Non-linear optimisation

FMRIB Technial Report TR07JA1

Jesper L. R. Andersson, Mark Jenkinson and Stephen Smith
FMRIB Centre, Oxford, United Kingdom

Correspondence and reprint requests should be sent to:
Jesper Andersson
FMRIB Centre
JR Hospital
Headington
Oxford OX3 9DU
phone: 44 1865 222 782
fax: 44 1865 222 717
mail: jesper@fmrib.ox.ac.uk

28 June 2007

1 INTRODUCTION

This is a short technical report reviewing non-linear minimisation in the context of the non-linear toolbox that has been develped for use in future development of FSL. We have focused more on an intuitive "narrative" starting with Newtons method and then progressing along a plausible "plot" rather than mathematical and historical stringency. An example of that is our view of the Levenberg and Levenberg-Marquardt methods as general means of modifying the Hessian (or the approximation passing for the Hessian) rather than specific to the Gauss-Newton approximation.

2 THEORY

2.1 Nonlinear minimisation of cost-function

Let us say we have some function f that depends on some set of parameters $\mathbf{w}$, and possibly some set of data $\mathbf{y}$, and that we want to find the particular parameters $\mathbf{w}$ that minimises the value of f. The difference between "parameters" and "data" is sometimes quite clear (such as when finding the Maximum-Likelihood parameters given some data). In other cases it may be less clear and the distinction is simply one of parameters being the variables that we wish to minimise f and the data being constants.

2.2 Newton's method

With a couple of exceptions, that will not be treated in this report, all methods for non-linear optimisation can be said to be based on Newton's method. In practice it is seldom used in its original form, having been replaced by variants that address some of its pitfalls, but these are typically precisely "variants". It is therefore crucial to understand the principles underlying Newton's method if one is to understand and appreciate its successors.

For any function scalar $f(\mathbf{w})$ we can perfom a Taylor-expansion around some point $\mathbf{w}_0$ such that its value at some "nearby" point $\mathbf{w}$ is given by

$$f(\mathbf{w}) \approx f(\mathbf{w}_0) + \nabla f(\mathbf{w}_0)(\mathbf{w} - \mathbf{w}_0) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_0)^T \mathbf{H}(\mathbf{w}_0)(\mathbf{w} - \mathbf{w}_0) \tag{1}$$

where $\mathbf{H}$ is the Hessian matrix whos ijth element is given by

$$\mathbf{H}_{ij} = \frac{\partial^2 f}{\partial \mathbf{w}_i \partial \mathbf{w}_j} \tag{2}$$

Note also that we have used the notation $\nabla f(\mathbf{w})$ and $\mathbf{H}(\mathbf{w})$ in lieu of the possibly more correct $\nabla f|_{\mathbf{w}}$ and $\mathbf{H}|_{\mathbf{w}}$. A stationary point is found by differentiating the approximation of $f(\mathbf{w})$ given by equation 1 with respect to $\mathbf{w}$ yielding

$$\nabla f(\mathbf{w}) \approx \nabla f(\mathbf{w}_0) + \mathbf{H}(\mathbf{w}_0)(\mathbf{w} - \mathbf{w}_0) \tag{3}$$

setting that derivative to zero to obtain

$$\mathbf{w} - \mathbf{w}_0 = -\mathbf{H}^{-1}(\mathbf{w}_0)\nabla f(\mathbf{w}_0) \tag{4}$$

where $\mathbf{w}-\mathbf{w}_0$ is the step that would take you to a stationary point of f provided it was sufficiently well approximated by a quadratic form near $\mathbf{w}_0$. When $\mathbf{w}_0$ is "too far" away from the minimum the step above may not be sufficient to take us to it, but will hopefully still be a step in the right direction. The step described by equation 4 can then be used in an iterative sequence

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{H}^{-1}(\mathbf{w}_k)\nabla f(\mathbf{w}_k) \tag{5}$$

which combined with some suitable criterion for converegence will lead to the solution.

This, known as a Newton-Raphson type minimisation scheme, is the starting point for most optimisation methods.

2.2.1 Gauss-Newton minimisation

A very common approximation to Newton-Raphson minimisation is called Gauss-Newton and is predicated on there existing some underlying vector-valued function $\mathbf{h}(\mathbf{w})$ such that h is an $\Re^m \rightarrow \Re^n$ mapping. The scalar valued objective/cost-function is then assumed to be

$$f(\mathbf{w}) = \mathbf{h}(\mathbf{w})^T\mathbf{h}(\mathbf{w}) \tag{6}$$

i.e. the sum of squares of the n individual functions. In this case a useful approximation to the Newton updating scheme (equation 5) is given by

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}(\mathbf{w}_k)^T\mathbf{J}(\mathbf{w}_k)\right)^{-1}\mathbf{J}(\mathbf{w}_k)\mathbf{h}(\mathbf{w}_k) \tag{7}$$

where $\mathbf{J}$ is the Jacobian matrix of $\mathbf{g}$ given by

$$\mathbf{J} = \begin{pmatrix} \frac{\partial h_1}{\partial w_1} & \frac{\partial h_1}{\partial w_2} & \cdots & \frac{\partial h_1}{\partial w_m} \\ \frac{\partial h_2}{\partial w_1} & \frac{\partial h_2}{\partial w_2} & \cdots & \frac{\partial h_2}{\partial w_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_n}{\partial w_1} & \frac{\partial h_n}{\partial w_2} & \cdots & \frac{\partial h_n}{\partial w_m} \end{pmatrix} \tag{8}$$

where the partial derivatives are all estimated at the point $\mathbf{w}_k$. The Gauss-Newton approximation is often used when each element of the function $\mathbf{h}(\mathbf{w})$ is the deviation between some observed value y_i and a model prediction $g_i(\mathbf{w})$. In that case

$$h_i(\mathbf{w}) = y_i - g_i(\mathbf{w}) \tag{9}$$

or in matrix notation

$$\mathbf{h}(\mathbf{w}) = \mathbf{y} - \mathbf{g}(\mathbf{w}) \tag{10}$$

A first order Taylor-expansion of $\mathbf{h}(\mathbf{w})$ around some point $\mathbf{w}_0$ is given by

$$\mathbf{h}(\mathbf{w}) \approx \mathbf{y} - \mathbf{g}(\mathbf{w}_0) - \mathbf{J}(\mathbf{w}_0)(\mathbf{w} - \mathbf{w}_0) \tag{11}$$

Let us for the time being ignore the $\approx$ and assume *iid* normal distributed errors

$$\mathbf{y} - \mathbf{g}(\mathbf{w}_0) = \mathbf{J}(\mathbf{w}_0)(\mathbf{w} - \mathbf{w}_0) + \mathbf{e} \qquad \mathbf{e} \sim N(0, \sigma^2) \tag{12}$$

which is solved in a Maximum-Likelihood (least squares) sense by

$$\widehat{(\mathbf{w}-\mathbf{w}_0)} = \left(\mathbf{J}(\mathbf{w}_0)^T\mathbf{J}(\mathbf{w}_0)\right)^{-1}\mathbf{J}(\mathbf{w}_0)^T(\mathbf{y}-\mathbf{g}(\mathbf{w}_0)) = \left(\mathbf{J}(\mathbf{w}_0)^T\mathbf{J}(\mathbf{w}_0)\right)^{-1}\mathbf{J}(\mathbf{w}_0)^T\mathbf{h}(\mathbf{w}_0) \tag{13}$$

Because the Taylor-expansion is only an approximation of $\mathbf{h}(\mathbf{w})$ this will need to be repeated, leading to the updating scheme described by equation 7. This scheme is of course easily adapted to an arbitrary (known) variance structure (*i.e.* $\mathbf{e} \sim N(0, \mathbf{V})$) by replacing equation 13 with the corresponding weighted least squares solution.

When comparing the Newton (equation 5) and Gauss-Newton (equation 7) steps we note that $\mathbf{J}^T\mathbf{J}$ has taken the "role" of $\mathbf{H}$ and $J^T(\mathbf{y}-\mathbf{g}(\mathbf{w}))$ that of ∇f. This may seem a little surprising (note that there is no trace of second derivatives in $\mathbf{J}$) so let us look at that a little, starting with $\mathbf{J}^T(\mathbf{y}-\mathbf{g}(\mathbf{w}))$. We start by noting that in sum notation $\mathbf{J}^T(\mathbf{y}-\mathbf{g}(\mathbf{w}))$ can be written as

$$\left[\mathbf{J}^T(\mathbf{y}-\mathbf{g}(\mathbf{w}))\right]_j = -\sum_{i=1}^{N}\frac{\partial g_i}{\partial w_j}(y_i - g_i(\mathbf{w})) \tag{14}$$

and then compare that to the derivative of f

$$\frac{\partial f}{\partial w_j} = \frac{\partial}{\partial w_j}\sum_{i=1}^{N}(f_i - g_i(\mathbf{w}))^2 = -2\sum_{i=1}^{N}\frac{\partial g_i}{\partial w_j}(y_i - g_i(\mathbf{w})) \tag{15}$$

noting that they differ only with a factor of 2. Next we do the same thing for $\mathbf{J}^T\mathbf{J}$ which in sum notation is

$$\left[\mathbf{J}^T\mathbf{J}\right]_{jk} = \sum_{i=1}^{N}\frac{\partial g_i}{\partial w_j}\frac{\partial g_i}{\partial w_k} \tag{16}$$

and

$$[\mathbf{H}]_{jk} = 2\sum_{i=1}^{N}\frac{\partial g_i}{\partial w_j}\frac{\partial g_i}{\partial w_k} - 2\sum_{i=1}^{N}\frac{\partial^2 g_i}{\partial w_j \partial w_k}(y_i - g_i(\mathbf{w})) \tag{17}$$

We note that again we have a factor of 2 difference, but in addition we also have a term that is missing from the $\mathbf{J}^T\mathbf{J}$ approximation. Given that how good an approximation can it possibly be? Surprisingly very good is the answer. The second term is a weighted sum over the deviations between the model and the data, and whenever we are near to the "true" solution we would expect this to add up to a small(ish) number. But what about when we are not close to the solution? Well, if we consider specifically an element on the diagonal

$$[\mathbf{H}]_{jj} = 2\sum_{i=1}^{N}\left(\frac{\partial g_i}{\partial w_j}\right)^2 - 2\sum_{i=1}^{N}\frac{\partial^2 g_i}{\partial w_j^2}(y_i - g_i(\mathbf{w})) \tag{18}$$

we note that it is a sum of squares (that will always be positive) and a weighted sum of deviations that may take any sign. When we are far away from the solution $\mathbf{H}_{jj}$ will often be dominated by the second term and can hence take on negative values (and it will!). The consequence of a negative value on the diagonal is that $\mathbf{H}$ is no longer positive definite and will cause a step "upwards" along that particular parameter. Hence it is typically the case that the Gauss-Newton approximation is much more stable (with respect to poor starting guesstimates of $\mathbf{w}$) than the Newton scheme, and with almost as rapid convergence properties close to the "true" $\mathbf{w}$. The downside is that it can only be used for the case when f is a sum of squares.

2.2.2 Fisher scoring

A method known as "Fisher scoring" is, like the Gauss-Newton method, based on Newtons methot but with a slight modification of the Hessian. In the field of statistics, and when the objective function that we want to maximise is typically a probability (*e.g.* the likelihood function), the Hessian matrix is referred to as the "observed information matrix".

The "expected information matrix" or the "Fisher information matrix" refers to the matrix whos ijth element is $E_{\mathbf{w}}\left[\partial^2\mathcal{L}/\partial w_i \partial w_j\right]$, i.e. the expected value of the second derivative where the expectation is taken over the parameters of the function. One of its uses is to determine how much information a certain experiment would yield about the parameters $\mathbf{w}$. Specifically it allows one to calculate a lower bound for the uncertainty of a parameter estimate that can be obtained from a particular experiment. Because it is based on the expectation it can be calculated without any reference to the data, *i.e.* based only on the design of the experiment and on some assumed values of the parameters.

The Fisher scoring method is simply Newtons method with the Hessian (or the observed information matrix) replaced with the expected (or Fisher) information matrix. It is probably easiest understood from a small example. Let us say we have some data generated from a model consisting of a mean μ and Gaussian additive noise with variance σ^2. Hence our model is $y_i = \mu + e_i$ where $e_i \sim N(0, \sigma^2)$. The neg-log likelihood for a sample of size n is proportional to

$$\mathcal{L}(\mu, \sigma^2) \propto n \log(\sigma^2) + \frac{1}{\sigma^2}\sum_{l=1}^{n}(y_l - \mu)^2 \tag{19}$$

The gradient, Hessian and Fisher information matrix are then given by

$$\begin{aligned}
\nabla\mathcal{L} &= \begin{bmatrix} \frac{\partial\mathcal{L}}{\partial\mu} & \frac{\partial\mathcal{L}}{\partial\sigma^2} \end{bmatrix}^T = \begin{bmatrix} -\frac{2}{\sigma^2}\sum_{l=1}^{n}(y_l-\mu) & \frac{n}{\sigma^2} - \frac{1}{(\sigma^2)^2}\sum_{l=1}^{n}(y_l-\mu)^2 \end{bmatrix}^T \\
\mathbf{H} &= \begin{pmatrix} \frac{\partial^2\mathcal{L}}{\partial\mu^2} & \frac{\partial^2\mathcal{L}}{\partial\mu\partial\sigma^2} \\ \frac{\partial^2\mathcal{L}}{\partial\mu\partial\sigma^2} & \frac{\partial^2\mathcal{L}}{\partial(\sigma^2)^2} \end{pmatrix} = \begin{pmatrix} \frac{2n}{\sigma^2} & \frac{2}{(\sigma^2)^2}\sum_{l=i}^{n}(y_l-\mu) \\ \frac{2}{(\sigma^2)^2}\sum_{l=i}^{n}(y_l-\mu) & -\frac{n}{(\sigma^2)^2} + \frac{2}{(\sigma^2)^3}\sum_{l=1}^{n}(y_l-\mu)^2 \end{pmatrix} \\
\boldsymbol{\mathcal{I}} &= \begin{pmatrix} E\left(\frac{\partial^2\mathcal{L}}{\partial\mu^2}\right) & E\left(\frac{\partial^2\mathcal{L}}{\partial\mu\partial\sigma^2}\right) \\ E\left(\frac{\partial^2\mathcal{L}}{\partial\mu\partial\sigma^2}\right) & E\left(\frac{\partial^2\mathcal{L}}{\partial(\sigma^2)^2}\right) \end{pmatrix} = \begin{pmatrix} \frac{2n}{\sigma^2} & 0 \\ 0 & \frac{n}{(\sigma^2)^2} \end{pmatrix}
\end{aligned} \tag{20}$$

where the expectation is taken over the data given the parameter estimate. Note that this means that the Hessian is a function of the parameter estimate and the data, whereas the information matrix is a function only of the parameter estimate.

Data was generated according to this model and then "analysed" them using either Newtons method (*i.e.* $\mathbf{H}$) or Fisher scoring (*i.e.* $\boldsymbol{\mathcal{I}}$) using a variety of choices of starting guesstimates for $[\mu\ \sigma^2]$. It was clear that Fisher scoring was much more robust with respect to starting guesstimates than Newtons method. Looking at $\mathbf{H}$ in equation 20 it is evident why the Newton-Raphson scheme is so sensitive to, in particular, a too large initial value for σ^2. When the estimate of σ^2 becomes large, $\partial^2\mathcal{L}/\partial(\sigma^2)^2$ becomes negative and the Hessian is no longer positive definite, and that means taking a step against the direction of the (negative) gradient.

It is also of interest to consider a slightly more involved model with normal distributed error (*i.e.* a sum-of-squares cost function). To do so let us say we want to fit an exponential to some data.

$$y_i = \alpha e^{-\beta x_i} + e_i \qquad e_i \sim N(0, \sigma^2) \tag{21}$$

Let us furthermore assume that we are interested in finding $\mathbf{w} = [\alpha\ \beta]$. The neg-log-likelihood for this model is given by

$$\mathcal{L}(\theta) = -\log p(\mathbf{y}|\mathbf{w}, \sigma^2) = \frac{n}{2}\log 2\pi + \frac{n}{2}\log\sigma^2 + \frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y_i - w_1 e^{-w_2 x_i}\right)^2 \tag{22}$$

Since we are not interested in σ^2 we can for our purposes write this as

$$\mathcal{L}(\mathbf{w}) \propto \sum_{i=1}^{n}\left(y_i - w_1 e^{-w_2 x_i}\right)^2 \tag{23}$$

From this we can derive the gradient (that is common to Newtons method and to Fisher scoring), the Hessian and the Fisher information matrix

$$\begin{aligned}
\nabla\mathcal{L} &= \begin{pmatrix} -2\sum_{i=1}^{n} e^{-w_2 x_i}(y_i - w_1 e^{-w_2 x_i}) \\ 2\sum_{i=1}^{n} w_1 x_i e^{-w_2 x_i}(y_i - w_1 e^{-w_2 x_i}) \end{pmatrix} \\
\mathbf{H}_{11} &= 2\sum_{i=1}^{n} e^{-2w_2 x_i} \\
\mathbf{H}_{12} &= \mathbf{H}_{21} = 2\sum_{i=1}^{n} x_i e^{-w_2 x_i}(y_i - w_1 e^{-w_2 x_i}) - 2\sum_{i=1}^{n} w_1 x_i e^{-2w_2 x_i} \\
\mathbf{H}_{22} &= 2\sum_{i=1}^{n} w_1^2 x_i^2 e^{-2w_2 x_i} - 2\sum_{i=1}^{n} w_1 x_i^2 e^{-w_2 x_i}(y_i - w_1 e^{-w_2 x_i}) \\
\boldsymbol{\mathcal{I}} &= \begin{pmatrix} 2\sum_{i=1}^{n} e^{-2w_2 x_i} & -2\sum_{i=1}^{n} w_1 x_i e^{-2w_2 x_i} \\ -2\sum_{i=1}^{n} w_1 x_i e^{-2w_2 x_i} & 2\sum_{i=1}^{n} w_1^2 x_i^2 e^{-2w_2 x_i} \end{pmatrix}
\end{aligned} \tag{24}$$

As you can see it is almost embarassingly simple to take the expectation in this case. All we need to do is to "say" that we expect $\sum_{i=1}^{n} c_i(y_i - w_1 e^{-w_2 x_i})$ to be zero. Furthermore it can be seen that $\mathbf{H}_{22}$ could very easily have a negative value *e.g.* when w_1 is small or w_2 is large (compared to the true value). In contrast $\boldsymbol{\mathcal{I}}_{22}$ is a sum of squares and will always be positive. Let us now take a look at the corresponding entities for the Gauss-Newton case and see how they compare. In this case $\mathbf{h}$ is given by $h_i = (y_i - w_1 e^{-w_2 x_i})$ which means that

$$\begin{aligned}
\mathbf{J}^T\mathbf{h} &= \begin{pmatrix} -\sum\limits_{i=1}^{n} e^{-w_2 x_i}(y_i - w_1 e^{-w_2 x_i}) \\ \sum\limits_{i=1}^{n} w_1 x_i e^{-w_2 x_i}(y_i - w_1 e^{-w_2 x_i}) \end{pmatrix} \\
\mathbf{J}^T\mathbf{J} &= \begin{pmatrix} \sum\limits_{i=1}^{n} e^{-2w_2 x_i} & -\sum\limits_{i=1}^{n} w_1 x_i e^{-2w_2 x_i} \\ -\sum\limits_{i=1}^{n} w_1 x_i e^{-2w_2 x_i} & \sum\limits_{i=1}^{n} w_2^2 x_i^2 e^{-2w_2 x_i} \end{pmatrix}
\end{aligned} \qquad (25)$$

We can plainly see that $\nabla\boldsymbol{\mathcal{L}} = 2\mathbf{J}^T\mathbf{h}$ and $\boldsymbol{\mathcal{I}} = 2\mathbf{J}^T\mathbf{J}$ which means that the Gauss-Newton updating step $-\left(\mathbf{J}^T\mathbf{J}\right)^{-1}\mathbf{J}^T\mathbf{h}$ is identical to the $-\boldsymbol{\mathcal{I}}^{-1}\nabla\boldsymbol{\mathcal{L}}$ step of Fisher scoring.

What this means is that Gauss-Newton can be seen as a special case of Fisher scoring when applied to a likelihood for a model with normal distributed errors for which we are only interested in the parameters (rather than the "hyper-parameters" such as *e.g.* σ^2). Another way of putting it would be to say that Fisher scoring offers Gauss-Newton robustness with respect to starting guesstimates when the cost funtion is not a sum-of-squares. An example of that is the use of Fisher scoring for estimating diffusion parameters for a Rician noise model.

2.3 The Levenberg and Levenberg-Marquardt modifications

It is *not* a good thing when/if the hessian becomes "not positive definite" and the Levenberg and the Levenberg-Marquardt modifications represent (ever so) slightly different approaches to ensuring that it stays positive definite.

2.3.1 Levenberg

So, let us say we take the step $-\mathbf{H}^{-1}\nabla f$, taking us from $\mathbf{w}^{(k)}$ to $\mathbf{w}^{(k+1)}$, calculate $f(\mathbf{w}^{(k+1)})$ and to our horror discovers that it is actually larger than $f(\mathbf{w}^{(k)})$. How could that happen, and what do you do? Let us first assume that ∇f is not exactly zero, which means that there is some step (small though it might be) in the direction of ∇f that would lead to a decrease in f. Therefore if f does not decrease it will mean one of two things: either the step was too long (past the minimum) or it was in the wrong direction. To examine the first of those options let us consider a very simple diagonal jacobian with elements H_{11} and H_{22}. That means that the step would be $[(1/H_{11})g_1 \; (1/H_{22})g_2]$. One way to shorten that step would be to add some fudge factor λ to the elements on the diagonal, making the step $[(1/(H_{11}+\lambda))g_1 \; (1/(H_{22}+\lambda))g_2]$ instead.

Let us now look at the other option, *i.e.* that we are taking a step in the wrong direction. The role of the hessian in the update step is to "temper" the gradient a little so we would not expect or want the step $\mathbf{H}^{-1}\nabla f$ to be identical to ∇f, but we would at least want it to go in the same general direction. Formally this means that we want the inner product of the $\mathbf{H}^{-1}\nabla f$ step and the gradient ∇f to be positive, which is exactly the same as saying that we want $\mathbf{H}$ to be positive definite. The reason it is no longer positive definite could be because one of elements on the diagonal is negative, in which case adding the fudge factor λ (so that $H_{ii} \rightarrow H_{ii} + \lambda$) to it will solve that as long as $\lambda > |H_{ii}|$. The other possible reason might be of the off-diagonal elements are "large" compared to the diagonal example. For example for a 2 parameter problem we require that $H_{11}H_{22} > H_{12}^2$. Again it can be seen that if we add some factor λ to the diagonal elements it means that $H_{11}H_{22} \to (H_{11} + \lambda)(H_{22} + \lambda)$ and we will always be able to find a λ that makes that greater than H_{12}^2.

So, it seems that by adding $\lambda\mathbf{I}$ to the Hessian we should be able to solve all our problems (related to optimisation). Only question now is what should λ be? The strategy suggested by Levenberg is to start of with some "small" λ so that $\mathbf{H} + \lambda\mathbf{I}$ is dominated by $\mathbf{H}$, to find the next set of parameters values from $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - (\mathbf{H} + \lambda\mathbf{I})^{-1}\nabla f$, calculate $f(\mathbf{w}^{(k+1)})$ and compare it to $f(\mathbf{w}^{(k)})$. If it turns out that $f(\mathbf{w}^{(k+1)})$ is smaller than $f(\mathbf{w}^{(k)})$ then all is well and we pick $\mathbf{w}^{(k+1)}$ as the next point for which we want to calculate ∇f and $\mathbf{H}$. In addition to that we let $\lambda \to 0.1\lambda$ such that on the next iteration we are even closer to a "pure" hessian update step. If on the other hand it turns out that $f(\mathbf{w}^{(k+1)})$ is larger than $f(\mathbf{w}^{(k)})$ we are clearly on the wrong track and need to do something. We then let $\lambda \to 10\lambda$ and calculate a new suggestion for $\mathbf{w}^{(k+1)}$ with this larger lambda. This is repeated, each time increasing λ by a factor of 10, until a $\mathbf{w}^{(k+1)}$ is found for which $f(\mathbf{w}^{(k+1)})$ is smaller than $f(\mathbf{w}^{(k)})$.

2.3.2 Levenberg-Marquardt

The difference between the Levenberg and the Levenberg-Marquardt is quite subtle, so subtle in fact that you might think that Marquardt obtained his imortality very cheaply. I am sure he was a lovely guy though and we should all be happy for him. Where Levenberg suggested replacing $\mathbf{H}$ by $\mathbf{H} + \lambda\mathbf{I}$ Marquardts suggestion was to instead replace it by $\mathbf{H} + \lambda\,\mathrm{diag}(\mathbf{H})$, where $\mathrm{diag}(\mathbf{H})$ is a matrix with the same values as $\mathbf{H}$ on the diagonal and zero elsewhere. So, what then are the implications of this compared to the $\mathbf{H} + \lambda\mathbf{I}$ scheme? First of all we can note that if an element on the diagonal is negative then adding $\lambda\,\mathrm{diag}(\mathbf{H})$ is only going to make it more negative. From that we can conclude that the Levenberg-Marquardt scheme should only be used for the Gauss-Newton or Fisher scoring variants, and in the literature you will typically see it presented in association with the Gauss-Newton algorithm. So, in what sense is it an improvement on Levenberg then? To answer that we need to consider a little what the elements on the diagonal of $\mathbf{H}$ (*i.e.* $\partial^2 f/\partial w_i^2$) really mean. First of all let us remind ourselves that $\partial f/\partial w_i$ tells us how much f is expected to change if we take a unity step in the ith direction. See it as a prognosis of what would happen if yot took that step. Now then, where does that leave $\partial^2 f/\partial w_i^2$? It tells us *how fast that prognosis changes as we move along the ith direction.* So, let us say *e.g.* that $\partial f/\partial w_i = 10$ and $\partial^2 f/\partial w_i^2 = 0.1$ for a particular f and $\mathbf{w}$. Hence the prognisis is that a unity step along the (negative) ith direcion would decrase f by 10, and since that prognosis is expected to change very little over that range (0.1) we would expect to pretty much see a decrease close to 10. If on the other hand $\partial^2 f/\partial w_i^2 = 100$ it means that the prognosis is expected to change a lot over that range and we really don't know what to expect at the end of that step. So in a way the value of $\partial^2 f/\partial w_i^2$ tells you something about how much information that is really contained in $\partial f/\partial w_i$. If $\partial^2 f/\partial w_i^2$ is small it makes sense to take a long step in that direction because the derivative information is quite certain. If on the other hand $\partial^2 f/\partial w_i^2$ is large it makes sense to be more cautious and take a shorter step. Let us now consider a case where $\nabla f = [-1\ -1]^T$ and where $H_{11} = 1$, $H_{22} = 100$ and $H_{12} = 0$. The gradient indicates that we might want to take a step in the [1 1] direction and that if we were to take a unity step f would decrease approximately by 2. The hessian on the other hand indicates that while this prognosis is reasonably good in the **i** direction it is associated with considerable uncertainty in the **j** direction and consequently the step advocated by Newton is

$$\Delta\mathbf{w} = -\begin{pmatrix} 1 & 0 \\ 0 & 100 \end{pmatrix}^{-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{100} \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{1}{100} \end{pmatrix} \tag{26}$$

Marquardt's argument is that even though we might be far from the "true" $\mathbf{w}$ and/or a quadratic form might be a poor approximation of f the values on the diagonal of $\mathbf{H}$ should still contain the information alluded to above, *i.e.* is should say something about the value of the information in the derivative. If we use the Levenberg scheme $\mathbf{H} \rightarrow \mathbf{H} + \lambda\mathbf{I}$ for a large λ the values on the diagonal will all be $\approx \lambda$ and that information will be largely lost. If on the other hand we use the Levenberg-Marquardt scheme $\mathbf{H} \rightarrow \mathbf{H} + \lambda\,\mathrm{diag}(\mathbf{H})$ the relative magnitude of the vaules on the diagonal is preserved and hnece that information retained. It is not obvious to me what difference that makes in practice, but a fact is that the Levenberg-Marquardt modification has become a de-facto standard for optimisation of non-linear least squares problems.

2.4 Quasi-Newton methods

Strange name, inn'it? It is used for a family of algorithms that all start out with the same basic assumptions and ideas as Newton's method, but tries to achieve the same goal with less computational effort. Hence it is still based on approximating the function f with a second order Taylor expansion around some point $\mathbf{w}_i$ and than to calculate consecutive updates of the parameters according to

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \mathbf{H}^{-1}\nabla f \tag{27}$$

but without the hassle of calculating and/or inverting $\mathbf{H}$. If you think of for example non-linear registration we try to estimate 10.000 parameters for the "medium resolution" case, *i.e.* $\mathbf{w}$ is a 10.000×1 vector. This means that $\mathbf{H}$ contains $\approx 1/2 \times 10^8$ unique elements that need to be calculated. Let us furthermore assume that we are using a least squares cost-function so that we can use the Gauss-Newton approximation. That means that each of the $1/2 \times 10^8$ elements is of the form $\sum_{i=1}^{n}(\partial h_i/\partial w_j)(\partial h_i/\partial w_k)$ where n is the number of voxels in the image volume. It should be obvious that this represent a considerable computational effort, and if that isn't enough we also need to invert a 10.000×10.000 matrix. So from that it can easily be realised that a method that allows us to calculate the step (or an approximation to it) $\mathbf{H}^{-1}\nabla f$ without having to calculate $\mathbf{H}$, or even without having to represent it, would be very valuable. The algorithms in this category will typically attempt to calculate a vetor $\mathbf{p}$ that points in the same direction as $-\mathbf{H}^{-1}\nabla f$, but whos norm has some unknown relation to the norm of $-\mathbf{H}^{-1}\nabla f$. A line-search is then performed to find a scalar κ such that $\kappa\mathbf{p}$ is at a minimum along $\mathbf{p}$. That information is then used for calculating a new $\mathbf{p}$ for the next step. They fall into two distinct categories called "Conjugate Gradients methods" and "Variable Metric methods". From a practical perspective they differ in that Variable Metric methods still need to represent (and for some flavours invert) the Hessian while Conjugate Gradient methods do not.

2.4.1 Conjugate Gradient methods

The Conjugate Gradient style methods build on the fundamental concept of conjugate directions. The same is true for Conjugate Directions style non-linear minimisation methods (*i.e.* Powells method) and for Conjugate Gradient methods for solving large sparse systems of linear equations, so it is worth spending a minute to get once head around it. Imagine we started at some point $\mathbf{w}_0$ and then went along some direction $\mathbf{r}$ until we found the minimum of f at the point $\mathbf{w}_1 = \mathbf{w}_0 + \lambda\mathbf{r}$, *i.e.* we are now at the bottom of the valley in our direction $\mathbf{r}$. We now want to choose a new direction $\mathbf{p}$ to go along and find the minimum along that direction. One condition that we want to fulfill when we go along that new direction is that it must not (straight away at least) destroy the minimisation along the previous direction $\mathbf{r}$. How do we then ensure that? The first thing to observe is that at the point $\mathbf{w}_1$ (*i.e.* the minimum along $\mathbf{r}$) the gradient of f is orthogonal to $\mathbf{r}$, *i.e.* $\nabla f\mathbf{r} = 0$. To not destroy the minimisation along that direction is equivalent to want it to remain that way, *i.e.* we want to choose a direction $\mathbf{p}$ such that $\nabla f(\mathbf{w}_1 + \lambda\mathbf{p})\mathbf{r} = 0$ for all values of λ within some "reasonable" range. To find such a direction we need to know how ∇f changes as we go along the new direction $\mathbf{p}$. An approximate exression for ∇f as we take a step λ along $\mathbf{p}$ (cf equation 1) is given by

$$\nabla f(\mathbf{w_1} + \lambda\mathbf{p}) \approx \nabla f(\mathbf{w_1}) + \lambda\mathbf{p}^T\mathbf{H}(\mathbf{w_1}) \tag{28}$$

which means that our condition $\nabla f(\mathbf{w}_1 + \lambda\mathbf{p})\mathbf{r} = 0$ can be written

$$\left(\nabla f(\mathbf{w}) + \lambda\mathbf{p}^T\mathbf{H}(\mathbf{w})\right)\mathbf{r} = \nabla f(\mathbf{w})\mathbf{r} + \lambda\mathbf{p}^T\mathbf{H}(\mathbf{w})\mathbf{r} = 0 \tag{29}$$

and finally since we know that $\nabla f(\mathbf{w})\mathbf{r}$ is already zero (remember we are at the minimum along $\mathbf{r}$) this reduces to

$$\mathbf{p}^T\mathbf{H}\mathbf{r} = 0 \tag{30}$$

This is the "conjugacy criterion" that is so central for all these methods. If you didn't quite understand the arguments above, please take a moment and read them again.

So, the idea behind Conjugate Gradient methods is to construct a series of directions such that for all k the $k+1$th direction fulfills the criterion $\mathbf{p}_{k+1}\mathbf{H}\mathbf{p}$ where $\mathbf{p}_1$ is arbitrary and usually choosen to be $-\nabla f(\mathbf{w}_0)$ and perform line minimisations along each of the directions. Sounds simple enough. Until you realise that another of the "conditions" of the Conjugate Gradient methods is that we cannot/do not want to calculate $\mathbf{H}$. Given that, how then does one go about finding the directions $\mathbf{p}_{k+1}$?

The strategy starts with the realisation that in general it still a good idea to take a step in the direction of the negative gradient, *i.e.* along ∇f. The problem if we always just do that is precisely that we will ruin the minimisation along previous directions, which is why Steepest Descent type algorithms have such poor convergence properties. How can we then combine the best of Steepest Descent and the concept of conjugate directions introduced above. Well, whenever we are at some minimum along some direction $\mathbf{p}_k$ we know that the gradient of f at that point (let us denote it by ∇f_{k+1}) is orthogonal to the previous search direction $\mathbf{p}_k$, and we also know that if we are to fullfill the conjugacy criterion the new search direction $\mathbf{p}_{k+1}$ cannot in general be orthogonal to the previous. That means that we need to "complement" the (negative) gradient $-\nabla f_{k+1}$ with a contribution from the old direction. We can hence express the new search direction as

$$\mathbf{p}_{k+1} = -\nabla f_{k+1}^T + \beta \mathbf{p}_k \tag{31}$$

where β is some scalar that we need to determine. By combining equations 30 and 31 we get

$$\beta \mathbf{p}_k^T \mathbf{H} \mathbf{p}_k - \nabla f_{k+1} \mathbf{H} \mathbf{p}_k = 0 \tag{32}$$

which means

$$\beta = \frac{\nabla f_{k+1} \mathbf{H} \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{H} \mathbf{p}_k} \tag{33}$$

which would lead to the update rule

$$\mathbf{p}_{k+1} = -\nabla f_{k+1}^T + \frac{\nabla f_{k+1} \mathbf{H} \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{H} \mathbf{p}_k} \mathbf{p}_k \tag{34}$$

Did you spot the catch? Right, $\mathbf{H}$ is still in there. How then to get rid of it? Well, keep equation 34 in the back of your head for a minute and let us return briefly to equation 28 and use that to calculate the gradient at two points $\mathbf{w}$ and $\mathbf{w} + \lambda \mathbf{p}$. These would be $\nabla f(\mathbf{w})$ and $\nabla f(\mathbf{w}) + \lambda \mathbf{p}^T \mathbf{H}(\mathbf{w})$ respectively. If we now consider $\mathbf{p}$ to be $\mathbf{p}_k$ (*i.e.* the direction in which we took the kth step) and $\lambda \mathbf{p}$ as the step we took from point $\mathbf{w}_k$ to point $\mathbf{w}_{k+1}$ and that we there want to calculate the new direction $\mathbf{p}_{k+1}$. Our approximation above of the derivatives at $\mathbf{w}_k$ and $\mathbf{w}_{k+1}$ says that

$$\nabla f(\mathbf{w} + \lambda \mathbf{p}) - \nabla f(\mathbf{w}) \approx \lambda \mathbf{p}^T \mathbf{H}(\mathbf{w}) \tag{35}$$

But we have in fact calculated the gradient at the points $\mathbf{w}_k$ and $\mathbf{w}_{k+1}$ (equivalent to $\mathbf{w}_k + \lambda \mathbf{p}_k$), so if we substitute those into equation 35 we obtain

$$\mathbf{H} \mathbf{p}_k \approx \frac{1}{\lambda} \left(\nabla f_{k+1} - \nabla f_k \right)^T \tag{36}$$

and if we substitute this into equation 34 we obtain the Hestenes-Stiefel update

$$\mathbf{p}_{k+1} = -\nabla f_{k+1}^T + \frac{\nabla f_{k+1} \left(\nabla f_{k+1} - \nabla f_k \right)^T}{\mathbf{p}_k^T \left(\nabla f_{k+1} - \nabla f_k \right)^T} \mathbf{p}_k \tag{37}$$

There are various modifications that can be made to the update rule, *e.g.* by recognising that $\nabla f_{k+1} \mathbf{p}_k$ is zero at the present point. I will just mention the Polak-Ribiere form given by

$$\mathbf{p}_{k+1} = -\nabla f_{k+1}^T + \frac{\nabla f_{k+1} \left(\nabla f_{k+1} - \nabla f_k \right)^T}{\nabla f_k \nabla f_k^T} \mathbf{p}_k \tag{38}$$

and that seems to be favored by most authors.

2.4.2 Variable Metric methods

The Variable Metric methods avoids calculating the Hessian by gradually building an estimate of either $\mathbf{H}$ or of $\mathbf{H}^{-1}$ (wich we will call $\mathbf{A}$) based on information obtained from previous iterations. I will only touch on those that build an estimate of $\mathbf{H}^{-1}$ (or $\mathbf{A}$) since that will in addition spare us having to invert $\mathbf{H}$. The idea is to start with some arbitrary estimate of $\mathbf{A}$, for example the unity matrix. We would then take the following step in the parameter space

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{A}_k \nabla f(\mathbf{w}_k) \tag{39}$$

The problem with this is (of course) that initially it would be a minor miracle of this took us anywhere near the minimum since $\mathbf{A}$ is likely to be a very poor approximation to $\mathbf{H}^{-1}$. Since $\mathbf{A}$ is the unity matrix we have simply taken a step in the direction of the gradient, and more importantly the length of the step is also given by that gradient. This step length may well be several of orders of magnitude off and may well take us to a point far beyond the minimum, possibly to a point where the function f is much larger than at $\mathbf{w}_k$. Consider for example when $f_1(\mathbf{w}) = w_1^2 + w_2^4$ and compare that to $f_2(\mathbf{w}) = 1 \times 10^6 \left(w_1^2 + w_2^4\right)$. Clearly the minimum is at $\mathbf{w}_m = [0\ 0]^T$ for both f_1 and f_2, but at any point $\mathbf{w}$ the gradient (and hence also the implied step length) is 1×10^6 larger for f_2 than for f_1. Therefore, instead of taking the step implied by equation 39 we use it to define a search direction $\mathbf{p}_k$, *i.e.*

$$\mathbf{p}_k = -\mathbf{A}_k \nabla f(\mathbf{w}_k) \tag{40}$$

and then one uses a line-minimisation algorithm to find a scalar κ such that $f(\mathbf{w}_k + \kappa \mathbf{p}_k)$ is a minimum of f along the line given by $\mathbf{w}_k + \kappa \mathbf{p}_k$ for $\lambda > 0$. Once we know the value of κ we know (provided f is a quadratic in the region around $\mathbf{w}_k$) that the "true" $\mathbf{H}^{-1}$ should have fulfilled

$$\mathbf{w}_{k+1} - \mathbf{w}_k = \kappa \mathbf{p}_k = -\mathbf{A}_k \nabla f(\mathbf{w}_k) \tag{41}$$

The "trick" with variable metric methods now is to use this to find $\mathbf{A}_{k+1}$, a better approximation to $\mathbf{H}^{-1}$. There are of course a very large number of matrices that would fulfill equation 41 above, so which one to chose? One suggestion is

$$\begin{aligned}\mathbf{A}_{k+1} &= \frac{(\mathbf{w}_{k+1} - \mathbf{w}_k)(\mathbf{w}_{k+1} - \mathbf{w}_k)^T}{(\mathbf{w}_{k+1} - \mathbf{w}_k)^T (\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k))} \\ &- \frac{(\mathbf{A}_k (\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k)))(\mathbf{A}_k (\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k)))^T}{(\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k))^T \mathbf{A}_i (\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k))}\end{aligned} \tag{42}$$

which one can easily verify by replacing $\mathbf{A}_k$ in equation 41 with $\mathbf{A}_{k+1}$ in equation 42. This is known as the DFP updating formula. An alternative form is obtained by adding yet another term to equation 42 resulting in

$$\mathbf{A}_{k+1} = \cdots + (\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k))^T \mathbf{A}_i (\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k)) \mathbf{u}\mathbf{u}^T \tag{43}$$

where $\cdots$ indicates the update in equation 42 and

$$\begin{aligned}\mathbf{u} &= \frac{\mathbf{w}_{k+1} - \mathbf{w}_k}{(\mathbf{w}_{k+1} - \mathbf{w}_k)^T (\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k))} \\ &- \frac{\mathbf{A}_i (\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k))}{(\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k))^T \mathbf{A}_i (\nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k))}\end{aligned} \tag{44}$$

Again it is easy to use insertion to verify that this satisfies equation 41. This scheme is known as BFGS, and is allegedly a little more robust than DFP.

For this class of algorithms it has been shown that (conditional on there being a single minimum in the parameter space) the line minimisation (yielding $\mathbf{w}_{k+1}$ in equation 41) does not need to find the "true" minimum along the direction $\mathbf{p}_k$. It is sufficient that it finds a point along this line that satisfies the "Wolfe-condition".

2.5 Implementation within FSL

The methods outlined above have all been implemented as part of an "FSL non-linear optimisation library" that will hopefully facilitate future development of methods that need to perform non-linear optimisation. It consists of two classes `NonlinParam` and `NonlinCF` and a global function `nonlin` that takes an object each of `NonlinParam` and `NonlinCF` as input.

2.5.1 The `NonlinCF` class

The `NonlinCF` class is a virtual base class that is the "heart" of the nonlin library. We will simplify our discussion initially by demonstrating a slightly modified version of `NonlinCF`.

```
class NonlinCF
{
private:
  ...
public:
  NonlinCF() {}
  virtual ~NonlinCF() {}
  virtual NEWMAT::ColumnVector grad(const NEWMAT::ColumnVector& w) const;
  virtual NEWMAT::Matrix       hess(const NEWMAT::ColumnVector& w) const;
  virtual double               cf(const NEWMAT::ColumnVector& w) const = 0;
};
```

Note how the interface consists of three functions returning $f(\mathbf{w})$ (`cf`), $\nabla f\,|_{\mathbf{w}}$ (`grad`) and $\mathbf{H}\,|_{\mathbf{w}}$ (`hess`). The function returning $f(\mathbf{w})$ is pure virtual, since we cannot know what function the application programmer wants to minimise, and consequently `NonlinCF` is an abstract base class. The minimum amount of work that an application programmer has to do is to sub-class `NonlinCF` and supply the `cf` function.

As a concrete example let us assume we have some data consisting of a vector of times $\mathbf{t}$ and some measurements $\mathbf{y}$ performed at those times. Let us further assume that we believe that the model

$$y_i = w_1 e^{-w_2 t_i} + e_i, \quad e_i \sim N(0, \sigma^2) \tag{45}$$

describes our data and that we want to infer on w_1 and w_2. We would do that by finding the values for w_1 and w_2 that minimises the difference between the model predictions and the observed data $\mathbf{y}$. To accomplish this we create class which we may call `OneExpCF`, and which down to its bare bones might look something like

```
class OneExpCF: public NonlinCF
{
public:
  OneExpCF(const ColumnVector& pt, const ColumnVector& py) : t(pt), y(py) {
    /* Should do some error checking here */
  }
  ~OneExpCF() ;
  virtual double cf(const ColumnVector& p) const;
private:
  ColumnVector   t;    // Independent data (times) goes here
  ColumnVector   y;    // "Measured" data goes here
};

double OneExpCF::cf(const ColumnVector& w) const
{
  double cfv = 0.0;
  for (int i=1; i<=t.Nrows(); i++) {
    double err = y(i) - w(1)*exp(-w(2)*t(i));
    cfv += err*err;
  }
  return(cfv);
}
```

After that all we would need to do is to call the global function `nonlin`, passing it an instance of `OneExpCF`

```
ColumnVector   t, y;

.../* Get t and y from a file, the user or something. */

OneExpCF      mycf(t,y);
NonlinParam   mypar(2,NL_LM);

NonlinOut status = nonlin(mypar,mycf);

cout % << ‘‘w1 = ‘‘ << (mypar.Par())(1) << ‘‘and w2 = ‘‘ << (mypar.Par())(2)’’;
```

So, what then does `NonlinParam` do?

2.5.2 The `NonlinParam` class

`NonlinParam` is really nothing more than a glorified `struct` that passes information about minimisation method, convergence criteria *etc* to `nonlin`, and that returns results and diagnostic output from `nonlin`. The constructor for `NonlinParam` takes several tens of input arguments, but all except 2 of those have default values and an object of type `NonlinParam` is typically created like

```
int            no_of_par = 2;  // Two parameters to find values for
NLMethod       nlm = NL_LM;    // Use Levenberg-Marquardt
```

```
NonlinParam   my_par(no_of_par,nlm);
```

and then possibly modified using some of the access functions

```
my_par.SetGaussNewtonType(LM_L);  // Use Levenberg rather than Levenberg-Marquardt

ColumnVector  spar(2);
spar(1) = 1.0; spar(2) = 0.1;
my_par.SetStartingEstimate(spar); // Use this as starting estimate
```

The optimisation is then performed by a call to `nonlin` and the results are examined using the access function `.Par()`

```
Nonlinout status = nonlin(my_par,OneExpCF);

if (status == NL_MAXITER) {
  cout % << "Sorry, optimisation failed" << endl;
}
else {
  cout % << "The solution is w = " << my_par.Par() << endl;
}
```

There are also access functions that allows one to examine the results and the route taken to get there in more detail

```
my_par.LogPar();
my_par.LogCF();

Nonlinout status = nonlin(my_par,OneExpCF);

if (status == NL_MAXITER) {
  cout % << "Sorry, optimisation failed" << endl;
}
else {
  cout % << "The solution is w = " << my_par.Par() << endl;
  cout % << "And this is how we got there" << endl;
  for (int i=0; i<my_par.CFHistory().size(); i++) {
    cout % << "w1 = " << ((my_par.ParHistory())[i])(1);
    cout % << "w2 = " << ((my_par.ParHistory())[i])(2);
    cout % << ":  cf = " << (my_par.CFHistory())[i] << endl;
}
```

2.5.3 The actual implementation of `NonlinCF`

As alluded to above our description this far is actually a *slight* simplification of `NonlinCF`. The actual implementation looks like

```
class NonlinCF
{
```

```
private:
  ...
public:
  NonlinCF() {}
  virtual ~NonlinCF() {}
  virtual double sf() const return(1.0);
  virtual NEWMAT::ReturnMatrix grad(const NEWMAT::ColumnVector& p) const;
  virtual boost::shared_ptr<BFMatrix> hess(const NEWMAT::ColumnVector& p,
            boost::shared_ptr<BFMatrix> iptr=boost::shared_ptr<BFMatrix>()) const;
  virtual double cf(const NEWMAT::ColumnVector& p) const = 0;
};
```

As you can see the difference consists of the return values from `grad` and `hess` and in an additional parameter to `hess`. These differences are only there for efficiency, and the "principle" is really as defined in the previous sections. However, in order to implement efficient sub-classes of `NonlinCF` we need to touch also on these details.

The first difference is simply that `grad` now returns a value of type `NEWMAT::ReturnMatrix`. This is simply a standard way to invoking yet another constructor when returning an object in the Newmat hierarchy. The actual object being returned is still a `NEWMAT::ColumnVector`, the trick is just to declare the return value as being of typ `NEWMAT::ReturnMatrix`. An implementation of `grad` for the `OneExpCF` class might hence look like

```
NEWMAT::ReturnMatrix OneExpCF::grad(const NEWMAT::ColumnVector& p) const
{
  NEWMAT::ColumnVector gradv(p.Nrows());
  gradv = 0.0;
  for (int i=1; i<=x.Nrows(); i++) {
    double tmp = exp(-p(2)*x(i));
    gradv(1) -= 2.0*tmp*(y(i)-p(1)*tmp);
    gradv(2) += 2.0*p(1)*x(i)*tmp*(y(i)-p(1)*tmp);
  }

  gradv.Release();
  return(gradv);
}
```

where as you can see the only "strange" bits are the return type `ReturnMatrix` and the call `gradv.Release()` immediately prior to the return.

The second difference lies in that `hess` now returns something of type `boost::shared_ptr<BFMatrix>` rather than a `NEWMAT::Matrix`. `BFMatrix` is an abstract base class with two derived classes `FullBFMatrix` and `SparseBFMatrix`, which means that we are aiming for a polymorphic behaviour. In order to realise that `hess` has to return either a pointer or a reference to a base class. A reference would not be practical, which leaves us with returning a pointer. However, had we just allocated some object using `new` inside `hess` and returned the resulting pointer we would have delgated the responsibility of deleting that memory to the "user" (as in application programmer "user"), which is a memory leak waiting to happen.

Therefore we opted to return a "smart pointer", or specifically an object to type `boost::shared_ptr<T>`. A "smart pointer" is simply an object that for all practical purposes behaves like a pointer, but that ensures that when the pointer itself runs out of scope it deletes whatever memory it was pointing to. The `boost::shared_ptr<T>` takes this a step further such that if there are several pointers pointing to the same memory it ensures that that memory is not deleted until the last of those pointers goes out of scope. It is probably better you read about it yourselves at `http://www.boost.org/libs/smart_ptr/shared_ptr.htm`.

The second issue is the `BFMatrix` class itself. It is a "wrapper"/"container" class that contains either a full matrix of type `NEWMAT::Matrix` or a sparse matrix of type `MISCMATHS::SpMat`. As a "standard" user of `nonlin` you only really need to know how to create an object of type `FullBFMatrix`. If you have an application where the Hessian is truly sparse (less than 25% of values non-zero) I recommend you have a look at the document noninreg.pdf for examples.

So, how would we implement `hess` for the `OneExpCF` class?

```
boost::shared_ptr<BFMatrix> OneExpCF::hess(const NEWMAT::ColumnVector&   p,
                                           boost::shared_ptr<BFMatrix>   iptr) const
{
  boost::shared_ptr<BFMatrix>   hessm;

  if (iptr && iptr->Nrows()==p.Nrows() && iptr->Ncols()==p.Nrows()) hessm = iptr;
  else hessm = boost::shared_ptr<BFMatrix>(new FullBFMatrix(p.Nrows(),p.Nrows()));
  for (int i=1; i<=2; i++) {for (int j=1; j<=2; j++) hessm->Set(i,j,0.0);}

  for (int i=1; i<=x.Nrows(); i++) {
    double tmp = exp(-p(2)*x(i));
    double tmp2 = exp(-2.0*p(2)*x(i));
    hessm->AddTo(1,1,2.0*tmp2);
    hessm->AddTo(1,2,2.0*x(i)*y(i)*tmp - 4.0*p(1)*x(i)*tmp2);
    hessm->AddTo(2,2,4.0*SQR(p(1))*SQR(x(i))*tmp2 - 2.0*p(1)*SQR(x(i))*y(i)*tmp);
  }
  hessm->Set(2,1,hessm->Peek(1,2));

  return(hessm);
}
```

The crucial statements here are

```
hessm = boost::shared_ptr<BFMatrix>(new FullBFMatrix(p.Nrows(),p.Nrows()));
```

which allocates memory for an `p.Nrows()`-by-`p.Nrows()` matrix and points a "smart pointer" to it, and the staements of type `hessm->Set(i,j,val)`, `hessm->AddTo(i,j,val)` and `hessm->Peek(i,j)` that sets, adds to and examines the location i,j in the matrix `hessm`.

The final difference lies in the `iptr` input parameter. It is simply there to allow for reuse of the memory that was used to store `hessm` in the previous call. It is probably of little consequence for most applications, but quite nice to have as a possibility when the Hessian starts to grow to hundreds of Megabytes.