COMPGV19: Tutorial 7

Table of Contents

Marta Betcke and Kiko Rul·lan

Exercise 3 Nonlinear Least Squares Minimization

Play with the options as indicated in the script to demonstate:

a) the differences between fitting an easy and a difficult moded,

b) fitting with a wrong model

c) robustness of LM as opposed to GN. Adding enough noise will break GN while LM will still converge to a solution


```
close all; clear all;
rng(1);
```

Generate the ground truth

```
%==============================
% Choose model to fit {'easy', 'difficult'}
model = 'easy';
%==============================
switch(lower(model))
  case 'easy'
    % Easy model
    % phi(x, t) = (x1 + x2*t^2)*exp(-t*x3)
    phi = @(x, t) (x(1) + t.^2.*x(2)).*exp(-t*x(3));
    JacobianPhi = @(x, t) [exp(-t*x(3)) t.^2.*exp(-t*x(3)) (x(1) +
 t.^2*x(2)).*(-t).*exp(-t*x(3))];
    xTrue = [3; 150; 2];

  case 'difficult'
    % Difficult model (the linear term in the paranthesis gets
 dominated)
    % phi(x, t) = (x1 + x2*t + x3*t^2)*exp(-t*x4)
    phi = @(x, t) (x(1) + t.*x(2) + t.^2.*x(3)).*exp(-t*x(4));
```

```
    JacobianPhi = @(x, t) [exp(-t*x(4)) t.*exp(-t*x(4)) t.^2.*exp(-
t*x(4)) (x(1) + t.*x(2) + t.^2.*x(3)).*(-t).*exp(-t*x(4))];
    xTrue = [3; -1; 150; 2];
end

disp(['xTrue: [' num2str(xTrue') ']'])

% Equispaced sampling points
nT = 200;
t = linspace(0,4,nT+1)'; t = t(2:end);
```

```
xTrue: [3  150    2]
```

Simulate the large residual problem using wrong model

```
%==============================
% right model
phiTrue = phi;
% wrong model (parameters of the perturbation chosen for the 'easy'
 model)
% phiTrue = @(x, t) (x(1) + t.^2.*x(2)).*exp(-t*x(3)) + (5 +
 2*sin(pi*t));
%==============================
```


Noiseless signal

```
signal = phiTrue(xTrue, t);

% Add noise
%==============================
%signal = signal + 1*randn(size(signal));                    %
 absolute additive Gaussian noise
signal = signal + 0.05*max(abs(signal))*randn(size(signal)); %
 relative to max amplitude additive GN
% signal = signal.*(1 + 0.01*randn(size(signal)););          %
 multiplicative GN

% Increasing noise will break GN (even if LSQR is used as a solver)
%signal = signal + 0.1*max(abs(signal))*randn(size(signal)); %
 relative to max amplitude additive GN
%==============================
```


Define the function

Auxiliary function

```
F.f = @(x) 0.5*sum((phi(x, t) - signal).^2);
F.r = @(x) phi(x, t) - signal;
F.J = @(x) JacobianPhi(x, t);
F.df = @(x) (F.J(x)')*F.r(x);
F.d2f = @(x) (F.J(x)')*F.J(x);
```

Eigenvalues of the Hessian at the solution

```
disp(['Eigenvalues of the approximation of the Hessian at the
 solution: ' num2str(eig(F.d2f(xTrue))',4)])
```

```
Eigenvalues of the approximation of the Hessian at the solution:
 0.1343      11.37   5.032e+04
```

Gauss-Newton line search

Initialisation

```
alpha0 = 1; %0.5;
tol = 1e-4;
maxIter = 200;
x0 = ones(length(xTrue), 1);

% Line Search parameters
lsOptsSteep.c1 = 1e-4;
lsOptsSteep.c2 = 0.9;
lsFun = @(x_k, p_k, alpha0) lineSearch(F, x_k, p_k, alpha0,
 lsOptsSteep);
% Gauss Newton call
disp('Gauss Newton:')
[xGN, fGN, nIterGN, infoGN] = descentLineSearch(F, 'gauss', lsFun,
 alpha0, x0, tol, maxIter);
disp(['xGN = [' num2str(xGN', 4) '],    fGN = ' num2str(fGN, 4) ',
 nIterGN = ' num2str(nIterGN)])
```



```
Gauss Newton:
xGN = [2.717      151.3      2.008],    fGN = 104,    nIterGN = 10
```


Levenberg-Marquardt trust region

Initialisation

```
Delta = 100;
eta = 0.1; %from interval (0, 0.25)
tol = 1e-4;
maxIter = 1000;
x0 = ones(length(xTrue), 1);
% Levenberg-Marguardt solver
disp('Levenberg-Marguardt:')
trFun = @(F, x_k, Delta) solverCMlevenberg(F, x_k, Delta, maxIter);
[xLM, fLM, nIterLM, infoLM] = trustRegion(F, x0, trFun, Delta, eta,
 tol, maxIter, 0, 0);
disp(['xLM = [' num2str(xLM', 4) '],    fLM = ' num2str(fLM, 4) ',
 nIterLM = ' num2str(nIterLM)])
```

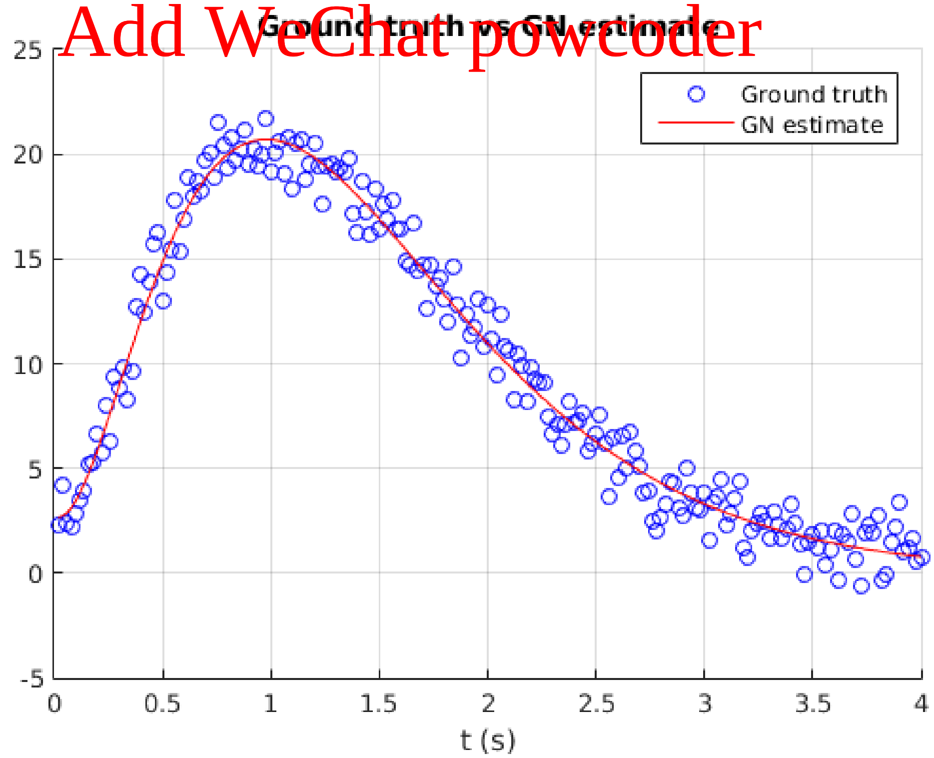
```
Levenberg-Marguardt:
In solverCMlevember: lambda = 0.5777, ||p|| = 50, Delta = 50, nIter =
 4
In solverCMlevember: lambda = 15.67, ||p|| = 12.5, Delta = 12.5, nIter
 = 6
```

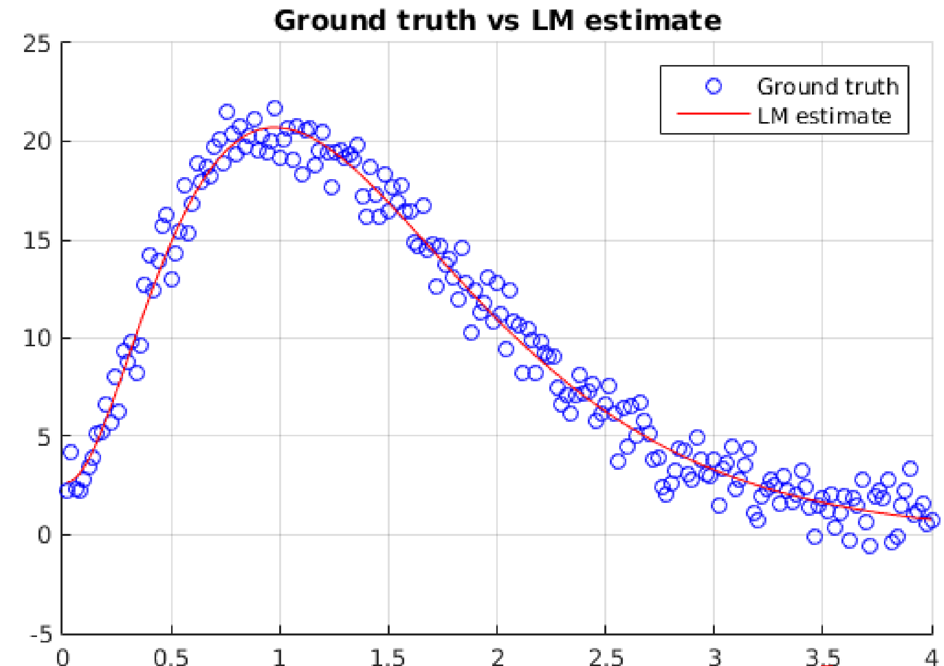
```
In solverCMlevember: lambda = 321.4, ||p|| = 3.125, Delta = 3.125,
 nIter = 6
In solverCMlevember: lambda = 2117, ||p|| = 0.7813, Delta = 0.7812,
 nIter = 5
In solverCMlevember: lambda = 1311, ||p|| = 0.7813, Delta = 0.7812,
 nIter = 4
In solverCMlevember: lambda = 521.4, ||p|| = 1.563, Delta = 1.562,
 nIter = 4
In solverCMlevember: lambda = 181.8, ||p|| = 3.125, Delta = 3.125,
 nIter = 4
In solverCMlevember: lambda = 117.2, ||p|| = 3.125, Delta = 3.125,
 nIter = 5
In solverCMlevember: lambda = 29.47, ||p|| = 6.25, Delta = 6.25, nIter
 = 5
In solverCMlevember: lambda = 10.88, ||p|| = 6.25, Delta = 6.25, nIter
 = 5
In solverCMlevember: lambda = 4.096, ||p|| = 12.5, Delta = 12.5, nIter
 = 4
In solverCMlevember: lambda = 1.288, ||p|| = 25, Delta = 25, nIter = 4
In solverCMlevember: lambda = 0.7564, ||p|| = 25, Delta = 25, nIter =
 4
In solverCMlevember: lambda = 0.4214, ||p|| = 25, Delta = 25, nIter =
 3
In solverCMlevember: lambda = 0.169, ||p|| = 25, Delta = 25, nIter = 3
In solverCMlevember: lambda = 0, ||p|| = 24.23, Delta = 25, nIter = 2
In solverCMlevember: lambda = 0, ||p|| = 1.24, Delta = 25, nIter = 2
In solverCMlevember: lambda = 0, ||p|| = 0.0003157, Delta = 25, nIter
 = 2
xLM = [2.717      151.4      2.008],   fLM = 104,   nIterLM = 18
```

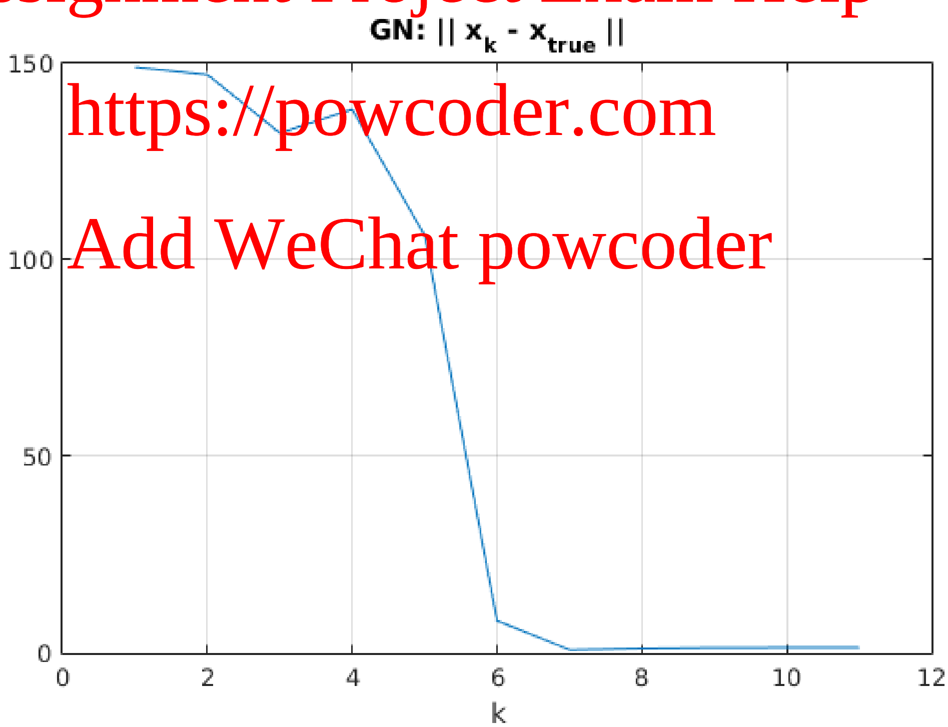

Plot the results

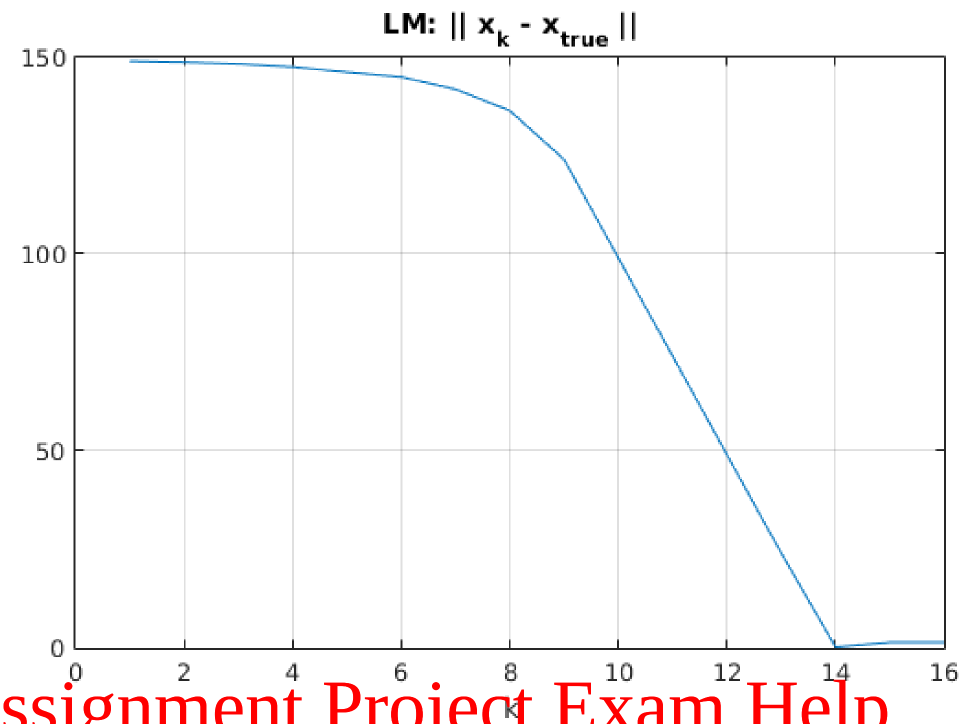
```
position1 = [100 800 600 400];
position2 = [100 100 600 400];
position3 = [800 800 600 400];
position4 = [800 100 600 400];
% Computed signal
estGN = phi(xGN, t);
estLM = phi(xLM, t);

% Plot - Gauss estimation
figure;
hold on;
plot(t, signal, 'ob');
plot(t, estGN, '-r');
title('Ground truth vs GN estimate');
xlabel('t (s)');
legend('Ground truth', 'GN estimate');
grid on;
set(gcf, 'pos', position1);

% Plot - Levenberg-Marquardt estimation
figure;
```

```
hold on;
plot(t, signal, 'ob');
plot(t, estLM, '-r');
title('Ground truth vs LM estimate');
xlabel('t (s)');
legend('Ground truth', 'LM estimate');
grid on;
set(gcf, 'pos', position2);

% Plot convergence - Gauss convergence
figure;
plot(sqrt(sum((infoGN.xs - repmat(xTrue, 1,
 size(infoGN.xs,2))).^2,1)));
hold on;
title('GN: || x_k - x_{true} ||');
xlabel('k');
grid on;
set(gcf, 'pos', position3);

% Plot convergence - Levemberg convergence
figure;
plot(sqrt(sum((infoLM.xs - repmat(xTrue, 1,
 size(infoLM.xs,2))).^2,1)));
hold on;
title('LM: || x_k - x_{true} ||');
xlabel('k');
grid on;
set(gcf, 'pos', position4);
```


Ground truth vs LM estimate
Ground truth
LM estimate
t (s)


GN: || x_k - x_{true} ||
k



============================= Subfunctions =============================

solverCMlevenberg.m

Computes the Levenberg-Marquardt direction for the least squares minimisation


```
function p = solverCMlevenberg(F, x_k, Delta, maxIter)
% SOLVERCMLEVENBERG Levenberg-Marguardt solver for constraint
 trustregion problem
%function p = solverCMlevenberg(F, x_k, Delta, maxIter)
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
%   - d2f: Hessian handler
%   - J: handler for the jacobian of r
%   - r: residual function
% x_k: current iterate
% Delta: upper limit on trust region radius
% maxIter: maximum number of iterations
%
% OUTPUT
% p: step (direction times lenght)
%
% Based on Algorithm 4.3 in Nocedal Wright
% Copyright (C) 2017 Marta M. Betcke, Kiko Rul·lan

% Initialise
lambda = eps;
```

```
nIter = 0;
% Compute the QR factorisation at x_k
J = F.J(x_k);
r = F.r(x_k);
[m, n] = size(J);
[Q_ini, R_ini] = qr(J); % Q: m x m orthogonal, R: m x n upper
 triangular
%maxEigenval = max(eig(R'*R)); % limit the value of lambda

p = 0;
while (nIter < maxIter && abs(norm(p)-Delta) > 1e-8 && lambda > 0)
%while (nIter < maxIter && abs(norm(p)-Delta)/Delta > 0.05 && (lambda
 > 0))
    % Update the Cholesky factorisation
    Q = Q_ini;
    R = R_ini;

    for i = 1:n
      % Construct i-th row of sqrt(lambda)*I
      row = zeros(1, n);
      row(i) = sqrt(lambda);
      % Insert i-th row of sqrt(lambda)*I at position m+i below R and
 update QR decomposition
      [Q, R] = qrinsert(Q, R, m+i, row, 'row');
    end

    % Solve (R'*R) p = (-J'*r) for L-M direction p
    p = R\(R'\(-J'*r));
    % Compute q (eigenvector, see description of Algorithm 4.3 Nocedal
 Wright)
    q = R'\p;
    % Update lambda (the Lagrange multiplayer for the trust region
 problem
    % and the shift to make J'*J spd). Note that J'*J is at least
 positive semidefinite
    % so any positive shift will make it spd.
    %lambda = max(0, lambda + (norm(p)./norm(q)).^2*(norm(p) - Delta)/
Delta);
    lambda = max(0, lambda + (sum(p.^2)./sum(q.^2))*(norm(p) - Delta)/
Delta);
    % if lambda == 0, GNstep = true; end

    nIter = nIter+1;
end

% GN step
if lambda == 0
  R = R_ini;
  % Solve (R'*R) p = (-J'*r) for GN direction p
  p = R\(R'\(-J'*r));
  nIter = nIter+1;
end
```

```
disp(['In solverCMlevember: lambda = ' num2str(lambda, 4) ', ||p||
 = ' num2str(norm(p), 4) ', Delta = ' num2str(Delta, 4) ', nIter = '
 num2str(nIter) ]);

%% Debug GN step
%[p_GN, flagLSQR,relresLSQR,iterLSQR,resvecLSQR,lsvecLSQR] = lsqr(J, -
r, 1e-6, 1000);
%difP = p_GN - p;
%disp(['lambda = ' num2str(lambda, 4) ', norm(p) = ' num2str(norm(p),
 4) ', Delta = ' num2str(Delta, 4) ', difP = ' num2str(norm(difP)) ]);
%pause
```

descentLineSearch.m

Includes the Gauss-Newton solver

```
function [xMin, fMin, nIter, info] = descentLineSearch(F, descent, ls,
 alpha0, x0, tol, maxIter)
% DESCENTLINESEARCH Wrapper function executing descent with line
 search
% [xMin, fMin, nIter, info] = descentLineSearch(F, descent, ls,
 alpha0, x0, tol, maxIter)
%
% INPUTS
% F: structure with fields
%   - f: function handler
%   - df: gradient handler
%   - d2f: Hessian handler
%   - J: Jacobian handler (Gauss-Newton method)
%   - r: residual handler (Gauss-Newton method)
% descent: specifies descent direction {'steepest', 'newton', 'newton-
cg', 'newton-reg', 'gauss'}
% alpha0: initial step length
% x0: initial iterate
% tol: stopping condition on relative error norm tolerance
%      norm(x_Prev - x_k)/norm(x_k) < tol;
% maxIter: maximum number of iterations
%
% OUTPUTS
% xMin, fMin: minimum and value of f at the minimum
% nIter: number of iterations
% info: structure with information about the iteration
%   - xs: iterate history
%   - alphas: step lengths history
%
% Copyright (C) 2017  Marta M. Betcke, Kiko Rullan

% Initialization
nIter = 0;
normError = 1;
```

```
x_k = x0;
info.xs = x0;
info.alphas = alpha0;

% Loop until convergence or maximum number of iterations
while (normError >= tol && nIter <= maxIter)

  % Increment iterations
    nIter = nIter + 1;

    % Compute descent direction
    switch lower(descent)
      case 'steepest'
        p_k = -F.df(x_k); % steepest descent direction
      case 'newton'
        p_k = -F.d2f(x_k)\F.df(x_k); % Newton direction
      case 'newton-cg'
        % Conjugate gradient method
        df_k = F.df(x_k); % gradient
        B_k = F.d2f(x_k); % hessian
        eps_k = min(0.5, sqrt(norm(df_k)))*norm(df_k);
        z_j = 0*df_k;
        r_j = df_k;
        d_j = -df_k;
        stopCond = false;
        nIterCG = 0;
        while (~stopCond && nIterCG < maxIter/10)
          if (d_j)'*B_k*d_j < 0
            if nIterCG == 0; p_k = d_j;
            else p_k = z_j; end;
            stopCond = true;
            eps_k = 0;
          end
          norm_r_j = r_j'*r_j;
          a_j = norm_r_j/(d_j'*B_k*d_j);
          z_j = z_j + a_j*d_j;
          r_j = r_j + a_j*B_k*d_j;
          if sqrt(r_j'*r_j) < eps_k; stopCond = true; end;
          b_j = r_j'*r_j/norm_r_j;
          d_j = -r_j + b_j*d_j;
          p_k = z_j;
          nIterCG = nIterCG + 1;
        end
      case 'newton-reg' % Newton regularised for non-linear equations
        p_k = solverCMlevenberg(F, x_k, 0.1, maxIter/10);
      case 'gauss' % Gauss-Newton algorithm
        % Solve min_p || J(x_k) p + r(x_k) ||
        % J(x_k)
        J_k = F.J(x_k);
        %% Solve normal euqations (squares condition number)
        %p_k = -(J_k'*J_k)\(J_k'*F.r(x_k));
        % Solve linearised least squares problem with LSQR
        [p_k, flagLSQR,relresLSQR,iterLSQR,resvecLSQR,lsvecLSQR] =
 lsqr(J_k, -F.r(x_k), 1e-6, 1000);
```

```
    end

    % Call line search given by handle ls for computing step length
    alpha_k = ls(x_k, p_k, alpha0);

    % Update x_k and f_k
    x_k_1 = x_k;
    x_k = x_k + alpha_k*p_k;

    % Compute relative error norm
    normError = norm(x_k - x_k_1)/norm(x_k_1);

    % Store iteration info
    info.xs = [info.xs x_k];
    info.alphas = [info.alphas alpha_k];

end

% Assign output values
xMin = x_k;
fMin = F.f(x_k);
```


Published with MATLAB® R2015a