**Numerical Optimization**

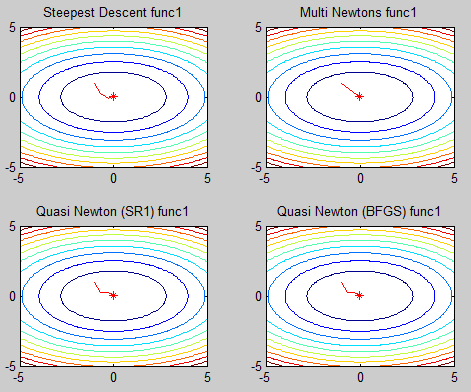
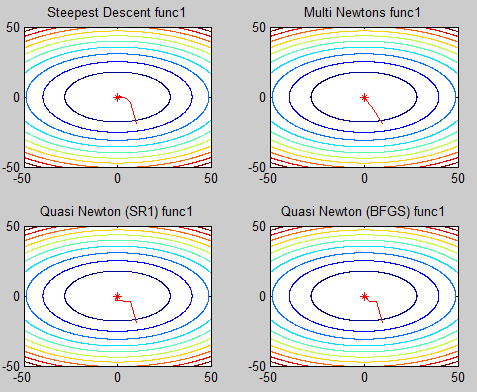
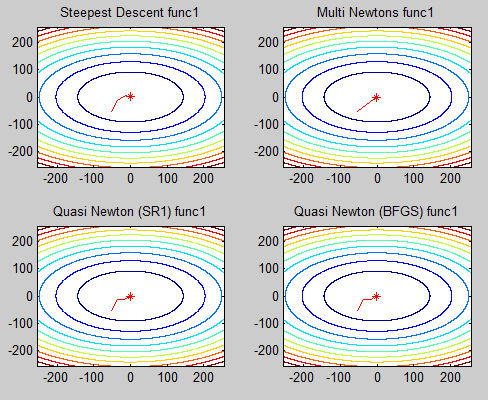
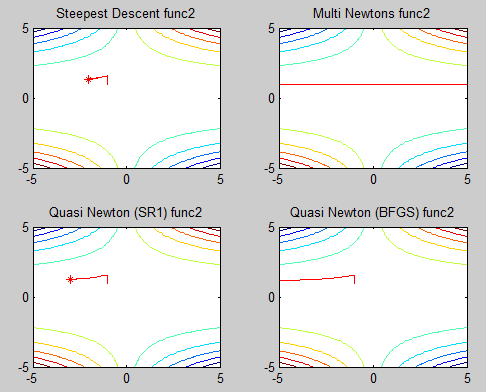
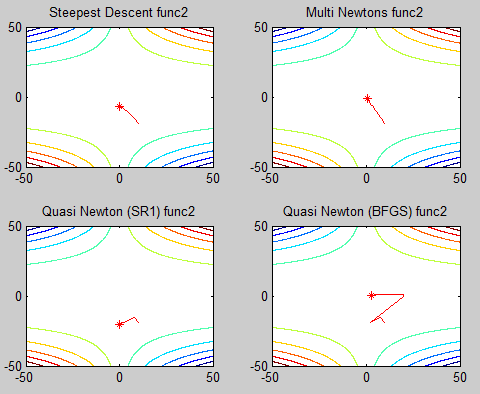
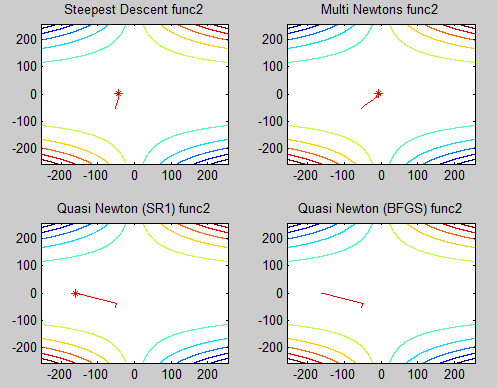
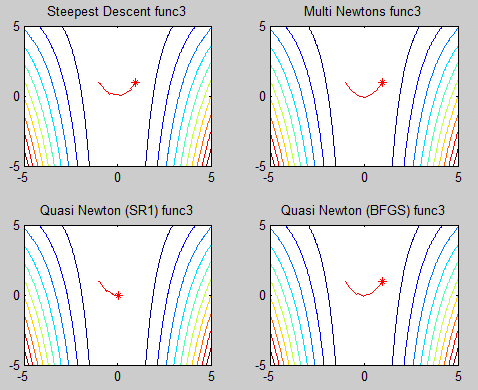
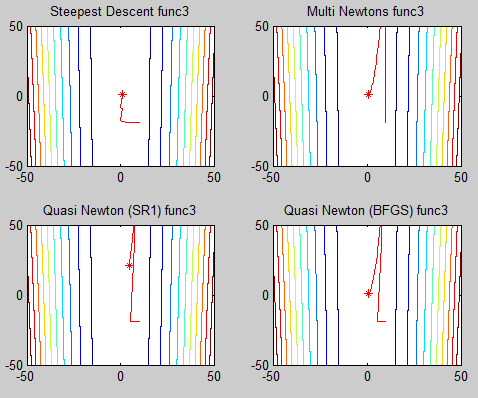
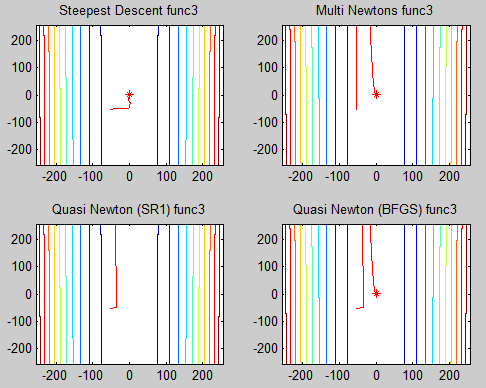
**Programming #4**

**Electrical Engineering & Computer Science**

**20171014 Young-Chul Yoon**

1. **Introduction  
   1) Goal of the assignment  
   -** Implement Steepest Descent, Newton’s, Quasi Newton’s(SR1, BFGS1) methods  
   - Compare their performance for the following three problems  
   - First start at (-1.0, 1.0) at each function. Then use different starting points

**2) Description about implementation and setting**I implemented methods using MATLAB 2014a. Each method is separately implemented as function. I used symbolic function to make 3 problems and diff function, which is included in MATLAB, to implement gradient and Hessian matrix. To visualize the convergence of each method, I made a contour plot and drew the movement of point on it.  
**1) Global setting**  
- starting points : (-1, 1), (10, -19), (-400, -270)  
- step length : smart step length seeking in lecture note  
- stopping criterion : (gradient of current point < 1.0e-3), time limit  
- performance check : time  
- time limit : maximum 10 seconds per method **2) Quasi Newton’s method**  
- Initial H : identity matrix  
- SR1 break down avoidance  
**< 3 problems >**  
1)   
2)   
3)

1. **Result in program  
   (star symbol means last point)**function 1, starting point [-1, 1]  
   function 1, starting point [10, -19]function 1, starting point [-53, -51]function 2, starting point [-1, 1]function2, starting point [10, -19]function2, starting point [-53, -51]  
     
   function3, starting point [-1, 1]function3, starting point [10, -19]  
     
   function3, starting point [-53, -51]
2. **Analysis**
3. Steepest descent method

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| problem | starting point | solution point | solution | time |
|  | [-1, 1] | (-0.000087, 0.000023) | 0.000000 | 0.254180sec |
| [10, -19] | (0.000084, -0.000030) | 0.000000 | 0.298858sec |
| [-53, -51] | (-0.000168, 0.000046) | 0.000000 | 0.317841sec |
|  | [-1, 1] | (-2.042985, 1.355537) | 1.066015 | 10.030303sec |
| [10, -19] | (0.006442, -7.091306) | 8.791692 | 10.000219sec |
| [-53, -51] | (-42.834744, 1.022671) | 0.487324 | 3.752598sec |
|  | [-1, 1] | (0.977440, 0.955061) | 0.001538 | 10.030057sec |
| [10, -19] | (0.975068, 0.950394) | 0.001878 | 10.012786sec |
| [-53, -51] | (0.965845, 0.932491) | 0.003513 | 10.006094sec |

2) Newtons method

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| problem | starting point | solution point | solution | time |
|  | [-1, 1] | (-0.000055, 0.000055) | 0.000000 | 0.140790sec |
| [10, -19] | (0.000029, -0.000055) | 0.000000 | 0.183752sec |
| [-53, -51] | (-0.000058, -0.000055) | 0.000000 | 0.194901sec |
|  | [-1, 1] | (NaN, NaN) | NaN | 10.156012sec |
| [10, -19] | (0.734925, -1.349914) | 8.205359 | 10.154434sec |
| [-53, -51] | (-5.443027, 1.161507) | 0.720563 | 10.153640sec |
|  | [-1, 1] | (0.999975, 0.999948) | 0.000000 | 0.315714sec |
| [10, -19] | (1.000011, 1.000020) | 0.000000 | 0.735325sec |
| [-53, -51] | (0.999987, 0.999973) | 0.000000 | 2.395584sec |

3) Quasi Newtons method (SR1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| problem | starting point | solution point | solution | time |
|  | [-1, 1] | (-0.000045, 0.000034) | 0.000000 | 0.179502sec |
| [10, -19] | (-0.000018, -0.000069) | 0.000000 | 0.203370sec |
| [-53, -51] | (-0.000047, -0.000034) | 0.000000 | 0.229288sec |
|  | [-1, 1] | (-2.994554, 1.261790) | 0.897164 | 10.014631sec |
| [10, -19] | (0.000295, -20.411101) | 7.873828 | 10.002526sec |
| [-53, -51] | (-160.258046, -0.872831) | 165219.740 | 10.009902sec |
|  | [-1, 1] | (0.077121, -0.010370) | 2.581746 | 10.015494sec |
| [10, -19] | (4.729235, 22.368344) | 41.722293 | 10.000772sec |
| [-53, -51] | (-43.509399, 1893.087706) | 5943.299384 | 10.008977sec |

4) Quasi Newtons method (BFGS)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| problem | starting point | solution point | solution | time |
|  | [-1, 1] | (-0.000108, 0.000071) | 0.000000 | 0.162348sec |
| [10, -19] | (0.000065, -0.000048) | 0.000000 | 0.208103sec |
| [-53, -51] | (-0.000113, -0.000073) | 0.000000 | 0.223335sec |
|  | [-1, 1] | (-13192.266833, 1.000075) | 0.452125 | 10.012004sec |
| [10, -19] | (3.000303, 0.500097) | 0.000000 | 1.993303sec |
| [-53, -51] | (NaN, NaN) | NaN | 10.170345sec |
|  | [-1, 1] | (0.999977, 0.999953) | 0.000000 | 0.505487sec |
| [10, -19] | (1.000014, 1.000027) | 0.000000 | 1.100282sec |
| [-53, -51] | (0.999989, 0.999975) | 0.000000 | 2.678658sec |

First of all Quasi Newtons method(BFGS) is the most efficient algorithm. It is easily confirmed in tables. In most case, it converge to almost minimum point(f = 0). It only diverge in just one case.   
 Steepest descent method always converge(Global convergence). But, as you can see in the time column, It takes much time than other function. Though it is terminated by time limit(10 seconds) in 5 cases, it doesn’t diverge. It means that it takes huge time near solution.  
 Newton’s method works well and also fast. I guess the reason is that our problem is not that difficult problem. I’m little suspicious about limitness of Newton’s method in lecture note, that it takes complex Hessian matrix calculation in every iteration. In my implementation, I first define the Hessian matrix before iteration and just put x,y value into defined Hessian matrix. So, it doesn’t need any complex Hessian matrix calculation in every iteration.  
 Quasi Newtons method(SR1) doesn’t work well. I implemented breaking down avoidance algorithm and used it in our code. Because of this algorithm, it doesn’t diverge to infinite(or NaN). But it doesn’t work efficiently and stuck in time limit(10seconds) frequently. I think it may work well in several cases, if it doesn’t use breaking down avoidance algorithm.  
And also, I couldn’t use the direct method that calculate H without inverse calculation of B. Because breaking down avoidance process need B. This could be another reason of low performance.

1. **Code Implementation**

**< Setting and Main function >**

|  |
| --- |
| %% Copyright (C) 2017 Young-Chul Yoon  %% All rights reserved.  %% Multivariate Optimization with derivative  clc;  clear all  base = [pwd, '\'];  addpath(genpath(base));  syms x y;  %% 3 functions  f = 2\*x^2+5\*y^2;  g = (1.5-x+x\*y)^2 + (2.25-x+x\*(y^2))^2 + (2.625-x+x\*(y^3))^2;  h = 100\*(y-x^2)^2 + 3\*(1-x)^2;  f\_set = {f, g, h};  %% Global setting  start\_pt = [-53, -51]';  f\_n = 3;  t\_limit = 10;  %% 3 functions for contour plot  x = -5\*abs(max(start\_pt)):0.1:5\*abs(max(start\_pt));  y = -5\*abs(max(start\_pt)):0.1:5\*abs(max(start\_pt));  [X, Y] = meshgrid(x,y);  Z1 = 2\*X.^2+5\*Y.^2;  Z2 = (1.5-X+X.\*Y).^2 + (2.25-X+X.\*(Y.^2))^2 + (2.625-X+X.\*(Y.^3))^2;  Z3 = 100\*(Y-X.^2)^2 + 3\*(1-X).^2;  cont\_set = {Z1, Z2, Z3};  %% Steepest Descent method  subplot(2,2,1);contour(X,Y,cont\_set{f\_n}, 10);  title(['Steepest Descent func',num2str(f\_n)]);  [sol, t] = steepest\_descent(f\_set{f\_n}, start\_pt, t\_limit);  x = sol(1);  y = sol(2);  fprintf('< Steepest Descent >\n');  fprintf('Solution point : (%f, %f)\n', x, y);  fprintf('Solution value : %f\n', eval(f\_set{f\_n}));  fprintf('Time spent : %fsec\n', t);  %% Newtons method  subplot(2,2,2);contour(X,Y,cont\_set{f\_n}, 10);  title(['Multi Newtons func',num2str(f\_n)]);  [sol, t] = multi\_newtons(f\_set{f\_n}, start\_pt, t\_limit);  x = sol(1);  y = sol(2);  fprintf('< Multi Newtons >\n');  fprintf('Solution point : (%f, %f)\n', x, y);  fprintf('Solution value : %f\n', eval(f\_set{f\_n}));  fprintf('Time spent : %fsec\n', t);  %% Quasi Newtons method (SR1)  subplot(2,2,3);contour(X,Y,cont\_set{f\_n}, 10);  title(['Quasi Newton (SR1) func',num2str(f\_n)]);  [sol, t] = quasi\_newton(f\_set{f\_n}, start\_pt, t\_limit, 'sr1');  x = sol(1);  y = sol(2);  fprintf('< Quasi Newton SR1 >\n');  fprintf('Solution point : (%f, %f)\n', x, y);  fprintf('Solution value : %f\n', eval(f\_set{f\_n}));  fprintf('Time spent : %fsec\n', t);  %% Quasi Newtons method (BFGS)  subplot(2,2,4);contour(X,Y,cont\_set{f\_n}, 10);  title(['Quasi Newton (BFGS) func',num2str(f\_n)]);  [sol, t] = quasi\_newton(f\_set{f\_n}, start\_pt, t\_limit, 'bfgs');  x = sol(1);  y = sol(2);  fprintf('< Quasi Newton BFGS >\n');  fprintf('Solution point : (%f, %f)\n', x, y);  fprintf('Solution value : %f\n', eval(f\_set{f\_n}));  fprintf('Time spent : %fsec\n', t); |

**< Steepest descent method >**

|  |
| --- |
| function [solution, t\_accum] = steepest\_descent(f, start\_pt, t\_limit)  %% Copyright (C) 2017 Young-Chul Yoon  %% All rights reserved.  %% Steepest descent method  temp\_st = tic;  pt = start\_pt;  der\_x = diff(f, 'x');  der\_y = diff(f, 'y');  grad = [der\_x, der\_y]';  t\_accum = toc(temp\_st);  pt\_set = [];  while true  pt\_set = [pt\_set, pt];  x = pt(1);  y = pt(2);  temp\_st = tic;  if norm(eval(grad)) < 1.0e-3 | t\_accum > t\_limit  solution = pt;  break;  end  p = -eval(grad);  step\_length = find\_step\_length(f, pt, grad, p);  pt = pt + step\_length\*p;  t\_accum = t\_accum + toc(temp\_st);  end  %% Draw converging process  hold on  plot(pt\_set(1, :), pt\_set(2, :), 'Color', 'r');  plot(pt\_set(1, end), pt\_set(2, end),'r\*')  hold off  end |

**< Newtons method >**

|  |
| --- |
| function [solution, t\_accum] = multi\_newtons(f, start\_pt, t\_limit)  %% Copyright (C) 2017 Young-Chul Yoon  %% All rights reserved.  %% Newtons method for multivariate problem  temp\_st = tic;  pt = start\_pt;  der\_x = diff(f, 'x');  der\_y = diff(f, 'y');  grad = [der\_x, der\_y]';  %% Define hessian matrix  der\_xx = diff(f, 'x', 'x');  der\_xy = diff(f, 'x', 'y');  der\_yx = diff(f, 'y', 'x');  der\_yy = diff(f, 'y', 'y');  H = [der\_xx, der\_xy;der\_yx, der\_yy];  t\_accum = toc(temp\_st);  pt\_set = [];  while true  pt\_set = [pt\_set, pt];  x = pt(1);  y = pt(2);  temp\_st = tic;  %% Stopping criterion  if norm(eval(grad)) < 1.0e-3 | t\_accum > t\_limit  solution = pt';  break;  end  p = - inv(eval(H))\*eval(grad);  %% Step length calculation  step\_length = find\_step\_length(f, pt, grad, p);  pt = pt + p.\*step\_length;  t\_accum = t\_accum + toc(temp\_st);  end  %% Draw converging process  hold on  plot(pt\_set(1, :), pt\_set(2, :), 'Color', 'r');  plot(pt\_set(1, end), pt\_set(2, end),'r\*')  hold off  end |

**< Quasi Newtons method (SR1, BFGS) >**

|  |
| --- |
| function [ solution, t\_accum ] = quasi\_newton(f, start\_pt, t\_limit, mode)  %% Copyright (C) 2017 Young-Chul Yoon  %% All rights reserved.  %% Quasi newtons method (SR1, BFGS)  temp\_st = tic;  der\_x = diff(f, 'x');  der\_y = diff(f, 'y');  grad = [der\_x, der\_y]';  %% Setting initial value  pt = start\_pt;  B = eye(2);  x = pt(1);  y = pt(2);  pt\_set = [];  t\_accum = toc(temp\_st);;  %% SR1  if strcmp(mode, 'sr1')  while true  temp\_st = tic;  pt\_set = [pt\_set, pt];  %% Stopping criterion  if norm(eval(grad)) < 1.0e-3 | t\_accum > t\_limit  solution = pt';  break;  end  %% Derive H from inverse of B  H = inv(B);  p = -H\*eval(grad);    prev\_pt = pt;  %% Step length calculation  step\_length = find\_step\_length(f, pt, grad, p);  s = step\_length\*p;  pt = pt + s;    x = prev\_pt(1);  y = prev\_pt(2);  old\_grad = eval(grad);  x = pt(1);  y = pt(2);  mv\_grad = eval(grad);    y\_k = mv\_grad - old\_grad;  prev\_B = B;  divisor = (y\_k - B\*s)'\*s;  B = B + (y\_k - B\*s)\*(y\_k - B\*s)'./divisor;  %% Strategy to avoid break down  if divisor < 0.5\*norm(s)\*norm(y\_k-B\*s) | divisor <= 0  B = prev\_B;  end  t\_accum = t\_accum + toc(temp\_st);  end  %% BFGS  elseif strcmp(mode, 'bfgs')  %% In BFGS, we directly derive H  H = B;  while true  temp\_st = tic;  pt\_set = [pt\_set, pt];  %% Stopping criterion  if norm(eval(grad)) < 1.0e-3 | t\_accum > t\_limit  solution = pt;  break;  end  p = -H\*eval(grad);    prev\_pt = pt;  %% Step length calculation  step\_length = find\_step\_length(f, pt, grad, p);  pt = pt + step\_length\*p;  x = prev\_pt(1);  y = prev\_pt(2);  old\_grad = eval(grad);  x = pt(1);  y = pt(2);  mv\_grad = eval(grad);    y\_k = mv\_grad - old\_grad;  s = pt - prev\_pt;    rho = 1./(y\_k'\*s);  H = (eye(2)-rho\*s\*y\_k')\*H\*(eye(2)-rho\*y\_k\*s')+rho\*s\*s';  t\_accum = t\_accum + toc(temp\_st);  end  else  disp('Not implemented method (Use sr1 or bfgs');  end  %% Draw converging process  hold on  plot(pt\_set(1, :), pt\_set(2, :), 'Color', 'r');  plot(pt\_set(1, end), pt\_set(2, end),'r\*')  hold off  end |

**< find step length algorithm >**

|  |
| --- |
| function [ step\_length ] = find\_step\_length(f, pt, grad, p)  %% Copyright (C) 2017 Young-Chul Yoon  %% All rights reserved.  alpha = 10;  c = 0.5;  rho = 0.5;  t\_accum = 0;  while true  temp\_st = tic;  x = pt(1);  y = pt(2);  right = eval(f) + c\*alpha\*eval(grad')\*p;  x = pt(1) + alpha\*p(1);  y = pt(2) + alpha\*p(2);  left = eval(f);    if left <= right | t\_accum > 0.5  step\_length = alpha;  return;  else  alpha = alpha\*rho;  end  t\_accum = t\_accum + toc(temp\_st);  end |