The Chinese University of Hong Kong, Shenzhen
SDS · School of Data Science

Xiao Li · Junfeng Wu · Ming Yan · Fall Semester 2023

MAT 3007 – Optimization

Assignment 8

Due: 11:59pm, Dec. 8 (Friday), 2023

Instructions:

- Homework problems must be carefully and clearly answered to receive full credit. Complete sentences that establish a clear logical progression are highly recommended.
- Please submit your assignment on Blackboard.
- The homework must be written in English.
- Late submission will not be graded.
- Each student must not copy homework solutions from another student or from any other source.

Problem 1 (A One-Dimensional Problem): (approx. *20* points)
We consider the optimization problem

$$\min_x \ f(x) := \frac{1}{4}(x^2-1)^2 + \frac{1}{2}(x-2)^2 \qquad \text{s.t.} \qquad x \in [0,2].$$

Implement the bisection **or** the golden section method to solve this problem and output a solution with accuracy at least 10^{-5}.

Solution : The general `MATLAB` code for the bisection and golden section method can be found in Listing 1–2.

Notice that the golden section method does not require knowledge of f'. Hence, applying the golden section method can save some preparatory calculations. We can run and compare the bisection and golden section method for **Problem 1**:

The golden section method returns the solution $x_{\mathtt{g}}^* = 1.2599213$ after 26 iterations. The bisection method requires 19 iterations and it returns $x_{\mathtt{b}}^* = 1.259922$. A plot of the function f and the solution can be found below in Figure 1.

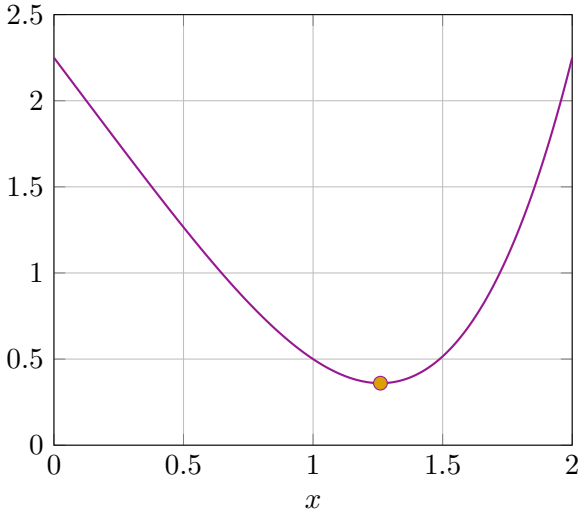

Figure 1: Plot of the function f of **Problem 1**.

Problem 2 (The Gradient Method): (approx. *30* points)
In this exercise, we want to solve the optimization problem

$$\min_{x\in\mathbb{R}^2} \; f(x) := 2x_1^4 + \frac{2}{3}x_1^3 + x_1^2 - 2x_1^2x_2 + \frac{4}{3}x_2^2. \tag{1}$$

via the gradient descent method.

Implement the gradient method that was presented in the lecture as a function `gradient_method` in `MATLAB` or `Python`.

The following input functions and parameters should be considered:

- `obj`, `grad` – function handles that calculate and return the objective function $f(x)$ and the gradient $\nabla f(x)$ at an input vector $x \in \mathbb{R}^n$. You can treat these handles as functions or fields of a class or structure `f` or you can use f and ∇f directly in your code. (For example, your function can have the form `gradient_method(obj,grad,...)`).
- x^0 – the initial point.
- `tol` – a tolerance parameter. The method should stop whenever the current iterate x^k satisfies the criterion $\|\nabla f(x^k)\| \leq$ `tol`.

We want to investigate the performance of the gradient method for different step size strategies. In particular, we want to test and compare backtracking and exact line search. The following parameters will be relevant for these strategies:

- $\sigma, \gamma \in (0, 1)$ – parameters for backtracking and the Armijo condition. (At iteration k, we choose α_k as the largest element in $\{1, \sigma, \sigma^2, \dots\}$ satisfying the condition $f(x^k - \alpha_k \nabla f(x^k)) - f(x^k) \leq -\gamma\alpha_k \cdot \|\nabla f(x^k)\|^2$).
- You can use the golden section method to determine the exact step size α_k. The parameters for the golden section method are: `maxit` (maximum number of iterations), $\texttt{tol}_\alpha$ (stopping tolerance for α, i.e. $\alpha_r - \alpha_l <$ $\texttt{tol}_\alpha$), $[0, \texttt{a}]$ (the interval of the step size with starting point 0 and `a`).

You can organize the latter parameters in an appropriate `options` class or structure. It is also possible to implement separate algorithms for backtracking and exact line search. The method(s) should return the final iterate x^k that satisfies the stopping criterion.

a) Apply the gradient method with backtracking and parameters $(\sigma, \gamma) = (0.5, 0.1)$ and exact line search ($\texttt{maxit} = 100$, $\texttt{tol}_\alpha = 10^{-6}$, $\texttt{a} = 2$) to solve the problem $\min_x f(x)$.

   The algorithms should use the stopping tolerance $\texttt{tol} = 10^{-5}$. Test the methods using the initial point $x^0 = (1, 1)^\top$ and report the behavior and performance of the methods. In particular, compare the number of iterations and the point to which the different gradient descent methods converged.

b) Let us define the set of initial points

$$\mathcal{X}^0 := \left\{ \begin{pmatrix} -3 \\ -3 \end{pmatrix}, \begin{pmatrix} 3 \\ -3 \end{pmatrix}, \begin{pmatrix} -3 \\ 3 \end{pmatrix}, \begin{pmatrix} 3 \\ 3 \end{pmatrix} \right\}$$

   Run the methods:

   - Gradient descent method with backtracking and $(\sigma, \gamma) = (0.5, 0.1)$,
   - Gradient method with exact line search and $\texttt{maxit} = 100$, $\texttt{tol}_\alpha = 10^{-6}$, $\texttt{a} = 2$,

   again for every initial point in the set $\mathcal{X}^0$ using the tolerance $\texttt{tol} = 10^{-5}$. For each algorithm/step size strategy create a single figure that contains all of the solution paths generated for the different initial points. The initial points and limit points should be clearly visible. Add a contour plot of the function f in the background of each figure.

Solution :

a) The function f has a global minimizer $x^* = (0, 0)^T$.

   The numerical results are summarized in Table 1 and the corresponding `MATLAB` code can be found in Listing 3–4.

Backtracking: $\gamma = 0.1$, $\sigma = 0.5$				
Initial Point	Iter.	Obj. Value	Final Iterate	Comment
$x^0 = (1, 1)^\top$	14	$3.9669 \cdot 10^{-12}$	$[0; 1.7249 \cdot 10^{-6}]$	Conv. to x^*

Exact Step Sizes: $\texttt{maxit} = 100$, $\texttt{tol} = 10^{-6}$, $\texttt{a} = 2$				
Initial Point	Iter.	Obj. Value	Final Iterate	Comment
$x^0 = (1, 1)^\top$	8	$6.3427 \cdot 10^{-13}$	$[-7.6262 \cdot 10^{-8}; 6.8654 \cdot 10^{-7}]$	Conv. to x^*

Table 1: Comparison of the gradient method using different step sizes strategies.

b) The solution paths are summarized and shown in Figure 2 and Figure 3. Exemplary code is presented in Listing 4. A more detailed table with the different results is presented in Table 2

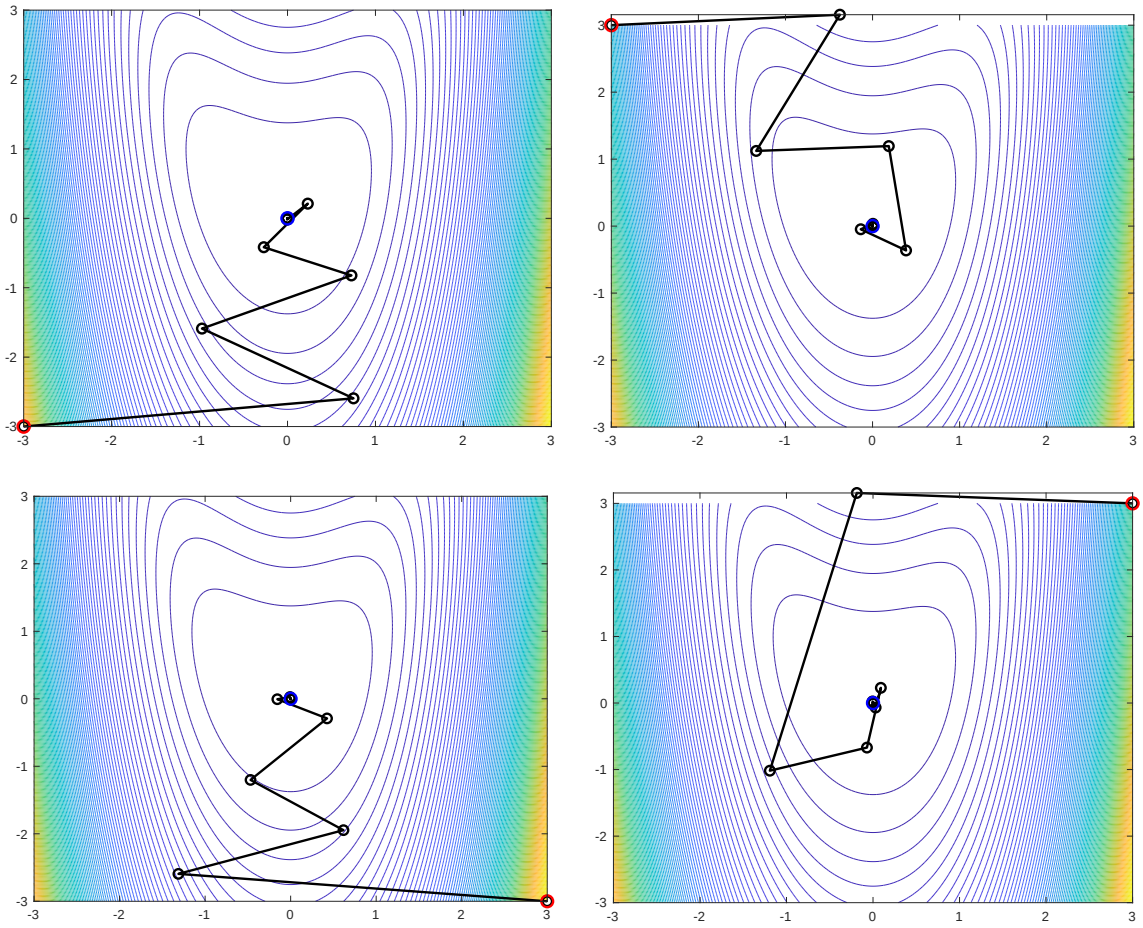

Figure 2: Problem 2: Solution path using backtracking (x axis stands for x_1 and y axis stands for x_2)

Problem 3 (A Limited-Memory Version of Adagrad): (approx. *25* points)
In this exercise, we investigate a limited-memory version of the gradient descent method with adaptive diagonal scaling. The update of the method is given by

$$x^{k+1} = x^k - \alpha_k D_k^{-1} \nabla f(x^k), \tag{2}$$

where $\alpha_k \geq 0$ is a suitable step size and $D_k \in \mathbb{R}^{n \times n}$ is a diagonal matrix that is chosen as follows: $D_k = \text{diag}(v_1^k, v_2^k, ..., v_n^k)$ and

$$v_i^k = \sqrt{\epsilon + \sum_{j=t_m(k)}^{k} (\nabla f(x^j))_i^2}, \quad t_m(k) = \max\{0, k-m\}, \quad \forall\ i = 1, ..., n.$$

Here, $\epsilon > 0$, the memory constant $m \in \mathbb{N}$, and the initial point $x^0 \in \mathbb{R}^n$ are given parameters.

a) Show that the direction $d^k = -D_k^{-1} \nabla f(x^k)$ is a descent direction for all $k \in \mathbb{N}$ (assuming $\nabla f(x^k) \neq 0$).

b) Write a `MATLAB` or `Python` code and implement the gradient method with adaptive diagonal scaling (Adagrad) and backtracking (i.e., implement the abstract descent method with d^k as descent direction). Run Adagrad on problem (1) introduced in **Problem 2** and compare

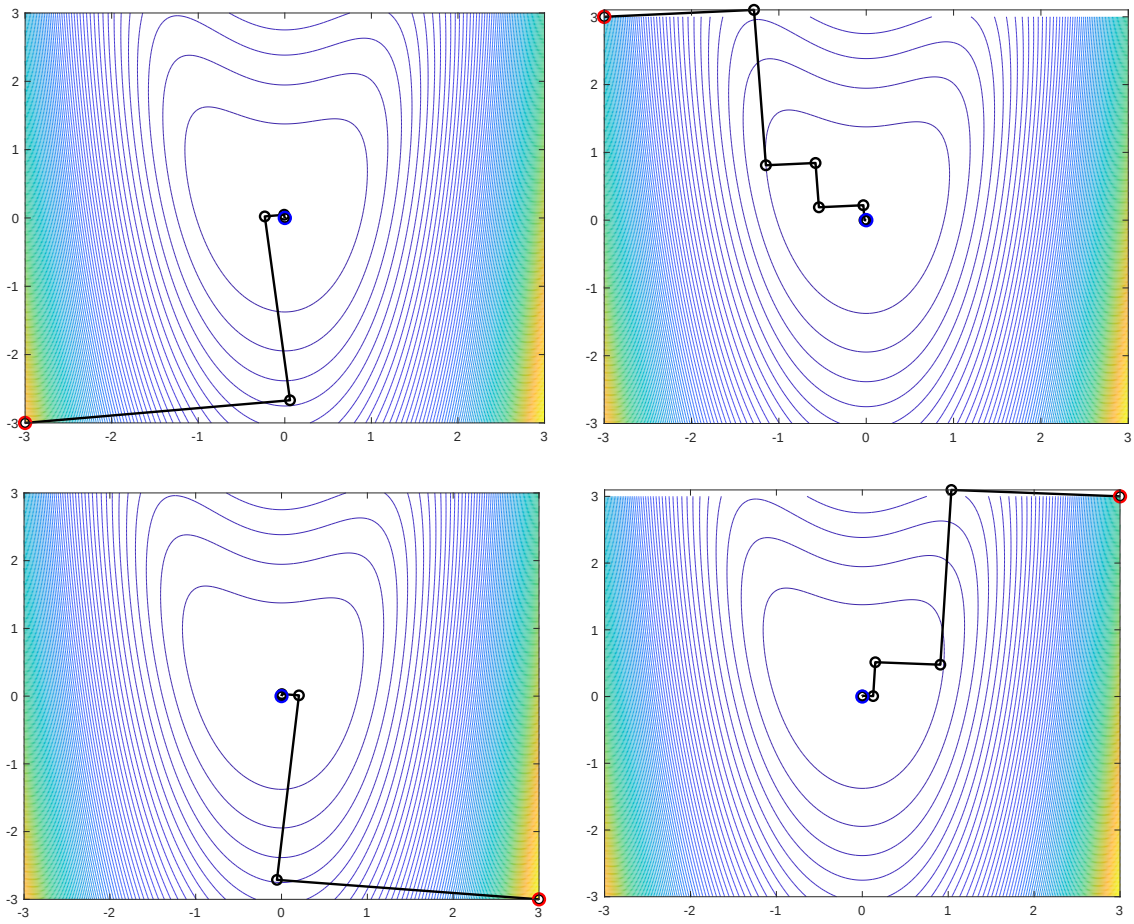

Figure 3: Problem 2: Solution path using exact line search (x axis stands for x_1 and y axis stands for x_2)

its performance with the tested approaches in **Problem 2** using the same initial points and stopping tolerance as in **Problem 2** b). You can use the following parameters:

- $(\sigma, \gamma) = (0.5, 0.1)$ – parameter for backtracking.
- $\epsilon = 10^{-6}$ – scaling parameter in the update (2).
- $m = 25$ – memory parameter.

Generate a plot of the different solution paths of Adagrad using the different initial points from $\mathcal{X}^0$ and add a contour plot of f in the background.

c) How does Adagrad behave when different memories m are chosen? (Suitable other choices might include $m \in \{5, 10, 15, 25, ..., \texttt{maxit}\}$)?

Solution :

a) Notice that we have $v_i^k \geq \sqrt{\epsilon} > 0$ for all i and k. Hence, this yields

$$\nabla f(x^k)^\top d^k = -\nabla f(x^k)^\top D_k^{-1} \nabla f(x^k) = -\sum_{i=1}^n (\nabla f(x^k)_i)^2 / v_i^k < 0$$

	Backtracking: $(0.1, 0.5)$			Exact Step Sizes: $(100, 10^{-6}, 2)$		
Initial Point	Iter.	Obj. Value	Comment	Iter.	Obj. Value	Comment
$(-3,-3)^\top$	15	$9.5248 \cdot 10^{-12}$	Conv. to x^*	7	$6.1935 \cdot 10^{-12}$	Conv. to x^*
$(3,-3)^\top$	16	$2.3606 \cdot 10^{-12}$	Conv. to x^*	7	$7.8622 \cdot 10^{-13}$	Conv. to x^*
$(-3,3)^\top$	16	$3.9693 \cdot 10^{-12}$	Conv. to x^*	10	$4.0554 \cdot 10^{-14}$	Conv. to x^*
$(3,3)^\top$	15	$1.7034 \cdot 10^{-11}$	Conv. to x^*	8	$1.0555 \cdot 10^{-11}$	Conv. to x^*
Average	15.5			8		

Table 2: Comparison of the gradient method using different step sizes strategies for $\mathcal{X}^0$.

b) An exemplary code for Adagrad is shown in Listing 5. The solution paths for $m = 25$ and $\mathcal{X}^0$ are presented in Figure 4 (a).

c) Table 3 contains details. The memory parameter $m = 5$ produces the best results.

	$m = 5$		$m = 10$		$m = 15$		$m = 25$	
Ini. Point	Iter.	Obj. Value	Iter.	Obj. Value	Iter.	Obj. Value	Iter.	Obj. Value
$(-3,-3)$	27	$3.2 \cdot 10^{-12}$	37	$3.6 \cdot 10^{-12}$	51	$7.5 \cdot 10^{-12}$	79	$1.1 \cdot 10^{-11}$
$(3,-3)$	27	$1.5 \cdot 10^{-11}$	40	$1.1 \cdot 10^{-12}$	54	$5.5 \cdot 10^{-12}$	66	$7.6 \cdot 10^{-12}$
$(-3,3)$	25	$4.1 \cdot 10^{-12}$	33	$1.4 \cdot 10^{-11}$	47	$4.6 \cdot 10^{-12}$	78	$1.5 \cdot 10^{-11}$
$(3,3)$	25	$1.7 \cdot 10^{-12}$	39	$2.6 \cdot 10^{-13}$	42	$1.4 \cdot 10^{-12}$	81	$3.2 \cdot 10^{-12}$
Average	26		37.25		48.5		76	

Table 3: Comparison of Adagrad using different memory parameters m for $\mathcal{X}^0$.

Problem 4 (Globalized Newton's Method): (approx. *25* points)
Implement the globalized Newton method with backtracking that was presented in the lecture as a function `newton_glob` in `MATLAB` or `Python`.

The pseudo-code for the full Newton method is given as follows:

Algorithm 1: The Globalized Newton Method

Initialization: Select an initial point $x^0 \in \mathbb{R}^n$ and parameter $\gamma, \gamma_1, \gamma_2, \sigma \in (0,1)$ and `tol`.
for $k = 0, 1, \ldots$ **do**
  If $\|\nabla f(x^k)\| \leq$ `tol`, then STOP and x^k is the output.
  Compute the Newton direction s^k as solution of the linear system of equations:

$$\nabla^2 f(x^k) s^k = -\nabla f(x^k).$$

  If $-\nabla f(x^k)^\top s^k \geq \gamma_1 \min\{1, \|s^k\|^{\gamma_2}\} \|s^k\|^2$, then accept the Newton direction and set $d^k = s^k$. Otherwise set $d^k = -\nabla f(x^k)$.
  Choose a step size α_k by backtracking and calculate $x^{k+1} = x^k + \alpha_k d^k$.

The following input functions and parameters should be considered:

- `obj`, `grad`, `hess` – function handles that calculate and return the objective function $f(x)$, the gradient $\nabla f(x)$, and the Hessian $\nabla^2 f(x)$ at an input vector $x \in \mathbb{R}^n$. You can treat these handles as functions or fields of a class or structure `f` or you can use f, ∇f, and $\nabla^2 f$ from part a) and b) directly in the algorithm. (For example, your function can have the form `newton_glob(obj,grad,hess,...)`).

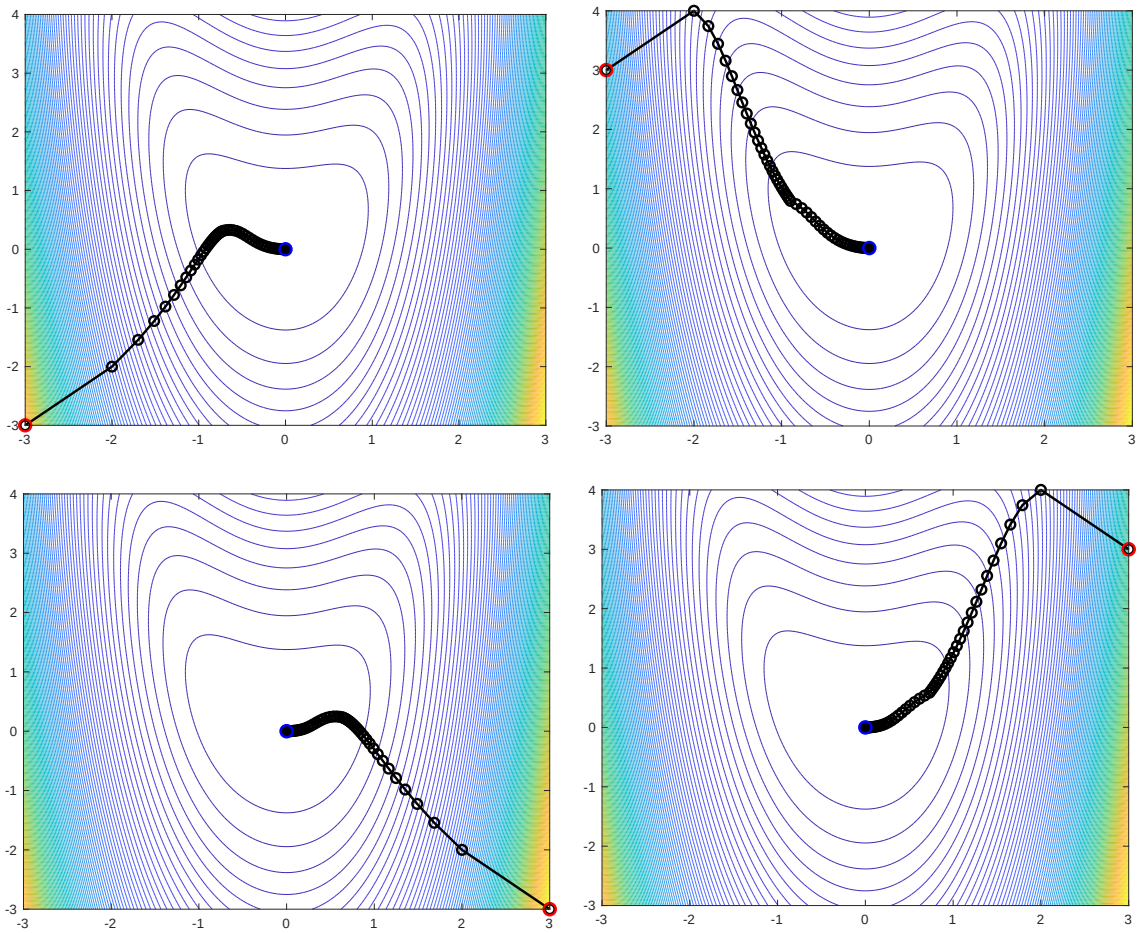

Figure 4: Problem 3: Solution path using Adagrad (x axis stands for x_1 and y axis stands for x_2)

- x^0 – the initial point.
- `tol` – a tolerance parameter. The method should stop whenever the current iterate x^k satisfies the criterion $\|\nabla f(x^k)\| \leq$ `tol`.
- $\gamma_1, \gamma_2 > 0$ – parameters for the Newton condition.
- $\sigma, \gamma \in (0, 1)$ – parameters for backtracking and the Armijo condition.

You can again organize the latter parameters in an appropriate `options` class or structure. You can use the backslash operator `A\b` in `MATLAB` or `numpy.linalg.solve(A,b)` to solve the linear system of equations $Ax = b$. If the computed Newton step $s^k = -\nabla^2 f(x^k)^{-1} \nabla f(x^k)$ is a descent direction and satisfies

$$-\nabla f(x^k)^\top s^k \geq \gamma_1 \min\{1, \|s^k\|^{\gamma_2}\} \|s^k\|^2,$$

we accept it as next direction d^k. Otherwise, the gradient direction $d^k = -\nabla f(x^k)$ is chosen. The method should return the final iterate x^k that satisfies the stopping criterion.

a) Test your approach on the Rosenbrock function $f : \mathbb{R}^2 \to \mathbb{R}$

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

with initial point $x^0 = (-1, -0.5)^\top$ and parameter $(\sigma, \gamma) = (0.5, 10^{-4})$ and $(\gamma_1, \gamma_2) = (10^{-6}, 0.1)$. (Notice that γ is smaller here). Besides the globalized Newton method also run

Globalized Newton Method					
	tol: 10^{-5}		tol: 10^{-8}		
Initial Point	Iter.	Obj. Value	Iter.	Obj. Value	Comment
$(-3,-3)^\top$	10	$6.864 \cdot 10^{-18}$	10	$6.864 \cdot 10^{-18}$	Conv. to x^*
$(3,-3)^\top$	10	$7.760 \cdot 10^{-16}$	11	$7.366 \cdot 10^{-25}$	Conv. to x^*
$(-3,3)^\top$	10	$8.316 \cdot 10^{-20}$	10	$8.316 \cdot 10^{-20}$	Conv. to x^*
$(3,3)^\top$	10	$3.458 \cdot 10^{-17}$	11	$3.242 \cdot 10^{-33}$	Conv. to x^*
Average	10		10.5		

Table 4: Comparison of the globalized Newton method using different tolerances for $\mathcal{X}^0$.

the gradient method with backtracking ($(\sigma, \gamma) = (0.5, 10^{-4})$) on this problem and compare the performance of the two approaches using the tolerance $\texttt{tol} = 10^{-7}$.

Does the Newton method always utilize the Newton direction? Does the method always use full step sizes $\alpha_k = 1$?

b) Repeat the performance test described in **Problem 2** b) and **Problem 3** b) for problem (1) using the globalized Newton method. You can use the parameter $(\sigma, \gamma) = (0.5, 0.1)$, $(\gamma_1, \gamma_2) = (10^{-6}, 0.1)$, and $\texttt{tol} = 10^{-5}$.

Plot all of the solution paths obtained by Newton's method for the different initial points in $\mathcal{X}^0$ in one figure (with a contour plot of f in the background).

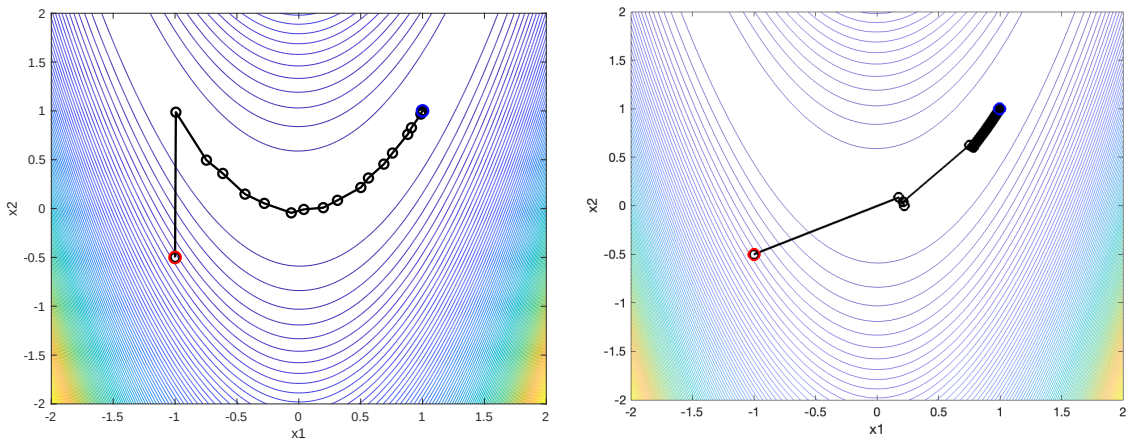

Figure 5: Problem 4: Solution path using Newton method and gradient method (The left one is the Newton method, the right one is the gradient method)

Solution :

a) The solution paths of the Newton method and gradient method are summarized and shown in Figure 5 . The corresponding `MATLAB` code can be found in Listing 6 and 7.

The Newton method requires 21 iterations (0.008 seconds) to recover a solution x^k satisfying $\|\nabla f(x^k)\| \leq 10^{-7}$. The gradient method (with Armijo linesearch) requires 17334 iterations (0.288 seconds) to reach a solution with similar accuracy. Hence, the globalized Newton method performs much more efficiently on this example. The Newton method always

performs Newton steps with step sizes ranging from $\frac{1}{8}$, $\frac{1}{2}$, to 1. During the last steps, we always select the full step size $\alpha_k = 1$.

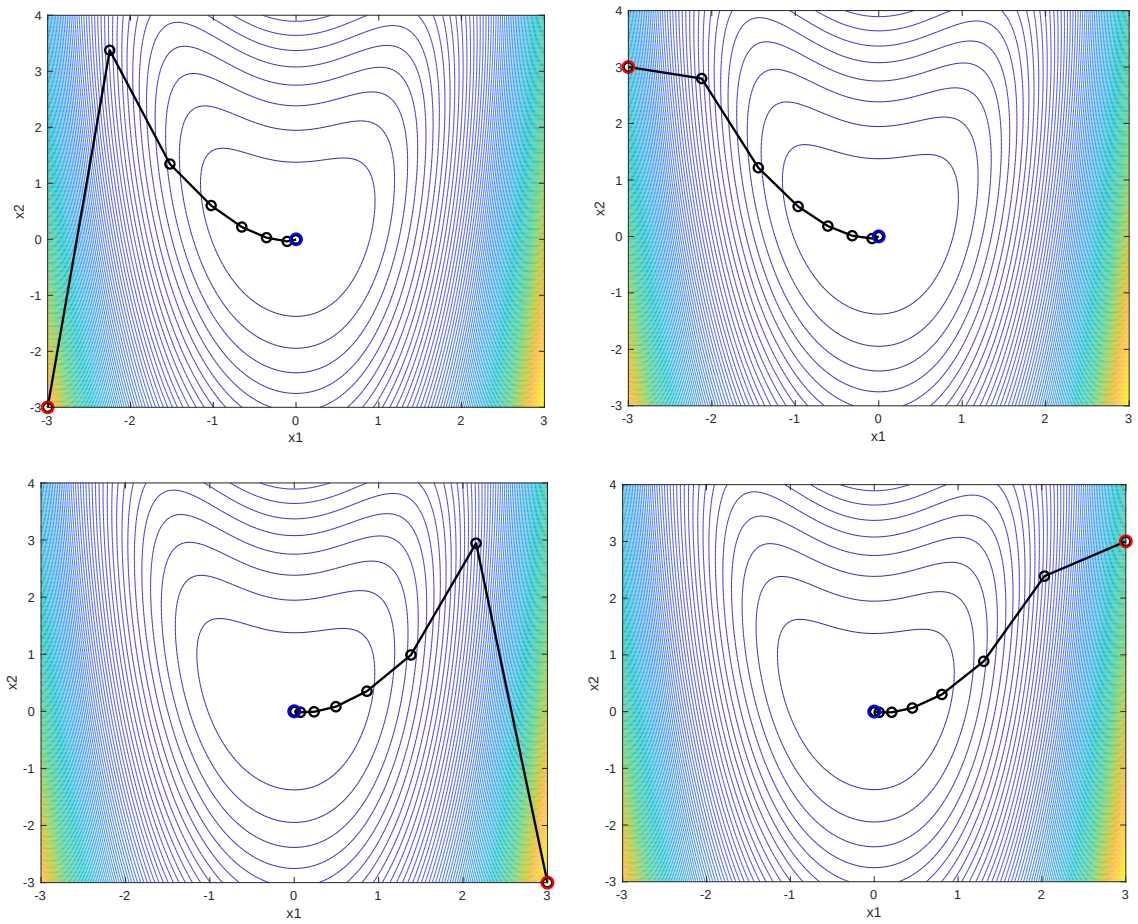

Figure 6: Problem 4: Solution path using Newton method

b) Results are summarized in Figure 6 and Table 4. Clearly, Newton's method again requires the least number of iterations to show convergence.

Listing 1: **Problem 1** – MATLAB code: Bisection method

```
function [x,gx] = bisection(g,xl,xr,options)

% Compute intial function values
gr  = g(xr); gl  = g(xl); sl = sign(gl);

if gl*gr > 0
    fprintf(1,'The input data not suitable!');
    x = []; gx = [];
    return
end

if options.Display
    fprintf(1,'\n— — — bisection algorithm; \n— — — [tol = %1.2e/ maxit = %4i]\n',options.
        TolX,options.MaxIter);
    fprintf(1,'ITER ; X ; |G(X)| ; |XR—XL|\n');
end

for i = 1:options.MaxIter
    xm  = (xl + xr)/2;
    gm  = g(xm);

    if options.Display
        fprintf(1,'[%4i] ; %1.8e ; %1.2e ; %1.2e \n',i,xm,abs(gm),abs(xl—xr));
    end

    if abs(xl—xr) < options.TolX % || abs(gm) < options.tol
        x = xm; gx = gm;
        return
    end

    if gm > 0
        if sl < 0, xr = xm; else, xl = xm; end
    else
        if sl < 0, xl = xm; else, xr = xm; end
    end
end

% g = @(x) x^3—2;
% options = optimset('Display','on', 'TolX',1e—5, 'MaxIter',100);
% [x,gx] = bisection(g,0,2,options)


% the plot generating f
% x = linspace(0,2,200);
% f = @(x) (x.^2—1).^2/4 + (x—2).^2/2;
% y = f(x);
% plot(x,y);hold on
% grid on;
% plot(1.26, 0.36012, 'bo', 'MarkerSize', 10, 'LineWidth', 2);
```

Listing 2: **Problem 1** – MATLAB code: Golden section method

```
function x = ausection(f,xl,xr,options)

% Compute intial function values
phi = (3—sqrt(5))/2;
```

```
5
6   xln = phi*xr + (1-phi)*xl;
7   xrn = (1-phi)*xr + phi*xl;
8
9   fln = f(xln);
10  frn = f(xrn);
11
12  if options.Display
13      fprintf(1,'\n- - - golden section algorithm; \n- - - [tol = %1.2e/ maxit = %4i]\n',
            options.TolX,options.MaxIter);
14      fprintf(1,'ITER ; X ; F(X) ; |XR-XL|\n');
15  end
16
17  for i = 1:options.MaxIter
18
19      if fln < frn
20          xr  = xrn; xrn = xln;
21          xln = phi*xr + (1-phi)*xl;
22          frn = fln; fln = f(xln);
23      else
24          xl  = xln; xln = xrn;
25          xrn = (1-phi)*xr + phi*xl;
26          fln = frn; frn = f(xrn);
27      end
28
29      if options.Display
30          fprintf(1,'[%4i] ; %1.8e ; %1.2e ; %1.2e \n',i,(xl+xr)/2,f((xl+xr)/2),abs(xl-xr));
31      end
32
33      if abs(xl-xr) < options.TolX
34          x = (xl+xr)/2;
35          return
36      end
37  end
38
39  % f = @(x) (x^2-1)^2/4 + (x-2)^2/2;
40  % options = optimset('Display','on', 'TolX',1e-5, 'MaxIter',100);
41  % x = ausection(f,0,2,options)
```

Listing 3: **Problem 2** – MATLAB code: Gradient method

```
function [x,out] = gradient_method(f,x0,tol,opts)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OPTIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic;

if ~isfield(opts,'maxit')
    opts.maxit  = 10000;
end

if ~isfield(opts,'mode')
    opts.mode       = 'fixed_step_size';
    opts.alpha      = 0.1;
elseif strcmp(opts.mode,'fixed_step_size')
    if ~isfield(opts,'alpha')
        opts.alpha  = 0.1;
    elseif ~isscalar(opts.alpha) || ~isreal(opts.alpha) || opts.alpha <= 0 || opts.alpha > Inf
        error('step size alpha must be in (0,Inf)!');
    end
elseif strcmp(opts.mode,'armijo_linesearch')
    if ~isfield(opts,'gamma')
        opts.gamma  = 0.1;
    elseif ~isscalar(opts.gamma) || ~isreal(opts.gamma) || opts.gamma <= 0 || opts.gamma >= 1
        error('parameter gamma must be in (0,1)!');
    end
    if ~isfield(opts,'s')
        opts.s  = 1;
    elseif ~isscalar(opts.s) || ~isreal(opts.s) || opts.s <= 0 || opts.s > Inf
        error('parameter s must be in (0,Inf)!');
    end
elseif strcmp(opts.mode,'exact_linesearch')
    if ~isfield(opts,'opts_au')
        opts_au.Display = false;
        opts_au.MaxIter = 100;
        opts_au.TolX    = 1e-6;
        opts_au.s       = 2;
    end
end

x           = x0;

if strcmp(opts.mode,'fixed_step_size')
    alpha   = opts.alpha;
else
    alpha   = 0;
end

% prepare trace in output
if opts.trace
    [trace.res, trace.time]  = deal(zeros(opts.maxit,1));
    if length(x) == 2
```

```
        trace.x              = zeros(opts.maxit,2);
    end
end

if opts.print
    fprintf(1,'--- gradient method with %s; n = %g\n',opts.mode,length(x));
    fprintf(1,'ITER ; OBJ.VAL ; G.NORM ; STEP.SIZE\n');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MAIN LOOP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for iter = 1:opts.maxit

    %-------------------------------------------------------------------------
    % calculate gradient
    %-------------------------------------------------------------------------
    g   = f.grad(x);
    ng  = norm(g);

    if opts.print
        obj_val   = f.obj(x);
        fprintf(1,'[%5i] ; %1.6f ; %1.4e ; %1.2f\n',iter,obj_val,ng,alpha);
    end

    % save information for graphic output
    if opts.trace
        trace.res(iter)     = ng;
        trace.time(iter)    = toc;
        if length(x) == 2
            trace.x(iter,:) = x';
        end
    end

    %-------------------------------------------------------------------------
    % stopping criterion
    %-------------------------------------------------------------------------

    if ng <= tol
        break
    end

    %-------------------------------------------------------------------------
    % step size and main update
    %-------------------------------------------------------------------------
    switch opts.mode
        case 'fixed_step_size'
            x           = x - opts.alpha*g;
        case 'armijo_linesearch'
            if iter == 1
                f_old   = f.obj(x);
            end
            alpha       = opts.s;
            x_old       = x;
            x           = x_old - alpha*g;
```

```
            f_new       = f.obj(x);
            a_counter   = 1;

            while f_new - f_old > - alpha*opts.gamma*ng^2 && a_counter <= 100
                alpha       = alpha/2;
                x           = x_old - alpha*g;
                f_new       = f.obj(x);
                a_counter   = a_counter + 1;
            end

            f_old       = f_new;
        case 'exact_linesearch'
            x_old       = x;
            phi         = @(alpha) f.obj(x_old - alpha*g);

            alpha       = ausection(phi,0,opts_au.s,opts_au);

            x           = x_old - alpha*g;
        case 'diminishing'
            alp     = opts.alpha(iter);
            x       = x - alp*g;

    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GENERATE OUTPUT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

out.time            = toc;
out.iter            = iter;

if opts.trace
    trace.res       = trace.res(1:iter);
    trace.time      = trace.time(1:iter);
    if length(x) == 2
        trace.x     = trace.x(1:iter,:);
    end
    out.trace       = trace;
    out.x           = x;
end

end


%%
% clear all;clc
% f.obj = @(x) 2*x(1)^4+2/3*x(1)^3+x(1)^2-2*x(1)^2*x(2)+4/3*x(2)^2;
% f.grad = @(x) [8*x(1)^3 + 2*x(1)^2 + 2*x(1)-4*x(1)*x(2);-2*x(1)^2 + 8/3*x(2)];
% opts = struct('maxit',1000,'mode','fixed_step_size','trace',true,'print',true);
% opts = struct('maxit',100,'mode','armijo_linesearch','trace',true,'print',true);
% opts = struct('maxit',100,'mode','exact_linesearch','trace',true,'print',true);
% [x,out] = gradient_method(f,[-3;-3],1e-5,opts);


% plot
```

```
% h = @(x) 2*x(:,1).^4 + 2/3*x(:,1).^3 + x(:,1).^2 - 2*x(:,1).^2.*x(:,2) + 4/3*x(:,2).^2;
% x1 = linspace(-3,3,300);
% x2 = linspace(-3,3,300);
% [X1,X2] = meshgrid(x1,x2);
% Z = h([X1(:), X2(:)]);
% Z = reshape(Z, size(X1));
% figure
% contour(X1,X2,Z,100);hold on
% plot(out.trace.x(:,1),out.trace.x(:,2), 'ko-', 'MarkerSize', 8, 'LineWidth', 2);
% % Emphasize the initial point (first point)
% plot(out.trace.x(1, 1), out.trace.x(1, 2), 'ro', 'MarkerSize', 10, 'LineWidth', 2);
%
% % Emphasize the final point (last point)
% plot(out.trace.x(end, 1), out.trace.x(end, 2), 'bo', 'MarkerSize', 10, 'LineWidth', 2);
```

Listing 4: **Problem 3** – MATLAB code: Adagrad

```
function [x,out] = adagrad(f,x0,tol,opts)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OPTIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic;

if ~isfield(opts,'maxit')
    maxit       = 1000;
else
    maxit       = opts.maxit;
end

if ~isfield(opts,'mode')
    opts.mode       = 'fixed_step_size';
    opts.alpha      = 0.1;
elseif strcmp(opts.mode,'fixed_step_size')
    if ~isfield(opts,'alpha')
        opts.alpha  = 0.1;
    elseif ~isscalar(opts.alpha) || ~isreal(opts.alpha) || opts.alpha <= 0 || opts.alpha >
         Inf
        error('step size sigma must be in (0,Inf)!');
    end
elseif strcmp(opts.mode,'armijo_linesearch')
    gamma           = opts.gamma;
end

x   = x0;

% prepare trace in output
if opts.trace
    [trace.res, trace.time]  = deal(zeros(maxit,1));
    if length(x) == 2
        trace.x             = zeros(maxit,2);
    end
end

save_g  = zeros(length(x),opts.m);
sum_g   = zeros(length(x),1);
counter = 1;

if opts.print
   fprintf(1,'- - - adagrad; step size: %s; n = %g; \n',opts.mode,length(x));
   fprintf(1,'ITER ; OBJ.VAL ; G.NORM ; STEP.SIZE ; COU \n');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MAIN LOOP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for iter = 1:opts.maxit

    %------------------------------------------------------------------
    % calculate gradient
```

```
    %-------------------------------------------------------------------------
    g   = f.grad(x);
    ng  = norm(g);

    if iter <= opts.m
        save_g(:,counter)   = g;
        sum_g               = sum_g + g.^2;
        counter             = mod(counter,opts.m)+1;
    else
        sum_g               = sum_g - save_g(:,counter).^2 + g.^2;
        save_g(:,counter)   = g;
        counter             = mod(counter,opts.m)+1;
    end

    d  = g./sqrt(opts.eps + sum_g);

    % save information for graphic output
    if opts.trace
        trace.res(iter)     = norm(g);
        trace.time(iter)    = toc;
        if length(x) == 2
            trace.x(iter,:) = x';
        end
    end

    if opts.print
        if iter == 1
            fprintf(1,'[%4i] ; %1.6f ; %1.4e ; %1.2e ; %g \n',iter,f.obj(x),ng,0,counter);
        else
            fprintf(1,'[%4i] ; %1.6f ; %1.4e ; %1.2e ; %g \n',iter,f.obj(x),ng,alp,counter
                );
        end
    end

    %-------------------------------------------------------------------------
    % stopping criterion
    %-------------------------------------------------------------------------
    if ng <= tol
        break
    end

    %-------------------------------------------------------------------------
    % calculate new iterate
    %-------------------------------------------------------------------------
    switch opts.mode
        case 'fixed_step_size'
            alp     = opts.alpha;
            x       = x - alp*d;
        case 'armijo_linesearch'
            if iter == 1
                f_old   = f.obj(x);
            end
            alp         = opts.s;
            x_old       = x;
            x           = x_old - alp*d;
            f_new       = f.obj(x);
```

```
            gtd         = g'*d;
            a_counter   = 1;

            while f_new - f_old > - alp*gamma*gtd && a_counter <= 100
                alp         = opts.sigma*alp;
                x           = x_old - alp*d;
                f_new       = f.obj(x);
                a_counter   = a_counter + 1;
            end

            f_old       = f_new;
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GENERATE OUTPUT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

out.time            = toc;
out.iter            = iter;

if opts.trace
    trace.res       = trace.res(1:iter);
    trace.time      = trace.time(1:iter);
    if length(x) == 2
        trace.x     = trace.x(1:iter,:);
    end
    out.trace       = trace;
end

end


%%
% f.obj = @(x) 2*x(1)^4+2/3*x(1)^3+x(1)^2-2*x(1)^2*x(2)+4/3*x(2)^2;
% f.grad = @(x) [8*x(1)^3 + 2*x(1)^2 + 2*x(1)-4*x(1)*x(2);-2*x(1)^2 + 8/3*x(2)];
% opts = struct('maxit',1000,'mode','armijo-linesearch','trace',true,'print',true,'m',25,'
    eps',1e-6,'gamma',0.1,'sigma',0.5,'s',1);
% [x,out] = adagrad(f,[-3;-3],1e-5,opts);

% plot
% h = @(x) 2*x(:,1).^4 + 2/3*x(:,1).^3 + x(:,1).^2 - 2*x(:,1).^2.*x(:,2) + 4/3*x(:,2).^2;
% x1 = linspace(-3,3,300);
% x2 = linspace(-3,4,300);
% [X1,X2] = meshgrid(x1,x2);
% Z = h([X1(:), X2(:)]);
% Z = reshape(Z, size(X1));
% figure
% contour(X1,X2,Z,100);hold on
% plot(out.trace.x(:,1),out.trace.x(:,2), 'ko-', 'MarkerSize', 8, 'LineWidth', 2);
% % Emphasize the initial point (first point)
% plot(out.trace.x(1, 1), out.trace.x(1, 2), 'ro', 'MarkerSize', 10, 'LineWidth', 2);
%
% % Emphasize the final point (last point)
% plot(out.trace.x(end, 1), out.trace.x(end, 2), 'bo', 'MarkerSize', 10, 'LineWidth', 2);
```

Listing 5: **Problem 4** – MATLAB code: Newton's method

```
function [x,out] = newton_glob(f,x0,tol,opts)

% === INPUT ==========
% f      a structure for the objective function
%   .obj(x)    returns the function value at x
%   .grad(x)   returns the gradient of f at x
%   .hess(x)   returns the hessian of f at x
% x0     initial point
% tol    tolerance parameter
% opts   a structure with options
% === OUTPUT =========
% x      a potential stationary point of min_x f(x)

tic;

x      = x0;
f_old  = f.obj(x);
type   = 'N';
alpha  = -1;

if opts.print
    fprintf(1,'--- globalized newton method; n = %g\n',length(x));
    fprintf(1,'ITER ; OBJ.VAL ; G.NORM ; ALPHA ; TYPE \n');
end

% prepare trace in output
if opts.trace
    [trace.res, trace.time]  = deal(zeros(opts.maxit,1));
    if length(x) == 2
        trace.x              = zeros(opts.maxit,2);
    end
end

% main loop
for  iter = 1:opts.maxit
    x_old   = x;
    g       = f.grad(x);
    ng      = norm(g);

    if opts.print
        fprintf(1,'[%4i] ; %2.6f ; %1.4e ; %1.3f ; %s\n',iter,f_old,ng,alpha,type);
    end

    % save information for graphic output
    if opts.trace
        trace.res(iter)     = ng;
        trace.time(iter)    = toc;
        if length(x) == 2
            trace.x(iter,:) = x';
        end
    end

    if ng <= tol
       break;
    end
```

```
56
57      d       = - f.hess(x)\g;
58
59      gtd     = d'*g;
60      nd      = norm(d);
61
62      if - gtd < 1e-6*min(1,nd^0.1)*nd^2
63          d       = -g;
64          gtd     = -ng^2;
65          type    = 'G';
66      else
67          type    = 'N';
68      end
69
70      alpha   = 1;
71      x       = x_old + alpha*d;
72      f_new   = f.obj(x);
73      acount  = 1;
74
75      while f_new - f_old > alpha*opts.gamma*gtd && acount <= 100
76          alpha   = alpha/2;
77          x       = x_old + alpha*d;
78          f_new   = f.obj(x);
79          acount  = acount + 1;
80      end
81
82      f_old   = f_new;
83  end
84
85  out.time            = toc;
86  out.iter            = iter;
87
88  if opts.trace
89      trace.res       = trace.res(1:iter);
90      trace.time      = trace.time(1:iter);
91      if length(x) == 2
92          trace.x     = trace.x(1:iter,:);
93      end
94      out.trace       = trace;
95  end
96
97  %%
98  % f.obj = @(x) 2*x(1)^4+2/3*x(1)^3+x(1)^2-2*x(1)^2*x(2)+4/3*x(2)^2;
99  % f.grad = @(x) [8*x(1)^3 + 2*x(1)^2 + 2*x(1)-4*x(1)*x(2);-2*x(1)^2 + 8/3*x(2)];
100 % f.hess = @(x) [24*x(1)^2 + 4*x(1) + 2 - 4*x(2), -4*x(1); -4*x(1), 8/3];
101 % % f.obj = @(x) 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
102 % % f.grad = @(x) [400*x(1)^3-400*x(1)*x(2) + 2*x(1)-2;200*(x(2) - x(1)^2)];
103 % % f.hess = @(x) [1200*x(1)^2 - 400*x(2) + 2,-400*x(1);-400*x(1),200];
104 % opts = struct('maxit',100,'trace',true,'print',true,'gamma',0.1);
105 % [x,out] = newton_glob(f,[-3;-3],1e-8,opts);
106
107 % plot
108 % h = @(x) 2*x(:,1).^4 + 2/3*x(:,1).^3 + x(:,1).^2 - 2*x(:,1).^2.*x(:,2) + 4/3*x(:,2).^2;
109 % % h = @(x) 100*(x(:,2)-x(:,1).^2).^2 + (1-x(:,1)).^2;
110 % x1 = linspace(-3,3,300);
111 % x2 = linspace(-3,4,300);
```

```
% [X1,X2] = meshgrid(x1,x2);
% Z = h([X1(:), X2(:)]);
% Z = reshape(Z, size(X1));
% figure
% contour(X1,X2,Z,100);hold on
% plot(out.trace.x(:,1),out.trace.x(:,2), 'ko-', 'MarkerSize', 8, 'LineWidth', 2);
% % Emphasize the initial point (first point)
% plot(out.trace.x(1, 1), out.trace.x(1, 2), 'ro', 'MarkerSize', 10, 'LineWidth', 2);
%
% % Emphasize the final point (last point)
% plot(out.trace.x(end, 1), out.trace.x(end, 2), 'bo', 'MarkerSize', 10, 'LineWidth', 2);
% xlabel('x1')
% ylabel('x2')
```