Assignment 3

Callum Lau, 19102521

Numerical Optimisation

March 28, 2020

1 EXERCISE 1

1.1 (b)

The code for updating the inverse Hessian H_k is given below:

```
    switch lower(descent)
      case 'bfgs'
        s_k = x_k - x_k_1;
        y_k = F.df(x_k) - F.df(x_k_1);

        if nIter == 1
          % Update initial guess H_0.
          disp(['Rescaling H0 with ' num2str((s_k'*y_k)/(y_k'*y_k)) ])
          H_k = @(x) (s_k'*y_k)/(y_k'*y_k) * x;
        end

        if nIter > 1
            rho_k = 1/(y_k'*s_k);
            H_k = @(x) (I - rho_k*s_k*y_k')*H_k((I -  rho_k*y_k*s_k')*x) + ...
                rho_k*s_k*s_k'*x;
        end

        if extractH
            % Extraction of H_k as handler
            info.H{length(info.H)+1} = H_k;
        end
    end
```

The key method which makes the implementation efficient is using the H_ks exclusively as *function handles* which can compute the value of $H_k x$ for some vector x instead of storing the matrix itself. As a result there are no matrix-matrix computations of order $O(N^3)$ and only matrix-vector computations of order $O(N^2)$. Secondly, each matrix used in each function handle is only instantiated in memory when it is called and so the memory complexity is only $O(N)$ instead of $O(KN)$ for K iterations of the algorithm (the case when the matrix are stored every iteration). Therefore the implementation is both speed and memory efficient.

2 EXERCISE 3

2.1 (a)

The parameters used for the BFGS and SR1 methods are given below:

	$\boldsymbol{x}_0$	`tolerance`	`maxIter`	α_0	`c`$_1$ (1st W.C.)	`c`$_2$ (2nd W.C.)
`BFGS`	$(10,10)^T$	`1e-10`	200	1	`1e-2`	0.2

	$\boldsymbol{x}_0$	`tolerance`	`maxIter`	Δ	η
`SR1`:	$(10,10)^T$	`1e-10`	200	1	0.1

The values for $\boldsymbol{x}_0$, `tol` and `maxIter` were chosen so that we could have a fair comparison of the performance between the two methods. As suggested, since computational evidence suggests it is more economical to perform fairly innacurate line search for the BFGS method, a value of `1e-2` was chosen for the `c`$_1$ (sufficient decrease condition) constant. Furthermore a value of $\alpha_0 = 1$ is important since the theory says that this step length will eventually be accepted, ensuring superlinear convergence. The trajectories of the iterates in the BFGS (red) and SR-1 (green) methods are plotted below:

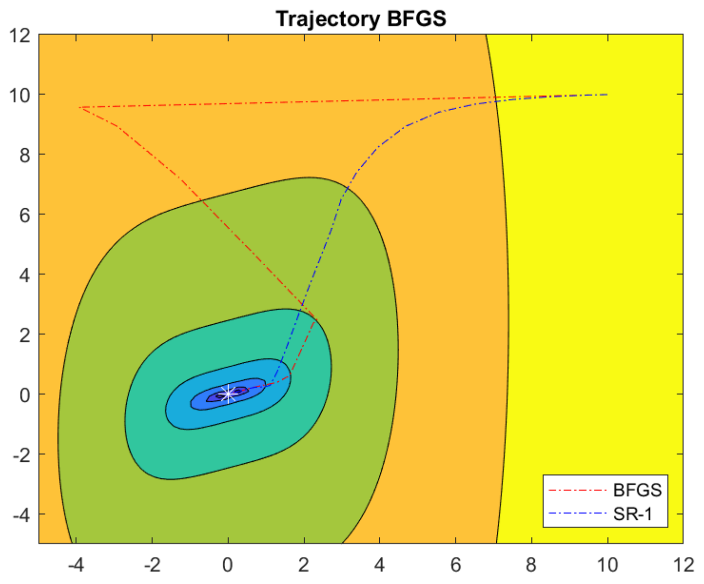
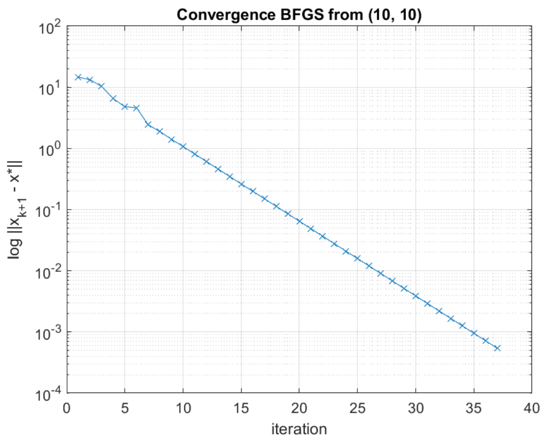
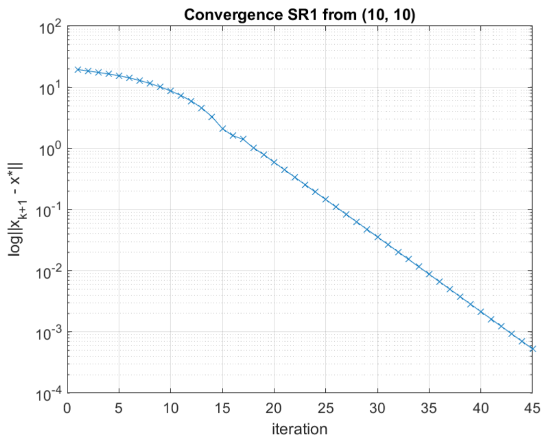


Figure 1: Trajectories of BFGS and SR1 methods

Observe that SR-1 seems to take a more direcy/smoother route to the minimum than BFGS. Both algorithms however take a similar number of iterations to reach the minimum. BFGS took 37 iterations whereas SR-1 took 45 for this problem. The performance of these algorithms applied to this problem and with these starting conditions is therefore very similar. For more precise analysis, we can analyse the convergence rates of the algorithms. Plots of the convergence of the iterates are provided below:



(a) BFGS $||x_{k+1} - x^*||$



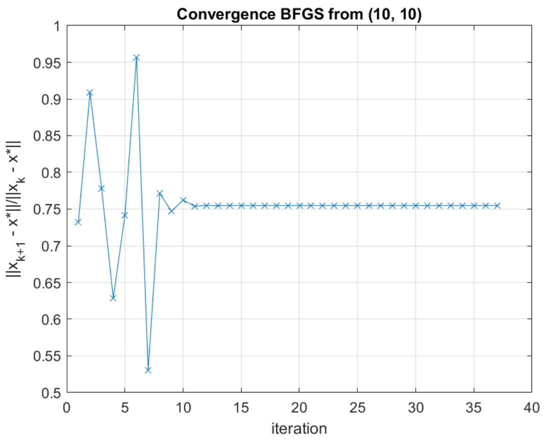
(b) SR1 $||x_{k+1} - x^*||$

Figure 2: Convergence of iterates on log scale

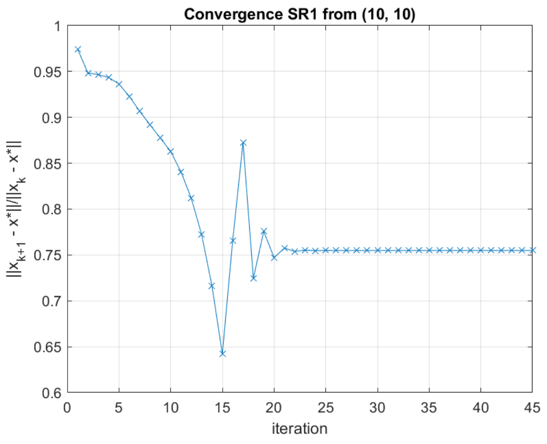
The values $||x_{k+1} - x^*||$ are plotted on a log scale, and show a straight line slope. This indicates that the rate of convergence is **linear**. The convergence of the iterates can also be analysed by plotting the values:

$$\frac{||x_{k+1} - x^*||}{||x_k - x^*||} \tag{1}$$

which are given below:



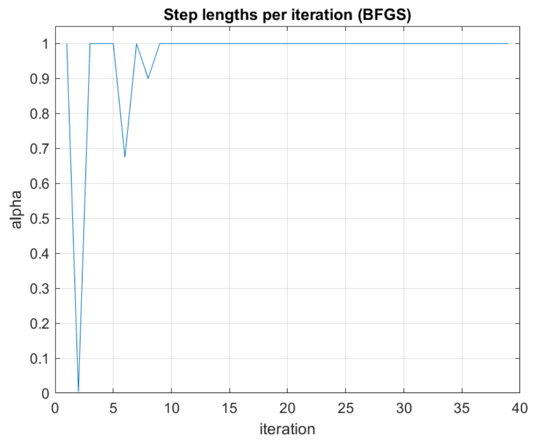
(a) BFGS $\frac{||x_{k+1}-x^*||}{||x_k-x^*||}$



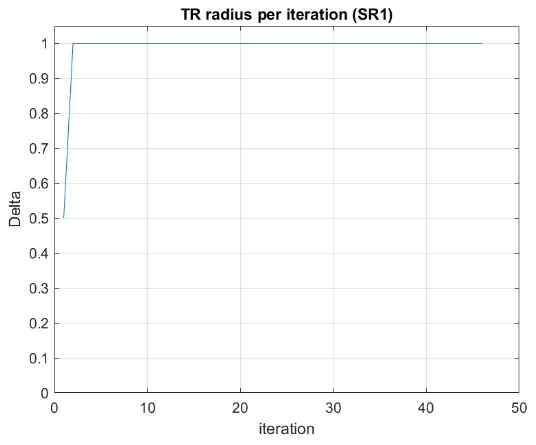
(b) SR1 $\frac{||x_{k+1}-x^*||}{||x_k-x^*||}$

Figure 3: Convergence of iterates

We observe that for both algorithms the ratio tends to around $0.75 \in (0, 1]$ which also indicates the practical rate of convergence for BFGS and SR1 is **linear**. This differs from the theory which states that the theoretical convergence rate of BFGS is *superlinear* and the theoretical convergence rate of SR-1 is $n + 1$-step *superlinear*. To understand why the practical convergence rate is slower than the theoretical we provide plots of the step lengths α_k (for BFGS) and trust region radii Δ_k (for SR-1), as well as the condition number of the Hessians per iteration:

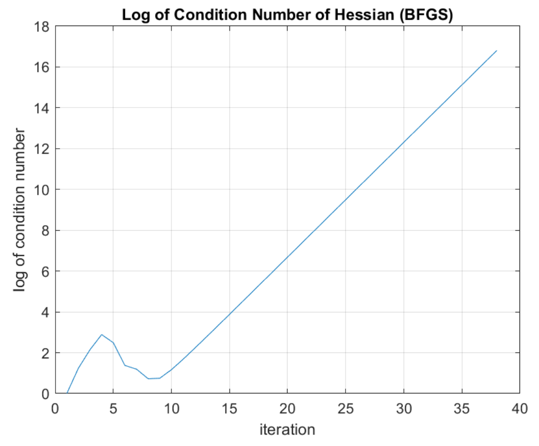


(a) BFGS step length α_k

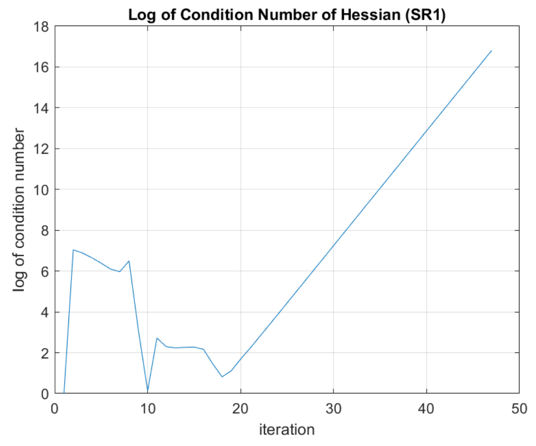


(b) SR1 trust region radii Δ_k

Figure 4: Step lengths and radii



(a) BFGS condition number



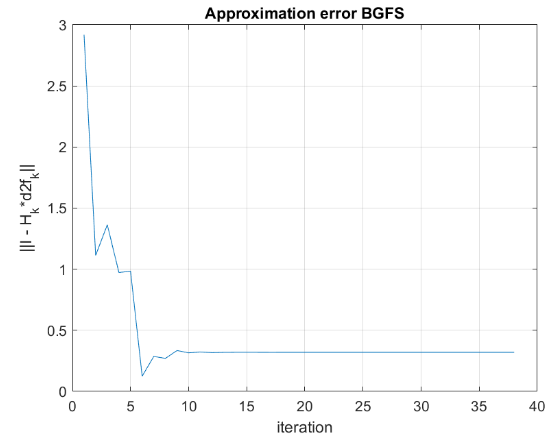
(b) SR1 condition number

Figure 5: Condition number of Hessians per iteration

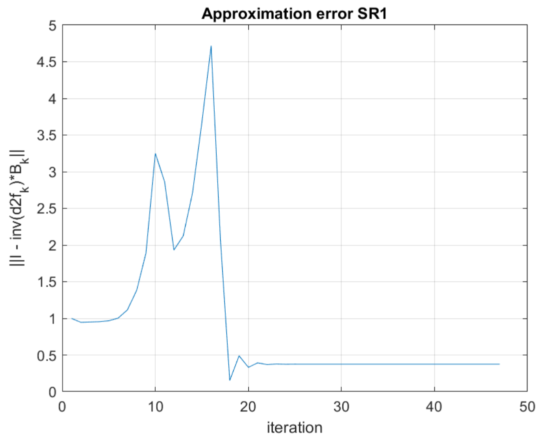
We see that as the iterations increase and the iterates x_k get closer to the minimum x^*, the Condition Number of the Hessians for both algorithms increase hugely - meaning they become increasingly ill-conditioned. Secondly we note that the step lengths remain constant and the trust region becomes inactive for both algorithms for a long time before convergece, suggesting the proximity of the iterates to the minimum. The Hessian therefore is describing the local curvature near the solution x^*, which in the case of large condition number, means that the curvature along one direction is much larger than another - i.e. the trajectory is entering a 'long valley' shape. This makes it hard to the iterates to convergence to the true solution as they continually overstep the true gradient direction, and the ill-conditioning of the matrix makes it more likely that numerical errors will make the descent directions even less accurate. In summary then the convergence rate becomes much lower than suggested by theory.

2.2 (b)

The errors between the approximation of the Hessians H_k^{BFGS} , $[B_k^{SR1}]^{-1}$ and their true value $\nabla^2 f(x_k)$ at iterations k are plotted below -



(a) $\{||I - H_k^{\text{BFGS}} \nabla^2 f(x_k)||_2\}_{k \geq 0}$



(b) $\{||I - (\nabla^2 f(x_k))^{-1} B_k^{\text{SR1}}||_2\}_{k \geq 0}$

Figure 6: Hessian approximation error

The key qualititave detail is that the poor approximation for the Hessian decreases eventually for a certain number of iteration, ≈ 5 in the case of BFGS and ≈ 20 in the case of SR1, before stabilising and never decreasing to zero, which is what we'd expect when the approximations become nearly exact. This is likely because of the linear convergence as discussed before and the poor conditioning of the Hessian approximations.

3 Exercise 6

3.1 (a)

In the model we have that

$$n(t_j) = \varphi(\boldsymbol{x}; t_j) - \tilde{\varphi}(\boldsymbol{x}; t_j) \tag{2}$$

where $n(t_j) \sim \mathcal{N}(0, \sigma^2) \doteq g_\sigma$. Then the likelihood for the observations $\tilde{\varphi}(t_j), j = 1, 2, \ldots, 200$ (no longer parametrised by $\boldsymbol{x}$) given parameter vector $\boldsymbol{x}$ is -

$$\pi(\tilde{\varphi}(t)|\boldsymbol{x}) = \prod_{j=1}^{200} g_\sigma(n(t_j)) = \prod_{j=1}^{200} g_\sigma(\varphi(\boldsymbol{x}; t_j) - \tilde{\varphi}(t_j)) \tag{3}$$

The *maximum likelihood* estimate for $\boldsymbol{x}$ is given by Bayes rule with uniform prior -

$$\begin{aligned}
\boldsymbol{x}_{\text{ML}} &= \max_{\boldsymbol{x}} \pi(\boldsymbol{x}|\tilde{\varphi}(t)) = \max_{\boldsymbol{x}} \pi(\tilde{\varphi}(\boldsymbol{x}; t)|\pi(\boldsymbol{x})) && (4)\\
&= \max_{\boldsymbol{x}} \prod_{j=1}^{200} g_\sigma(\varphi(\boldsymbol{x}; t_j) - \tilde{\varphi}(t_j)) && (5)\\
&= \max_{\boldsymbol{x}} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2} \sum_{j=1}^{200} (\varphi(\boldsymbol{x}; t_j) - \tilde{\varphi}(t_j))^2\right) && (6)
\end{aligned}$$

$$= \min_{\boldsymbol{x}} \frac{1}{2} \sum_{j=1}^{200} (\varphi(\boldsymbol{x}; t_j) - \tilde{\varphi}(t_j))^2 \doteq \min_{\boldsymbol{x}} \frac{1}{2} \sum_{j=1}^{200} r_j^2(\boldsymbol{x}) \tag{7}$$

which is precisely the form of the least-squares model for residuals r_j and model φ. The Jacobian is given by

$$J(\boldsymbol{x}) = \left[\frac{\partial r_j}{\partial x_i}\right]_{ij} = \begin{bmatrix} \nabla r_1^T(\boldsymbol{x}) \\ \dots \\ \nabla r_{200}^T(\boldsymbol{x}) \end{bmatrix} \tag{8}$$

where

$$\nabla r_j(\boldsymbol{x}) = \begin{bmatrix} \exp(-x_3 t_j) \\ t_j^2 \exp(-x_3 t_j) \\ -t_j(x_1 + x_2 t_j^2) \exp(-x_3 t_j) \end{bmatrix} \tag{9}$$

3.2 (b)

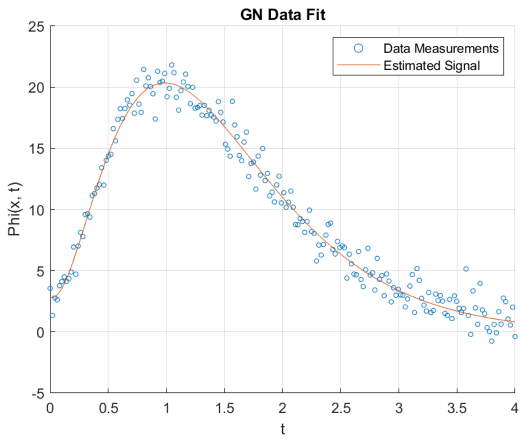
After simulating the data $\tilde{\varphi}(t_j)$ for $t \in (0, 4]$ the Jacobian and residual functions were formed as above. Both the Gauss-Newton and Levenberg-Marquardt methods were applied to the non-linear least squares problem of finding the parameter vector $\boldsymbol{x}$ which minimises equation (6). The parameters used in both methods are provided below.

	$\boldsymbol{x}_0$	`tolerance`	`maxIter`	α_0	`c`$_1$ (1st W.C.)	`c`$_2$ (2nd W.C.)
`Gauss-Newton`	$(2,2,2)^T$	`1e-10`	200	1	`1e-4`	0.1

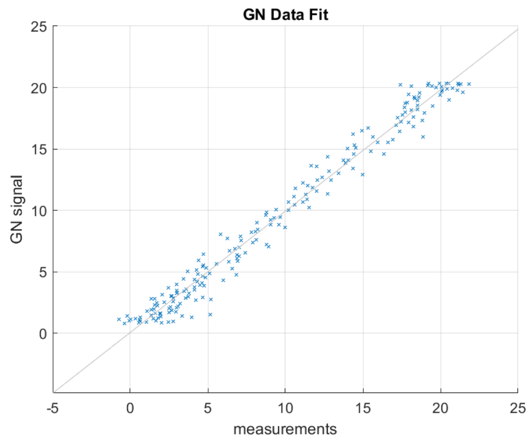
	$\boldsymbol{x}_0$	`tolerance`	`maxIter`	Δ	η
`Levenberg-Marquardt`:	$(2,2,2)^T$	`1e-10`	200	50	1

- A large value for $\Delta = 50$ (the maximum trust region size) was used since the starting point $\boldsymbol{x}_0 = (2,2,2)^T$ was far off from parameter $\boldsymbol{x}^* = (3, 150, 2)^T$ that generated the data. In order to ensure quick convergence therefore, the search region must be large enough to allow for a large initial descent step. Indeed, we found that if $\Delta = 1$, the algorithm would saturate the trust region bound at every iteration, leading to very slow convergence.
- The second Wolfe condition parameter (constant in strong curvature condition) was set `c`$_2$ = 0.1 since this value is recommended for Newton-type algorithms.

Each algorithm was then used to solve the minimisation problem with the initial parameters as set above. The Gauss-Newton method took **9** iterations, whereas the Levenberg-Marquadt method took **16** iterations. The minimising parameter vector found by methods were the same to 4 d.p. and on one such run the value was $\boldsymbol{x}_{\min} = (2.9989, 152.8286, 1.9972)^T$. The corresponding curves of $\varphi(\boldsymbol{x}_{\min}; t_j)$ for $j = 1, \dots, 200$ plotted against the data points $\tilde{\varphi}(\boldsymbol{x}^*; t_j)$ are shown below



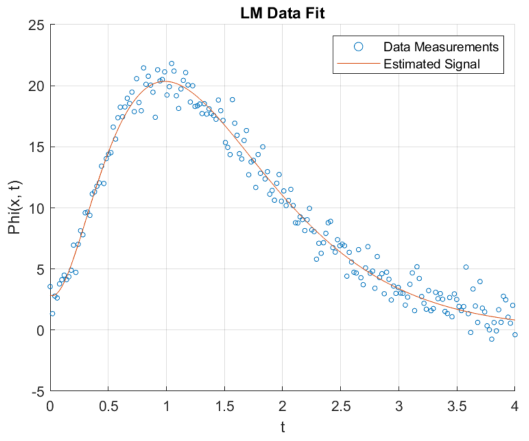
(a) Estimated Signal line with measurements



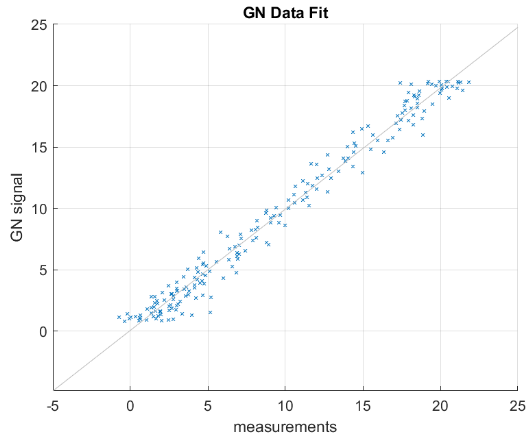
(b) Estimated Signal versus measurements

Figure 7: Gauss Newton plots

Since the parameter vector $\boldsymbol{x}_{\min}$ found by both algorithms was exactly the same, the Levenberg-Marquadt plots look identical:



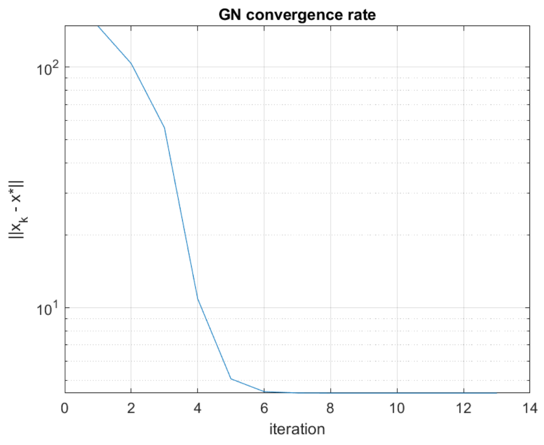
(a) Estimated Signal line with measurements



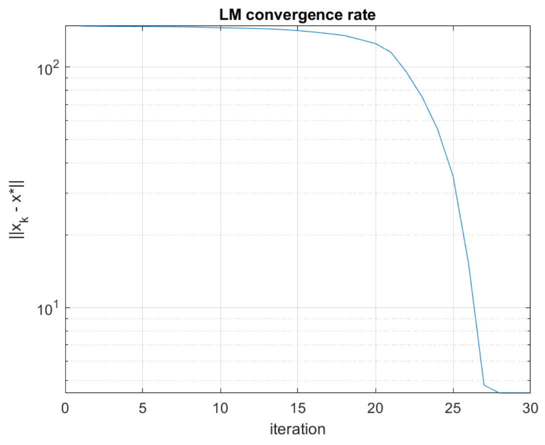
(b) Estimated Signal versus measurements

Figure 8: Levenberg-Marquadt plots

In both cases we see that the Estimated Signal values versus Measurements is close to a $y = x$ line indicating the high accuracy of our model (which we would not know already if given only the data measurements). Furthermore, convergence plots in terms of the values $||x_{k+1} - x^*||$ are plotted below



(a) Gauss-Newton



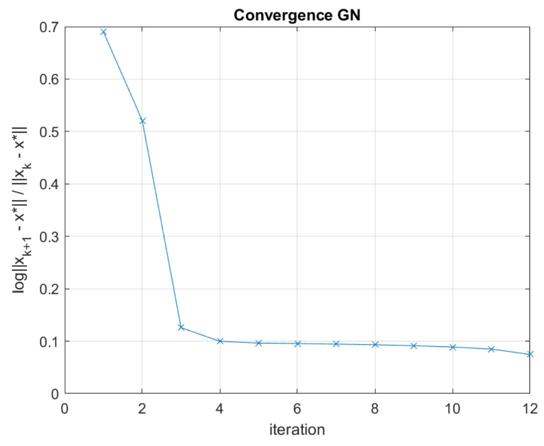
(b) Levenberg-Marquardt

Figure 9: Plots of $||x_k - x^*||$

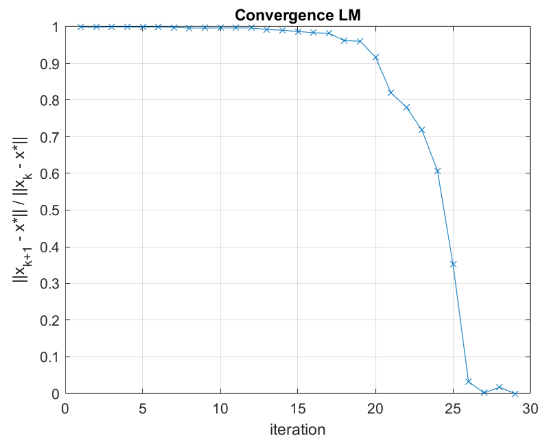
In order to assess the convergence rate of Gauss-Newton we evualate the factor $||[J^TJ(x^*)]^{-1}H(x^*)||$ at x^*. In this case the factor $||[J^TJ(x^*)]^{-1}H(x^*)|| \approx 3.1 > 1$. This implies that the convergence rate of the algorithm for a full Newton step:

$$||x_k + p_k^{\mathrm{GN}} - x^*|| = ||[J^TJ(x^*)]^{-1}H(x^*)||||x_k - x^*|| + O(||x_k - x^*||^2) \tag{10}$$

cannot be guarenteed to be quadratic for the Gauss-Newton methods. Indeed this is what we observe in the converge plots of the GN methods where superlinear local convergence is not observed (after the 6th iteration the algorithm convergences very slowly to the solution). Furthermore we also plot the ratio $\frac{||x_{k+1}-x^*||}{||x_k-x^*||}$ for both algorithms:



(a) Gauss-Newton



(b) Levenberg-Marquardt

Figure 10: Plots of $\frac{||x_k-x^*||}{||x_{k-1}-x^*||}$

For the LM plot we observe that the value of $\frac{||x_k-x^*||}{||x_{k-1}-x^*||}$ does finally reach zero, which potentially suggests that the LM method is indeed achieving superlinear local convergence. This is in comparison to the Gauss-Newton convergence rate which has a value of around 0.1. Therefore we conclude that GN has **linear** local convergence and LM has **superlinear** local convergence for this problem.