Numerical Optimisation: Assignment 7

Student Number:13064947

February 2018

1 Exercise 1

Submitted via Cody Coursework

2 Exercise 2

Optional zero mark exercise - come back to it perhaps. Although exercise 3, does require the plugging in of this function to the trust region function and the additional calculation of the Hessian anyway.

3 Exercise 3

Consider a model $\phi(x_1, x_2, x_3; t) = (x_1 + x_2 t^2)e^{-x_3 t}$ with parameters (x_1, x_2, x_3). Simulate the measurements sampling this model for a fixed choice of parameters $(x_1, x_2, x_3) = (3, 150, 2)$ at 200 equi-spaced points in $t_i \in (0, 4]$ and adding gaussian noise $n(t_i) \sim \mathcal{N}(0, \sigma^2)$, where $\sigma = 0.05 \max_{t_i} |\phi(t_i)|$.

$$\tilde{\phi}(x_1, x_2, x_3; t_j) = (x_1 + x_2 t^2)e^{-x_3 t} + n(t_j)$$

For completeness the code stub for the simulation of the model is given below:

```
%Generate the simulated values with the known parameters

t = linspace(0,4,200);

Phi = @(x) (x(1)+x(2)*(t.^2)).*exp(-x(3)*t);

%Clearly the model needs to learn these parameters
unnoised = Phi([3,150,2]);

sigma = 0.05*max(unnoised);
noise = normrnd(0,sigma, [1,200]);
%these will be targets used for residuals etc.
y = unnoised + noise;
```

3.1 Formulate the least-squares problem for fitting the model ϕ and derive its Jacobian.

The least squares formulation is

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x)$$

where $r_j(x)$ is our j-th residual. In this particular example:

$$r_j(x) = \phi(x, t_j) - y_j$$

with $\phi(x,t)$ being given above and x and t being vector values and t_j, y_j being the j-th instance.

In full

$$r_j(x) = (x_1 + x_2 t_j^2)e^{-x_3 t_j} - y_j$$

For the Jacobian

$$J(x) = \left[\frac{\partial r_j}{\partial x_i}\right]_{ij}$$

Thus in this example the Jacobian is made up of the following matrix (using column vectors). i.e. (The matrix is of dimension (200,3) in our case, with time for our rows and parameters for our columns).

$$J(x;t) = \left[\frac{\partial r}{\partial x_1}, \frac{\partial r}{\partial x_2}, \frac{\partial r}{\partial x_3}\right]$$

Taking the partials we end up with the following column vectors with t being the time vector. Which is also shown in the code-snippet for the jacobian for Gauss-Newton below.

$$J(x;t) = \left[e^{-x_3 t}, t^2 e^{-x_3 t}, -t(x_1 + x_2 t^2)e^{-x_3 t}\right]$$

3.2 Estimate the parameters (x_1, x_2, x_3) from your simulated measurements

Specify all the relevant parameters and explain the results. Visualise the fit by plotting the estimated signal versus the measurements.

3.2.1 Gauss-Newton

For Gauss-Newton we need a vector of residuals, the value of our function f, (in this case the function to be minimised is a sum of squared residuals). We also require the Jacobian calculated above, as well as the gradient. I find it helps to keep track of dimensionality. In this case with 200 data points the residuals will be $(200, 1)$, the Jacobian $(200, 3)$ and the gradient $(3, 1)$. Clearly the 'guessed' parameter values will be passed into these functions each time. The 200 coming from the number of datapoints and the 3 coming from the three parameters we are able to play with.

In this case we are doing line search so any comments are effectively a copy of previous assignments with strong wolf line search etc. The parameters for replication are listed below for repeatability purposes. I used two sets of iterations 100 and 200. I will comment on this later. It is not really needed here. But was needed for Levenberg-Marquardt using the trust region methodology. For all optimisations (GN or LM) I started at the intial guess $x0 = (1, 1, 1)$.

- Maximum iterations 100 and 200 (for GN and LM)
- Tolerance 1e-10. (For GN and LM)
- Alpha0 = 1.
- Optimisation parameters - c1 = 1e-04, c2 = 0.5

For completeness I have also included a very short code snippet showing the specific parts of the implementation needed for Gauss-Newton.

```
%Function handlers for residual, function value
%jacobian and gradient

%Residuals
F.r = @(x) (Phi(x)-y)';

%Function value
F.f = @(x) 0.5*sum(F.r(x).^2);

%Jacobian
F.J = @(x) [exp(-x(3)*t)' ...
            (t.^2.*exp(-x(3)*t))' ...
            (-t.*(x(1)+x(2)*t.^2).*exp(-x(3)*t))'];

%Derivative using Jacobian
F.df = @(x) F.J(x)'*F.r(x);
```

3.2.2 Levenberg-Marquardt

The difference as regards the Levenberg-Marquardt implementation was the use of a trust region methodology. The solver was provided '@solverCMlevenberg', so this was simply plugged into trust region code from previous assignments. There are minor parameter differences for the trust region method given below, all other parameters are the same as for GN.

- Eta = 0.1. (Step acceptance, relative progress threshold).
- Delta = 1 (Trust region radius).

The other key difference for Levenberg-Marquardt is of course the need to derive the Hessian. This is achieved using the following equation.

$$\nabla^2 f(x) = J(x)^T J(x) + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x)$$

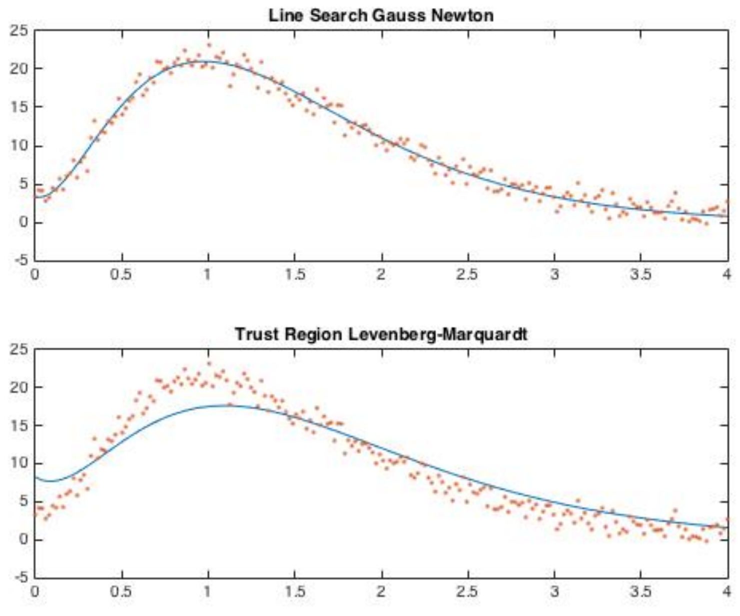
Again I show a very small code snippet showing the Hessian implementation using the equation above. I haven't shown the workings for the second derivatives of $r_j(x)$, however they can be seen in the code.

```
%Get the hessian for LM
%This is perhaps not so readable and a little too cute...
%in this particular example we would add 200 - 3x3 matrices
%and then add to the first part of the equation in the book

%However in this case all bar the bottom row is zero...so
%just do that row: add them all and plonk them on the bottom
%row of 3x3 matrix of zeros before finally adding

H2 = @(x) vertcat(zeros(2,3), ...
                   sum([(Phi(x)-y).*-t.*exp(-x(3)*t); ...
                       (Phi(x)-y).*(-t.^3).*exp(-x(3)*t);...
                       (Phi(x)-y).*(t.^2).*exp(-x(3)*t)],2)');

H1 = @(x) F.J(x)'*F.J(x);

%The final Hessian
F.d2f = @(x) (H1(x) + H2(x));
```

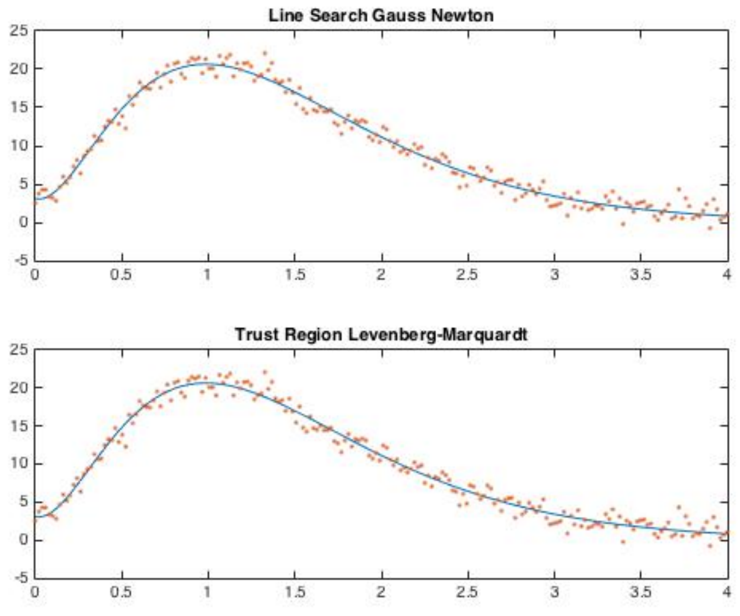
3.2.3 Display and explain results

I carelessly neglected to label the axes. So for clarity on all charts the x-axis is time. Recall that $t \in (0, 4]$, with 200 data points. The dotted values on each chart are the simulated values. The line through those dotted values is the result after calculating the co-efficients x_1, x_2, x_3.

The main result is how the optimisers converged. Gauss-Newton with line search was very quick and converged in only 15 iterations. As can be seen from fig 1a, after running both GN and LM for 100 iterations, Levenberg-Marquardt had still not fully converged. This can be seen be carefully examining the fit on the chart. After a rerun of 200 iterations fig 1b, we can see that Levenberg-Marquardt has by now converged (in fact it took 164 iterations). Note: The original incorrect solver took over 400 iterations. These results are with the new solver provided on (13/03). Given that LM was passed the Hessian (and hence curvature information), I am somewhat surprised by this result. However, my tolerances were tight 1e-10. It has been the case that trust-region methods have been somewhat 'slow and steady' and smoother, but longer in their convergence properties. At least using the parameters I have used so far in the assignments.



(a) GN v ML - 100 iterations



(b) GN v ML - 200 iterations