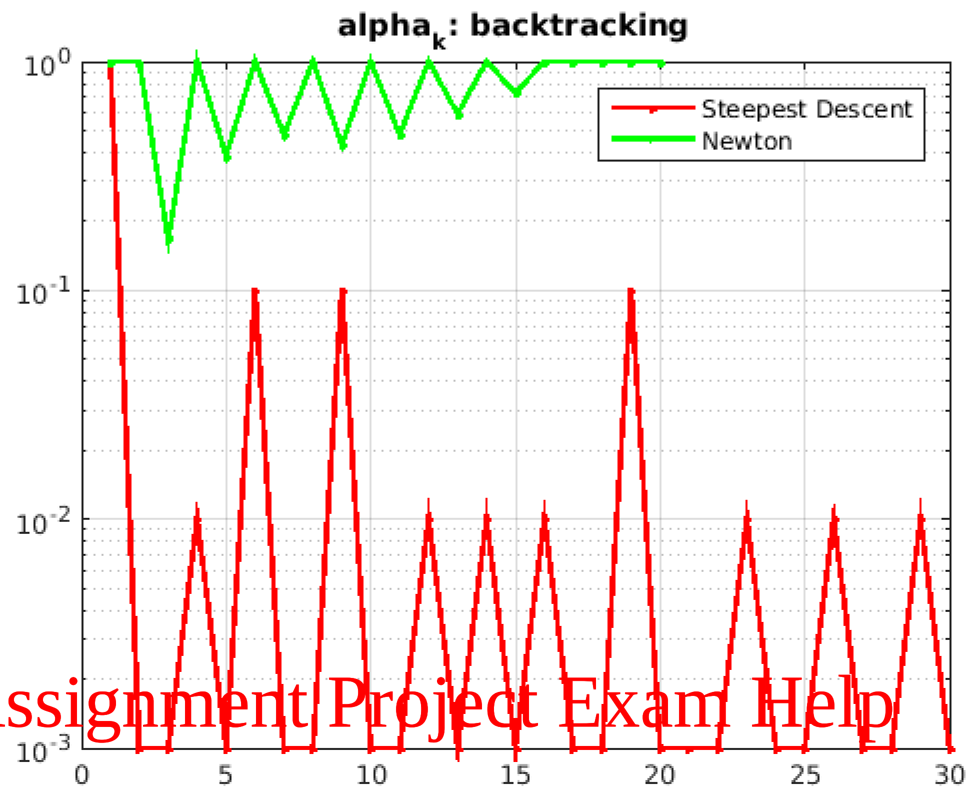



```
%=============================
% Point x0 = [-1.2; 1]
%=============================
x0 = [-1.2; 1];

% Steepest descent backtracking line search
lsOptsSteep.rho = 0.1;
lsOptsSteep.c1 = c1;
lsFun = @(x_k, p_k, alpha0) backtracking(F, x_k, p_k, alpha0,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F, 'steepest', lsFun, alpha0, x0, tol, maxIter);

% Newton backtracking line search
lsOptsNewton.rho = 0.9;
lsOptsNewton.c1 = c1;
lsFun = @(x_k, p_k, alpha0) backtracking(F, x_k, p_k, alpha0,
 lsOptsNewton);
[xNewton, fNewton, nIterNewton, infoNewton] =
 descentLineSearch(F, 'newton', lsFun, alpha0, x0, tol, maxIter);

% Define grid for visualisation
n = 300;
x = linspace(-1.5,1.5,n+1);
y = linspace(-0.5,2.5,n+1);
[X,Y] = meshgrid(x,y);

% Iterate plot
```

```
figure;
hold on;
plot(infoSteep.xs(1, :), infoSteep.xs(2, :), '-or', 'LineWidth',
 2, 'MarkerSize', 3); % only first 10 iterations
plot(infoNewton.xs(1, :), infoNewton.xs(2, :), '-*g', 'LineWidth',
 2, 'MarkerSize', 3); % only first 10 iterations
contour(X, Y, log(max(rosenbrock(X,Y), 1e-3)), 20);
title('x_k: descent with line search')
legend('Steepest descent', 'Newton');
```

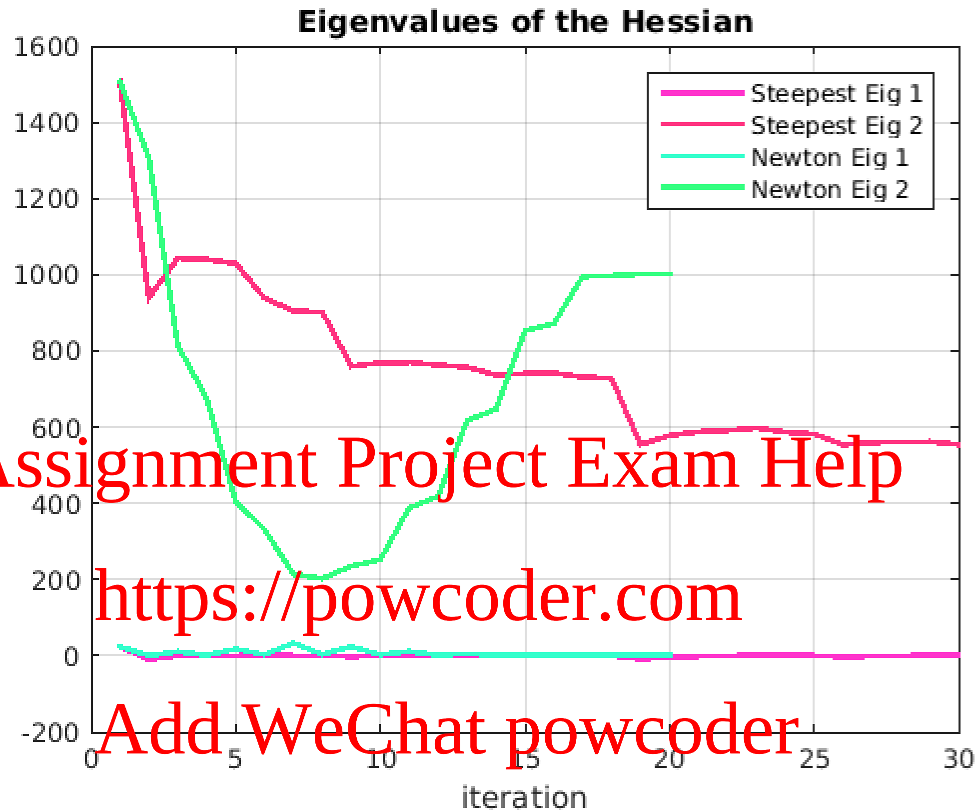


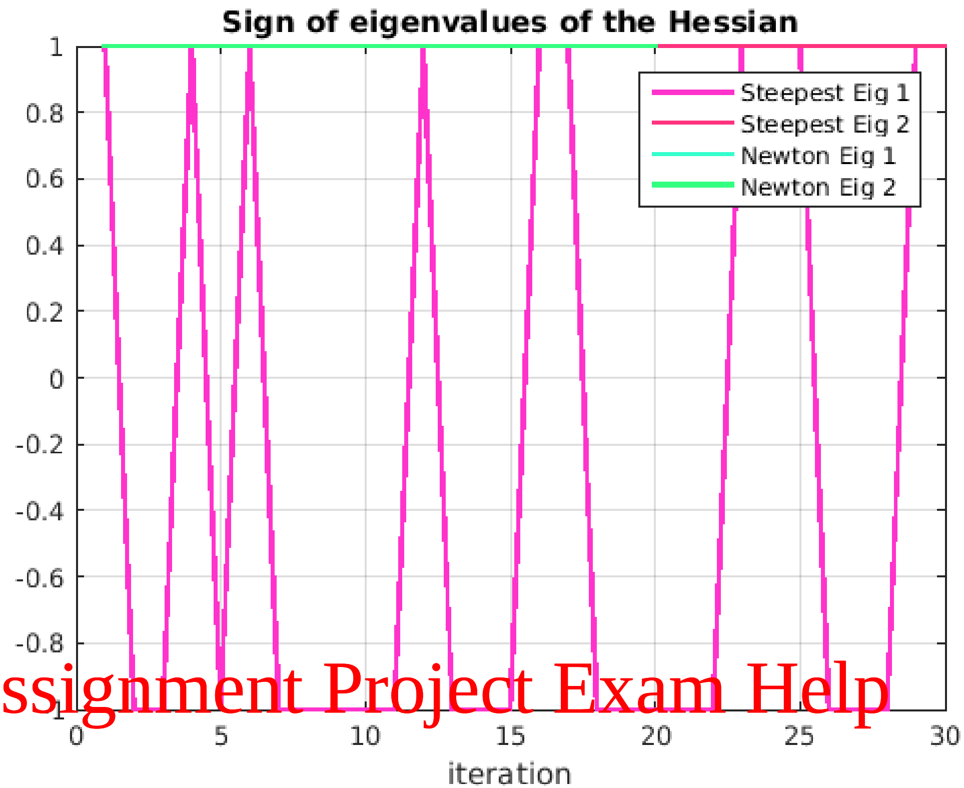

```
% Step length plot
figure;
semilogy(infoSteep.alphas(1:30), '-or', 'LineWidth', 2, 'MarkerSize',
 2); hold on;
semilogy(infoNewton.alphas(1:end), '-*g', 'LineWidth',
 2, 'MarkerSize', 2);
grid on;
title('alpha_k: backtracking');
legend('Steepest Descent', 'Newton');
```




```
% Plot eigenvalues
eigSteep = [];
eigNewton = [];
for i = 1:size(infoSteep.xs, 2)
    eigSteep = [eigSteep eig(F.d2f(infoSteep.xs(:, i)))];
end
for i = 1:size(infoNewton.xs, 2)
    eigNewton = [eigNewton eig(F.d2f(infoNewton.xs(:, i)))];
end
figure;
plot(eigSteep(1, 1:30), 'Color', [1 0.2 0.8], 'LineWidth', 2);
 hold on;
plot(eigSteep(2, 1:30), 'Color', [1 0.2 0.5], 'LineWidth', 2);
plot(eigNewton(1, :), 'Color', [0.2 1 0.8], 'LineWidth', 2);
plot(eigNewton(2, :), 'Color', [0.2 1 0.5], 'LineWidth', 2);
legend('Steepest Eig 1', 'Steepest Eig 2', 'Newton Eig 1', 'Newton Eig
 2');
title('Eigenvalues of the Hessian');
grid on;
xlabel('iteration');

figure;
plot(sign(eigSteep(1, 1:30)), 'Color', [1 0.2 0.8], 'LineWidth', 2);
 hold on;
plot(sign(eigSteep(2, 1:30)), 'Color', [1 0.2 0.5], 'LineWidth', 2);
plot(sign(eigNewton(1, :)), 'Color', [0.2 1 0.8], 'LineWidth', 2);
```

```
plot(sign(eigNewton(2, :)), 'Color', [0.2 1 0.5], 'LineWidth', 2);
legend('Steepest Eig 1', 'Steepest Eig 2', 'Newton Eig 1', 'Newton Eig
 2');
title('Sign of eigenvalues of the Hessian');
grid on;
xlabel('iteration');
```

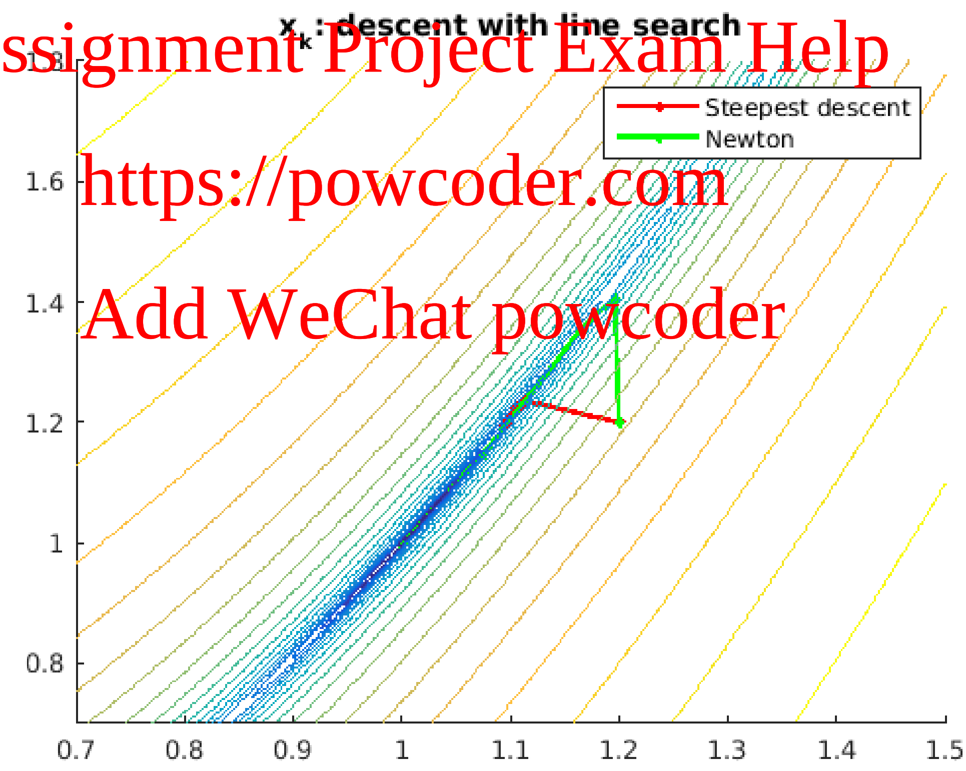




Line search satisfying strong Wolfe conditions

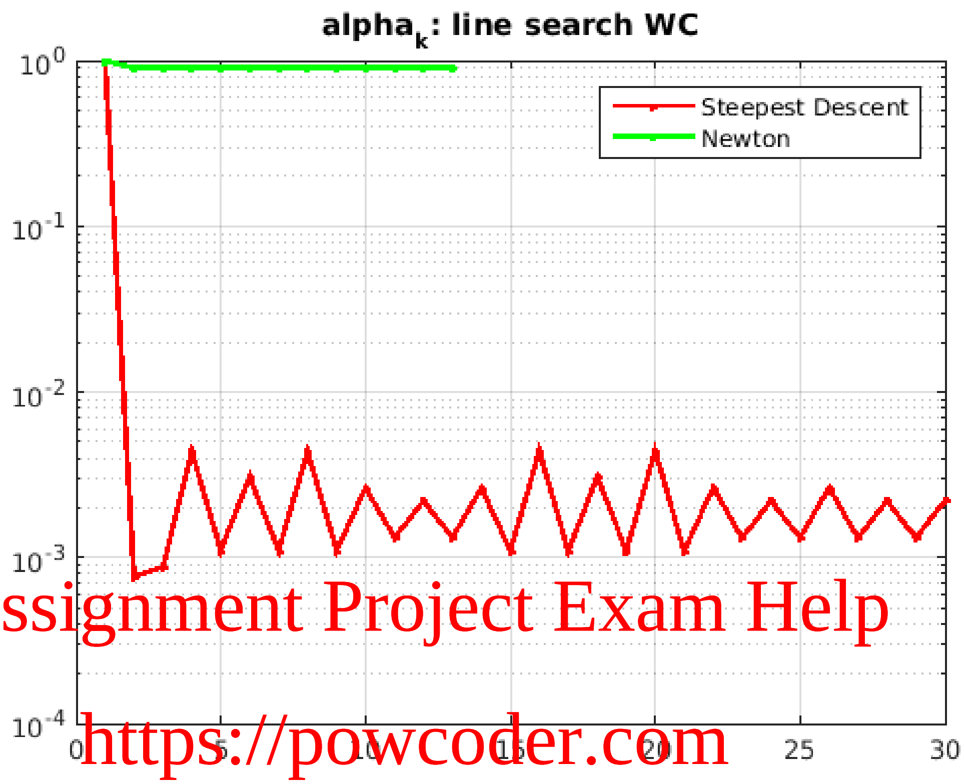

```
% Initialisation
alpha_max = alpha0;
c1 = 1e-4; %0.01;
%c2 = 0.9; % choose 0.1 for steepest descent, 0.9 for Newton
tol = 1e-6;

%=============================
% Point x0 = [1.2; 1.2]
%=============================
x0 = [1.2; 1.2];

% Steepest descent line search strong WC
lsOptsSteep.c1 = c1;
lsOptsSteep.c2 = 0.1;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha_max,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F, 'steepest', lsFun, alpha0, x0, tol, maxIter);

% Newton line search strong WC
lsOptsNewton.c1 = c1;
lsOptsNewton.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha_max,
 lsOptsNewton);
```

```
[xNewton, fNewton, nIterNewton, infoNewton] =
 descentLineSearch(F, 'newton', lsFun, alpha0, x0, tol, maxIter);

% Define grid for visualisation
n = 300;
x = linspace(0.7,1.5,n+1);
y = linspace(0.7,1.8,n+1);
[X,Y] = meshgrid(x,y);


% Iterate plot
figure;
hold on;
plot(infoSteep.xs(1, :), infoSteep.xs(2, :), '-or', 'LineWidth',
 2, 'MarkerSize', 3); % only first 10 iterations
plot(infoNewton.xs(1, :), infoNewton.xs(2, :), '-*g', 'LineWidth',
 2, 'MarkerSize', 3); % only first 10 iterations
contour(X, Y, log(max(rosenbrock(X,Y), 1e-3)), 20);
title('x_k: descent with line search')
legend('Steepest descent', 'Newton');
```



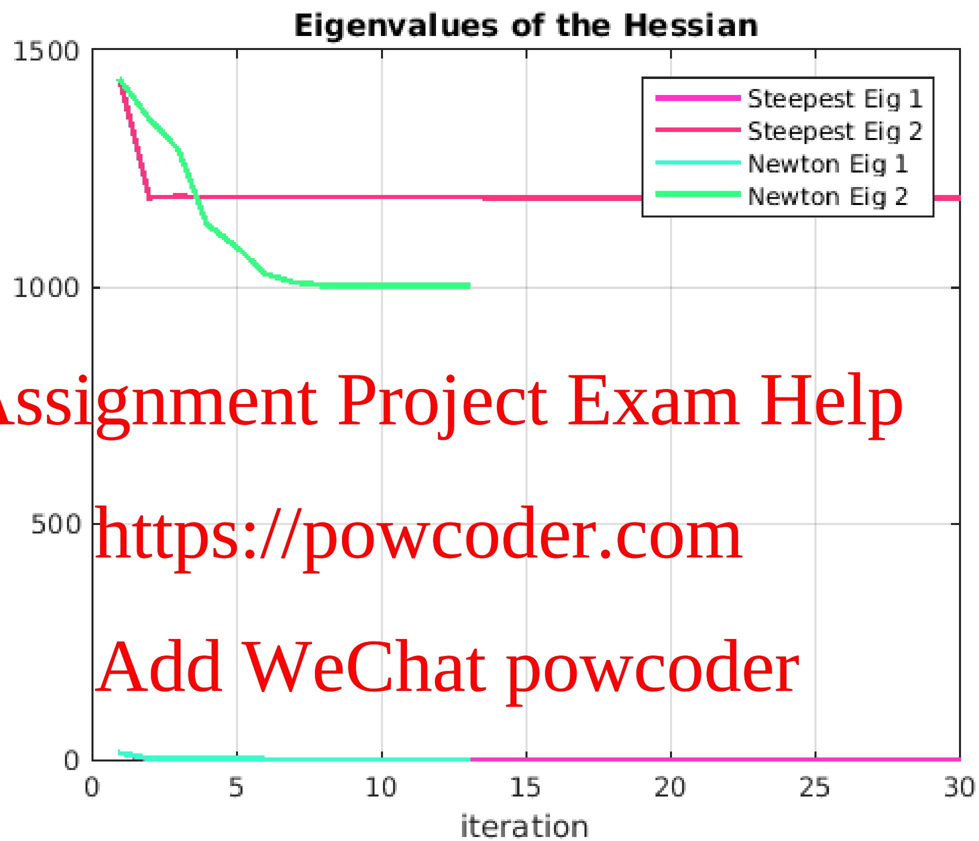

```
% Step length plot
figure;
semilogy(infoSteep.alphas(1:30), '-or', 'LineWidth', 2, 'MarkerSize',
 2); hold on;
semilogy(infoNewton.alphas(1:end), '-*g', 'LineWidth',
 2, 'MarkerSize', 2);
grid on;
```

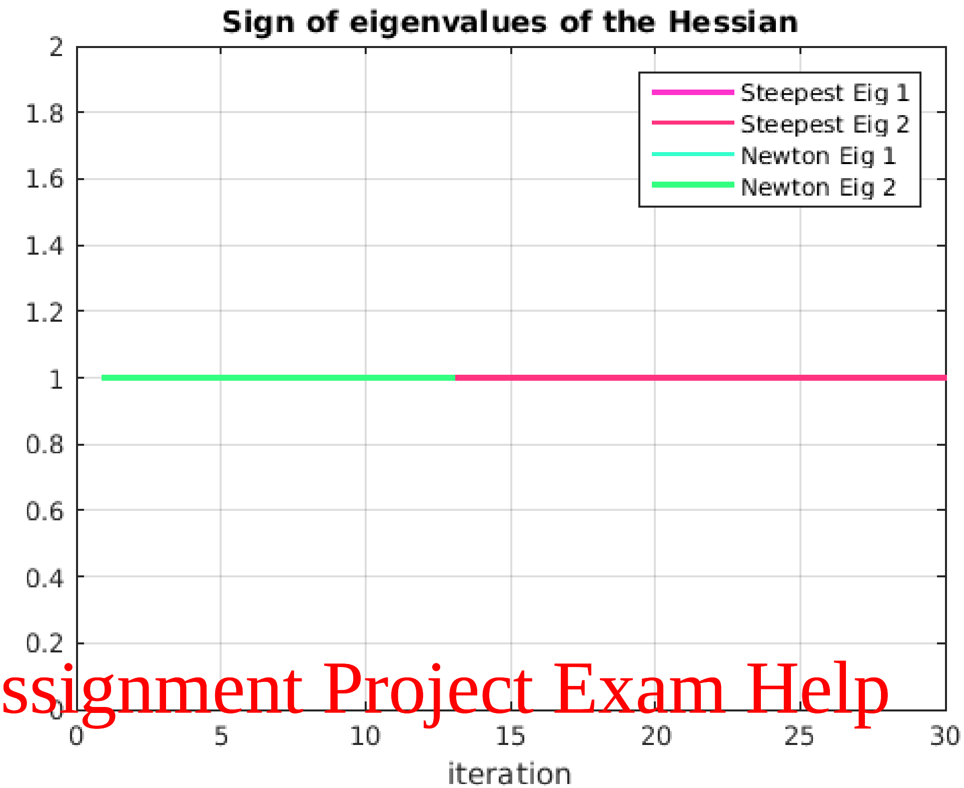
```
title('alpha_k: line search WC');
legend('Steepest Descent', 'Newton');
```




```
% Plot eigenvalues
eigSteep = [];
eigNewton = [];
for i = 1:size(infoSteep.xs, 2)
    eigSteep = [eigSteep eig(F.d2f(infoSteep.xs(:, i)))];
end
for i = 1:size(infoNewton.xs, 2)
    eigNewton = [eigNewton eig(F.d2f(infoNewton.xs(:, i)))];
end
figure;
plot(eigSteep(1, 1:30), 'Color', [1 0.2 0.8], 'LineWidth', 2);
 hold on;
plot(eigSteep(2, 1:30), 'Color', [1 0.2 0.5], 'LineWidth', 2);
plot(eigNewton(1, :), 'Color', [0.2 1 0.8], 'LineWidth', 2);
plot(eigNewton(2, :), 'Color', [0.2 1 0.5], 'LineWidth', 2);
legend('Steepest Eig 1', 'Steepest Eig 2', 'Newton Eig 1', 'Newton Eig
 2');
title('Eigenvalues of the Hessian');
grid on;
xlabel('iteration');

figure;
plot(sign(eigSteep(1, 1:30)), 'Color', [1 0.2 0.8], 'LineWidth', 2);
 hold on;
```

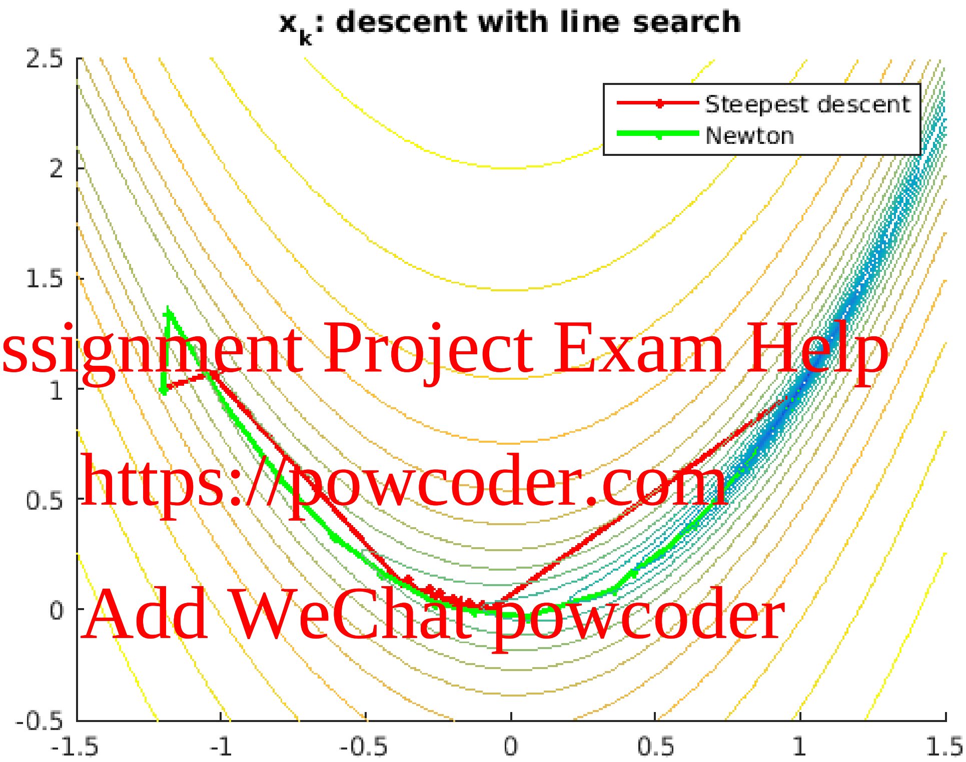
```
plot(sign(eigSteep(2, 1:30)), 'Color', [1 0.2 0.5], 'LineWidth', 2);
plot(sign(eigNewton(1, :)), 'Color', [0.2 1 0.8], 'LineWidth', 2);
plot(sign(eigNewton(2, :)), 'Color', [0.2 1 0.5], 'LineWidth', 2);
legend('Steepest Eig 1', 'Steepest Eig 2', 'Newton Eig 1', 'Newton Eig
 2');
title('Sign of eigenvalues of the Hessian');
grid on;
xlabel('iteration');
```



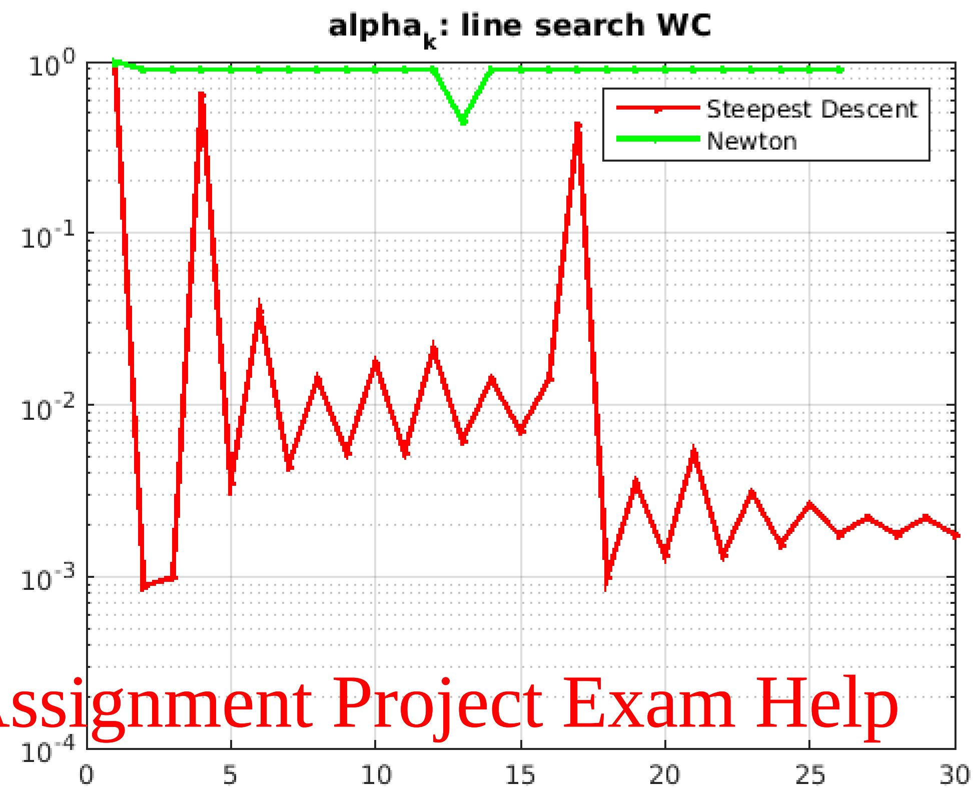



```
%==============================
% Point x0 = [-1.2; 1]
%==============================
x0 = [-1.2; 1];

% Steepest descent line search strong WC
lsOptsSteep.c1 = c1;
lsOptsSteep.c2 = 0.1;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha_max,
 lsOptsSteep);
[xSteep, fSteep, nIterSteep, infoSteep] =
 descentLineSearch(F, 'steepest', lsFun, alpha0, x0, tol, maxIter);

% Newton g line search strong WC
lsOptsNewton.c1 = c1;
lsOptsNewton.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha_max,
 lsOptsNewton);
[xNewton, fNewton, nIterNewton, infoNewton] =
 descentLineSearch(F, 'newton', lsFun, alpha0, x0, tol, maxIter);

% Define grid for visualisation
n = 300;
x = linspace(-1.5,1.5,n+1);
y = linspace(-0.5,2.5,n+1);
[X,Y] = meshgrid(x,y);
```

```
% Iterate plot
figure;
hold on;
plot(infoSteep.xs(1, :), infoSteep.xs(2, :), '-or', 'LineWidth',
 2, 'MarkerSize', 3); % only first 10 iterations
plot(infoNewton.xs(1, :), infoNewton.xs(2, :), '-*g', 'LineWidth',
 2, 'MarkerSize', 3); % only first 10 iterations
contour(X, Y, log(max(rosenbrock(X,Y), 1e-3)), 20);
title('x_k: descent with line search')
legend('Steepest descent', 'Newton');
```

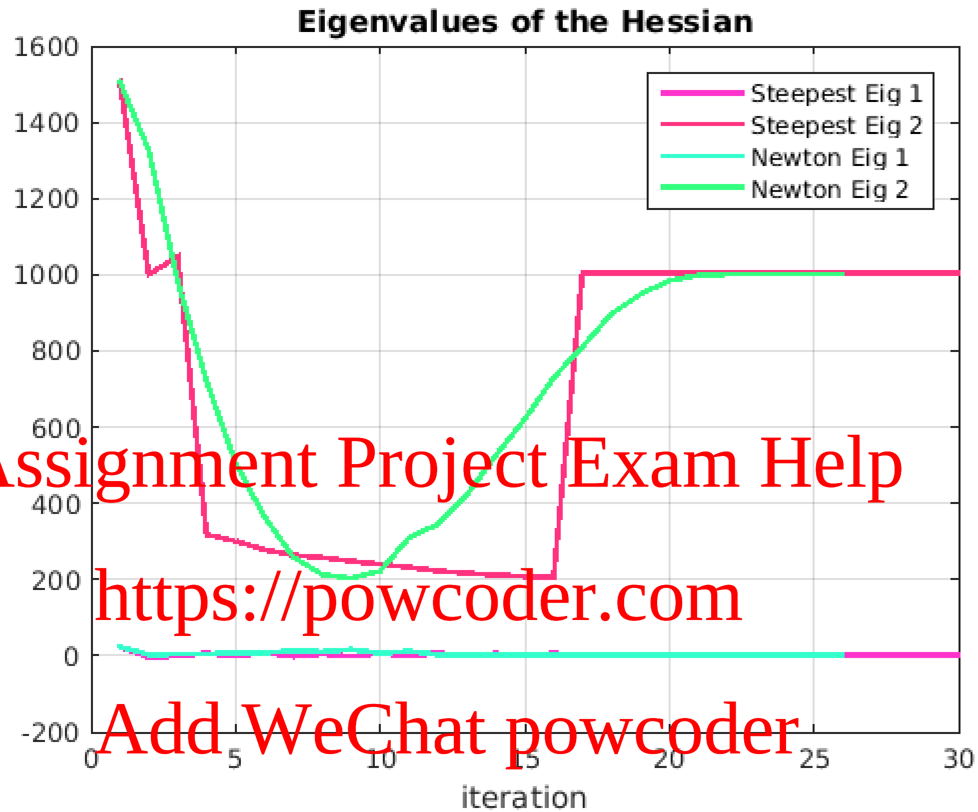


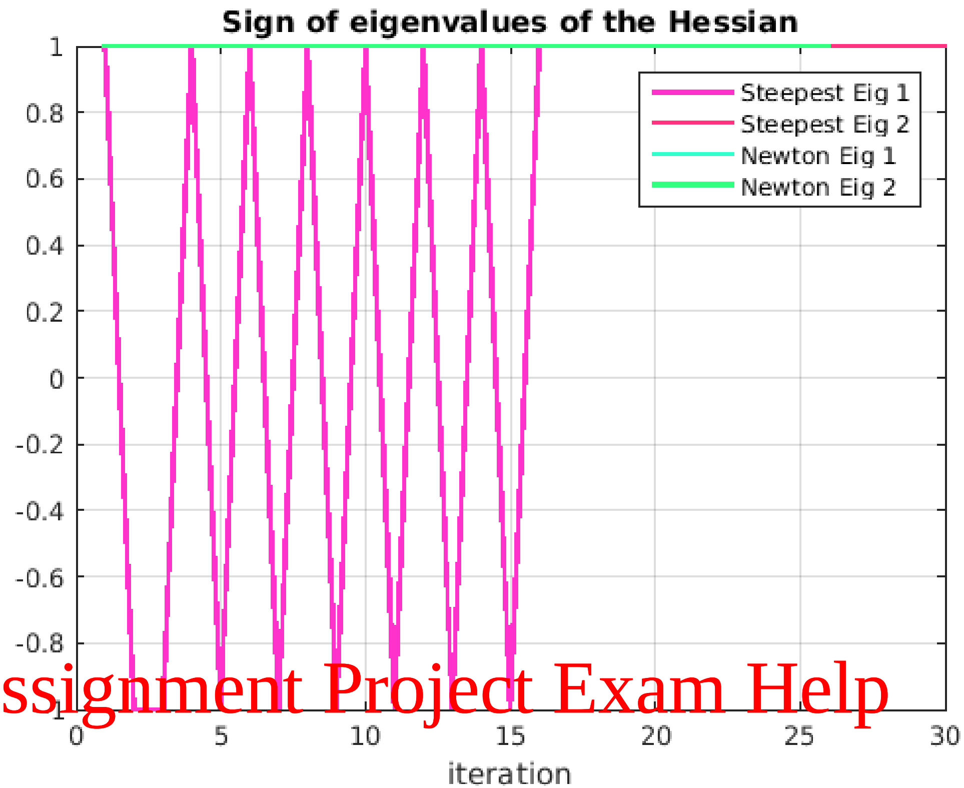

```
% Step length plot
figure;
semilogy(infoSteep.alphas(1:30), '-or', 'LineWidth', 2, 'MarkerSize',
 2); hold on;
semilogy(infoNewton.alphas(1:end), '-*g', 'LineWidth',
 2, 'MarkerSize', 2);
grid on;
title('alpha_k: line search WC');
legend('Steepest Descent', 'Newton');
```




```
% Plot eigenvalues
eigSteep = [];
eigNewton = [];
for i = 1:size(infoSteep.xs, 2)
    eigSteep = [eigSteep eig(F.d2f(infoSteep.xs(:, i)))];
end
for i = 1:size(infoNewton.xs, 2)
    eigNewton = [eigNewton eig(F.d2f(infoNewton.xs(:, i)))];
end
figure;
plot(eigSteep(1, 1:30), 'Color', [1 0.2 0.8], 'LineWidth', 2);
 hold on;
plot(eigSteep(2, 1:30), 'Color', [1 0.2 0.5], 'LineWidth', 2);
plot(eigNewton(1, :), 'Color', [0.2 1 0.8], 'LineWidth', 2);
plot(eigNewton(2, :), 'Color', [0.2 1 0.5], 'LineWidth', 2);
legend('Steepest Eig 1', 'Steepest Eig 2', 'Newton Eig 1', 'Newton Eig
 2');
title('Eigenvalues of the Hessian');
grid on;
xlabel('iteration');

figure;
plot(sign(eigSteep(1, 1:30)), 'Color', [1 0.2 0.8], 'LineWidth', 2);
 hold on;
plot(sign(eigSteep(2, 1:30)), 'Color', [1 0.2 0.5], 'LineWidth', 2);
plot(sign(eigNewton(1, :)), 'Color', [0.2 1 0.8], 'LineWidth', 2);
```

```
plot(sign(eigNewton(2, :)), 'Color', [0.2 1 0.5], 'LineWidth', 2);
legend('Steepest Eig 1', 'Steepest Eig 2', 'Newton Eig 1', 'Newton Eig
 2');
title('Sign of eigenvalues of the Hessian');
grid on;
xlabel('iteration');
```





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

descentLineSearch.m

Wrapper function executing iteration with descent direction and line search method

```
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'}
% ls: function handle for computing the step length
% 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 minimal allowed step
%      norm(x_k - x_k_1)/norm(x_k) < tol;
% maxIter: maximum number of iterations
%
% OUTPUTS
% xMin, fMin: minimum and value of f at the minimum
% nIter: number of iterations
% info: structure with information about the iteration
%   - xs: iterate history
%   - alphas: step lengths history
%
% Copyright (C) 2017  Marta M. Betcke, Kiko Rullan

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

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

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

  % Increment iterations
    nIter = nIter + 1;

    % Compute descent direction
    switch lower(descent)
      case 'steepest'
        p_k = -F.df(x_k); % steepest descent direction
      case 'newton'
        p_k = -F.d2f(x_k)\F.df(x_k); % Newton direction
    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;

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

    switch stopType
      case 'step'
        % Compute relative step length
        normStep = norm(x_k - x_k_1)/norm(x_k_1);
        stopCond = (normStep < tol);
```

```
      case 'grad'
        stopCond = (norm(F.df(x_k), 'inf') < tol*(1 +
 tol*abs(F.f(x_k))));
    end

end

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

backtracking.m

Backtracking line search

```
function [alpha, info] = backtracking(F, x_k, p, alpha0, opts)
% BACKTRACKING Backtracking line search to satisfy sufficient decrease
 condition
% [alpha, info] = backtracking(F, x_k, p, alpha0, opts)
%
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
% x_k: current iterate
% p: descent direction
% alpha0: initial step length
% opts: backtracking specific option structure with fields
%   - rho: in (0,1) backtraking step length reduction factor
%   - c1: constant in sufficient decrease condition f(x_k + alpha_k*p)
 > f(x_k) + c1*alpha_k*(df_k'*p)
%         Typically chosen small, (default 1e-4).
%
% OUTPUTS
% alpha: step length
% nIter: number of iterations
% info: structure with information about the backtracking iteration
%   - alphas: step lengths history
%
% Copyright (C) 2017  Marta M. Betcke, Kiko Rullan


% Default values
if nargin < 5 || ~isfield(opts, 'c1')
  opts.c1 = 1e-4;
end
% Choose
% rho = 0.1 for steepest descent, conjugate gradients
% rho = 0.9 for Newton, Quasi-Newton
if nargin < 5 || ~isfield(opts, 'rho')
```

```
  opts.rho = 0.9;  % Newton
end
if nargin < 4
  alpha0 = 1; % Newton
end

% Initialize info structure
info.alphas = alpha0;
info.rho = opts.rho;
info.c1 = opts.c1;

% Initial step length
alpha = alpha0;

% Compute f, grad f at x_k
f_k = F.f(x_k);
df_k = F.df(x_k);

% Backtracking linesearch for computing step length
while F.f(x_k + alpha*p) > f_k + opts.c1*alpha*(df_k')*p
  alpha = opts.rho*alpha;
  info.alphas = [info.alphas alpha];
end
```


lineSearch.m

Line search algorithm find steps satisfying strong Wolfe conditions (Wright, Nocedal Algorithm 3.5)


```
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 = zoomInt(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 = zoomInt(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;
```

Zoom function used by the line search above (Wright, Nocedal Algorithm 3.6)

```
function [alpha, info] = zoomInt(Phi, alpha_l, alpha_h, c1, c2)
% ZOOMINT Zoom algorithm for line search with strong Wolfe conditions
% alpha = zoomInt(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';
tol = eps;

% 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 = 100;
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 abs(alpha_h - alpha_l) < tol
      alpha = alpha_j;
      stop = true;
      warning('Line search stopped because the interval became to
 small. Return centre of the interval.')
    end

    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 reestablish it
          % -> [alpha_j, alpha_l]
            alpha_h = alpha_l;
        end
        alpha_l = alpha_j;
    end
  end
```

Published with MATLAB® R2015a