The University of Texas
at Austin

EE381V Large Scale Optimization

Problem Set 4

Edited by LaTeX

Department of Computer Science

STUDENT
Jimmy Lin
xl5224

COURSE COORDINATOR
Sujay Sanghavi

UNIQUE NUMBER
17350

RELEASE DATE
October 18, 2014

DUE DATE
October 23, 2014

TIME SPENT
10 hours

October 25, 2014

Table of Contents

List of Figures

Part I
Matlab and Computational Assignment

1 Conjugate Gradient Algorithm

1.1 M_1

Command to be executed in matlab:

```
>> load ConjugateGradient.mat
>> CGS(M1, b1, x)
```

Plot

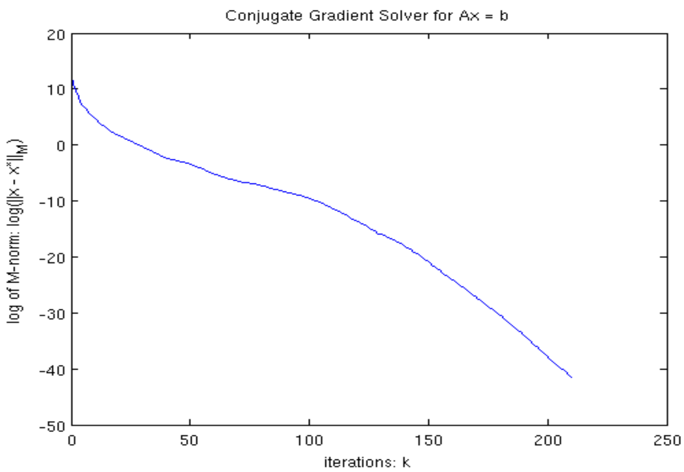


Figure 1: Conjugate Gradient Solver for M_1

1.2 M_2

Command to be executed in matlab:

```
>> load ConjugateGradient.mat
>> CGS(M2, b2, x)
```

Plot

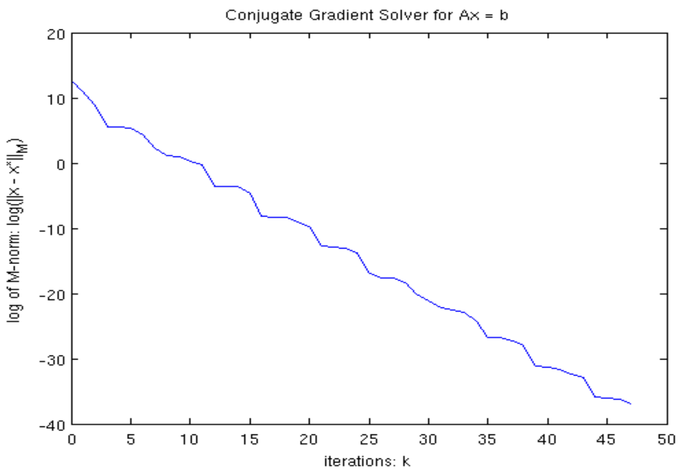


Figure 2: Conjugate Gradient Solver for M_2

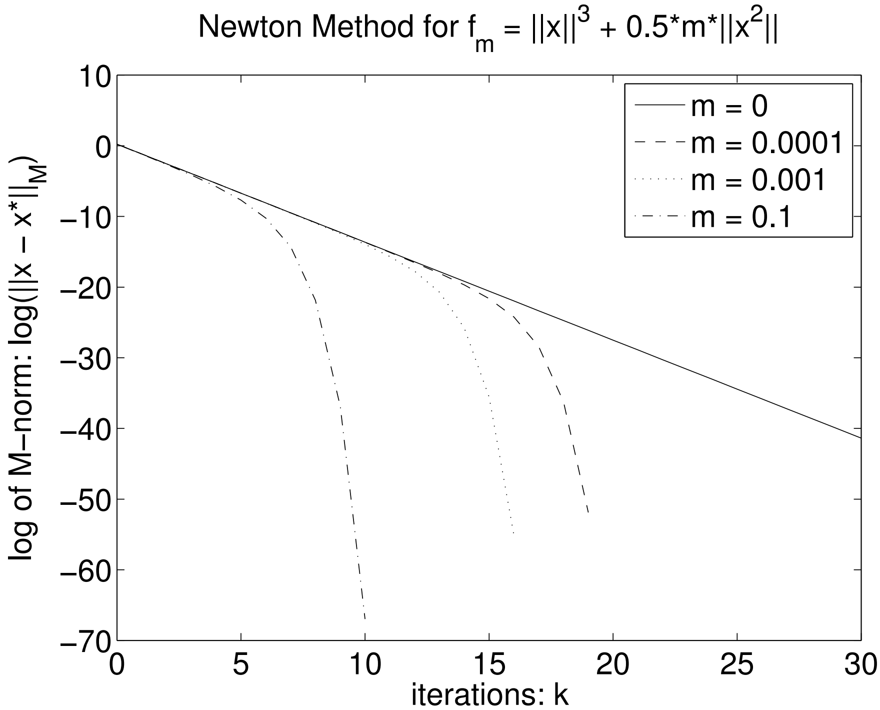
2 Newtons Method

2.1 plots for various m

Command to run:

```
>> Newton(0, 'k-')
>> hold on
>> Newton(0.0001, 'k--')
>> Newton(0.001, 'k:')
>> Newton(0.1, 'k-.')
>> Newton(0, 'k-')
```

Plots Note that the initial point here is $0.5 * ones(5, 1)$.



2.2 Explanation

From the figure, we can see that the $f_m(\cdot)$ goes to quadratic convergence phrase with fewer number of iterations if m is greater. The intuitive explanation is that the larger m will cause quadratic convergence condition $||\nabla f(x)|| < \frac{m^2}{L}$ easier to satisfy. Note that one extreme is when $m = 0$, our newton method solver never goes into quadratic convergence phrase (only linear convergence phrase).

3 Central Path

3.1 Find a function F

$$F = -\sum_{i=1}^{4} log(x_i) - log(4 - x_1 - 3x_2 - x_4) - log(3 - 2x_1 - x_2) - log(3 - x_2 - 4x_3 - x_4) \quad (1)$$

3.2 Find analytic center x_F^*

$$x_F^* = arg \min_{x \in domF} F(x) \quad (2)$$
$$= (0.5488, 0.3091, 0.2543, 0.6485) \quad (3)$$

3.3 Generate a central path

The commands we used for generating a central path with various α

```
>> t_init = 5; alpha1=0.01; alpha2=0.1; alpha3=0.5; alpha4=1e-3;
>> CP(t_init, alpha1, 'k-');
>> hold on
>> CP(t_init, alpha2, 'k--');
>> CP(t_init, alpha3, 'k:');
>> CP(t_init, alpha4, 'k-.');
```

3.4 Plot the error $log(||x^{(k)} - x^*||)$ w.r.t k

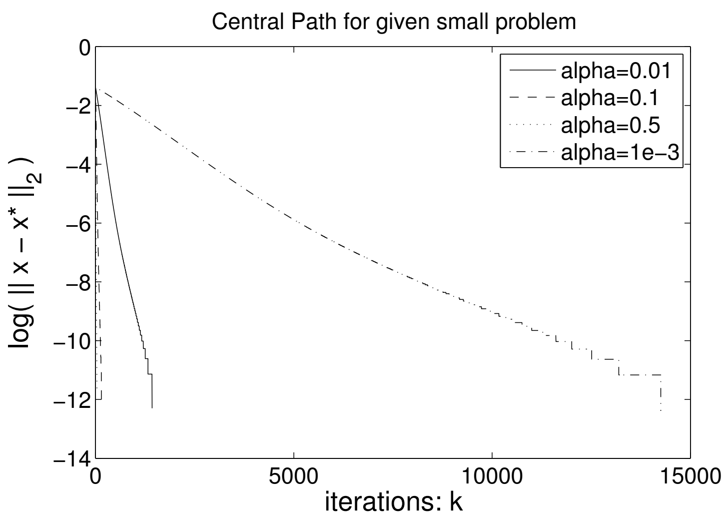
In order to plot the errors, we first employ CVX to obtain the optimal feasible point x^* by

```
>> CVX_solve_CP
```

For implementation details of this program, please go to appendix.
Then we have optimal feasible point for plotting $log(||x^{(k)} - x^*||)$

$$x^* = (0, 0, 0, 0) \quad (4)$$

Plots



4 Larger Linear Program

4.1 Find a function F

For given $A^{m \times n}$

$$F = -\sum_{i=1}^{N} log(x_i) - \sum_{i=1}^{m} log(b - a_i^T x) \tag{5}$$

4.2 Find analytic center x_F^*

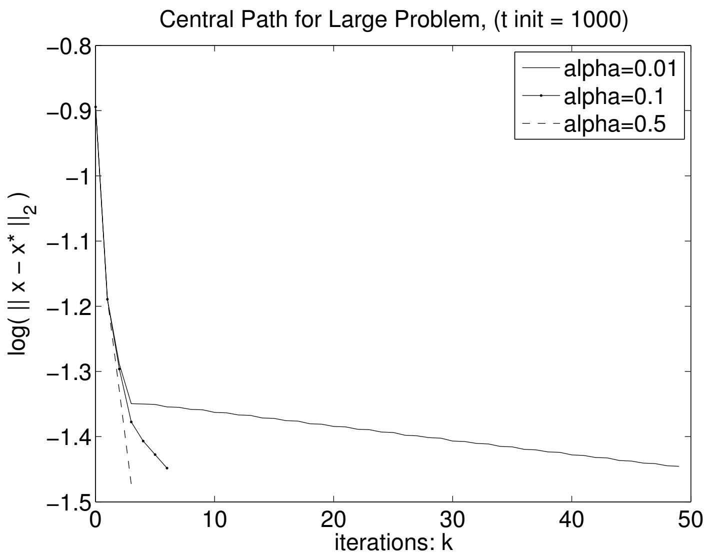
Newton's method start from $x_0 = 0.01 * ones(100, 1)$. We first employ CVX to compute a feasible point and then use newton_solver to figure out the analytic center. Since it has 50 elements, we are not going to post it here.

4.3 Generate a central path

Using similar commands with ones in response to previous question.

4.4 Plot the error $log(||x^{(k)} - x^*||)$ w.r.t k

Plot



5 Gradient and Newton

Command to be executed:

```
>> Rosenbrock
```

Plot

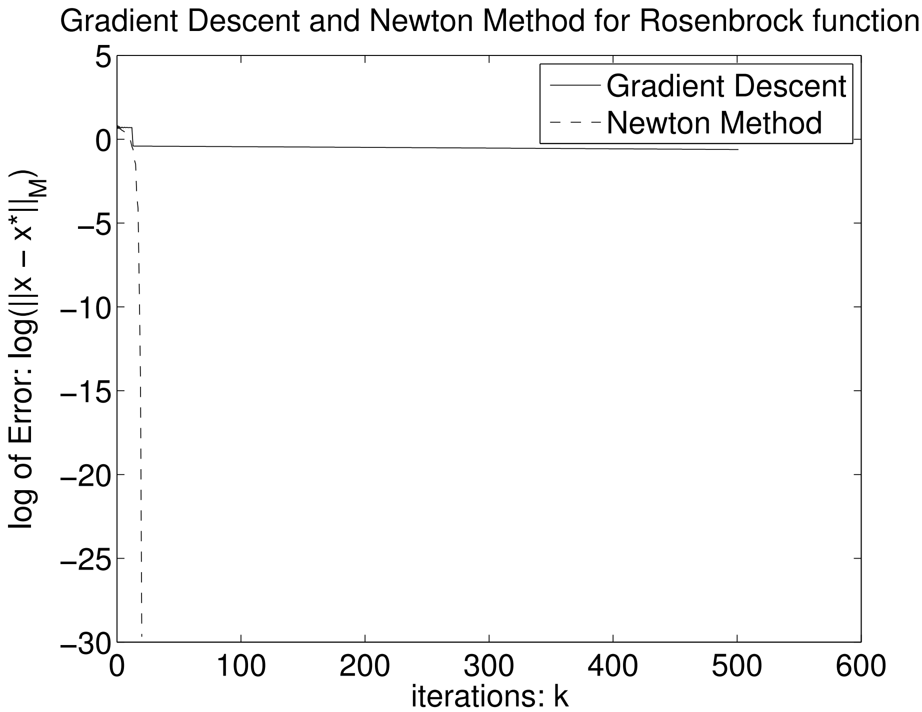


Figure 3: Gradient Descent and Newton Method on Rosenbrock function

Part II
Written Problems

6 $\alpha-$holder

Now we derive the rate of convergence with step size $t = 1$,

$$
\begin{aligned}
||\nabla f(x^+)||_2 &= ||\nabla f(x + \Delta x_{nt}) - \nabla f(x) - \nabla^2 f(x)\Delta x_{nt}||_2 && (6)\\
&= \left|\left| \int_0^1 \big(\nabla^2 f(x + t\Delta x_{nt}) - \nabla^2 f(x)\big)\Delta x_{nt} dt \right|\right|_2 && (7)\\
&\leq \int_0^1 ||\big(\nabla^2 f(x + t\Delta x_{nt}) - \nabla^2 f(x)\big)||_2 \cdot ||\Delta x_{nt}||_2 \cdot dt && (8)\\
&\leq \int_0^1 H||t\Delta x_{nt}||_2^\alpha \cdot ||\Delta x_{nt}||_2 \cdot dt && (9)\\
&= H||\Delta x_{nt}||_2^{1+\alpha} \cdot \frac{1}{1+\alpha} t^{1+\alpha}\Big|_0^1 && (10)\\
&= \frac{H}{1+\alpha}||\Delta x_{nt}||_2^{1+\alpha} && (11)\\
&= \frac{H}{1+\alpha}||\nabla^2 f(x)^{-1}\nabla f(x)||_2^{1+\alpha} && (12)\\
&\leq \frac{H}{(1+\alpha)m^{1+\alpha}}||\nabla f(x)||_2^{1+\alpha} && (13)\\
& && (14)
\end{aligned}
$$

Thus, we can easily find a constant C, such that

$$C \cdot ||\nabla f(x^+)||_2 \leq \Big(C \cdot ||\nabla f(x)||_2\Big)^{1+\alpha} \quad (15)$$

where $C = (\frac{H}{(1+\alpha)m^{1+\alpha}})^{-\alpha}$.
Recursively apply (15) and simulate the same inference of (9.34) and (9.35) in Boyd's book, we can conclude that once $||\nabla f(x^{(k)})||$ is small enough, then the convergence will go into convergence phrase with order at $1 + \alpha$.

A Codes Printout

A.1 Conjugate Gradient Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1. Conjugate Gradient Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function CGS(M, b, x_opt)
    [R, C] = size(M);
    assert (R == C, 'M should be square matrix.');
    assert (R == size(b,1), 'Dim of M and b should be consistent.');

    EPSILON = 10e-10; % how close solution do we need
    x_0 = zeros(size(x_opt)); % initial point

    listK = [];
    listlogMdiff = [];

    k = 0;
    x = x_0;
    r = b - M * x; % residual
    p = r;
    while 1,
        diff = x - x_opt;
        logMdiff = log(diff' * M * diff);

        fprintf ('iteration: %d, log(||x - x*||_M) = %f \n', k, logMdiff)
        listK = [listK k];
        listlogMdiff = [listlogMdiff logMdiff];

        alpha = (r' * r) / (p' * M * p);
        x = x + alpha * p;
        r_new = r - alpha * M * p;
        if r_new < EPSILON,
            break
        end
        beta = (r_new' * r_new) / (r' * r);
        p = r_new + beta * p;
        r = r_new;
        k = k + 1;
    end
    plot (listK, listlogMdiff)
    title('Conjugate Gradient Solver for Ax = b')
    xlabel('iterations: k')
    ylabel('log of M-norm: log(||x - x*||_M)')
end
```

A.2 Newtons Method

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2. Newton Method
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function Newton(m, marker)
    x_opt = zeros(5, 1); % initial point
    t = 1; % step size fixed at 1
    x_0 = [0.5 0.5 0.5 0.5 0.5]';

    listK = [];
    listlogMdiff = [];

    k = 0;
    x = x_0;
    while 1,
        diff = x - x_opt;
        logMdiff = log(norm(diff,2)^2);

        fprintf ('iteration: %d, log(||x - x*||_2^2) = %f \n', k, logMdiff)
        listK = [listK k];
        listlogMdiff = [listlogMdiff logMdiff];

        grad = grad_func(x, m);
        hess = hess_func(x, m);
        x = x - t * inv(hess) * grad;

        k = k + 1;
        %if grad' * hess * grad <= eps,
        if logMdiff < -40,
            break
        end
    end
    plot (listK, listlogMdiff, marker)
    title('Newton Method for f_m', 'fontsize', 18)
    xlabel('iterations: k', 'fontsize', 18)
    ylabel('log of M-norm: log(||x - x*||_M)', 'fontsize', 18)
    set(gca, 'fontsize', 18)
end

function val = func(x, m)
    normx = norm(x, 2);
    val = normx^3 + 0.5 * m * normx^2;
end

function val = grad_func(x, m)
    val = 3 * norm(x, 2) * x + m * x;
end

function val = hess_func(x, m)
    len = size(x,1);
    normx = norm(x, 2);
    val = (3 / normx) * x * x' + ( 3* normx + m) * eye(len, len);
end
```

A.3 Newton Solver For Central Path Generation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3. General Newton's method solver for Central Path problem
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function x_opt = Newton_Solver (b, A, x_0, t, c)
    eta = 0.1;
    k = 0;
    x = x_0;
    x_opt = x;
    beta = 0.8;
    while k < 100,
        grad = grad_func(b, A, x, t, c);
        hess = hess_func(b, A, x, t, c);
        delta_x = - inv(hess) * grad;
        x = x + eta * delta_x;
        if norm(x-x_opt, 2)/norm(x, 2) <= 1e-3 ,
            break;
        else
            k = k + 1;
            x_opt = x;
        end
    end
    x_opt = x;
end
function val = func (b, A, x, t, c)
    N = size(x, 1); % number of variables
    C = size(A, 1); % number of constraint
    val = 0.0;
    for i = 1:N,
        val = val - log(x(i));
    end
    for i = 1:C,
        val = val - log(b(i) - A(i,:)*x);
    end
    val = val + t* c' * x;
end
function val = grad_func (b, A, x, t, c)
    N = size(x, 1);
    C = size(A, 1); % number of constraint
    val = zeros(N, 1);
    val = val - 1 ./ x;
    for i = 1:C,
        val = val - (1 / (b(i) - A(i,:)*x)) * (-A(i,:)');
    end
    val = val + t * c;
end
function val = hess_func (b, A, x, t, c)
    N = size(x, 1);
    C = size(A, 1); % number of constraint
    val = zeros(N, N);
    for i = 1:N,
        val(i, i) = val(i, i)+ (1/x(i))^2;
    end
    for i = 1:C,
        val = val + (1 / (b(i) - A(i,:)*x))^2 * (A(i,:)*A(i,:)');
    end
end
```

A.4 Small Linear Program

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3. Central Path
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function x = CP (t_init, alpha, marker)
    % acquired global optima from cvx command
    x_opt = zeros(4, 1);
    % create barrier system of equation
    A = [1 3 0 1; 2 1 0 0; 0 1 4 1];
    b = [4 3 3]';
    c = [2 4 1 1]';
    % find analytical center
    x_F_init = [0.2, 0.2, 0.2, 0.2]';
    t = 0;
    x_F = Newton_Solver (b, A, x_F_init, t, c);
    % generate a central path
    listK = [0];
    listLogNorm2 = [log(norm(x_F-x_opt, 2))];

    k = 1;
    t = t_init;
    x = x_F;
    while 1,
        x = Newton_Solver (b, A, x, t, c);

        logNorm2 = log(norm(x - x_opt, 2));
        listK = [listK k];
        listLogNorm2 = [listLogNorm2 logNorm2];

        if norm(x - x_opt, 2) < 1e-5,
            break
        end
        t = t * (1 + alpha);
        k = k + 1;
    end
    semilogx (listK, listLogNorm2, marker)
    xlabel('iterations: k', 'fontsize', 18)
    ylabel('log( || x - x* ||_2 )', 'fontsize', 18)
    title('Central Path for the large problem', 'fontsize', 18)
    set(gca, 'fontsize', 18)
end
```

A.5 Larger Linear Program

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3. Central Path for the large linear program
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function x = CP_large (c, A, b, t_init, alpha, marker)
    % acquired global optima from cvx command
    [M, N] = size(A);
    x_opt = CVX_solve_CP_large (c, A, b)

    % find feasible starting point
    %x_F_init = feasible(b, A, c, ones(50, 1)');
    x_F_init = 10e-9 * ones(50, 1);
    % find analytical center
    x_F = Newton_Solver (b, A, x_F_init, 0, c')
    % generate a central path
    listK = [0];
    listLogNorm2 = [log(norm(x_F-x_opt, 2))];

    k = 1;
    t = t_init;
    x = x_F;
    while k < 50,
        x = Newton_Solver (b, A, x, t, c');

        logNorm2 = log(norm(x - x_opt, 2));
        listK = [listK k];
        listLogNorm2 = [listLogNorm2 logNorm2];

        if norm(x - x_opt, 2) < 1e-5,
            break
        end
        t = t * (1 + alpha);
        k = k + 1;
    end
    plot (listK, listLogNorm2, marker)
    xlabel('iterations: k', 'fontsize', 18)
    ylabel('log( || x - x* ||_2 )', 'fontsize', 18)
    title('Central Path for the large problem', 'fontsize', 18)
    set(gca, 'fontsize', 18)
end
```

A.6 Gradient and Newton

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5. Gradient descent and Newton on Rosenbrock function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function Rosenbrock ()
    EPSILON = 10e-10; % how close solution do we need
    alpha = 0.1;
    beta = 0.95;

    x_0 = [-1.2, 1]'; % initial point
    x_opt = [1 1]';

    [listK_GD, listError_GD] = gradient_descent (@func, @grad_func, ...
                                             x_0, x_opt, alpha, beta);
    [listK_NM, listError_NM] = newton_method (@func, @grad_func, @hess_func, ...
                                             x_0, x_opt, alpha, beta);

    plot (listK_GD, listError_GD, 'k-', listK_NM, listError_NM, 'k--')
    set(gca, 'fontsize', 18)
    legend ('Gradient Descent', 'Newton Method')
    title('Gradient Descent and Newton Method for Rosenbrock function')
    xlabel('iterations: k')
    ylabel('log of Error: log(||x - x*||_M)')
end

%% gradient descent with BTLS
function [listK, listError] = gradient_descent (f, grad_f, x_0, x_opt, alpha, beta)
    listK = [];
    listError = [];
    x = x_0;  % initial point
    k = 0;
    while 1,
        Error = log(norm(x_opt-x, 2));

        fprintf ('iteration: %d, Error = %f \n', k, Error)
        listK = [listK k];
        listError = [listError Error];
        if k > 500,
            break
        end

        gradient = grad_f(x);
        delta_x = -1.0 * gradient;
        t = 1.0;
        while f(x+t*delta_x) > f(x) + alpha*t*gradient'*delta_x,
           t = beta * t;
        end
        x = x + t * delta_x;
        k = k + 1;
    end
end
%% newton method with BTLS
function [listK, listError] = newton_method (f, grad_f, hess_f, x_0, x_opt, alpha, beta)
    listK = [];
    listError = [];
    x = x_0;  % initial point
    k = 0;
    while 1,
        % Error = f(x);
        Error = log(norm(x_opt-x, 2));

        fprintf ('iteration: %d, Error = %f \n', k, Error)
        listK = [listK k];
        listError = [listError Error];
```

```
        if k > 50,
            break
        end

        gradient = grad_f(x);
        hessian = hess_f(x);
        delta_x = -1.0 * inv(hessian) * gradient;
        t = 1.0;
        while f(x+t*delta_x) > f(x) + alpha*t*gradient'*delta_x,
           t = beta * t;
        end
        x = x + t * delta_x;
        k = k + 1;
    end
end

function [x_1, x_2] = parse (x)
    [m, n] = size(x);
    assert (m == 2 && n == 1);
    x_1 = x(1);
    x_2 = x(2);
end
function y = func (x)
    [x_1, x_2] = parse(x);
    y = 100 * (x_2 - x_1^2)^2 + (1-x_1)^2;
end
function y = grad_func (x)
    [x_1, x_2] = parse(x);
    y = zeros(2,1);
    y(1) = -400 * x_1 * (x_2-x_1^2) - 2 * (1-x_1);
    y(2) = 200 * (x_2 - x_1^2);
end
function y = hess_func (x)
    [x_1, x_2] = parse(x);
    y = zeros(2,2);
    y(1,1) = -400 * (x_2 - 3*x_1^2 ) + 2;
    y(1,2) = -400 * x_1;
    y(2,1) = y (1,2);
    y(2,2) = 200;
end
```