MAT 3007 — Optimization

Solutions 8

Problem 1 (A One-Dimensional Problem): (approx. *20* points)
Consider the minimization problem

$$\min_{x\in\mathbb{R}} \ f(x) \qquad \text{s.t.} \qquad x \in [0,1],$$

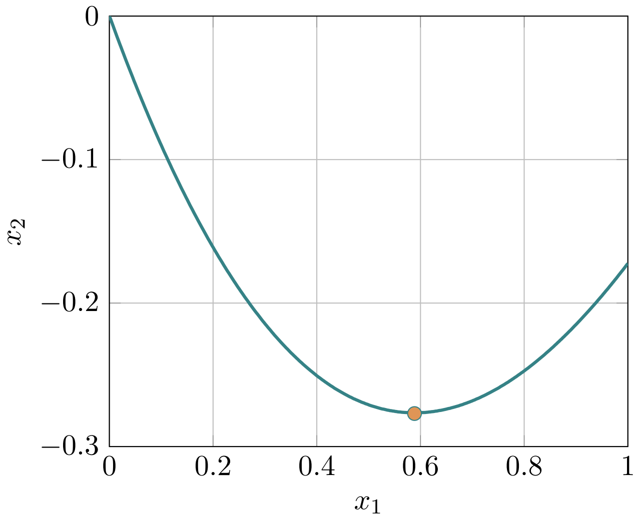
where f is given by $f(x) := e^{-x} - \cos(x)$. Solve this problem using the bisection **or** the golden section method. Compare the number of iterations required to recover a solution in $[0,1]$ with accuracy less or equal than 10^{-5}.

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**:

```
% demo_p1

% options
options.maxit   = 100;
options.tol     = 1e-5;
options.display = true;

% functions
f               = @(x) exp(-x)-cos(x);
g               = @(x) -exp(-x)+sin(x);

xl              = 0;
xr              = 1;

[xa]            = ausection(f,xl,xr,options);
[xb,~]          = bisection(g,xl,xr,options);
```

The golden section method returns the solution $x^*_{\mathtt{g}} = 5.88530599 \cdot 10^{-1}$ after 24 iterations. The bisection method requires 18 iterations and it returns $x^*_{\mathtt{b}} = 5.88535309 \cdot 10^{-1}$. A plot of the function f and the solution can be found below.



Problem 2 (Descent Directions): (approx. *20* points)
Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function and consider $x \in \mathbb{R}^n$ with $\nabla f(x) \neq 0$. Verify the following statements:

a) Set $d = -(\nabla f(x)_j) \cdot e_j = -\frac{\partial f}{\partial x_j}(x) \cdot e_j$, where $e_j \in \mathbb{R}^n$ is the j-th unit vector and $j \in \{1, \dots, n\}$ is an index satisfying

$$\left| \frac{\partial f}{\partial x_j}(x) \right| = \max_{1 \le i \le n} \left| \frac{\partial f}{\partial x_i}(x) \right| = \|\nabla f(x)\|_\infty.$$

Then, d is a descent direction of f at x.

b) The direction $d = -\nabla f(x)/\|\nabla f(x)\|$ is a descent direction of f at x.

c) Let f be twice continuously differentiable and define $d_i = -(\nabla f(x)_i)/\max\{\nabla^2 f(x)_{ii}, \varepsilon\}$ for all $i \in \{1, \dots, n\}$ and for some $\varepsilon > 0$. Then, d is well-defined (we do not divide by zero) and it is a descent direction of f at x.

Solution :

a) We calculate:

$$\nabla f(x)^\top d = -\frac{\partial f}{\partial x_j}(x) \cdot \nabla f(x)^\top e_j = -\left[\frac{\partial f}{\partial x_j}(x) \right]^2 = -\|\nabla f(x)\|_\infty^2.$$

Due to $\nabla f(x) \neq 0$, we have $\|\nabla f(x)\|_\infty \neq 0$ which implies $\nabla f(x)^\top d < 0$. Hence, d is a descent direction of f at x.

b) We have $\nabla f(x)^\top d = -\|\nabla f(x)\| < 0$. Hence, d is a descent direction.

c) We have $\max\{\nabla^2 f(x)_{ii}, \varepsilon\} \ge \varepsilon > 0$. Hence, we do not divide by zero and d is well-defined. We further have

$$\nabla f(x)^\top d = -\sum_{i=1}^n \frac{(\nabla f(x)_i)^2}{\max\{\nabla^2 f(x)_{ii}, \varepsilon\}} < 0.$$

Here, the last estimate follows from the fact $\max\{\nabla^2 f(x)_{ii}, \varepsilon\} > 0$ for all i and $\nabla f(x) \neq 0$. Thus, d is a descent direction of f at x.

Problem 3 (The Gradient Method): (approx. *35* points)

In this exercise, we want to solve the optimization problem

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

via the gradient descent method. (This is problem 1 discussed in sheet 6).

Implement the gradient method that was presented in the lectures 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 $\|\nabla f(x^k)\| \leq \texttt{tol}$.

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), `tol` (stopping tolerance), $[0, \texttt{a}]$ (the interval of the step size).

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 ($\texttt{maxit} = 100$, $\texttt{tol} = 10^{-6}$, $\texttt{a} = 2$) 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 $x^0 = (3, -3)^\top$ and report the performance of the methods, i.e., compare the number of iterations and the point to which the different gradient methods converged.

b) Let us define the set of ten initial points

$$\mathcal{X}^0 := \left\{ \begin{pmatrix} -4 \\ -4 \end{pmatrix}, \begin{pmatrix} -4 \\ 0 \end{pmatrix}, \begin{pmatrix} -4 \\ 4 \end{pmatrix}, \begin{pmatrix} -2 \\ -4 \end{pmatrix}, \begin{pmatrix} -2 \\ 4 \end{pmatrix}, \begin{pmatrix} 2 \\ -4 \end{pmatrix}, \begin{pmatrix} 2 \\ 4 \end{pmatrix}, \begin{pmatrix} 4 \\ -4 \end{pmatrix}, \begin{pmatrix} 4 \\ 0 \end{pmatrix}, \begin{pmatrix} 4 \\ 4 \end{pmatrix} \right\}.$$

   Run the methods:

   – Gradient descent method with backtracking and $(\sigma, \gamma) = (0.5, 0.1)$,

   – Gradient method with exact line search and $\texttt{maxit} = 100$, $\texttt{tol} = 10^{-6}$, $\texttt{a} = 2$,

   – General descent method using the direction d from **Problem 2 a)** as descent direction (you can either use backtracking $(\sigma, \gamma) = (0.5, 0.1)$ or exact line search ($\texttt{maxit} = 100$, $\texttt{tol} = 10^{-6}$, $\texttt{a} = 2$) to determine the step sizes) 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) As shown in **Problem 1** on **Sheet 6**, the function f has two stationary points $x^* = (0,0)^\top$ and $y^* = (-2,1)^\top$. Here, x^* is a (degenerate) saddle point while y^* is a strict local minimum of f.

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
$x^0 = (3,-3)^\top$	37	-4.000000	$[-2.0000; 1.0000]$	Conv. to y^*

Exact Step Sizes: $\texttt{maxit} = 100$, $\texttt{tol} = 10^{-6}$, $\texttt{a} = 2$				
Initial Point	Iter.	Obj. Value	Final Iterate	Comment
$x^0 = (3,-3)^\top$	15	-4.000000	$[-2.0000; 1.0000]$	Conv. to y^*

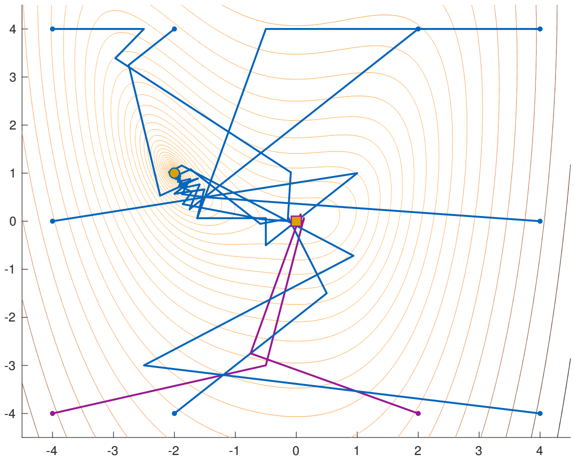
Table 1: Comparison of the gradient method using different step sizes strategies.

In this example, the two gradient methods converge very quickly to the solution y^* requiring only 15 and 37 iterations. The gradient method with exact line search is converges slightly quicker.

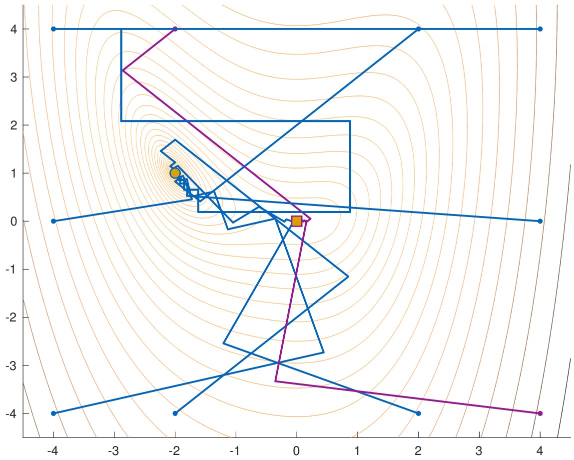
b) The solution paths are summarized and shown in Figure 1. Exemplary code is presented in Listing 4. We test the general descent method with directions d generated as in **Problem 2 a)** using exact line search. A more detailed table with the different results is presented below in Table 2

	Backtracking: $(0.1, 0.5)$			Exact Step Sizes: $(100, 10^{-6}, 2)$		
Initial Point	Iter.	Obj. Value	Comment	Iter.	Obj. Value	Comment
$(-4,-4)^\top$	500	$2.067 \cdot 10^{-09}$	Conv. to x^*	19	$-4.00 \cdot 10^0$	Conv. to y^*
$(-4,0)^\top$	36	$-4.00 \cdot 10^0$	Conv. to y^*	20	$-4.00 \cdot 10^0$	Conv. to y^*
$(-4,4)^\top$	39	$-4.00 \cdot 10^0$	Conv. to y^*	36	$-4.00 \cdot 10^0$	Conv. to y^*
$(-2,-4)^\top$	41	$-4.00 \cdot 10^0$	Conv. to y^*	24	$-4.00 \cdot 10^0$	Conv. to y^*
$(-2,4)^\top$	32	$-4.00 \cdot 10^0$	Conv. to y^*	613	$2.552 \cdot 10^{-09}$	Conv. to x^*
$(2,-4)^\top$	396	$4.149 \cdot 10^{-09}$	Conv. to x^*	23	$-4.00 \cdot 10^0$	Conv. to y^*
$(2,4)^\top$	31	$-4.00 \cdot 10^0$	Conv. to y^*	22	$-4.00 \cdot 10^0$	Conv. to y^*
$(4,-4)^\top$	35	$-4.00 \cdot 10^0$	Conv. to y^*	121	$3.948 \cdot 10^{-09}$	Conv. to x^*
$(4,0)^\top$	37	$-4.00 \cdot 10^0$	Conv. to y^*	34	$-4.00 \cdot 10^0$	Conv. to y^*
$(4,4)^\top$	34	$-4.00 \cdot 10^0$	Conv. to y^*	36	$-4.00 \cdot 10^0$	Conv. to y^*
Average	118.1			94.8		

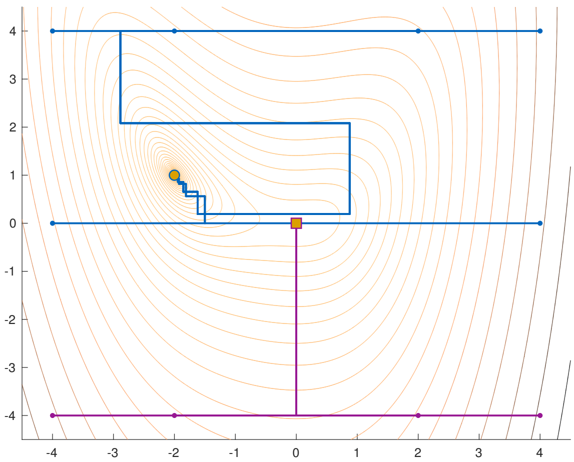
Table 2: Comparison of the gradient method using different step sizes strategies for $\mathcal{X}^0$.



(a) Backtracking for $\mathcal{X}^0$



(b) Exact Step Sizes for $\mathcal{X}^0$



(c) Descent Method with d from **P2 a)**

Figure 1: Solution paths for the gradient method with backtracking, exact line search, and a different descent direction (with exact line search) 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 below. The following input functions and parameters should be considered:

- 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`.
- `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,...)`).
- $\gamma_1, \gamma_2 > 0$ – parameters for the Newton condition.

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 $k = 0, 1, \dots$ **do**

- If $\|\nabla f(x^k)\| \le$ `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 $-\nabla f(x^k)^\top s^k \ge \gamma_1 \min\{1, \|s^k\|^{\gamma_2}\}\|s^k\|^2$, 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 $x^{k+1} = x^k + \alpha_k d^k$.

- $\sigma, \gamma \in (0,1)$ – 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
$$-\nabla f(x^k)^\top s^k \ge \gamma_1 \min\{1, \|s^k\|^{\gamma_2}\}\|s^k\|^2,$$
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 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 `tol` $= 10^{-7}$.

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

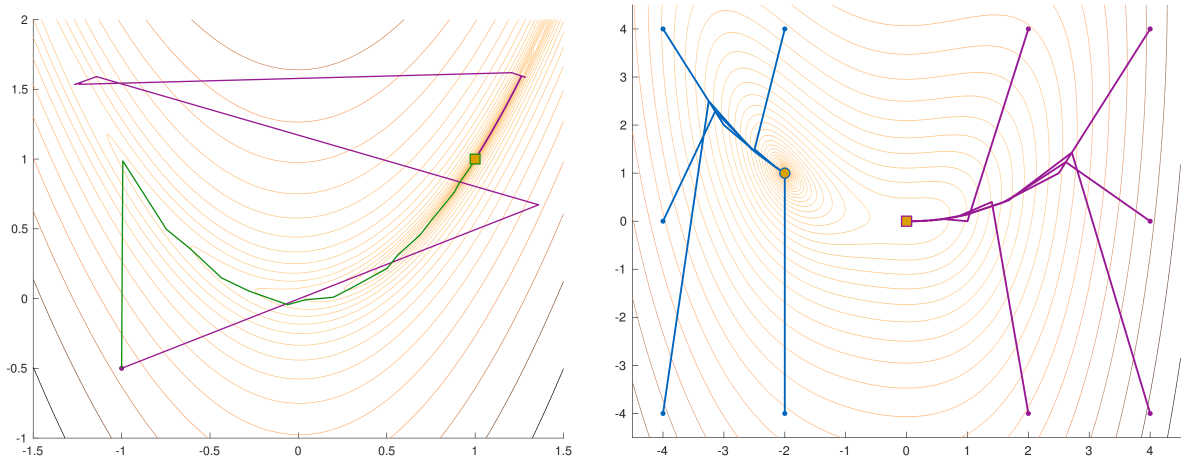
b) Repeat the performance test from **Problem 3 b)** for problem (1) with the globalized Newton method. You can use $(\sigma, \gamma) = (0.5, 0.1)$, $(\gamma_1, \gamma_2) = (10^{-6}, 0.1)$, and `tol` $= 10^{-5}$.

   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).

Solution :

a) The solution paths of the Newton method and gradient method are summarized and shown in Figure 2 (a). The corresponding `MATLAB` code can be found in Listing 6 and 7.

   The Newton method requires 21 iterations (0.0035 seconds) to recover a solution x^k satisfying $\|\nabla f(x^k)\| \le 10^{-7}$. The gradient method (with Armijo linesearch) requires 16897 iterations (1.1200 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 $\alpha_k = 1$.



(a) Newton's Method (green) vs Gradient Method (purple) on the Rosenbrock function

(b) Newton's Method for **Problem 3**

Figure 2: Solution paths of Newton's method for the problems in **Problem 4**

Globalized Newton Method					
	`tol`: 10^{-5}		`tol`: 10^{-8}		
Initial Point	Iter.	Obj. Value	Iter.	Obj. Value	Comment
$(-4,-4)^\top$	7	$-4.00 \cdot 10^0$	8	$-4.00 \cdot 10^0$	Conv. to y^*
$(-4,0)^\top$	7	$-4.00 \cdot 10^0$	8	$-4.00 \cdot 10^0$	Conv. to y^*
$(-4,4)^\top$	7	$-4.00 \cdot 10^0$	8	$-4.00 \cdot 10^0$	Conv. to y^*
$(-2,-4)^\top$	2	$-4.00 \cdot 10^0$	2	$-4.00 \cdot 10^0$	Conv. to y^*
$(-2,4)^\top$	6	$-4.00 \cdot 10^0$	7	$-4.00 \cdot 10^0$	Conv. to y^*
$(2,-4)^\top$	13	$3.270 \cdot 10^{-09}$	18	$9.996 \cdot 10^{-14}$	Conv. to x^*
$(2,4)^\top$	13	$9.205 \cdot 10^{-10}$	18	$2.812 \cdot 10^{-14}$	Conv. to x^*
$(4,-4)^\top$	15	$8.998 \cdot 10^{-10}$	20	$2.749 \cdot 10^{-14}$	Conv. to x^*
$(4,0)^\top$	15	$7.694 \cdot 10^{-10}$	20	$2.350 \cdot 10^{-14}$	Conv. to x^*
$(4,4)^\top$	15	$6.375 \cdot 10^{-10}$	20	$1.948 \cdot 10^{-14}$	Conv. to x^*
Average	10		12.9		

Table 3: Comparison of the globalized Newton method using different tolerances for $\mathcal{X}^0$.

b) Results are summarized in Figure 2 (b) and Table 3. Clearly, Newton's method again requires the least number of iterations to show convergence.

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

```matlab
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.tol,options.maxit);
    fprintf(1,'ITER ; X ; |G(X)| ; |XR-XL|\n');
end

for i = 1:options.maxit
    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.tol % || 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
```

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

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

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

xln = phi*xr + (1-phi)*xl;
xrn = (1-phi)*xr + phi*xl;

fln = f(xln);
```

```
frn = f(xrn);

if options.display
    fprintf(1,'\n--- golden section algorithm; \n--- [tol = %1.2e/ maxit = %4i]\n',options.tol,options.maxit);
    fprintf(1,'ITER ; X ; F(X) ; |XR-XL|\n');
end

for i = 1:options.maxit

    if fln < frn
        xr  = xrn;
        xrn = xln;
        xln = phi*xr + (1-phi)*xl;
        frn = fln;
        fln = f(xln);
    else
        xl  = xln;
        xln = xrn;
        xrn = (1-phi)*xr + phi*xl;
        fln = frn;
        frn = f(xrn);
    end

    if options.display
        fprintf(1,'[%4i] ; %1.8e ; %1.2e ; %1.2e \n',i,(xl+xr)/2,f((xl+xr)/2),abs(xl-xr));
    end

    if abs(xl-xr) < options.tol
        x = (xl+xr)/2;
        return
    end
end
```

Listing 3: **Problem 3** – 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      = 1;
elseif strcmp(opts.mode,'fixed-step-size')
    if ~isfield(opts,'alpha')
        opts.alpha  = 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.maxit   = 100;
        opts_au.tol     = 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'
            %[~,ind]     = max(abs(g));
            %d           = zeros(length(x0),1);
            %d(ind(1))   = 1;
            %g           = g(ind(1))*d;

            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;
end

end
```

Listing 4: **Problem 3** – `MATLAB` code: demo–file

```

% test function
f.obj       = @(x) x(1)^4+2*(x(1)-x(2))*x(1)^2+4*x(2)^2;
f.grad      = @(x) [4*x(1)^3+6*x(1)^2-4*x(1)*x(2);-2*x(1)^2+8*x(2)];
f.hess      = @(x) [12*x(1)^2+12*x(1)-4*x(2),-4*x(1);-4*x(1),8];
f.obj_print = @(x,y) x.^4+2*(x-y).*x.^2+4*y.^2;
```

```
% initial point
x0_ref      = [-4,-4,-4,-2,-2,2,2,4,4,4;-4,0,4,-4,4,-4,4,-4,0,4];
%x0_ref      = [3;-3];

% options
opts.maxit  = 10000;
opts.trace  = true;
opts.print  = false;
opts.mode   = 'armijo-linesearch';
%opts.mode   = 'exact-linesearch';

opts.s      = 1;
opts.sigma  = 0.5;
opts.gamma  = 0.1;

nr_ini      = 10;

xl = -4.5; xr = 4.5; yl = -4.5; yr = 4.5;

[X,Y]       = meshgrid(xl:0.01:xr,yl:0.01:yr);
Z           = f.obj_print(X,Y);
max_z       = max(max(Z));

figure;
hold on

contour(X,Y,Z,logspace(-2,3,40)-4)

%fprintf(1,'- - - gradient method; step size: %s; n = %g; \n',opts.mode,2);
%fprintf(1,'ITER ; OBJ.VAL ; G.NORM ; [X1/X2]\n');

a_iter = 0;

for i = 1:nr_ini
     x0          = x0_ref(:,i);

     [x,out]     = gradient_method(f,x0,1e-5,opts);
     %[x,out]    = newton_glob(f,x0,1e-5,opts);

     if norm(x-[0;0]) <= 0.1
         clr = [152,24,147]/255;
     elseif norm(x-[-2;1]) <= 0.1
         clr = [0,101,188]/255;
     else
         clr = [1,1,1];
     end

     %fprintf(1,'[%5i] ; %1.6e ; %1.6e ; [%1.6e/%1.6e]\n',out.iter,f.obj(x),norm(f.grad(x)
         ),x(1),x(2));
     plot3(x0(1),x0(2),max_z,'.','Color',clr,'MarkerSize',13);
     plot3(out.trace.x(:,1),out.trace.x(:,2),max_z*ones(size(out.trace.x,1),1),'-','Color'
         ,clr,'LineWidth',1.5,'MarkerSize',10);

     a_iter = a_iter + out.iter;
end
```

```
a_iter = a_iter/nr_ini;

plot3(0,0,1.1*max_z,'s','MarkerSize',10,'MarkerFaceColor',[219,160,1]/255,'MarkerEdgeColor
    ',[152,24,147]/255);
plot3(-2,1,1.1*max_z,'o','MarkerSize',8,'MarkerFaceColor',[219,160,1]/255,'MarkerEdgeColor
    ',[0,101,188]/255);

hold off
colormap(flipud(copper))

axis([xl xr yl yr])

set(gcf,'Renderer', 'painters');
saveas(gcf,'plot_p3-01','epsc');
```

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
```

```
    d       = - f.hess(x)\g;

    gtd     = d'*g;
    nd      = norm(d);

    if - gtd < 1e-6*min(1,nd^0.1)*nd^2
        d       = -g;
        gtd     = -ng^2;
        type    = 'G';
    else
        type    = 'N';
    end

    alpha   = 1;
    x       = x_old + alpha*d;
    f_new   = f.obj(x);
    acount  = 1;

    while f_new - f_old > alpha*opts.gamma*gtd && acount <= 100
        alpha   = alpha/2;
        x       = x_old + alpha*d;
        f_new   = f.obj(x);
        acount  = acount + 1;
    end

    f_old   = f_new;
end

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
```

Listing 6: **Problem 4** – `MATLAB` code: demo file and plotting

```

% test function
f.obj       = @(x) 100*(x(2)-x(1)^2)^2 + 1*(1-x(1))^2;
f.grad      = @(x) [400*(x(1)^2-x(2))*x(1)+2*(x(1)-1);200*(x(2)-x(1)^2)];
f.hess      = @(x) [400*(3*x(1)^2-x(2))+2,-400*x(1);-400*x(1),200];
f.obj_print = @(x,y) 100*(y-x.^2).^2 + 1*(1-x).^2;

% initial point
x0_ref      = [-1;-0.5];

% options
opts.maxit  = 25000;
opts.trace  = true;
opts.print  = true;
```

```
opts.mode   = 'armijo-linesearch';

opts.s      = 1;
opts.sigma  = 0.5;
opts.gamma  = 1e-4;

nr_ini      = 1;

xl = -1.5; xr = 1.5; yl = -1; yr = 2;

[X,Y]       = meshgrid(xl:0.01:xr,yl:0.01:yr);
Z           = f.obj_print(X,Y);
max_z       = max(max(Z));

tol         = 1e-7;

figure;
hold on

ch          = contour(X,Y,Z,logspace(-2.5,4,30));

fprintf(1,'ITER ; OBJ.VAL ; G.NORM ; [X1/X2]\n');

for i = 1:nr_ini
     x0          = x0_ref(:,i);

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

     fprintf(1,'[%5i] ; %1.6e ; %1.6e ; [%1.6e/%1.6e]\n',out.iter,f.obj(x),norm(f.grad(x)),x(1),x(2));

     plot3(x0(1),x0(2),max_z,'.','Color',[152,24,147]/255,'MarkerSize',12);
     plot3(out.trace.x(:,1),out.trace.x(:,2),max_z*ones(size(out.trace.x,1),1),'-','Color',[152,24,147]/255,'LineWidth',1.1,'MarkerSize',10);

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

     fprintf(1,'[%5i] ; %1.6e ; %1.6e ; [%1.6e/%1.6e]\n',out.iter,f.obj(x),norm(f.grad(x)),x(1),x(2));

     plot3(x0(1),x0(2),max_z,'.','Color',[17,140,17]/255,'MarkerSize',12);
     plot3(out.trace.x(:,1),out.trace.x(:,2),max_z*ones(size(out.trace.x,1),1),'-','Color',[17,140,17]/255,'LineWidth',1.1,'MarkerSize',10);
end

plot3(1,1,1.1*max_z,'s','MarkerSize',10,'MarkerFaceColor',[219,160,1]/255,'MarkerEdgeColor',[17,140,17]/255);

hold off
colormap(flipud(copper))

axis([xl xr yl yr])

set(gcf,'Renderer', 'painters');
saveas(gcf,'plot_p4-ros','epsc');
```