TMA 4180 Optimeringsteori

Least Squares Optimization

Harald E. Krogstad, rev. 2010

This note was originally prepared for earlier versions of the course. Nocedal and Wright has a nice introduction to *Least Square* (LS) optimization problems in Chapter 10, and the note is now therefore only a small supplement. It reflects that LS problems are by far the most common case for unconstrained optimization. See also N&W, p. 245–250 for typical examples of LS problems.

Least Square problems have often their origin in fitting models to observations. In its simplest form, we know this from the problem of fitting a *regression line*, $y = ax + b$, through a set of data points $\{x_i, y_i\}$, $i = 1, \cdots, N$. When $N > 2$, it is in general impossible to put the line through all points, but we try to determine an optimal line, for example by determining the pair $\{a^*, b^*\}$ which minimizes the objective function

$$f(a,b) = \sum_{i=1}^{N} (ax_i + b - y_i)^2 . \tag{1}$$

In this simple case, it is easy to derive the solution analytically, but in general the solution has to be found numerically. Since these problems are so common, there has been a lot of work involved in adapting the general algorithms for unconstrained optimization to this special case. It is more effective to use specially adapted algorithms instead of the more general ones.

Below we first consider the *linear* LS problem, which some of you would know from linear analysis or multiple linear regression in statistics. The notation may differ somewhat from what you know, but the ideas are the same.

The *singular value decomposition* is an important tool for analyzing the linear problem, and the linear problem is also important for the solutions of the various iteration steps in non-linear LS problems. In the non-linear algorithms we once more meet basic tools like line search and trust region methods, which utilize the special form of the gradient and Hessian the LS problems have.

1 Linear Least Square Problems

1.1 The origin

Although LS problems stem from a variety of different situations, we shall consider a situation where n *factors* influence the result of an *experiment*. For each experiment we know the values of the factors, say $\{a_1, ..., a_n\}$, and we are able to measure the *result* β.

We are interested in finding a function $f(a_1, ..., a_n)$ which can be used to compute the result without having to repeat the experiment each time. In practice, this may be quite difficult:

- A lot of factors may be involved

- We are not quite sure whether a factor counts or not
- There may be factors unknown to us
- We have *no* idea how the function f looks
- It is expensive and/or time consuming to carry out experiments
- We have only limited time and budget to figure out a solution

All these points are part of real life, where you often find yourself in a group of colleagues where you are the least incompetent member.

The first guess if one does not know anything will often be to assume that there is *linear* dependency between the factors and the result. In other words, we are looking for a set of coefficients or weight factors $\{x_1, \cdots, x_n\}$ such that

$$a_1x_1 + a_2x_2 + ... + a_nx_n \approx \beta. \tag{2}$$

Note that things are turned around here: When we apply the formula after the x-s are determined, we put in $\{a_1, \cdots, a_n\}$ and get an estimate of β, denoted $\hat{\beta}$. The expression should in general also include a constant term, and this is obtained by letting $a_1 = 1$, so that the formula reads $\hat{\beta} = x_1 + a_2x_2 + \cdots + a_nx_n$.

In order to determine the x-s, we need to carry out a set of *calibration experiments*. If we do not know anything about the x-s, we would like to have $m > n$ experiments: If the number of experiments is less than n (often the situation in practice), there will in general be many different sets of weight factors fitting Eqn. 2, and if we have exactly n experiments, we could still satisfy Eqn. 2 exactly, but would have no idea how well the equation is satisfied in other cases.

Let us store the values of the factors in the $m \times n$ matrix A and the results in the m dimensional vector b, such that

$$\begin{aligned} A &= \{a_{i1}, a_{i2}, \cdots, a_{in}\}, \;\; i = 1, \cdots, m, & (3) \\ b &= \{\beta_1, \cdots, \beta_m\}'. & (4) \end{aligned}$$

Since it would be too good to be true if we really found a vector x such that $Ax = b$, we are satisfied if we are able to find x such that the sum of the squared deviations is minimized:

$$\min_x \sum_{i=1}^{m} \left(a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n - \beta_i\right)^2. \tag{5}$$

In matrix notation this is equivalent to finding the minimum of the function

$$f(x) = \frac{1}{2}\,\|Ax - b\|_2^2 = \frac{1}{2}(Ax - b)'(Ax - b). \tag{6}$$

1.2 Solution of the full rank problem

By carrying out the multiplications in Eqn. 6, we find that

$$f(x) = \frac{1}{2}(Ax-b)'(Ax-b) = \frac{1}{2}x'A'Ax \; - x'A'b + \frac{b'b}{2}, \tag{7}$$

which we, by now, should recognize as the *quadratic test problem.*

If $A'A$ happens to be positive definite (which requires that $m \geq n$), we already know that the solution, x^*, satisfies the equation

$$A'Ax = A'b. \tag{8}$$

(Convince yourself that all matrix multiplications make sense!). The system of equations in 8 is called the *Normal Equations.*

It turns out that $A'A$ is *positive definite if and only if A has full column rank.* In this case, where $m > n$, this means that $\operatorname{rank}(A) = n$, or that all columns of A are linearly independent. That is equivalent to $Ax = 0 \Leftrightarrow x = 0$, which in turn is equivalent to $x'A'Ax = \|Ax\|^2 = 0 \Leftrightarrow x = 0$.

Even if A has full rank mathematically, it is not necessarily reasonable numerically to solve the normal equations. The *condition number* of A is the ratio between its largest and smallest singular value (see below), and the condition number of $A'A$ will be the square of that. In order to avoid numerical problems, the best is to use a QR-factorization of A. The 2-norm is invariant under orthogonal transformations, and this reduces the problem to solving a linear system of equations with n unknowns which has the upper $n \times n$ sub-matrix of R as its coefficient matrix. A short summary of this is given in N&W p. 251, and a more extensive treatment is found in the book of Golub and vanLoan [2], Chapter 5.

Cases with few factors and a large number of experiments are not likely to experience any problems with the Normal Equations.

1.3 The rank deficient problem

If the rank of A, $r = \operatorname{rank}(A)$, is less than n, the problem will still have solutions, but they are not unique. The *null space* of A, $\mathcal{N}(A)$, is the set of all vectors z such that $Az = 0$, and when the column rank, $\operatorname{rank}(A)$, is less than n, $\mathcal{N}(A)$ contains non-zero vectors (recall the definition of rank!).

Obviously, if x^* is a solution to 6, then $x^* + z$, $z \in \mathcal{N}(A)$ will be a solution as well since $A'Az = 0$ and $f(x^* + z) = f(x^*)$.

The existence of solutions is not obvious, but follows by considering Eqn. 8 as an equation not in $\mathbb{R}^n$ but in the smaller *orthogonal complement of the null-space of* A, $\mathcal{N}(A)^{\perp}$. Check out all definitions you do not know/recall and try to carry out the following steps:

1. Show that $A'b$ and $A'Ax$ are members of $\mathcal{N}(A)^{\perp}$ by taking the scalar product with a vector $z \in \mathcal{N}(A)$.

2. Show that the matrix $A'A$ is non-singular when restricted to $\mathcal{N}(A)^{\perp}$ (This amounts to show that $\|Ay\| = 0 \iff y = 0$ when $y \in \mathcal{N}(A)^{\perp}$).

3. Show that there is a unique $x^* \in \mathcal{N}(A)^{\perp}$ such that $A'Ax^* = A'b$

(This may be a little tough if you are not familiar with linear algebra and linear transformations. It is not that important, – the solution exists).

The rank deficient case will occur if we try to explain our result by including too many factors. In practice, it is smart to start with few factors and then include one additional factor at a time. By supervising $\|Ax - b\|$, we get a feeling of when enough is enough.

Even if A is rank deficient, the function $f(x)$ in Eqn. 7 is still *convex* since $A'A \geq 0$, and the set of global minima, $\Gamma = \{y; f(y) = \min\}$, is therefore a closed convex set (We proved above that there are indeed optimal solutions, so Γ is definitely non-empty).

Are all these possible solutions equally good? Mathematically yes, but in practice we are using x for *predicting* or *estimating* β, and the predictor $\hat{\beta}$ for a given set of factors, a, is simply

$$\hat{\beta} = a'x^*. \tag{9}$$

Unnecessary large components in x^* will magnify any measuring errors in the factors a. Therefore, it is reasonable to use as small x^* as possible.

If we, on the convex set Γ, define the strictly convex function $g(y) = \|y\|_2^2$, the function will have a unique minimum, and *this* would be the special solution we are looking for:

$$x^* = \underset{A^tAx = A^tb}{\operatorname{argmin}} \|x\|_2^2 . \tag{10}$$

Can you prove that x^* has to be equal to the solution in $\mathcal{N}(A)^{\perp}$? (Recall that any solution may be written uniquely as $x^* = x_0 + z$, where $z \in \mathcal{N}(A)$ and $x_0 \in \mathcal{N}(A)^{\perp}$).

There are several numerical methods for finding x^* described in N&W, as well as in [2]. The most robust although not the cheapest computationally is based on the Singular Value Decomposition of A.

1.3.1 The Singular Value Decomposition

The *Singular Value Decomposition (SVD)* is a door-opener to a lot of practical algorithms, and very much used in modern applications of linear algebra.

The *reduced form* SVD of a general $m \times n$ matrix A is

$$A = U\Sigma V' = \sum_{i=1}^{r} \sigma_i u_i v_i', \tag{11}$$

where

$$U = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_r \\ | & | & & | \end{bmatrix}, \; V = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \dots & v_r \\ | & | & & | \end{bmatrix} \tag{12}$$

have orthogonal columns,

$$\begin{aligned} U'U &= I_{(r)}, \\ V'V &= I_{(r)}, \end{aligned} \tag{13}$$

and $r = \operatorname{rank}(A)$.

The $r \times r$ *diagonal* matrix Σ contains the so-called *singular values* of A. These singular values are the square root of the non-negative eigenvalues of $A'A$, or AA' (Note that the matrices U and V may be extended to full $m \times m$ and $n \times n$ orthogonal matrices and Σ to an $m \times n$ matrix, as shown in N&W, p. 598. We do *not* assume or need this below).

There are a few things to check here:

- The matrix A represents a linear transformation from $\mathbb{R}^n \to \mathbb{R}^m$ and, similarly, A' a linear transformation from $\mathbb{R}^m \to \mathbb{R}^n$.
- $v_1, v_2, ..., v_r \in \mathbb{R}^n$ and $u_1, u_2, ..., u_r \in \mathbb{R}^m$.
- U and V consist of the *eigenvectors* of AA' and $A'A$ that correspond to non-zero eigenvalues, $(AA')\,U = U\Sigma^2$ and $(A'A)\,V = V\Sigma^2$.
- The columns of V span an *orthogonal basis* for $\mathcal{N}(A)^\perp \subset \mathbb{R}^n$, and the vectors in U an orthogonal basis for the range of A, $\mathcal{R}(A) \subset \mathbb{R}^m$.

The matrix

$$A^+ = V\Sigma^{-1}U' \tag{14}$$

is called a *pseudo-inverse* or a *generalized inverse* of A, and is the most famous member among a large number of generalized inverses. This particular generalized inverse is called the *Moore-Penrose inverse* of A.

Exercise: Prove that A^+A acts as the identity matrix on $\mathcal{N}(A)^\perp$, and AA^+ as the identity matrix on $\mathcal{R}(A)$ (Hint: Use that all vectors $x \in \mathcal{N}(A)^\perp$ may be written $x = Vs$, $s \in \mathbb{R}^r$, and similarly $z = Ut, t \in \mathbb{R}^r$ for $z \in \mathcal{R}(A)$).

You could consult a textbook (E.g. [2] or your own linear algebra book) in order to brush this up. It is not so easy to grasp, – or to remember!

1.3.2 The solution in terms of the SVD

Given the SVD of A, the unique solution with the smallest norm of Eqn. 8 is

$$x^* = A^+b = V\Sigma^{-1}U'b = \sum_{i=1}^{r} \frac{u_i'b}{\sigma_i} v_i. \tag{15}$$

(Prove this yourself by showing that $A'A\,(A^+b) = A'b$!).

In Eqn. 15, we assume that the singular values are ordered according to their size, that is,

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r. \tag{16}$$

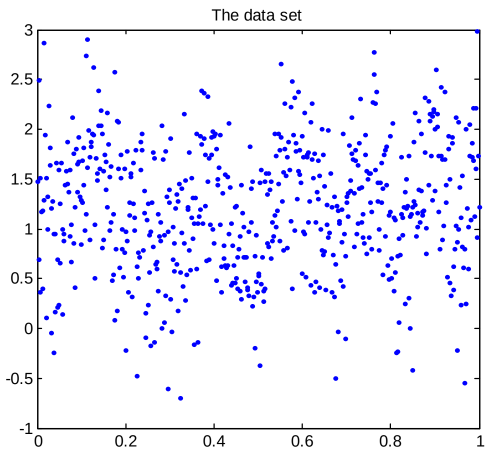

Figure 1: Simulated 600 data points with some underlying function added to a lot of noise.

This is the opposite ordering from what we have been using for eigenvalues.

Contrary to what you learn in the linear algebra courses, determining the rank of a big matrix is far from trivial. If the singular values decrease gradually to 0, it is quite difficult to say where to stop (perhaps when $\sigma_i < 10^{-16}$?) and define the rest to be 0. The nice thing about Eqn. 15 is that we can add one term at a time and stop when the solution starts to get unstable. Small singular values magnify errors in b, U or V, and this case is called an *ill-conditioned* problem.

It seems to be an inherent property of real life problems of some size that they all tend to be ill-conditioned. This will be discussed later for *Inverse Problems*.

1.3.3 A worked example

The following simple and somewhat unrealistic example shows how the SVD works. We have a large and very noisy data set of a function of one variable (y). The values of y range between 0 and 1, and the idea is to extract the function by fitting a polynomial. Typical data are shown in Fig. 1, and if this has been *real* data set, we would not even think of fitting something else than a constant value, or at most a straight line. The data are produced by adding independent normally distributed noise with standard deviation equal to 0.6 to the function values. Let us nevertheless take a very bold attitude and try to fit a polynomial of order 60 (!) by just running the Matlab functions **polyfit** and **polyval**:

```
p = polyfit(y,b,60);
Y = polyval(p,y);
plot(y,data,'.',y,Y);
```

In this case, the factors are the various powers of variable y, $a = \left\{1, y, y^2, \cdots, y^{60}\right\}$ and the weights x are the coefficients in the polynomial,

$$\hat{\beta} = x_1 + x_2 y + x_3 y^2 + \cdots + x_{61} y^{60}. \tag{17}$$

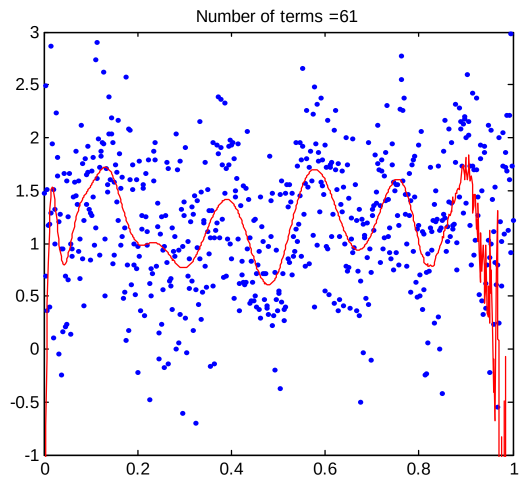

Figure 2: Data and 60th order polynomial fitted by **polyfit** in Matlab.

As expected, the result is really not that great, as Fig. 2 shows. Also Matlab protests and warns us that the matrices involved are close to singular. The program suggests removing some data points, or do some centering and scaling of the data. Nothing of this would actually help, – the obvious cure is clearly to reduce the order of the polynomial. But let us not give up that fast. We stick to 60th order and form the *factor matrix* A of dimension 600×61 (61 factors and 600 experiments).

$$A = \begin{bmatrix} 1 & y_1 & y_1^2 & \cdots & y_1^{60} \\ 1 & y_2 & y_2^2 & \cdots & y_2^{60} \\ \vdots & & \vdots & & \vdots \\ 1 & y_{600} & y_{600}^2 & \cdots & y_{600}^{60} \end{bmatrix}, \tag{18}$$

and the data vector $b = (b_1, b_2, \cdots, b_{600})'$.

We are then faced with the problem of finding the best loads $x = (x_1, x_2, \cdots, x_{61})'$ to the data, that is, solve the LS problem,

$$x^* = \arg \min_{x \in \mathbb{R}^{61}} \|Ax - b\|_2^2 \,. \tag{19}$$

Let us consider two possibilities:

- Use of the SVD
- Use of the normal equations

Both are straightforward to program, the SVD even as simple as `[U Sigma V] = svd(A)`! The rank (A) computed by Matlab is equal to 26, and this seems to be in rough agreement with a plot of the singular values shown on Fig. 3. It is tempting to use the solution on the SVD-form,

$$x^* = V\Sigma^{-1}U'b = \sum_{i=1}^{r} \frac{u_i' b}{\sigma_i} v_i. \tag{20}$$

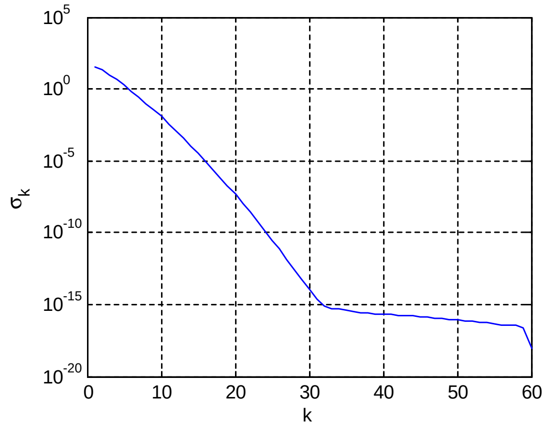

Figure 3: The singular values of the A-matrix. The machine accuracy seems to be reached after about 30 singular values, but the singular values never become exactly 0.

where we include one term at a time. This is displayed in Fig. 4, where the result is plotted every eight term. The graphs in the middle show some similarities, and this is even true for the graphs at the bottom, although using more terms in Eqn. 20 than the rank is not very meaningful. The SVD solution with $r = \operatorname{rank}(A)$ terms, the Normal equation solution, and the (up to now) unknown true function are shown in Fig. 5. The final plot, Fig. 6 shows the SVD solution and the fit of a polynomial of order r. The solutions are about equally good, but we have not tried to optimize this further. By running the program with different data, the optimal SVD solution seems, on the average, to be slightly better than the Normal Equation solution. The reader should be warned that the, after all, civilized behaviour of these results is in part due to Matlab's very carefully coded numerics!

2 Non-Linear Least Square Problems

In the general non-linear least square case we have a vector-valued function, $h(x) \in \mathbb{R}^m$ where $x \in \mathbb{R}^n$, and we want to find the minimum of

$$f(x) = \frac{1}{2}\left\|h(x)\right\|_2^2 = \frac{1}{2}\sum_{i=1}^{m} h_i(x)^2 = \frac{1}{2}h(x)'h(x). \tag{21}$$

The simple Example 10.1 in N&W on p. 247–249 is of this form.

All the gradients of the various components in h make up what is called the *Jacobian matrix* of h, defined by

$$J(x) = \left\{\frac{\partial h_i}{\partial x_j}(x)\right\}_{\substack{i=1,\dots,m,\\ j=1,\dots,n}} = \begin{bmatrix} - & \nabla h_1(x) & - \\ - & \nabla h_2(x) & - \\ & \vdots & \\ - & \nabla h_m(x) & - \end{bmatrix}. \tag{22}$$

Recall that we assume that ∇h_i are n-dimensional *row* vectors. By taking derivatives term by

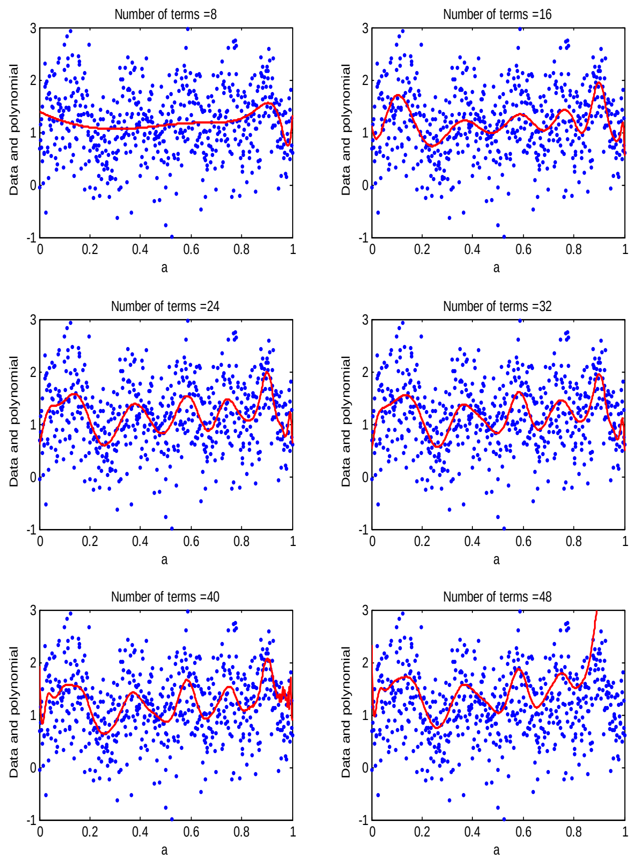

Figure 4: SVD solutions truncated at the term indicated. All polynomials have order 60 (The data set here differs from the above).

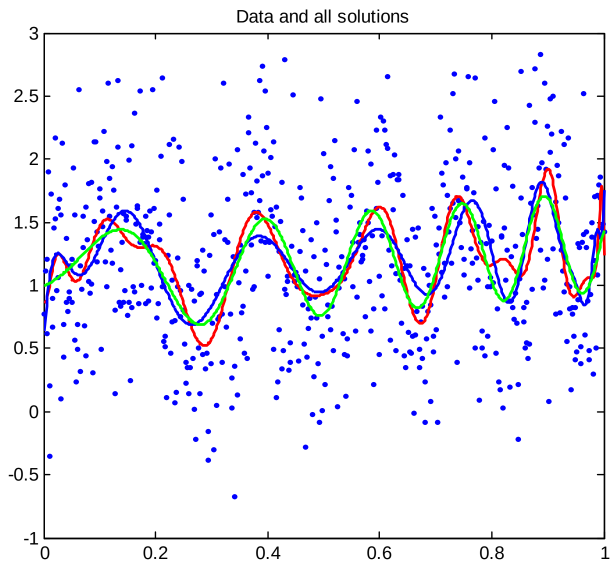

Figure 5: Data and all polynomials: Red: SVD solution with r terms; Blue: Normal Equations; Green: Underlying exact function.

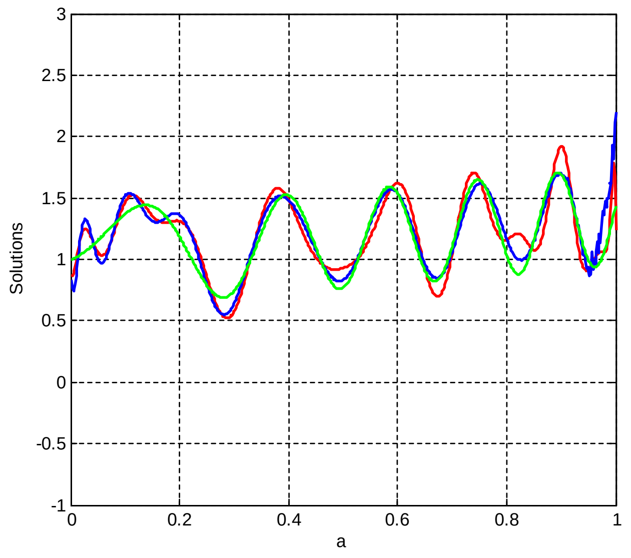

Figure 6: Red: SVD solution of order $r = \text{rank}\,(A)$. Blue: A simple polynomial fit of order r. Green: Input function.

term, we easily obtain (do it yourself!) that the gradient and the Hessian of f are

$$\begin{aligned} \nabla f(x)' &= J'(x)h(x), && (23) \\ \nabla^2 f(x) &= J'(x)J(x) + \sum_{i=1}^{m} h_i(x)\nabla^2 h_i(x), && (24) \end{aligned}$$

where $\left\{\nabla^2 h_i\right\}$ are the Hessian matrices for h_i , $i = 1, \cdots, m$.

Note that if $h_i(x)$ are linear functions, say $h(x) = Ax - b$, then $f(x) = \frac{1}{2}\left\|Ax - b\right\|_2^2$, and

$$\begin{aligned} \nabla f(x)' &= A'(Ax - b) = A'h(x), \\ \nabla^2 f(x) &= A'A. && (25) \end{aligned}$$

We observe that A corresponds to the Jacobian, J, whereas the Hessian in the general non-linear case gets an additional term

$$B = \sum_{i=1}^{m} h_i(x)\nabla^2 h_i(x). \tag{26}$$

If the minimization leads to small values of $h_i(x)$, or the problem is nearly linear such that $\nabla^2 h_i(x)$-s are really small, the first term in $\nabla^2 f(x)$ will dominate and

$$\nabla^2 f(x) \approx J(x)'J(x). \tag{27}$$

This is utilized in *Gauss-Newtons method.* We remember that the iteration step in Newton's method could be written

$$\nabla^2 f(x_k)(x_{k+1} - x_k) = -\nabla f(x_k)', \tag{28}$$

or, in terms of the *search direction*, p_k,

$$\nabla^2 f(x_k)p_k = -\nabla f(x_k)'. \tag{29}$$

In the Gauss-Newtons method, the search direction p_k for a line search, $x_{k+1} = x_k + \alpha_k p_k$, is obtained by approximating the Hessian by the *first part* of the sum in Eqn. 24,

$$J(x_k)'J(x_k)p_k = -J(x_k)'h(x_k). \tag{30}$$

We observe from the previous section that this is the same as solving the *linear* LS problem

$$\min_p \left\|J(x_k)p + h(x_k)\right\|_2 . \tag{31}$$

Practical experience has shown that if $J(x)$ has full rank and $h(x)$ really gets small close to the solution, $J(x_k)'J(x_k)$ becomes a good approximation to the exact Hessian and Gauss-Newton converges about as fast as full Newton. This is illustrated for Gauss-Newton on the Rosenbrock Banana Function on Fig. 7, copied from Matlab Optimization Toolbox documentation. The guide says that only 48 function evaluations were needed, even when the gradient was computed numerically. If, on the contrary, $J(x)$ is rank deficient, or the residuals (the components of h) are not small, the performance may be very poor. This is discussed in N&W, p. 256–257.

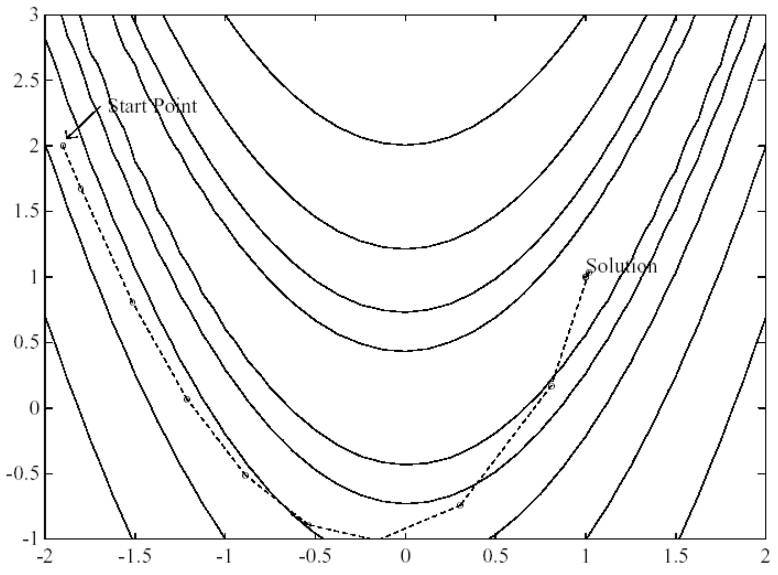

Figure 7: This figure, copied from the documentation of Matlab *Optimization Toolbox*, shows that Gauss-Newton is very efficient when the last part of the Hessian matrix may be ignored.

Whereas Gauss-Newton is a *line search* method, it is also reasonable use the same approximate Hessian in a *trust region* setting. The quadratic approximation to the objective is then

$$m\left(p\right) = f_k + \left(J_k^{\prime} h_k\right)^{\prime} p + \frac{1}{2} p^{\prime} \left(J_k^{\prime} J_k\right) p, \tag{32}$$

and with a spherical domain D with radius Δ, we have the same type of model problem as before,

$$p^* = \operatorname{argmin}_{s \in D} m\left(p\right). \tag{33}$$

If $J(x_k)$ has full rank, the solution of 33 will be identical to the Gauss-Newton solution,

$$p_{GN} = -\left(J_k^{\prime} J_k\right)^{-1} \left({J_k}^{\prime} h_k\right), \tag{34}$$

as long as $\|p_{GN}\| \leq \Delta$. Otherwise, the solution has to be on the boundary of D and obtained from

$$\left(J_k^{\prime} J_k + \lambda_k I\right) p = -{J_k}^{\prime} h_k. \tag{35}$$

with a value on λ_k that ensures $\|p\| = \Delta$. Note that in this case, the solution of the constrained problem is straightforward (in principle): $J_k^{\prime} J_k \geq 0$, and $J_k^{\prime} J_k + \lambda I > 0$ for all $\lambda > 0$. Moreover, $\|p_\lambda^*\| \underset{\lambda \to \infty}{\longrightarrow} 0$.

This trust region algorithm is called *Levenberg-Marquardt's method,* and is the standard method for non-linear LS problems and one of the algorithm implemented in Optimization Toolbox in MATLAB.

If it is possible to compute the matrix B in a reasonable way, it will in general be preferable to use it, as discussed in [1]. Such methods are implemented in the NAG library of numerical algorithms.

References

[1] Gill, P.E. and W. Murray: Algorithms for the solution of the nonlinear least-square problems, *SIAM J. Num. Anal.* Vol. 15(1978) pp. 977-992.

[2] Golub, G.H. and C.F. VanLoan: *Matrix Computations*, Johns Hopkins University Press, 1989.

[3] Luenberger, D.G.: *Linear and Nonlinear Programming*, 2nd ed., Addison Westley, 1984.