An Implementation of the Levenberg-Marquardt Algorithm

Lukas Finschi

Eidgenössische Technische Hochschule Zürich

Institut für Operations Research
Clausiusstrasse 45
CH-8092 Zürich

April 16, 1996

Abstract

Minimization of a quadratic function over a sphere can be done in a very efficient way by use of the Levenberg-Marquardt algorithm. This paper gives a short introduction to the techniques of this algorithm and presents an implementation of it. This implementation was realized in C and extended with links to Mathematica and S-Plus.

Contents

1 Introduction

1.1 Two Quadratic Problems

We consider the following two quadratic minimization problems:

$$\min_{\|\delta\| \le h} q(\delta) \tag{1}$$

$$\min_{\|\delta\| = h} q(\delta) \tag{2}$$

where

$$q : \mathbb{R}^n \to \mathbb{R}, \quad \delta \mapsto q(\delta) := \frac{1}{2}\delta^T G \delta + g^T \delta \tag{3}$$

is an arbitrary quadratic function with $G \in \mathbb{R}^{n \times n}$ a symmetric matrix, $g \in \mathbb{R}^n$, and $h > 0$; the norm is the (euclidean) ℓ_2-norm defined by $\|\delta\| := \sqrt{\delta^T \delta}$.

1.2 The Basic Theorem

Since G is symmetric, there exist n eigenvectors $v_1, v_2, \ldots, v_n$ of G with corresponding eigenvalues $\lambda_{max} := \lambda_1 \ge \lambda_2 \ge \cdots \ge \lambda_n =: \lambda_{min}$. We will use this notation for the rest of this paper.

The problem given by (1) will be solved (numerically) by use of the Levenberg-Marquardt algorithm, which is based on the following theorem:

Theorem 1 *Given problem (1) and a $\delta \in \mathbb{R}^n$ with $\|\delta\| \le h$. Then:*
δ is a solution of (1) if and only if there exists a $\nu \ge 0$, such that

(i) $G + \nu \mathbb{1}$ *is positive semidefinite*

(ii) $(G + \nu \mathbb{1})\delta = -g$

(iii) if $\nu > 0$, *then* $\|\delta\| = h$

If such a ν exists, then it is unique (in particular: ν does not depend on the solution δ, only on G, g, and h).

A proof of theorem 1 can be found in [1] and in [2].

Before we study problem (1), we show how the second problem given by (2) can be reduced to the first.

1.3 Peripheral Minimization

For $\mu > \lambda_{min}$, $\tilde{G} := G - \mu \mathbb{1}$ is not positive semidefinite. Then theorem 1 says, that for every solution $\tilde{\delta}$ of

$$\min_{\|\delta\| \le h} \frac{1}{2}\delta^T \tilde{G} \delta + g^T \delta \tag{4}$$

we have $\|\tilde{\delta}\| = h$ (since $\nu > 0$). Then $\tilde{\delta}$ is a solution of

$$\min_{\|\delta\| = h} \frac{1}{2}\delta^T \tilde{G} \delta + g^T \delta \tag{5}$$

and because of

$$\frac{1}{2}\delta^T \tilde{G} \delta + g^T \delta = \frac{1}{2}\delta^T G \delta + g^T \delta - \frac{1}{2}\mu \|\delta\|^2 \tag{6}$$

a solution of (2) (remark: the term $-\frac{1}{2}\mu\|\delta\|^2$ in (6) will be constant in (5) since $\|\delta\| = h$). So, a solution of (2) can be found in the following way:

Algorithm 1
Find a $\mu > \lambda_{min}$ (e.g. by use of Gershgorin discs)
$\tilde{G} := G - \mu \mathbb{1}$
Solve (4) $\Rightarrow$ solution $\tilde{\delta}$ (with minimal value $\tilde{q}(\tilde{\delta})$)
Then $\tilde{\delta}$ is a solution of (2) and $q(\tilde{\delta}) = \tilde{q}(\tilde{\delta}) + \frac{1}{2}\mu h^2$.

For an implementation of algorithm 1 see appendix A (procedure `lm_opt_peripheral`).
As theorem 1 gives a characterization of the solutions of (1), the following theorem does the same for problem (2):

Theorem 2 *Given problem (2) and a $\delta \in \mathbb{R}^n$ with $\|\delta\| = h$. Then:*
δ is a solution of (2) if and only if there exists a $\nu \in \mathbb{R}$, such that

(i) $G + \nu\mathbb{1}$ is positive semidefinite

(ii) $(G + \nu\mathbb{1})\delta = -g$

If such a ν exists, then it is unique.

Proof:
(*Only if)* Let δ be a solution of (2), $\mu > \lambda_{min}$, and $\tilde{G} := G - \mu\mathbb{1}$ (then $\tilde{G}$ is not positive semidefinite). In (6) the term $-\frac{1}{2}\mu\|\delta\|^2$ is constant for all δ with $\|\delta\| = h$, so the given δ is a solution of (5) and (with theorem 1 (i) and (iii)) a solution of (4). By theorem 1 there exists a $\tilde{\nu} \geq \mu - \lambda_{min} > 0$, such that $(\tilde{G} + \tilde{\nu}\mathbb{1})\delta = -g$ holds. With $\nu := \tilde{\nu} - \mu$ we have $\tilde{G} + \tilde{\nu}\mathbb{1} = G + \nu\mathbb{1}$, so $G + \nu\mathbb{1}$ is positive semidefinite and $(G + \nu\mathbb{1})\delta = -g$ holds.
(*If)* Let $\nu \in \mathbb{R}$ be a solution of (i) and (ii), and let $\mu > \lambda_{min}$. Set $\tilde{G} := G - \mu\mathbb{1}$ and $\tilde{\nu} := \nu + \mu$, then we know (because of $\lambda_{min} + \nu \geq 0$) that $\tilde{\nu} > 0$. Since $\tilde{G} + \tilde{\nu}\mathbb{1} = G + \nu\mathbb{1}$, $\tilde{G} + \tilde{\nu}\mathbb{1}$ is positive semidefinite and $(\tilde{G} + \tilde{\nu}\mathbb{1})\delta = -g$. Then (with theorem 1), δ is a solution of (4) and, as seen before, a solution of (5) and of (2).
(*Uniqueness)* For $i \in \{1, 2\}$ let ν_i and δ_i be solutions of (i) and (ii), i.e. $\nu_i \geq -\lambda_{min}$ and $(G+\nu_i\mathbb{1})\delta_i = -g$. Let be $\mu > \lambda_{min}$ and $\tilde{G} := G - \mu\mathbb{1}$, then as above we find $\tilde{\nu}_i$ as in theorem 1. From the uniqueness of the parameter in theorem 1, we have $\tilde{\nu}_1 = \tilde{\nu}_2 =: \tilde{\nu}$. From $(\tilde{G} + \tilde{\nu}\mathbb{1})\delta_i = (G + (\tilde{\nu} - \mu)\mathbb{1})\delta_i = -g = (G + \nu_i\mathbb{1})\delta_i$ we conclude $\nu_i\delta_i = (\tilde{\nu} - \mu)\delta_i$. Because of $\|\delta_i\| = h > 0$, we have $\delta_i \neq 0$ and $\nu_i = \tilde{\nu} - \mu$, i.e. $\nu_1 = \nu_2$.

For the rest of this paper, we will concentrate on problem (1).

1.4 The Idea of the Levenberg-Marquardt Algorithm

We want to find a solution of problem (1) and will do this by use of theorem 1.

First we have to find the ν, such that in theorem 1 (i), (ii), and (iii) hold, so we seek in the intervall $(\max\{0, -\lambda_{min}\}, \infty)$ for a ν^* with $\|\delta\| = \|(G + \nu^*\mathbb{1})^{-1}(-g)\| = h$ (in the Levenberg-Marquardt algorithm this is done using the efficient technique of subsection 2.1); if such a ν^* exists, it is unique and we have $\nu > -\lambda_{min}$ (in the implementation, this is called the `NORMAL CASE`); otherwise the ν of theorem 1 is equal to $\max\{0, -\lambda_{min}\}$: either $\nu = 0$ (`CASE NU == 0`) or $\nu = -\lambda_{min} > 0$ (`HARDCASE`).

In each possible case we like to find a solution δ, so we have to study the set of solutions depending on ν:

Either we have $\nu > -\lambda_{min}$, then $G + \nu\mathbb{1}$ is positive definite and the solution δ of (1) is unique, given by $(G + \nu\mathbb{1})^{-1}(-g)$.

Or $\nu = -\lambda_{min}$, then $\delta \in \mathbb{R}^n$ is a solution of (1) if and only if it is of the form $\delta = \bar{\delta} + \delta'$, where

$$\bar{\delta} := \lim_{\substack{\nu \to -\lambda_{min} \\ \nu > -\lambda_{min}}} (G + \nu\mathbb{1})^{-1}(-g) = \sum_{\substack{i=1 \\ \lambda_i \neq \lambda_{min}}}^{n} (-v_i^T g)\frac{1}{\lambda_i - \lambda_{min}} v_i$$

and $\delta' = \sum_{\lambda_i = \lambda_{min}} d_i v_i$ for arbitrary d_i (with the restriction that $\nu > 0$ implies $\|\delta'\| = \sqrt{h^2 - \|\bar{\delta}\|^2}$).
In this case (i.e. $\nu = -\lambda_{min}$) we have $\delta' \perp \bar{\delta}$ and $\delta' \perp g$, and $q(\delta) = q(\bar{\delta}) + \frac{\lambda_{min}}{2}(h^2 - \|\bar{\delta}\|^2)$.

In fact, we will do the following:

(a) In the `NORMAL CASE`: $\delta = (G + \nu\mathbb{1})^{-1}(-g)$ is the unique solution.

(b) In the `CASE NU == 0`: if $\lambda_{min} > 0$ then $\delta = G^{-1}(-g)$ is the solution, otherwise ($\lambda_{min} = 0$) we choose $\delta = \bar{\delta}$.

(c) In the `HARDCASE` we have to find an eigenvector $\beta \in \mathbb{R}^n$ belonging to the eigenvalue λ_{min} (i.e. $G\beta = \lambda_{min}\beta$) and a $\tau \in \mathbb{R}$ such that $\|\bar{\delta} + \tau\beta\| = h$, and then $\delta = \bar{\delta} + \tau\beta$ is a solution of (1).

2 Some Techniques of the Algorithm

2.1 Efficient Use of the Newton Method

We will assume in this subsection, that $g \neq 0$ and that ν^* as in subsection 1.4 exists (i.e. $\nu^* \in (\max\{0, -\lambda_{min}\}, \infty)$ and $\|(G + \nu^* \mathbb{1})^{-1}(-g)\| = h$).

We define

$$\eta : (\max\{0, -\lambda_{min}\}, \infty) \to \mathbb{R}, \quad \nu \mapsto \eta(\nu) := \|(G + \nu\mathbb{1})^{-1}(-g)\| \tag{7}$$

This function η is positive, strictly decreasing and convex. We could find ν^* by applying the Newton method on $\eta(\nu) - h$; in fact, this may be not very efficient. It is better to apply the Newton method on $\frac{1}{\eta(\nu)} - \frac{1}{h}$ (then we find for one Newton step $\nu_{next} = \nu - \frac{\eta(\nu)-h}{\eta'(\nu)} \cdot \frac{\eta(\nu)}{h}$; we can say, this method is better than the Newton method applied on $\eta(\nu) - h$ by the factor $\frac{\eta(\nu)}{h}$ in each step). Since the function $\nu \mapsto \frac{1}{\eta(\nu)}$, defined on the same range as η, is positive, strictly increasing und concave, we have for every ν a Newton step with $\nu_{next} \leq \nu^*$ (which may be used for lower bounds, see implementation appendix A, lines 492 - 497).

It is not a big problem to compute $\eta'(\nu) \equiv \frac{d\eta}{d\nu}$: Since we use Cholesky decompositions $LL^T = G + \nu\mathbb{1}$ to compute $\eta(\nu)$ and $\delta = (G + \nu\mathbb{1})^{-1}(-g)$ is already known, we just have to compute $L^{-1}\delta$, then we have $\eta'(\nu) = \frac{-(L^{-1}\delta)^T(L^{-1}\delta)}{\|\delta\|}$.

2.2 Lower and Upper Bounds for the Parameter ν

Another important technique is the use of lower and upper bounds for the ν we want to find. This idea will be used to control convergence.

At beginning, we set $\nu_{low} = 0$ and $\nu_{high} = \max\{0, \frac{\|g\|}{h} - \lambda^-\}$, where $\lambda^- \leq \lambda_{min}$ can be found by using Gershgorin discs. Execution of the implemented algorithm will produce a sequence of values for ν which allows us

(a) to decide, whether $\nu = 0$ is what we are looking for,

(b) to reduce the intervall $[\nu_{low}, \nu_{high}]$ to a very small intervall containing the desired ν, or

(c) to find a ν with $\eta(\nu) \approx h$ (whenever such a ν exists in the intervall $(\max\{0, -\lambda_{min}\}, \infty)$).

If we stop because of (a) (`CASE NU == 0`) or (c) (`NORMAL CASE`), we do not care about the length of the intervall $[\nu_{low}, \nu_{high}]$; otherwise the length of the intervall $[\nu_{low}, \nu_{high}]$ is the criterion to stop the algorithm (then we would be in the `HARDCASE`).

2.3 The Squareroot-Approximation

A third technique tries to improve efficiency in the `HARDCASE`, where the function η gives no information about the ν we want to find, so we would have to use a bisection method to find $\nu = -\lambda_{min}$. As we compute Cholesky decompositions of $G + \nu\mathbb{1}$ anyway, we may compute

$$mindiag(\nu) := \text{the smallest value of an element on the diagonal of L} \tag{8}$$

where $LL^T = G + \nu\mathbb{1}$. Experience shows that $\nu \mapsto mindiag(\nu)$ can be approximated by $\nu \mapsto c \cdot \sqrt{\nu + \lambda_{min}}$ for some (unknown) constant $c > 0$, so we have for two different $\nu_i > -\lambda_{min}$ $f_i := mindiag(\nu_i) \approx c \cdot \sqrt{\nu_i + \lambda_{min}}$, $i \in \{1, 2\}$, which leads to

$$-\lambda_{min} \approx \frac{\nu_1 - \left(\frac{f_1}{f_2}\right)^2 \nu_2}{1 - \left(\frac{f_1}{f_2}\right)^2}$$

(in the implementation appendix A, especially see lines 514 - 516).

2.4 The Trick of the Tiny Steps

Another trick which can be helpful in the `HARDCASE` is the following: Experience shows that after a few iterations of the parameter ν one bound of the intervall $[\nu_{low}, \nu_{high}]$ is a quite good approximation of $-\lambda_{min}$, where the other bound is of slower convergence. Then a very small step up or down (in the implementation called a `tiny_step`) may suffice to reduce $[\nu_{low}, \nu_{high}]$ in a next step a lot and, if we are lucky, one of the stop criteria may terminate the algorithm (in the implementation appendix A, especially see lines 518 - 530).

3 Implementation of the Algorithm

The most important ideas of the algorithm were presented in section 2; in this section we discuss further aspects of the implementation.

For a source code (in C) of the whole algorithm, see appendix A (procedure `lm_opt`). Most of this code should speak for itself, so further discussion will be reduced to few points.

3.1 Bisection and Termination

In order to limit the number of executions of the `while`-loop (lines 463 - 536 in the listing), after a certain number of executions of this loop (counted by variable `pparaiter`, especially see line 467) all special methods (e.g. Newton, Squareroot, Tiny Steps) are put aside and only an ordinary bisection method is applied for the rest. Since all loops of the algorithm are executed only finitely often, the execution will terminate; e.g. the `while`-loop (lines 463 - 536) will be terminated, since (at least when bisection is applied) the following holds:

(a) if $\nu_{low} = 0$ for all steps: criteria $\nu_{high} >$ `NU_EPS` will cause termination after a limited number of steps

(b) otherwise: $\nu_{high} - \nu_{low} > \nu_{high}$`NU_EPS` will be violated after a finite number of steps

3.2 Strategy in the `while`-loop

In the first execution of the `while`-loop (lines 463 - 536 in the listing), we have $\nu \geq -\lambda_{min}$ (see line 456), so (if not $\nu = -\lambda_{min}$, a rare and unproblematic case) we are sure to find a Cholesky decomposition without more search.

As long as we do not know whether ν^* exists or not (see subsection 1.4), we concentrate on finding $\max\{0, -\lambda_{min}\}$, i.e. in the second step we try $\nu = 0$ (see lines 455 and 512), in the following steps (but not longer as we find a $\nu \in (-\lambda_{min}, \nu^*)$) we use the squareroot-approximation method (see subsection 2.3). Even if ν^* exists and we do not know, the algorithm is very efficient since the Newton method of subsection 2.1 is applied on the lower bound ν_{low} anyway.

If once we have found a $\nu \in (-\lambda_{min}, \nu^*)$, i.e. we are sure that ν^* exists, we concentrate on finding ν^* using the Newton method (see subsection 2.1); a flag will be set (`nu_low_gt_milami = TRUE`, see lines 507 and 512).

3.3 Finding an Eigenvector β

In the `HARDCASE`, we have to find a $\beta \in \mathbb{R}^n$ with $G\beta = \lambda_{min}\beta$; in fact, our β will be an approximation of such an eigenvector. We compute a Cholesky decomposition of a nearly singular $G + \nu\mathbb{1} = LL^T$ (i.e. $\nu \approx -\lambda_{min}$), so we hope that $mindiag(\nu) \approx 0$ (see (8)). Let be min_index the smallest i with $L_{ii} = mindiag(\nu)$. Then we set $\beta_i = 0$ for $i > min\_index$, $\beta_{min\_index} = 1$, and β_i with $i < min\_index$ is determined by $L^T\beta = \gamma$, where $\gamma_i = 0$ for $i \neq min\_index$ and $\gamma_{min\_index} = mindiag(\nu)$ (in the implementation appendix A, lines 548 - 560).

3.4 The Sign of τ in $\bar{\delta} + \tau\beta$

In the `HARDCASE`, we have to find τ with $\|\bar{\delta} + \tau\beta\| = h$. There are two solutions, which should be of the same quality since in theory we have $\beta \perp \bar{\delta}$. But in practice our β is not more than an approximation and one solution of τ is better than the other, which can be seen by the sign of $\beta^T\bar{\delta}$ (in the implementation appendix A, lines 561 - 571).

4 Results

The following tests give an impression of the behaviour of the algorithm. It has to be noted that the results depend on the test problems and on the machines.

4.1 Generating Problem Sets

For problem (1), one problem set (consisting of 32 problems) was generated like the following:

(a) The matrix G and the vector g are filled with random numbers (uniformly distributed in $[0, 1]$), then $G_{sing} = G - \lambda_{min}\mathbb{1}$ is computed (i.e. G_{sing} is (nearly) singular). The test problems are given by G_μ and g and h, where $G_\mu = G_{sing} + \mu\mathbb{1}$ and $h = \|(G_\mu + \nu\mathbb{1})^{-1}(-g)\|$ for $\mu, \nu \in \{0, 0.00001, 0.00101, 0.10101, 10.10101\}$ (every possible pair (μ, ν) except $(0, 0)$) (for `NORMAL CASE` tests). This will give 24 test problems for (a).

(b) The same as in (a) (reuse of all data) for the special case of $\nu = 0$, but $h = 2 \cdot \|(G_\mu + \nu\mathbb{1})^{-1}\delta\|$ (for `CASE NU == 0` tests). This will give another 4 test problems.

(c) The G and g of (a) are reused as G and $\bar{\delta}$ and a vector β is computed from G_{sing} as described in subsection 3.3. Then the test problems are given by G_ν and $g = -(G_\nu + \nu\mathbb{1})(\bar{\delta} + \beta)$ and $h = \|\bar{\delta} + \beta\|$, where $G_\nu = G_{sing} - \nu\mathbb{1}$ for $\nu \in \{0.00001, 0.00101, 0.10101, 10.10101\}$ (for `HARDCASE` tests). So we have 4 test problems in (c).

.

For problem (2), the test problems were generated in the same way as for problem (1), but with the following changes:

(a) The set $\{0, 0.00001, \ldots, 10.10101\}$ was replaced for μ (not for ν) by $\{0.01, 1.00001\}$. This will give 10 test problems for (a).

(b) The `CASE NU == 0` will not appear, so it was not tested.

(c) Here nothing was changed, so we have 4 test problems in (c).

. By this, one problem set for problem (2) consists of 14 problems. The test set was reduced because of the longer duration of the execution.

4.2 Test Series and Results

Both problems ((1) and (2)) were tested with series of problem sets (see subsection 4.1) as the following:

Dimension	1, 2, 3, 4, 8, 16, and 32	100 and 200	300	400 and 500
Number of sets	1000	100	10	3
Number of problems (1)	32000	3200	320	96
Number of problems (2)	14000	1400	140	42

The machine was a HP 9000 735/125, the constants had the values

`DBL_MAX`	$1.7976931348623157 \cdot 10^{+308}$	Maximum `double`
`DBL_EPSILON`	$2.2204460492503131 \cdot 10^{-16}$	Smallest `double` ε that $1.0 + \varepsilon \neq 1.0$
`NU_EPS`	$1.1102230246251565 \cdot 10^{-15}$	see implementation appendix A
`H_EPS`	$2.2204460492503131 \cdot 10^{-15}$	see implementation appendix A

The results of the test are given in the two tables below.

The first line shows the average time used for solving one test problem (not a whole test set of 32 (or 14) tests). "Time" means not CPU time; it is the time of the whole test computation, including the time for generating the problems. The second line gives the ratio t/n^2 (where t is the time of line 1 and n the dimension of the problem) and shows, that the complexity of the problem in practice increases (more or less) quadratically in the dimension. The time measurements are to be taken with care since many machine effects may be included (like swapping data for problems in high dimensions).

The third line gives the average number of Cholesky decompositions made solving one test problem, where the average was computed over all `NORMAL CASE` test problems (i.e. for all tests that happened to be treated as `NORMAL CASE` by procedure `lm_opt`; this has not to be the same as all tests generated as in subsection 4.1 (a)). The number of Cholesky decompositions is a good measure for the complexity "modulo dimension".

The fourth line is the same as the third but for `HARDCASE` instead of `NORMAL CASE`. For `CASE NU == 0` the number of Cholesky decompositions is constantly 2 (not in the table).

The last line gives the percentage of tests that happened to be `HARDCASE`. In fact not every test generated as `NORMAL CASE` (see subsection 4.1 (a)) is recognized as such by procedure `lm_opt`; it could be a `HARDCASE` too. For low dimensions (observed for $n = 1, 2, 3, 4$) a test generated as `HARDCASE` (subsection 4.1 (c)) could be solved as `NORMAL CASE`. Only the `CASE NU == 0` problems

are always in this same case when solved.

Table of results for problem (1):

Dimension n	1	2	3	4	8	16	32
milliseconds	0.2	0.3	0.3	0.3	0.8	3.0	14.3
milliseconds/n^2	0.1875	0.0781	0.0341	0.0195	0.0122	0.0117	0.0140
Chol.dec. `NORMAL CASE`	1.21	4.09	4.39	4.50	4.49	4.59	4.58
Chol.dec. `HARDCASE`	-	14.25	15.54	15.91	17.77	17.63	17.20
% `HARDCASE`	0	11.0	17.4	18.8	17.8	18.7	20.0

Dimension n	100	200	300	400	500
milliseconds	319.7	1138.1	7503.1	15270.8	34479.2
milliseconds/n^2	0.0320	0.0285	0.0834	0.0954	0.1379
Chol.dec. `NORMAL CASE`	4.93	5.29	5.29	5.06	5.31
Chol.dec. `HARDCASE`	18.29	17.31	18.02	21.35	18.95
% `HARDCASE`	19.8	23.2	20.6	20.8	19.8

Table of results for problem (2):

Dimension n	1	2	3	4	8	16	32
milliseconds	0.1	0.1	0.2	0.5	1.0	5.9	35.3
milliseconds/n^2	0.0714	0.0357	0.0238	0.0313	0.0156	0.0232	0.0345
Chol.dec. `NORMAL CASE`	3.00	5.50	5.48	5.16	5.81	6.85	5.09
Chol.dec. `HARDCASE`	-	23.12	24.52	25.13	26.40	27.72	29.04
% `HARDCASE`	0	29.0	48.3	53.4	55.9	78.9	85.5

Dimension n	100	200	300	400	500
milliseconds	888.6	9610.0	38378.6	86904.8	197404.8
milliseconds/n^2	0.0889	0.2403	0.4264	0.5432	0.7896
Chol.dec. `NORMAL CASE`	6.35	(17.00)	(10.00)	(11.00)	-
Chol.dec. `HARDCASE`	28.04	28.61	28.44	26.56	28.26
% `HARDCASE`	85.6	99.9	99.3	97.6	100.0

The maximum number of Cholesky decompositions for one test problem ever observed was 102.

The precision of the results is as follows:

(a) For `NORMAL CASE` problems, the solution δ is unique. The error of the computed solution δ_{comp} can be measured as relative error $\frac{\|\delta - \delta_{comp}\|}{\|\delta\|}$. The maximum value for this error ever observed was lower than $2.32 \cdot 10^{-13}$.

(b) For `CASE NU == 0` problems, the procedure always found the correct solution; this is caused by the numerical identity of the way of generating and solving the `CASE NU == 0` problems.

(c) For `HARDCASE` problems, we use an approximation for an eigenvector β, so the accuracy of the solutions is lower than for `NORMAL CASE` problems. Because of the multitude of the solutions, we can not take the same measure for the error as in (a), but we take the error $\frac{|q(\delta) - q(\delta_{comp})|}{|q(\delta_{comp})|}$. The maximum value for this error ever observed was lower than $1.28 \cdot 10^{-9}$.

References

[1] R. Fletcher:
Practical Methods of Optimization, Second Edition,
John Wiley & Sons Ltd. 1987, p. 100 - 107
ISBN 0 471 91547 5

[2] David M. Gay:
Computing Optimal Locally Constrained Steps,
SIAM J. Sci. Stat. Comput., Vol. 2, No. 2, June 1981, p. 186 - 197

A Source Code of the C Program

```
/* Eidgenoessische Technische Hochschule Zuerich

   Institut fuer Operations Research
   Clausiusstrasse 45
   CH-8092 Zuerich

   lmlib: Library for Levenberg-Marquardt algorithm

   Lukas Finschi, 20.11.1995 - 16.4.1996

   Procedures in lmlib.c: - eval_q
                          - lowerbound_lambdamin
                          - upperbound_lambdamax
                          - dcholdc
                          - dcholsl
                          - dchlhsl
                          - lm_opt
                          - lm_opt_peripheral

   Remarks:

      1. This library solves numerically a special quadratic optimization
         problem using the method of the Levenberg-Marquardt algorithm with
         the technique of More (see ref. 3, 4), implemented with special
         techniques for higher efficiency (see ref. 5).

      2. The data structure for matrices and vectors is the structure used in
         "Numerical Recipes in C" (see ref. 1, 2):
         - For vectors and matrices, you are free to use as range of the
           indices 1..dim instead of 0..(dim-1). Notation: e.g.
           matrix[1..dim][1..dim] is a matrix of dimension dim x dim.
         - A matrix is represented as an array of an array (referencing twice)
           as follows: let be "double **matrix;" our declaration, then
                       matrix       is the pointer of the matrix,
                       matrix[i]    is the pointer of the i-th row,
                       matrix[i][j] is the (double-)element with indices i, j.
         - A symmetric matrix A is used only in the upper-right part (incl. the
           diagonal), i.e. only elements A[i][j] with i<=j are used.
           If a Cholesky-decomposition is done for such a matrix A, i.e.
           A = L * L^T, then the generated matrix L will be stored in the
           lower-left part of A (not incl. the diagonal) and in a seperate
           vector for the diagonal elements.
           The procedures solving linear equations like Ax=b or Lx=b will
           use this representation of L.

      3. For allocating matrices and vectors and computing Cholesky-
         decompositions and solutions of linear equation systems some
         procedures of "Numerical Recipes in C" are used (partially modified).

      4. There also exists a program that allows to use part of this library in
         Mathematica (via MathLink).

      5. Furthermore there exists a program that allows to use part of this
         library in S-Plus.

      6. See also comments to each procedure below.

   References:
```

```
60         1. W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery:
61            Numerical Recipes in C, The Art of Scientific Computing,
62            Second Edition, Cambridge University Press 1988, 1992, 1994
63            ISBN 0 521 43108 5
64
65         2. W. T. Vetterling, S. A. Teukolsky, W. H. Press, B. P. Flannery:
66            Numerical Recipes, Example Book (C),
67            Second Edition, Cambridge University Press 1988, 1992
68            ISBN 0 521 43720 2
69
70         3. R. Fletcher: Practical Methods of Optimization, Second Edition,
71            John Wiley & Sons Ltd. 1987, p. 100 - 107
72            ISBN 0 471 91547 5
73
74         4. David M. Gay: Computing Optimal Locally Constrained Steps,
75            SIAM J. Sci. Stat. Comput., Vol. 2, No. 2, June 1981, p. 186 - 197
76
77         5. Lukas Finschi: An Implementation of the Levenberg-Marquardt Algorithm,
78            Institut fuer Operations Research, Clausiusstrasse 45, CH-8092 Zuerich
79            1996
80
81      Compiling:
82
83         This library has to be compiled/linked with "nrutil.c" from the
84         "Numerical Recipes in C" package.
85   */
86
87   /* ================================================================== */
88   /* Includings: Header-Files of the used C programming package:
89                       - math.h:   sqrt         (used in dcholdc, lm_opt)
90                                   fabs         (used in lowerbound_lambdamin,
91                                                      upperbound_lambdamax,
92                                                      lm_opt)
93                       - float.h:  DBL_EPSILON  (used for NU_EPS, H_EPS)
94                                   DBL_MAX      (used in lm_opt)
95                   Header-Files of the "Numerical Recipes in C" package:
96                       - nrutil.h: dmatrix      (used in lm_opt)
97                                   dvector      (used in lm_opt)
98                                   free_dmatrix (used in lm_opt)
99                                   free_dvector (used in lm_opt)
100  */
101
102  #include <math.h>
103  #include <float.h>
104
105  #ifndef NR_END
106  #include "nrutil.h"
107  #endif
108
109  /* ================================================================== */
110
111  #ifndef NU_EPS                                              /** NU_EPS  **/
112  #define NU_EPS  5*DBL_EPSILON  /* never >= 1 !!! */
113  #endif                                                      /** NU_EPS  **/
114
115  #ifndef H_EPS                                               /** H_EPS   **/
116  #define H_EPS   10*DBL_EPSILON /* never >= 1 !!! */
117  #endif                                                      /** H_EPS   **/
118
119  #ifndef TRUE                                                /** TRUE    **/
120  #define TRUE     1
```

```
#endif                                                          /** TRUE    **/

#ifndef FALSE                                                   /** FALSE   **/
#define FALSE    0
#endif                                                          /** FALSE   **/

#ifndef UNKNOWN                                                 /** UNKNOWN **/
#define UNKNOWN  0
#endif                                                          /** UNKNOWN **/

/* ========================================================================== */
/* Procedure: double eval_q(double **mg, double *vg, double *vd, int dim)

   Input:     mg:  matrix G[1..dim][1..dim] (symmetric;
                       only entries mg[i][j] with i<=j are used)
              vg:  vector g[1..dim]
              vd:  vector delta[1..dim]
              dim: dimension (has to be >= 1)

   Output:    value of q(delta) = 0.5*(delta^T)*G*delta + (g^T)*delta
              =0.5*sum_over_i_j(mg[i][j]*vd[i]*vd[j])+sum_over_i(vg[i]*vd[i])

   Comments:  The inputs mg, vg, vd (given by reference) are not changed
              by eval_q (read only).

   Mnemonics: eval_q: "evaluates the function q"
*/

double eval_q(double **mg, double *vg, double *vd, int dim)
{
   double value;
   int i, j;

   value = 0;
   for(i=1; i<=dim; i++) {
      value += (mg[i][i]*vd[i]*vd[i])/2;
      for(j=i+1; j<=dim; j++) value += mg[i][j]*vd[i]*vd[j];
      value += vg[i]*vd[i];
   }
   return value;
}

/* ========================================================================== */
/* Procedure: lowerbound_lambdamin(double **mg, int dim)

   Input:     mg:  matrix G[1..dim][1..dim] (symmetric;
                       only entries mg[i][j] with i<=j are used)
              dim: dimension (has to be >= 1)

   Output:    a lower bound for the minimal eigenvalue of G

   Comments:  Since G is symmetric, all eigenvalues lambda_i of G are real
              numbers. Let lambda_min be the smallest of them, i.e.
              lambda_min <= lambda_i for all i. The procedure computes (using
              Gershgorin discs) an number
              lowerbound_lambdamin(double **mg, int dim) <= lambda_min.

              The input mg is not changed by this procedure (read only).

   Mnemonics: lowerbound_lambdamin: "lower bound for lambda_min"
*/
```

```
182
183 double lowerbound_lambdamin(double **mg, int dim)
184 {
185    int i, j;
186    double min, temp;
187
188    for(i=1; i<=dim; i++) {
189       temp = mg[i][i];
190       for(j=1; j<i; j++) temp -= fabs(mg[j][i]);
191       for(j=i+1; j<=dim; j++) temp -= fabs(mg[i][j]);
192       if(i==1 || min>temp) min = temp;
193    }
194    return min;
195 }
196
197 /* ========================================================================= */
198 /* Procedure: upperbound_lambdamax(double **mg, int dim)
199
200    Input:     mg:  matrix G[1..dim][1..dim] (symmetric;
201                       only entries mg[i][j] with i<=j are used)
202               dim: dimension (has to be >= 1)
203
204    Output:    an upper bound for the maximal eigenvalue of G
205
206    Comments:  Since G is symmetric, all eigenvalues lambda_i of G are real
207               numbers. Let lambda_max be the largest of them, i.e.
208               lambda_i <= lambda_max for all i. The procedure computes (using
209               Gershgorin discs) an number
210               upperbound_lambdamax(double **mg, int dim) >= lambda_max.
211
212               The input mg is not changed by this procedure (read only).
213
214    Mnemonics: upperbound_lambdamax: "upper bound for lambda_max"
215 */
216
217 double upperbound_lambdamax(double **mg, int dim)
218 {
219    int i, j;
220    double max, temp;
221
222    for(i=1; i<=dim; i++) {
223       temp = mg[i][i];
224       for(j=1; j<i; j++) temp += fabs(mg[j][i]);
225       for(j=i+1; j<=dim; j++) temp += fabs(mg[i][j]);
226       if(i==1 || max<temp) max = temp;
227    }
228    return max;
229 }
230
231 /* ========================================================================= */
232 /* Procedure: int dcholdc(double **a, int dim, double *p)
233
234    Input:     a:   matrix A[1..dim][1..dim] (symmetric;
235                       only entries a[i][j] with i<=j are used)
236               dim: dimension (has to be >= 1)
237               p:   vector p[1..dim]
238
239    Output:    If the Cholesky-decomposition is possible: 0; else: 1.
240               In the first case the lower-left part of input a contains the
241               non-diagonal part of L and p contains the diagonal elements of L.
242
```

```
   Comments:  The procedure computes the Cholesky-decompositon of A, i.e.
              A = L * L^T (see also remark 2 at the beginning of this file).

              The decomposition is possible if and only if A is not positive
              definite, so this procedure is also a test for positive
              definiteness.

              This procedure is a "Numerical Recipes in C" procedure
              (modified for type "double").

              The upper-right part of input a (incl. the diagonal) is not
              changed by this procedure (read only).

   Mnemonics: dcholdc: "("double"-version of) Cholesky-decomposition"
*/

int dcholdc(double **a, int dim, double *p)
{
   int i, j, k;
   double sum;

   for(i=1; i<=dim; i++){
      for(j=i; j<=dim; j++){
         sum = a[i][j];
         for (k=i-1; k>=1; k--) sum -= a[i][k]*a[j][k];
         if(i==j) {
            if(sum<=0.0) return 1;
            p[i] = sqrt(sum);
         }
         else a[j][i] = sum/p[i];
      }
   }
   return 0;
}

/* ========================================================================== */
/* Procedure: void dcholsl(double **a, int dim, double p[], double b[],\
                     double x[])

   Input:     a:   matrix A[1..dim][1..dim] (as result of dcholdc;
                      only entries a[i][k] with k<i are used)
              dim: dimension (has to be >= 1)
              p:   vector p[1..dim] (diagonal of Cholesky-decomp.)
              b:   vector b[1..dim] (right hand side of equation)
              x:   vector x[1..dim] (where the solution is returned)

   Output:    overwrites the entries of x

   Comments:  The procedure computes the solution of Ax=b, where the Cholesky-
              decomposition A = L * L^T is given in inputs a and p (see
              comments of procedure dcholdc and remark 2 at the beginning of
              this file).

              This procedure is a "Numerical Recipes in C" procedure
              (modified for type "double").

              The inputs a, p, and b are not changed by this procedure (read
              only).

   Mnemonics: dcholsl: "("double"-version of) Cholesky-decomposition solver"
*/
```

```
304
305 void dcholsl(double **a, int dim, double p[], double b[], double x[])
306 {
307    int i, k;
308    double sum;
309
310    for (i=1; i<=dim; i++) { /* solve L * y = b (here: y stored in x) */
311       for (sum=b[i],k=i-1; k>=1; k--) sum -= a[i][k]*x[k];
312       x[i] = sum/p[i];
313    }
314    for (i=dim;i>=1;i--) {   /* solve L^T * x = y */
315       for (sum=x[i],k=i+1; k<=dim; k++) sum -= a[k][i]*x[k];
316       x[i] = sum/p[i];
317    }
318 }
319
320 /* ========================================================================= */
321 /* Procedure: void dchlhsl(double **a, int dim, double p[], double b[],\
322                      double x[])
323
324    Input:     a:   matrix A[1..dim][1..dim] (as result of dcholdc;
325                       only entries a[i][k] with k<i are used)
326               dim: dimension (has to be >= 1)
327               p:   vector p[1..dim] (diagonal of Cholesky-decomp.)
328               b:   vector b[1..dim] (right hand side of equation)
329               x:   vector x[1..dim] (where the solution is returned)
330
331    Output:    overwrites the entries of x
332
333    Comments:  The procedure computes the solution of Lx=b, where the Cholesky-
334               decomposition A = L * L^T is given in inputs a and p (see
335               comments of procedure dcholdc and remark 2 at the beginning of
336               this file).
337
338               This procedure is derived from dcholsl.
339
340               The inputs a, p, and b are not changed by this procedure (read
341               only).
342
343    Mnemonics: dchlhsl: "("double"-version of) Cholesky-decomp. half solver"
344                        ("half" means: half the way of solving L * L^T * x = b)
345 */
346
347 void dchlhsl(double **a, int dim, double p[], double b[], double x[])
348 {
349    int i, k;
350    double sum;
351
352    for (i=1; i<=dim; i++) {
353       for (sum=b[i],k=i-1; k>=1; k--) sum -= a[i][k]*x[k];
354       x[i]=sum/p[i];
355    }
356 }
357
358 /* ========================================================================= */
359 /* Procedure: void lm_opt(double **mg, double *vg, double *vd,\
360                      double *pval, double *pnu, double h, int dim,\
361                      int *pcase, int *pcholcount, int *pparaiter)
362
363    Input:     mg:         matrix G[1..dim][1..dim] (symmetric;
364                              only entries mg[i][j] with i<=j are used)
```

```
                vg:          vector g[1..dim]
                vd:          vector delta[1..dim] (return value: delta~)
                pval:        pointer to q(delta~) (return value)
                pnu:         pointer to nu~ (return value)
                h:           radius of the ball (has to be > 0)
                dim:         dimension (has to be >= 1)
                pcase:       (geometric) case of the optimization problem
                                (return value; one of: 1 (NORMAL CASE),
                                0 (CASE NU == 0), -1 (HARDCASE))
                pcholcount:  pointer to the count of Cholesky-decompositions
                                done during the execution of lm_opt (return value)
                pparaiter:   pointer to the count of iterations (i.e. executions
                                of the while-loop) done finding nu~ (return value)

    Output:     overwrites the inputs vd, pval, pnu, pcase, pcholcount, paraiter

    Comments:   This procedure solves numerically the following quadratic
                minimization problem:

                   min q(delta), such that norm(delta) <= h,

                where q(delta) = 0.5*(delta^T)*G*delta + (g^T)*delta, and
                norm(delta) = sqrt(delta^T * delta).

                lm_opt will find a solution delta~ and a nu~>=0, such that

                   (G + nu~) is positive semidefinite,
                   (G + nu~ * 1) * delta~ = -g,
                   if nu*>0, then norm(delta~)=h.

                For techniques used in this procedure, see ref. 5.

                This procedure does not modify the inputs mg and vg (read only).

    Mnemonics:  lm_opt: "Levenberg-Marquardt optimization"

                variables used in the procedure:
                   mg_nu:            matrix (G + nu * 1)
                   vcholdiag_mg_nu:  vector of the diagonal of L (result of the
                                        Cholesky-decomposition of mg_nu)
                   vb:               vector beta (used in two different contexts:
                                        i) calculation Newton-step to find nu~
                                       ii) computing vd + tau*vb (HARDCASE) )
                   nu:               the Levenberg-Marquardt parameter
                   nu_last, nu_next: last nu, next nu
                   nu_low, nu_high:  lower and upper bound of nu
                   norm_vd:          norm of the vector delta (= sqrt(vd^T * vd))
                   norm2_vb:         norm^2 of the vector beta (= vb^T * vb)
                   innprod_vd_vb:    inner prod. of delta and beta (= vd^T * vb)
                   tau:              used computing vd + tau*vb (HARDCASE)
                   mindiag, mindiag_last:
                                     value of the smallest diagonal element of L
                                        (result of the Cholesky-decomposition of
                                         mg_nu), last such value
                   tmp:              a temporary used variable (various contexts)
                   i, j:             integer used as indices (many contexts)
                   min_index:        index according to mindiag
                   vg_not_zero:      boolean (TRUE <=> vg is not 0)
                   nu_low_gt_milami: boolean (TRUE <=> nu_low > -lambda_min)
                   bisection:        boolean (TRUE  => use bisection method)
                   tiny_step_up:     boolean (for special convergence method)
```

```
426 */
427
428 void lm_opt(double **mg, double *vg, double *vd, double *pval, double *pnu,\
429             double h, int dim, int *pcase, int *pcholcount, int *pparaiter)
430 {
431    double **mg_nu, *vcholdiag_mg_nu, *vb;
432    double nu, nu_last, nu_next, nu_low, nu_high;
433    double norm_vd, norm2_vb, innprod_vd_vb, tau, mindiag, mindiag_last, tmp;
434    int i, j, min_index;
435    int vg_not_zero, nu_low_gt_milami, bisection, tiny_step_up;
436
437 /*                      Allocation and initialization of variables: */
438    mg_nu = dmatrix(1, dim, 1, dim);
439    vcholdiag_mg_nu = dvector(1, dim);
440    vb = dvector(1, dim);
441
442    for(i=1; i<=dim; i++) {
443       vd[i] = 0;
444       for(j=i+1; j<=dim; j++) mg_nu[i][j] = mg[i][j];
445    }
446    norm_vd = 0;
447    *pparaiter = 0;
448    *pcholcount = 0;
449    mindiag = 0;
450    nu_low_gt_milami = UNKNOWN;
451    bisection = FALSE;
452    tiny_step_up = TRUE;
453
454 /*                      Set initial values for nu_low, nu_next, nu_high: */
455    nu_low = 0;
456    nu_next = -lowerbound_lambdamin(mg, dim);
457    tmp = 0; for(i=1; i<=dim; i++) tmp += vg[i]*vg[i];
458    if(tmp==0) vg_not_zero = FALSE; else vg_not_zero = TRUE;
459    nu_high = sqrt(tmp)/h + nu_next;
460    if(nu_high<2*NU_EPS) nu_high = 2*NU_EPS;
461    if(nu_next<nu_low) nu_next = nu_high;
462
463 /* * * * * * * * * * * * * while: Find a good approximation for nu~: */
464    while(fabs(norm_vd-h) > H_EPS*h && nu_high > NU_EPS\
465          && (nu_high-nu_low>nu_high*NU_EPS || nu_low-nu>nu_high*NU_EPS)) {
466       *pparaiter += 1;
467       if(*pparaiter>30) bisection = TRUE;
468
469       nu_last = nu;
470       nu = nu_next;
471       mindiag_last = mindiag;
472
473 /*                      Find a Cholesky-decomposition of a mg_nu: */
474       for(i=1; i<=dim; i++) mg_nu[i][i] = mg[i][i]+nu;
475       while(dcholdc(mg_nu, dim, vcholdiag_mg_nu)!=0 \
476             && nu_high-nu_low>nu_high*NU_EPS) {
477          *pcholcount += 1;
478          nu_low = nu;
479          if(nu_high-nu_low<=nu_high*NU_EPS) nu = nu_high;
480          else if(bisection) nu = (nu_low+nu_high)/2;
481          else  nu = nu_low + 0.4*(nu_high-nu_low);
482          for(i=1; i<=dim; i++) mg_nu[i][i] = mg[i][i]+nu;
483       }
484       *pcholcount += 1;
485
486 /*                                Find new nu_low, nu_high: */
```

```
487       if(vg_not_zero) {
488          dcholsl(mg_nu, dim, vcholdiag_mg_nu, vg, vd);
489          for(i=1; i<=dim; i++) vd[i] = -vd[i];
490          norm_vd = 0; for(i=1; i<=dim; i++) norm_vd += vd[i]*vd[i];
491          norm_vd = sqrt(norm_vd);
492          if(!bisection) {
493             dchlhsl(mg_nu, dim, vcholdiag_mg_nu, vd, vb);
494             norm2_vb = 0; for(i=1; i<=dim; i++) norm2_vb += vb[i]*vb[i];
495             tmp = nu - (1-norm_vd/h)*(norm_vd*norm_vd/norm2_vb);
496             if(tmp>nu_low) nu_low = tmp;
497          }
498       }
499       if(!vg_not_zero || norm_vd<h) {
500          nu_high = nu;
501          mindiag = DBL_MAX;
502          for(i=1; i<=dim; i++)
503                if(vcholdiag_mg_nu[i]<mindiag) mindiag = vcholdiag_mg_nu[i];
504       }
505       else {
506          if(nu>nu_low) nu_low = nu;
507          nu_low_gt_milami = TRUE;
508       }
509
510 /*                          Find nu_next (in (nu_low, nu_high)): */
511       if(bisection) nu_next = (nu_low+nu_high)/2;
512       else if(nu_low_gt_milami || *pparaiter==1) nu_next = nu_low;
513       else {
514          tmp = mindiag/mindiag_last; tmp = tmp*tmp;
515          if(tmp==1) nu_next = (nu_low+nu_high)/2;
516          else nu_next = (nu-tmp*nu_last)/(1-tmp);
517          if(nu_next>nu_low && nu_next<nu_high) {
518             tmp = 0.01*(nu_high-nu_next);
519             if(tmp<nu_high*NU_EPS) {
520                if(tiny_step_up) {
521                   nu_next = nu_next + nu_high*NU_EPS;
522                   if(nu_next>nu_high) nu_next = nu_next - 2*nu_high*NU_EPS;
523                   else tiny_step_up = FALSE;
524                }
525                else {
526                   nu_next = nu_next - nu_high*NU_EPS;
527                   if(nu_next<nu_low) nu_next = nu_next + 2*nu_high*NU_EPS;
528                   else tiny_step_up = TRUE;
529                }
530             }
531             else nu_next = nu_next + tmp;
532          }
533          else nu_next = nu_low + 0.3*(nu_high-nu_low);
534       }
535    }
536 /* * * * * * * * * * * * end of while: Found a good approximation for nu~. */
537
538 /*              Find with nu~ (and vd(nu~)) a solution vd~: */
539    if(fabs(norm_vd-h)<=H_EPS*h){ /* NORMAL CASE; vd~ = vd(nu~) */
540       *pcase = 1;
541    }
542    else if(nu==0) {              /* CASE NU == 0; vd~ = vd(nu~) */
543       *pcase = 0;
544    }
545    else {                        /* HARDCASE; vd~ = vd(nu~) + tau*vb */
546       *pcase = -1;
547
```

```
548       mindiag = DBL_MAX;
549       for(i=1; i<=dim; i++)
550             if(vcholdiag_mg_nu[i]<mindiag) {
551                mindiag = vcholdiag_mg_nu[i];
552                min_index = i;
553             }
554       for(i=dim; i>min_index; i--) vb[i] = 0;
555       vb[min_index] = 1;
556       for(i=min_index-1; i>=1; i--) {
557          tmp = -mg_nu[min_index][i];
558          for(j=i+1; j<min_index; j++) tmp -= mg_nu[j][i]*vb[j];
559          vb[i] = tmp/vcholdiag_mg_nu[i];
560       }
561       norm2_vb = 0;
562       for(i=1; i<=dim; i++) norm2_vb += vb[i]*vb[i];
563
564       innprod_vd_vb = 0;
565       for(i=1; i<=dim; i++) innprod_vd_vb += vd[i]*vb[i];
566       tmp = innprod_vd_vb*innprod_vd_vb + norm2_vb*(h*h-norm_vd*norm_vd);
567       if(innprod_vd_vb>0)
568          tau = (-innprod_vd_vb + sqrt(tmp) ) /norm2_vb;
569       else
570          tau = (-innprod_vd_vb - sqrt(tmp) ) /norm2_vb;
571       for(i=1; i<=dim; i++) vd[i] = vd[i]+tau*vb[i];
572    }
573    *pval = eval_q(mg, vg, vd, dim);
574    *pnu = nu;
575
576    free_dmatrix(mg_nu, 1, dim, 1, dim);
577    free_dvector(vcholdiag_mg_nu, 1, dim);
578    free_dvector(vb, 1, dim);
579
580    return;
581 }
582
583 /* ========================================================================== */
584 /* Procedure: void lm_opt_peripheral(double **mg, double *vg, double *vd,\
585                      double *pval, double *pnu, double h, int dim,\
586                      int *pcase, int *pcholcount, int *pparaiter)
587
588    Input:     mg:         matrix G[1..dim][1..dim] (symmetric;
589                              only entries mg[i][j] with i<=j are used)
590               vg:         vector g[1..dim]
591               vd:         vector delta[1..dim] (return value: delta~)
592               pval:       pointer to q(delta~) (return value)
593               pnu:        pointer to nu~ (return value)
594               h:          radius of the ball (has to be > 0)
595               dim:        dimension (has to be >= 1)
596               pcase:      (geometric) case of the optimization problem
597                              (return value; one of: 1 (NORMAL CASE),
598                              0 (CASE NU == 0), -1 (HARDCASE))
599               pcholcount: pointer to the count of Cholesky-decompositions
600                              done during the execution of lm_opt (return value)
601               pparaiter:  pointer to the count of iterations (i.e. executions
602                              of the while-loop) done finding nu~ (return value)
603
604    Output:    overwrites the inputs vd, pval, pnu, pcase, pcholcount, paraiter
605
606    Comments:  This procedure solves numerically the following quadratic
607               minimization problem:
608
```

```
609                  min q(delta), such that norm(delta) = h,
610
611               where q(delta) = 0.5*(delta^T)*G*delta + (g^T)*delta and
612               norm(delta) = sqrt(delta^T * delta).
613
614               lm_opt_peripheral will find a solution delta~ and a nu~,
615               such that
616
617                  (G + nu~) is positive semidefinite,
618                  (G + nu~ * 1) * delta~ = -g,
619
620               For techniques used in this procedure, see ref. 5.
621
622               This procedure does not modify the input vg (read only),
623               but modifies the diagonal elements of mg!
624
625    Mnemonics: lm_opt_peripheral : "Levenberg-Marquardt optimization of the
626                                    peripheral minimization problem"
627 */
628
629 void lm_opt_peripheral(double **mg, double *vg, double *vd,\
630           double *pval, double *pnu, double h, int dim,\
631           int *pcase, int *pcholcount, int *pparaiter)
632 {
633    double mu;
634    int i;
635
636    mu = upperbound_lambdamax(mg, dim);
637    for(i=1; i<=dim; i++) mg[i][i] -= mu;
638
639    lm_opt(mg, vg, vd, pval, pnu, h, dim, pcase, pcholcount, pparaiter);
640
641    *pval += 0.5*mu*h*h;
642    *pnu -= mu;
643
644    return;
645 }
```

B Extension to Mathematica

The Makefile

```
1    #
2    # Makefile for lm_mathlink, the Mathematica extension of lmlib.c
3    #
4    # Remarks:
5    #
6    #   1. First the Template File (here "lm_mathlink.tm") has to be passed to
7    #      the MathLink Preprocessor mprep. To do this, type "make mprep"; a file
8    #      called "lm_mathlink.tm.c" will be generated.
9    #
10   #   2. After preprocessing the Template File (see rem. 1), the output of mprep
11   #      and the C Source File (here "lm_mathlink.c") are compiled/linked
12   #      together. To do so, type "make math"; the output file is called
13   #      "lm_mathlink". (Steps 1 and 2 could be combined.)
14   #
15   #   3. Having generated a MathLink Object File (see rem. 2), you can start
16   #      Mathematica and work; here an example session:
17   #         In[1]:= lmlink = Install["lm_mathlink"]
18   #
19   #         Out[1]= LinkObject[lm_mathlink, 1, 1]
20   #
21   #         In[2]:= LinkPatterns[lmlink]
22   #
23   #         Out[2]= {LevMarquardt[(m_)?MatrixQ, (v_)?ListQ, r_Real],
24   #
25   #         >    LevMarquardtEq[(m_)?MatrixQ, (v_)?ListQ, r_Real]}
26   #
27   #         In[3]:= ?LevMarquardt
28   #         Procedure: LevMarquardt[ G, g, r].
29   #
30   #         Input:     matrix G[1..dim][1..dim] (symmetric;
31   #                       only entries mg[i][j] with i<=j are used)
32   #                    vector g[1..dim]
33   #                    radius r of the ball (has to be > 0)
34   #
35   #         Output:    vector delta
36   #
37   #         Comments:  This procedure solves numerically the
38   #                    following quadratic minimization problem:
39   #
40   #                        min q[delta], such that Norm[delta] <= r,
41   #
42   #                    where
43   #                        q[delta] = 0.5*(delta^T)*G*delta + (g^T)*delta,
44   #                    and
45   #                        Norm[delta] = Sqrt[delta^T * delta].
46   #
47   #         In[3]:= G = {{5.0, 4.0}, {4.0, 5.0}}
48   #
49   #         Out[3]= {{5., 4.}, {4., 5.}}
50   #
51   #         In[4]:= g = {2.0, 3.0}
52   #
53   #         Out[4]= {2., 3.}
54   #
55   #         In[5]:= LevMarquardt[G, g, 3.0]
56   #
57   #         Out[5]= {0.222222222222222, -0.777778}
```

```
#
#         In[6]:= LevMarquardtEq[G, g, 3.0]
#
#         Out[6]= {1.79603579204218, -2.40297}
#
#         In[7]:= Uninstall[lmlink]
#
#         Out[7]= lm_mathlink
#
#         In[8]:= Quit
#
#
#      (Remark: Uninstall (see In[7]) is not needed before Quit.)

mprep:
        /users/software/math/Bin/MathLink/mprep \
lm_mathlink.tm > lm_mathlink.tm.c

math:
        gcc -I/users1/students/finschi/LevMar \
-I/users/software/math/Source/Includes -O3 \
lm_mathlink.c lm_mathlink.tm.c ../lmlib.c ../nrutil.c \
/users/software/math/Bin/MathLink/libML.a \
-lm -o lm_mathlink
```

The Template File

```
1    :Begin:
2    :Function:           lev_mar
3    :Pattern:            LevMarquardt[m_?MatrixQ, v_?ListQ, r_Real]
4    :Arguments:          { m, v, r }
5    :ArgumentTypes:      { Manual }
6    :ReturnType:         Manual
7    :End:
8
9    :Evaluate:           LevMarquardt::usage =
10                           "Procedure: LevMarquardt[ G, g, r].\n\nInput:     matrix G[1..dim][1..dim] (sy
11             where\n               q[delta] = 0.5*(delta^T)*G*delta + (g^T)*delta,\n           and\n
12
13   :Begin:
14   :Function:           lev_mar_eq
15   :Pattern:            LevMarquardtEq[m_?MatrixQ, v_?ListQ, r_Real]
16   :Arguments:          { m, v, r }
17   :ArgumentTypes:      { Manual }
18   :ReturnType:         Manual
19   :End:
20
21   :Evaluate:           LevMarquardtEq::usage =
22                           "Procedure: LevMarquardtEq[ G, g, r].\n\nInput:     matrix G[1..dim][1..dim] (
23             where\n               q[delta] = 0.5*(delta^T)*G*delta + (g^T)*delta,\n           and\n
24
```

The C File

```
1    /* IFOR Zurich: Levenberg-Marquardt Algorithm
2       "lm_mathlink.c": MathLink Extension of lmlib.c
3       Lukas Finschi, 29.1.1996 - 2.4.1996
4
5       Procedures in lm_mathlink.c: - lev_mar
6                                    - lev_mar_eq
7                                    - main
8
9    */
10
11   #include "nrutil.h"
12   #include "lm.h"
13   #include "mathlink.h"
14
15   void lev_mar(void)
16   {
17      double **mg, *mg_math, *vg, *vg_math, *vd, val, nu, h;
18      long *dim_math, dim_vg_math, depth;
19      int i, j, dim, lmcase, chol, para;
20      char **heads;
21
22      MLGetDoubleArray(stdlink, &mg_math, &dim_math, &heads, &depth);
23
24      dim = (int) dim_math[0];
25      mg = dmatrix(1, dim, 1, dim);
26      for(i=1; i<=dim; i++) for(j=1; j<=dim; j++)
27                mg[i][j] = mg_math[(i-1)*dim + (j-1)];
28
29      MLGetRealList(stdlink, &vg_math, &dim_vg_math);
30                     /* here is no test for dim_math == dim_vg_math */
31      vg = dvector(1, dim);
32      for(i=1; i<=dim; i++) vg[i] = vg_math[i-1];
33
34      MLGetReal(stdlink, &h);
35
36      vd = dvector(1, dim);
37
38      lm_opt(mg, vg, vd, &val, &nu, h, dim, &lmcase, &chol, &para);
39
40      MLPutRealList(stdlink, &(vd[1]), (long) dim);
41
42      free_dmatrix(mg, 1, dim, 1, dim);
43      free_dvector(vg, 1, dim);
44      free_dvector(vd, 1, dim);
45
46      MLDisownDoubleArray(stdlink, mg_math, dim_math, heads, depth);
47      MLDisownRealList(stdlink, vg_math, dim_vg_math);
48
49      return;
50   }
51
52   void lev_mar_eq(void)
53   {
54      double **mg, *mg_math, *vg, *vg_math, *vd, val, nu, h;
55      long *dim_math, dim_vg_math, depth;
56      int i, j, dim, lmcase, chol, para;
57      char **heads;
58
59      MLGetDoubleArray(stdlink, &mg_math, &dim_math, &heads, &depth);
```

```
60
61      dim = (int) dim_math[0];
62      mg = dmatrix(1, dim, 1, dim);
63      for(i=1; i<=dim; i++) for(j=1; j<=dim; j++)
64                mg[i][j] = mg_math[(i-1)*dim + (j-1)];
65
66      MLGetRealList(stdlink, &vg_math, &dim_vg_math);
67                      /* here is no test for dim_math == dim_vg_math */
68      vg = dvector(1, dim);
69      for(i=1; i<=dim; i++) vg[i] = vg_math[i-1];
70
71      MLGetReal(stdlink, &h);
72
73      vd = dvector(1, dim);
74
75      lm_opt_peripheral(mg, vg, vd, &val, &nu, h, dim, &lmcase, &chol, &para);
76
77      MLPutRealList(stdlink, &(vd[1]), (long) dim);
78
79      free_dmatrix(mg, 1, dim, 1, dim);
80      free_dvector(vg, 1, dim);
81      free_dvector(vd, 1, dim);
82
83      MLDisownDoubleArray(stdlink, mg_math, dim_math, heads, depth);
84      MLDisownRealList(stdlink, vg_math, dim_vg_math);
85
86      return;
87  }
88
89  int main(int argc, char *argv[])
90  {
91      return MLMain(argc, argv);
92  }
```

C Extension to S-Plus

The Makefile

```
#
# Makefile for lm_spluslink, the S-Plus extension of lmlib.c
#
# Remarks:
#
#   1. First the C Source File (here "lm_spluslink.c") has to be compiled.
#      To do this, type "make"; a file called "lm_spluslink.o" will be
#      generated.
#      Another possibility is to type "Splus COMPILE lm_spluslink.c"
#
#   2. After compilation (see rem. 1) you can start S-Plus and work; here
#      an example session:
#         > dyn.load2("lm_spluslink.o")
#         > G_matrix(c(5, 4, 4, 5), 2, 2)
#         > g_c(2, 3)
#         > r_3
#         > n_2
#         > d_c(0,0)
#         > .C("lev_mar", as.double(G), as.double(g), as.double(r),
#         + as.integer(n), as.double(d))[[5]]
#         [1]  0.2222222 -0.7777778
#         > .C("lev_mar_eq", as.double(G), as.double(g), as.double(r),
#         + as.integer(n), as.double(d))[[5]]
#         [1]  1.796036 -2.402968
#         > q()
#
#      (Remark: Instead of "dyn.load2" you may use "dyn.load".)

NAME = lm_spluslink
CFILES = $(NAME).c
OFILES =
LIBS =
CFLAGS = -I/users1/students/finschi/LevMar -c -O3
LDFLAGS =
CC = gcc

$(NAME): $(OFILES)
        $(CC) $(CFLAGS) $(LDFLAGS) $(OFILES) $(CFILES) $(LIBS)

```

The C File

```
1    /* IFOR Zurich: Levenberg-Marquardt Algorithm
2       "lm_spluslink.c": S-Plus Extension of lmlib.c
3       Lukas Finschi, 31.1.1996 - 16.4.1996
4
5       Procedures in lm_spluslink.c: - lev_mar
6                                     - lev_mar_eq
7
8    */
9
10   #include "nrutil.c"
11   #include "lmlib.c"
12
13   void lev_mar(double *mg_splus, double *vg_splus, double *r_splus,\
14          long *dim_splus, double *vd_splus)
15   {
16      double **mg, *vg, *vd, val, nu, h;
17      int i, j, dim, lmcase, chol, para;
18
19      dim = (int) *dim_splus;
20      mg = dmatrix(1, dim, 1, dim);
21      for(i=1; i<=dim; i++) for(j=1; j<=dim; j++)
22                mg[i][j] = mg_splus[i-1 + (j-1)*dim];
23                      /* Splus <-> C: "transpose"! */
24
25      vg = dvector(1, dim);
26      for(i=1; i<=dim; i++) vg[i] = vg_splus[i-1];
27
28      vd = dvector(1, dim);
29
30      h = *r_splus;
31
32      lm_opt(mg, vg, vd, &val, &nu, h, dim, &lmcase, &chol, &para);
33      for(i=1; i<=dim; i++) vd_splus[i-1] = vd[i];
34
35      free_dmatrix(mg, 1, dim, 1, dim);
36      free_dvector(vg, 1, dim);
37      free_dvector(vd, 1, dim);
38
39      return;
40   }
41
42
43   void lev_mar_eq(double *mg_splus, double *vg_splus, double *r_splus,\
44          long *dim_splus, double *vd_splus)
45   {
46      double **mg, *vg, *vd, val, nu, h;
47      int i, j, dim, lmcase, chol, para;
48
49      dim = (int) *dim_splus;
50      mg = dmatrix(1, dim, 1, dim);
51      for(i=1; i<=dim; i++) for(j=1; j<=dim; j++)
52                mg[i][j] = mg_splus[i-1 + (j-1)*dim];
53                      /* Splus <-> C: "transpose"! */
54
55      vg = dvector(1, dim);
56      for(i=1; i<=dim; i++) vg[i] = vg_splus[i-1];
57
58      vd = dvector(1, dim);
59
```

```
60      h = *r_splus;
61
62      lm_opt_peripheral(mg, vg, vd, &val, &nu, h, dim, &lmcase, &chol, &para);
63      for(i=1; i<=dim; i++) vd_splus[i-1] = vd[i];
64
65      free_dmatrix(mg, 1, dim, 1, dim);
66      free_dvector(vg, 1, dim);
67      free_dvector(vd, 1, dim);
68
69      return;
70  }
```