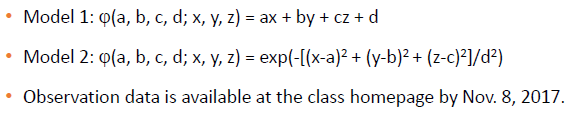
**Numerical Optimization**

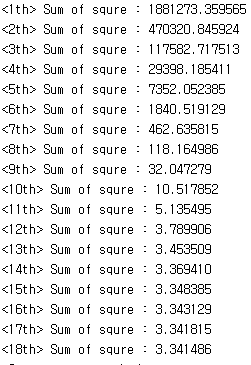
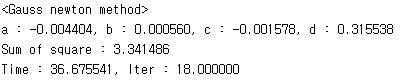
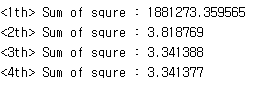
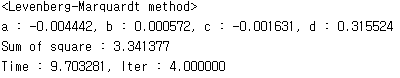
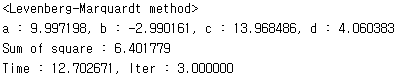
**Programming #6**

**Electrical Engineering & Computer Science**

**20171014 Young-Chul Yoon**

1. **Introduction  
   1) Goal of the assignment**- Implement Gauss-Newton’s and LM(Levenberg-Marquardt) method for the  
   following models and given observations  
     
   - Determine unknown parameters in the least square sense

**2) Description about implementation and setting**  
I used MATLAB R2014a(GIST free version). Because of version and toolbox problem, I couldn’t use many useful functions, which are available in upper version. So, I implemented Jacobian and Residue by myself. I used subs() function in MATLAB, which make it possible to selectively push the values in the symbolic variables. This made the the code little longer. Additionally, for better comparison of Gauss newton and Levenberg-Marquardt method, I didn’t use step-length computation in Gauss newton method. Also, visual graph is not available in this assignment, because of the high dimension(4).  
**1) Global setting**  
- Termination criterion : sum of square < 1.0e-3  
- Initial guess : [a, b, c, d] = [10, -3, 14, 4](func1), [-3, 8, 23, 9](func2)  
- Initial sum of square : infinite  
**2) Gauss-Newton**  
- Step length : 0.5 **3) Levenberg-Marquardt**  
- Initial lambda : 1  
- This code is similar to Gauss-Newton, except lambda computation part

1. **Result in program  
   (There are no visual graph here, because of the high dimension)  
   1)   
   < Gauss-Newton >  
   - Convergence process** **- Result** **< Levenberg-Marquardt >  
   - Converging process** **- Result**  
   **2)   
   < Gauss-Newton >  
   - Convergence process**Diverge **- Result**Diverge **< Levenberg-Marquardt >  
   - Converging process** **- Result**
2. **Analysis**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Function** | **Method** | **Solution** | **Sum of sq** | **Time** | **iter** |
|  | Gauss-Newt | [-0.0044, 0.00056,  -0.00158, 0.31554] | 3.3415 | 36.676 | 18 |
| L-M | [-0.0044, 0.00057,  -0.00163, 0.3155] | 3.3414 | 9.7 | 4 |
|  | Gauss-Newt | Diverge | | | |
| L-M | [9.997, -2.9902,  13.9685, 4.0604] | 6.40178 | 12.7 | 3 |

First of all, Levenberg-Marquardt(L-M) definitely perform better than Gauss-Newton method. In the simple function like func1, Gauss-Newton could found the solution without step-length computation. But, it took almost 4-times more time than L-M. In the complex function like func2, Gauss Newton couldn’t find the solution because of improper step-length. I tested using various initial assumption of [a,b,c,d], but it couldn’t find solution in most cases. In contrast, L-M always converge. This is the strength of L-M method. It doesn’t need any additional step-length calculation and assure the convergence.

1. **Code Implementation**

**< Setting and Main function >**

|  |
| --- |
| %% Copyright (C) 2017 Young-Chul Yoon  %% All rights reserved.  %% Gauss-Newton and Levenberg-Marquardt  clc;  clear all  base = [pwd, '\'];  addpath(genpath(base));  data = xlsread('observation\_data\_LSM\_LM\_2017.xlsx');  syms a b c d x y z;  %% 2 functions  f = a\*x + b\*y + c\*z + d;  g = exp(-((x-a)^2 + (y-b)^2 + (z-c)^2)/d^2);  func = g;  %{  [a, b, c, d, soq, t, iter] = gauss\_newton(func, data);  fprintf('<Gauss newton method>\n');  fprintf('a : %f, b : %f, c : %f, d : %f\n',a,b,c,d);  fprintf('Sum of square : %f\n', soq);  fprintf('Time : %f, Iter : %f\n\n', t, iter);  %}  [a, b, c, d, soq, t, iter] = l\_m(func, data);  fprintf('<Levenberg-Marquardt method>\n');  fprintf('a : %f, b : %f, c : %f, d : %f\n',a,b,c,d);  fprintf('Sum of square : %f\n', soq);  fprintf('Time : %f, Iter : %f\n', t, iter); |

**< Gauss Newton >**

|  |
| --- |
| function [a, b, c, d, sum\_sq, t, iter] = gauss\_newton(f, data)  %% Copyright (C) 2017 Young-Chul Yoon  %% All rights reserved.  %% Gauss-Newton  t\_st = tic;  x\_dat = data(:,1);  y\_dat = data(:,2);  z\_dat = data(:,3);  f\_dat = data(:,4);  m = length(data(:,1));  %% Initial guess  B = [-3, 8, 23, 9]';  sum\_sq = inf;  iter = 1;  while true  %% Calculate residue  r = [];  for i=1:m  arg = [x\_dat(i) y\_dat(i) z\_dat(i) B(1) B(2) B(3) B(4)];  temp\_r = f\_dat(i) - subs(f, {'x', 'y', 'z', 'a', 'b', 'c', 'd'}, arg);  r = [r;temp\_r];  end  r = eval(r);  %% Calculate jacobian  j = [];  for i=1:m  x = x\_dat(i);  y = y\_dat(i);  z = z\_dat(i);  arg = [x y z];  r\_f = subs(f, {'x', 'y', 'z'}, arg);  der\_a = diff(r\_f, 'a');  der\_a = subs(der\_a, {'a', 'b', 'c', 'd'}, B');  der\_b = diff(r\_f, 'b');  der\_b = subs(der\_b, {'a', 'b', 'c', 'd'}, B');  der\_c = diff(r\_f, 'c');  der\_c = subs(der\_c, {'a', 'b', 'c', 'd'}, B');  der\_d = diff(r\_f, 'd');  der\_d = subs(der\_d, {'a', 'b', 'c', 'd'}, B');  temp\_j = [der\_a, der\_b, der\_c, der\_d];  j = [j;temp\_j];  end  J = eval(j);    %% Compute new directioni  p = -inv(J'\*J)\*J'\*r;    %% Calculate new guess  B = B - 0.5\*p;    prev\_sum = sum\_sq;  %% Calculate sum of squares  sum\_sq = 0;  for i=1:m  sum\_sq = sum\_sq + r(i)^2;  end  fprintf('<%dth> Sum of squre : %f\n', iter, sum\_sq);    %% Terminate condition  if norm(sum\_sq - prev\_sum) < 1.0e-3  t = toc(t\_st);  a = B(1);  b = B(2);  c = B(3);  d = B(4);  return;  end  iter = iter + 1;  end |

**< Levenberg-Marquardt >**

|  |
| --- |
| function [a, b, c, d, sum\_sq, t, iter] = l\_m(f, data)  %% Copyright (C) 2017 Young-Chul Yoon  %% All rights reserved.  %% Levenberg-Marquardt  t\_st = tic;  x\_dat = data(:,1);  y\_dat = data(:,2);  z\_dat = data(:,3);  f\_dat = data(:,4);  m = length(data(:,1));  %% Initial guess  B = [10, -3, 14, 4]';  sum\_sq = inf;  sum = inf;  lambda = 1;  iter = 1;  while true  %% Calculate residue  r = [];  for i=1:m  arg = [x\_dat(i) y\_dat(i) z\_dat(i) B(1) B(2) B(3) B(4)];  temp\_r = f\_dat(i) - subs(f, {'x', 'y', 'z', 'a', 'b', 'c', 'd'}, arg);  r = [r;temp\_r];  end  r = eval(r);  %disp(r)  %% Calculate jacobian  j = [];  for i=1:m  x = x\_dat(i);  y = y\_dat(i);  z = z\_dat(i);  arg = [x y z];  r\_f = subs(f, {'x', 'y', 'z'}, arg);  der\_a = diff(r\_f, 'a');  der\_a = subs(der\_a, {'a', 'b', 'c', 'd'}, B');  der\_b = diff(r\_f, 'b');  der\_b = subs(der\_b, {'a', 'b', 'c', 'd'}, B');  der\_c = diff(r\_f, 'c');  der\_c = subs(der\_c, {'a', 'b', 'c', 'd'}, B');  der\_d = diff(r\_f, 'd');  der\_d = subs(der\_d, {'a', 'b', 'c', 'd'}, B');  temp\_j = [der\_a, der\_b, der\_c, der\_d];  j = [j;temp\_j];  end  J = eval(j);    prev\_B = B;  prev\_sum = sum;  while true  %% Compute direction  p = -inv(J'\*J + lambda\*eye(length(J(1,:))))\*J'\*r;  %% Calculate new guess  B = prev\_B - p;  sum = 0;  for i=1:m  arg = [B', [x\_dat(i), y\_dat(i), z\_dat(i)]];  sum = sum + subs(f, {'a', 'b', 'c', 'd', 'x', 'y', 'z'}, arg)^2;  end  sum = eval(sum);  %% When increased, continue loop  if sum > prev\_sum  lambda = lambda\*10;  %% When decreased, use lambda and newely derived assumptions  else  lambda = lambda/10;  break;  end    end  prev\_sum = sum\_sq;  %% Calculate sum of squares  sum\_sq = 0;  for i=1:m  sum\_sq = sum\_sq + r(i)^2;  end  fprintf('<%dth> Sum of squre : %f\n', iter, sum\_sq);    %% Terminate condition  if norm(sum\_sq - prev\_sum) < 1.0e-3  t = toc(t\_st);  a = B(1);  b = B(2);  c = B(3);  d = B(4);  return;  end  iter = iter + 1;  end |