stichting
mathematisch
centrum

MC

AFDELING NUMERIEKE WISKUNDE (DEPARTMENT OF NUMERICAL MATHEMATICS) NW 17/78 APRIL

J.C.P. BUS, B. VAN DOMSELAAR & J. KOK

# NONLINEAR LEAST SQUARES ESTIMATION

Second printing

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 65D10, 65H10

first printing 1975
second printing 1978

# Nonlinear least squares estimation

by

J.C.P. Bus, B. van Domselaar & J.Kok

## ABSTRACT

Two algorithms for minimizing a sum of squares are described and compared. The first one is the well-known Gauss-Newton algorithm. The second one is based on the algorithm given by Marquardt.

KEY WORDS & PHRASES: *nonlinear least squares, minimizing a sum of squares, overdetermined nonlinear systems, curve-fitting.*

CONTENTS

## 1. INTRODUCTION

In this report we describe and compare implementations of two algorithms for the calculation of a least squares solution of an overdetermined system of nonlinear equations. The first algorithm, given in section 3, is the well-known Gauss-Newton algorithm (see for instance HARTLEY [13]). The second, which is described in section 4, is based on the algorithm given by MARQUARDT [15].

Numerical results of these algorithms, together with those of some general minimization algorithms, are given in section 5, while conclusions based on numerical as well as theoretical considerations are stated in section 6.

Finally, these algorithms are described in the form of ALGOL 60 procedures in appendix.

## 2. STATEMENT OF THE PROBLEM

A wide variety of problems in numerical analysis can be solved by minimizing a sum of squares. In our opinion, the most important example is the problem of fitting the function

$$(2.1) \qquad g(t;x),$$

depending on a real variable t and a vector of parameters $x = (x_1,x_2,\dots,x_n)^T$, to m observations $(t_i,y_i)$, $i = 1,\dots,m$ $(m \geq n)$. Denote

$$Y = (y_1,\dots,y_m)^T$$

and the function $G\colon \mathbb{R}^n \to \mathbb{R}^m$ by

$$G(x) = (g(t_1;x),g(t_2;x),\dots,g(t_m;x))^T.$$

Then, the residual function $f\colon \mathbb{R}^n \to \mathbb{R}^m$, depending on x, may be defined by

(2.2) $$f(x) = G(x) - Y.$$

With this notation, the "curve fitting problem" can be formulated as:

(2.3) minimize $\|f(x)\|$ with respect to x, for some norm $\|.\|$ in $\mathbb{R}^m$.

Choosing the euclidean norm we obtain, by writing $f_i(x) = g(t_i;x) - y_i$, a so-called nonlinear least squares problem:

(2.4) *minimize the sum of squares* $F(x) = \sum_{i=1}^{m} (f_i(x))^2$, *with respect to* x,

or in vector notation:

(2.5) *minimize* $F(x) = f^T(x)f(x)$.

Hence, assuming that f is twice differentiable, we want to calculate $x \in \mathbb{R}^n$, such that

(2.6) $\nabla F(x) = 0$; $\nabla^2 F(x)$ is positive definite.

Denote the jacobian matrix of partial derivatives of the residual function by J(x). Since

(2.7) $$(J(x))_{ij} = \frac{\partial f_i(x)}{\partial x_j} = \frac{\partial g(t_i;x)}{\partial x_j},$$

we see that J(x) equals the jacobian matrix of the function G(x).

Substituting (2.5) in the first equation of (2.6) leads to

(2.8) $$J^T(x)f(x) = 0.$$

Hence, problem (2.5) can be replaced by the problem of finding x such that (2.8) is satisfied, provided that the second derivative of F(x) is positive definite at the solution.

In choosing a method for solving (2.8) we assume that analytical expressions for the elements of the jacobian matrix $J(x)$ are available. A well-known method for solving nonlinear equations is Newton's method. When applying to (2.8), this method is defined by

$$(2.9) \qquad x_{k+1} - x_k = - [H(x_k)]^{-1} J^T(x_k) f(x_k), \qquad k = 0,1,2,\ldots,$$

where

$$(2.10) \qquad H(x) = J^T(x)J(x) + \left[\frac{d}{dx} J^T(x)\right] f(x).$$

Hence, we should provide in each iteration the second derivative of the function $g(t;x)$, which is an $(m\times n\times n)$-tensor. However, realizing this will be very difficult or even impossible for most practical problems and a simplification of Newton's method is desirable.

## 3. THE GAUSS-NEWTON ALGORITHM

The method described in this section is essentially based on Newton's method for solving (2.8) as given by (2.9) and (2.10).

If we assume that $g(t;x)$ is a proper function for fitting the given observations, then the residual function is smooth and its norm is small. Therefore,

$$\left\| \frac{d}{dx} J^T(x) f(x) \right\|$$

will be small relative to $\|J(x)\|$. Hence it seems reasonable to approximate $H(x)$, given by (2.10), by the simpler expression

$$\bar{H}(x) = J^T(x)J(x).$$

Substituting $\bar{H}(x)$ for $H(x)$ in (2.9) we obtain

$$(3.1) \qquad J^T(x_k)J(x_k)\delta_k = -J^T(x_k)f(x_k), \qquad k = 0,1,2,\ldots,$$

where

$$(3.2)\qquad \delta_k = x_{k+1} - x_k.$$

The iterative method thus obtained is called the *Gauss-Newton method*. sidering (3.1) we see that it is, in fact, the so-called "normal equa belonging to the overdetermined system of linear equations

$$(3.3)\qquad J(x_k)\delta_k = -f(x_k).$$

As is well known (GOLUB [10]), one should not calculate $\delta_k$ by perform the matrix multiplication and solving the symmetric linear system (3. This might decrease the stability of the process, since the condition ber is squared. The right way to calculate $\delta_k$ is to obtain an orthogo decomposition of $J(x_k)$, in order to solve (3.3) directly. Our impleme tion of the Gauss-Newton algorithm uses Householder orthogonalisation (DEKKER [4], BUSINGER & GOLUB [3]). Thus, an orthogonal m-th order ma $Q_k$ and an m × n upper-triangular matrix $R_k$ are calculated such that

$$(3.4)\qquad J(x_k) = Q_kR_k.$$

Subsequently, $\delta_k$ is calculated by solving the n × n upper-triangular system, consisting of the first n equations of

$$(3.5)\qquad R_k\delta_k = -Q_k^Tf(x_k).$$

In many practical problems, it might occur that the algorithm de ed is unstable, since $J(x_k)$ is not necessarily of full rank n. Althou algorithm terminates as soon as $J(x_k)$ appears not to be of full rank tive to the machine precision (see DEKKER [4], p.65), we also use a s egy to control the step size. We choose

$$(3.6)\qquad x_{k+1} = x_k + \alpha_k\delta_k,$$

where $\alpha_k$ is chosen such that

$$(3.7) \qquad F(x_{k+1}) < F(x_k).$$

Our version of the Gauss-Newton algorithm, which will be called algorithm G in the sequel, is defined by three blocks G1, G2 and G3. After initialisation (G1), N + 1 iteration steps (G2) are performed, where N is defined by the stopping criteria (G3).

G1: *initialisation*;
let $x_0$ be a given approximation to the minimum of $F(x)$ and let $\varepsilon_{re}$, $\varepsilon_a$ and $\varepsilon_0$ be three given tolerance values;

G2: *iteration step*, $k = 0,1,\ldots,N$;
calculate an orthogonal m-th order matrix $Q_k$ and an $m \times n$ upper-triangular matrix $R_k$ such that (3.4) is satisfied; calculate the direction of search $d_k$ by solving the $n \times n$ upper-triangular linear system, consisting of the first n equations of

$$R_k d_k = -Q_k^T f(x_k);$$

if $F(x_k+d_k) \le F(x_k)$, then choose $\alpha_k = 1$, otherwise choose $\alpha_k = 2^{-r}$, where r is the smallest nonnegative integer, such that

$$F(x_k+2^{-r}d_k) < F(x_k)$$

and

$$F(x_k+2^{-r}d_k) < F(x_k+2^{(r+1)}d_k);$$

set $\delta_k = \alpha_k d_k$, $x_{k+1} = x_k + \delta_k$;

G3: *stopping criteria*;
the number of iteration steps equals N + 1, where N is the smallest nonnegative integer such that

$$F(x_N) \le \varepsilon_0^2 \qquad \text{or} \qquad \|\delta_k\| \le \|x_{k+1}\| \times \varepsilon_{re} + \varepsilon_a.$$

An implementation of algorithm G in ALGOL 60 is given in appendix.

The main disadvantage of this method is that it may break down for certain k, if the jacobian matrix $J(x_k)$ does not have full rank n, within the precision of calculation. Even when the algorithm does not break down, it may occur that $\|d_k\|$ becomes prohibitively large if the condition number of $J(x_k)$ is large. In that case, either the value of r, and therefore the number of function evaluations in the k-th step, will be large, or some local solution far from the initial guess may be found.

## 4. AN ALGORITHM BASED ON THE IDEAS OF LEVENBERG AND MARQUARDT

One way of reducing the difficulties of algorithm G is based on an idea of LEVENBERG [14], which is used by MARQUARDT [15] in his algorithm for solving nonlinear least squares problems. The idea is to calculate a step vector $\delta_k$, k = 0,1,2,..., according to

$$(4.1)\qquad (J^T(x_k)J(x_k)+\lambda_k M_k)\delta_k = -J^T(x_k)f(x_k),$$

where $\lambda_k$ is a nonnegative scalar and $M_k$ is some positive definite matrix. If we choose $\lambda_k = 0$, then $\delta_k$ equals the Gauss-Newton vector defined by (3.1). When we choose $M_k = I$, where I denotes the unit matrix, then, for any fixed k, if $\lambda_k$ tends to infinity then $\delta_k$ tends to the steepest descent direction

$$(4.2)\qquad -J^T(x_k)f(x_k),$$

and its norm tends to zero (MARQUARDT [15], theorems 2 and 3).

To obtain a reasonable value for $\lambda_k$, k = 0,1,2,..., we use the following strategy, due to GOLDSTEIN & PRICE [9] and also given by FLETCHER [6]. Define $s_k = s_k(\lambda)$ and $h_k = h_k(\lambda)$ by

$$(4.3)\qquad (J^T(x_k)J(x_k)+\lambda I)s_k = -J^T(x_k)f(x_k)$$

and

$$(4.4) \qquad h_k(\lambda) = \frac{F(x_k) - F(x_k+s_k)}{-s_k^T J^T(x_k) f(x_k)} .$$

Then, we choose the value of $\lambda_k$ such that

$$(4.5) \qquad h_k(\lambda_k) \geq \mu,$$

where $\mu$ is a given constant, $0 < \mu < 0.5$.

Using this strategy, it is possible that (4.3) has to be solved for more than one value of $\lambda$. An easy way of doing this is recommended by MARQUARDT [15]. He calculates the eigenvalues and eigenvectors of the matrix product $J^T(x_k)J(x_k)$. However, this is mathematically equivalent to the calculation of the singular values of $J(x_k)$ (GOLUB & REINSCH [12]). Since the latter is more stable we use it in our algorithm. So, we calculate a decomposition of $J(x_k)$

$$(4.6) \qquad J(x_k) = U_k \Sigma_k V_k^T,$$

where $U_k$ is an orthogonal m-th order matrix, $\Sigma_k$ is the $m \times n$ diagonal matrix of singular values $\sigma_1,\ldots,\sigma_n$, and $V_k$ is an n-th order orthogonal matrix. Substituting (4.6) in (4.3) leads to

$$(4.7) \qquad V_k(\Sigma_k^2+\lambda I)V_k^T s_k = -V_k \Sigma_k U_k^T f(x_k),$$

where $\Sigma_k^2 = \mathrm{diag}(\sigma_1^2,\ldots,\sigma_n^2)$. Hence, $s_k$ can be written as

$$(4.8) \qquad s_k(\lambda) = -V_k(\Sigma_k^2+\lambda I)^{-1}\Sigma_k U_k^T f(x_k),$$

provided $\lambda$ is chosen such that $\Sigma_k^2 + \lambda I$ is nonsingular, which is true for $\lambda > 0$. This shows that, once the singular value decomposition is performed, we only need a matrix-vector multiplication to obtain $s_k = s_k(\lambda)$ for various values of $\lambda$.

Using these ideas, we define our Marquardt-type algorithm, called algorithm M in the sequel, by the following three blocks $M_1$, $M_2$ and $M_3$. After initialisation ($M_1$), N + 1 iteration steps ($M_2$) are performed, where N is defined by the stopping criteria ($M_3$).

$M_1$: *initialisation*;

let $x_0$ be an approximation to the minimum of $F(x)$; let $\varepsilon_r$ and $\varepsilon_a$ be two given tolerance values and $\mu$, $\omega$, $\nu$ and $\xi$ given constants such that $0 < \mu << 0.5$, $\omega < 1$, $\nu > 1$ and $\xi > 0$;

$M_2$: *iteration step*, $k = 0,1,\dots,N$;

calculate the singular value decomposition given by (4.6); set $\rho_k = \xi\|\Sigma_k\|^2$; if $k = 0$, then set $\lambda_0^{(0)} = \lambda_{-1} = \rho_k$, otherwise (i.e. if $k > 0$) set

$$\lambda_k^{(0)} = \omega\lambda_{k-1}, \qquad \text{if } \lambda_{k-1} \leq \lambda_{k-2}$$

or

$$\lambda_k^{(0)} = \lambda_{k-1}, \qquad \text{if } \lambda_{k-1} > \lambda_{k-2};$$

if $h(\lambda_k^{(0)}) \geq \mu$, then set $\lambda_k = \lambda_k^{(0)}$, otherwise, set $\lambda_k = \nu^r\max(\lambda_k^{(0)},\rho_k)$, where $r$ is the smallest nonnegative integer satisfying

$$h(\nu^r\max(\lambda_k^{(0)},\rho_k)) \geq \mu;$$

set $\delta_k = s_k(\lambda_k)$ and $x_{k+1} = x_k + \delta_k$;

$M_3$: *stopping criteria*;

the number of iteration steps equals $N + 1$, where $N$ is the smallest nonnegative integer satisfying

$$F(x_{N-1}) - F(x_N) \leq \varepsilon_r F(x_N) + \varepsilon_a$$

or

$$F(x_N) \leq \varepsilon_a.$$

An implementation of algorithm M in ALGOL 60 is given in appendix.

It is clear that, with the strategy for choosing $\lambda_k$ as described in $M_2$, the value of $\lambda_k$ has a tendency of decreasing, in particular if the function $F(x)$ is convex, or if the jacobian matrix $J(x)$ has a relatively small condition number. In practice this means that algorithm M behaves almost like algorithm G in those cases where algorithm G behaves fine. Therefore, we may expect algorithm M to be nearly as efficient as algorithm G for well-conditioned problems. On the other hand, the strategy in algorithm M is such that it does not suffer from breaking down when the jacobian matrix $J(x_k)$ is singular, as algorithm G does. Therefore, algorithm M is expected to be very useful for practical problems, since in most cases little is known about the condition of the problem to be solved.

## 5. NUMERICAL COMPARISONS

In this section we compare four algorithms:

- Algorithm G, defined in section 3.
- Algorithm M, defined in section 4.
- A variable metric algorithm for general minimization. This algorithm, which uses a rank-one updating formula is given in BUS [2] and is called algorithm R in the sequel. For an ALGOL 60 implementation see BUS ([16], section 5.1.2).
- A rank-two variable metric algorithm (BUS [2]), which is essentially the same as the one given by FLETCHER [6]. See also BUS([16], section 5.1.2). This algorithm is called algorithm F.

We divide our test problems into two sets.

The first set consists of rather artificial problems, where the number of variables equals the number of observations. In fact, these functions are adopted from literature, where they were used for testing general minimization algorithms and algorithms for solving systems of nonlinear equations.

The second class consists of some real curve fitting problems.

*Class I: Some artificial test problems*

Rather than giving the function $g(t;x)$ and the observations $(t_i;y_i)$, $i = 1,\dots,m$ (see section 2), we define these problems by giving $f(x)$ (cf. (2.2))

$P_1$: Brown's function (see BROWN [1]).

$$f_i(x) = -(n+1) + 2x_i + \sum_{\substack{j=1 \\ j\neq i}}^{n} x_j, \qquad\qquad i = 1,\ldots,n-1,$$

$$f_n(x) = -1 + \prod_{j=1}^{n} x_j.$$

This function, in which all but the last equation are linear, has two known zeroes; the vector x where each component has the value 1 and the vector x where all components but the last one have the same value. For instance, for n = 5 the approximate solution is:

$$x = (-0.579,-0.579,-0.579,-0.579,8.90)^T.$$

This function has been tested for n = 5,10,15,20.
In all cases the starting guesses are

$$x_i = 0.5, \qquad\qquad i = 1,\ldots,n.$$

$P_2$: The double polynomial function (see FREUDENSTEIN & ROTH [8]).

$$f_1(x_1,x_2) = -13 + x_1 + ((-x_2+5)x_2-2)x_2,$$

$$f_2(x_1,x_2) = -29 + x_1 + ((x_2+1)x_2-14)x_2.$$

This function has a zero at $(5,4)^T$, but all procedures converged to a point that proved to be a local minimum of the euclidean norm of the residual function. The starting guess is $(15,-2)^T$.

$P_3$: A badly scaled problem (see POWELL [17]).

$$f_1(x_1,x_2) = 10000x_1x_2 - 1,$$

$$f_2(x_1,x_2) = e^{-x_1} + e^{-x_2} - 1.0001.$$

For this difficult problem it is preferable to scale the parameters $x_1$ and $x_2$, so that their magnitudes are comparable, but it is interesting to discover what happens when this advice is not followed. The function has a zero at

$$(1.098_{10}-5,9.106)^T$$

and the starting guess is $(0,1)^T$.

$P_4$: Powell's function (see POWELL [17]).

$$f_1(x_1,x_2) = x_1,$$

$$f_2(x_1,x_2) = \frac{10x_1}{x_1 + 0.1} + 2x_2^2.$$

This function has only one zero, viz. $(0,0)^T$, and this is the only stationary point of the euclidean norm of the residual function. The ill-conditionedness of the problem is illustrated by the eigenvalues of the matrix $J^T(x)J(x)$ on the line $x_2 = 0$:

$$\sigma_1 = 0, \qquad \sigma_2 = \frac{1}{(x_1+0.1)^4} + 1.$$

Hence, $J^T(x)J(x)$ is singular for $x_2 = 0$.
The starting guess is $(3,1)^T$.

*Class II: The curve fitting problems*

$P_5$: The fertilizer experiment (see HARTLEY [13]).

$$g(t;x) = x_1 + x_2 \exp(t\times x_3)$$

In this model $g(t;x)$ represents the yield of wheat corresponding to a rate of application of fertilizer t. The observations $(t_i,y_i)$, $i = 1,\dots,6$, are given in table 5.1. The parameter $x_1$ is the asymptotic yield for large rates of fertilizer application and $x_2$ is the exponential rate of response decrease.

The initial guess for the parameters is $(500,-140,-0.18)^T$.

Table 5.1

| $i$ | $t_i$ | $y_i$ |
|---|---|---|
| 1 | -5 | 127 |
| 2 | -3 | 151 |
| 3 | -1 | 379 |
| 4 | 1 | 421 |
| 5 | 3 | 460 |
| 6 | 5 | 426 |

$P_6$: The heart-infarct problem (see VAN DOMSELAAR [5]).
The problem involves fitting a reduced model of an enzyme effusion into the blood after a heart-infarct. The model has the following form:

$$g(t;x) = c\,\frac{x_4(x_1-a)}{a(x_4-x_1)}\,\frac{d_1(t)}{d_{lim}}\exp(-x_4t) + c\frac{d_2(t)}{d_{lim}}\exp(-at) + g_{lim},$$

where

$$g_{lim} = g(\infty,x) \text{ is the "normal effusion"},$$

$$d_1(t) = \int_0^t \frac{1}{\tau}\exp\left[-0.5\left(\frac{l_n(\tau) - x_3}{x_2}\right)^2 + x_4\tau\right] d\tau,$$

$$d_2(t) = \int_0^t \frac{1}{\tau}\exp\left[-0.5\left(\frac{l_n(\tau) - x_3}{x_2}\right)^2 + a\tau\right] d\tau$$

and

$$d_{lim} = \lim_{t\to\infty} d_2(t).$$

In this model, $x_1$ is a constant of demolition, $x_2$ a measure for the duration of excessive effusion and $x_3$ denotes the time at which the maximum effusion appears. There are 17 observations $(t_i,y_i)$. The starting guess is $(0.14,0.2,2.40,0.28)^T$.

$P_7$: Trigonometric functions (FLETCHER & POWELL [7]). This problem may be described in an easy way by giving G(x) (cf. section 2).

$$G(x) = As(x) + Bc(x),$$

where A and B are m × n matrices, whose elements are generated as random integers between -100 and +100, s(x) and c(x) are n-vectors, such that

$$s(x) = (\sin (x_1), \sin (x_2),\ldots, \sin (x_n))^T$$

and

$$c(x) = (\cos (x_1), \cos (x_2),\ldots, \cos (x_n))^T.$$

The observations are created as follows. The elements of a vector $x^*$ are generated as random numbers between $-\pi$ and $+\pi$. Then

$$Y = (I+bD)G(x^*),$$

where I is the m-th order unit matrix. D is some m-th order diagonal matrix whose elements are generated as random numbers in the interval [-0.5,0.5] and b is some scalar used for varying the magnitude of the disturbance in the data. We have chosen b = 0.01, 0.05, 0.1, 0.2, 0.5 and 1.
The starting guess is $x^* + 0.01\delta$, where the elements of the vector $\delta$ are random numbers between $-\frac{1}{2}\pi$ and $+\frac{1}{2}\pi$.
We tested the problem for (n,m) = (5,5), (5,10), (5,15), (10,10), (10,15) and (10,20).

$P_8$: The exponential fitting problem (GOLUB & PEREYRA [11]).

$$g(x,t) = x_1 + x_2 \exp (-x_3t) + x_4 \exp (-x_5t).$$

The number of observations is 33.
The starting guess is $(0.5,2.5,0.01,-1,0.02)^T$.

The testing has been performed on a CYBER 73 computer with a machine precision of $10^{-14}$ approximately.

In order to be able to compare the four algorithms, which use different stopping criteria, we used as a measure the number of function evaluations needed to fall below a certain value for the norm of the function f(x) (see (2.4)), which is specified in the tables under $\|f(x)\|$. The choice of this value depends heavily on the accuracy of the data as far as the second class of problems is concerned.

If not mentioned otherwise, algorithm M has been tested with $\xi = 0.01$, $\mu = 0.01$, $\omega = 0.5$ and $\nu = 10$.

The results of the first class of problems are given in table 5.2.

Table 5.2

Class I

| function | n | $\|f(x)\|$ | number of function evaluations needed | | | |
|---|---|---|---|---|---|---|
| | | | G | M | F | R |
| P1 | 5 | $10^{-10}$ | 17 | 12 | 16 | 17 |
| | 10 | $10^{-10}$ | 52 | 16 | 20 | 20 |
| | 15 | $10^{-10}$ | - | 18 | 24 | 25 |
| | 20 | $10^{-10}$ | - | 19 | 29 | 30 |
| P2 | 2 | 6.99887 | - | 15 | 10 | 9 |
| P3 | 2 | $10^{-10}$ | 150[a)] | 54 | 150[a)] | 150[a)] |
| P4 | 2 | $10^{-10}$ | 16 | 25 | 150[a)] | 150[a)] |

a) The precision asked for has not been attained.

*Discussion of P1.*

For $n = 5$, algorithm G found the zero that is given as an example in the description of the function, while the other procedures found $(1,1,\dots,1)^T$. For $n = 10$, algorithm G had much difficulty in improving the norm of the residual function at the initial guess $x_0$. For $n = 15$ and $n = 20$ algorithm G failed. This behaviour is affirmed by theory (see the end of section 3), since the condition number of $J^T(x_0)J(x_0)$ is proportional to $2^n$.

*Discussion of P2.*

None of the algorithms found the zero minumum at $(5,4)^T$. Algorithm G did not converge because of a singular jacobian matrix. The other algorithms found a relative minimum of the norm of the residual function at $(11.412,-0.89681)^T$.

*Discussion of P3.*

Algorithm M performed well on this problem, the other methods did not. This is due to the condition number of $J^T(x)J(x)$ at the solution, which is about $10^8$. After 150 function evaluations algorithm G reached $1.0_{10}{-3}$ for the norm of the residual function, algorithm R reached $0.4_{10}{-4}$ and algorithm F reached $0.6_{10}{-4}$.

*Discussion of P4.*

The rapid convergence of algorithm G was remarkable, although it was not quadratic. After 150 function evaluations, algorithms R and F reached only the value $0.4_{10}{-5}$ for the norm of $f(x)$. Algorithm M, however, seems to be too careful. As is seen in table 5.3 the number of function evaluations is highly dependent on the value of $\xi$. The less the value of $\xi$, the more the behaviour of algorithm M is alike that of algorithm G. Only the case $\xi = 10^{-3}$ has an exceptional behaviour. The iteration path for this case happens to contain a nearly stationary point, while the paths for the other cases happen to avoid this point.

In the tables 5.4 up to 5.7 we give results of the problems from class II. For the problems P5, P6 and P8 we have tried to give the user an impression of the progress in the reduction of the norm of the residual function relative to the number of function evaluations needed.

Table 5.3

| $\xi$ | $\|f(x)\|$ | number of funct. eval. needed |
|---|---|---|
| $10^{-6}$ | $10^{-10}$ | 25 |
| $10^{-5}$ | $10^{-10}$ | 29 |
| $10^{-4}$ | $10^{-10}$ | 32 |
| $10^{-3}$ | $0.51_{10^{-3}}$ | 150[a)] |
| $10^{-2}$ | $0.59_{10^{-6}}$ | 50 |
| $10^{-1}$ | $0.59_{10^{-6}}$ | 49 |
| 1 | $0.21_{10^{-5}}$ | 150[a)] |

a) The precision asked for has not been attained.

*Discussion of P5.*

Since the starting guess is chosen close to the solution $(523.3,-156.9,-0.1996)^T$, none of the methods had any difficulty in finding this solution as is shown in table 5.4. Here, the more robust algorithm M is clearly less efficient.

Table 5.4

| $\|f(x)\|$ | number of function evaluations needed | | | |
|---|---|---|---|---|
| | G | M | F | R |
| 116.25 | 2 | 14 | 5 | 6 |
| 115.73 | 4 | 20 | 7 | 8 |
| 115.715 | 7 | 23 | 10 | 11 |

*Discussion of P6.*

The correlation matrix at the solution of this problem showed a strong mutual dependence of the parameters. Therefore, they are very hard to determine. As shown in table 5.5 algorithm G did not succeed in finding a solution of this problem. Algorithm R was terminated because its execution time exceeded the time limit which was imposed upon all procedures. Algorithm F converged very slowly to a minimum. For algorithm M we used $\xi = 0.1$.

Table 5.5

| $\|f(x)\|$ | number of function evaluations needed | | | |
|---|---|---|---|---|
| | G | M | F | R |
| 59.653 | - | 2 | 57 | 15 |
| 52.968 | - | 7 | 59 | 16 |
| 49.314 | - | 19 | 62 | 41 |
| 49.233 | - | 23 | 101 | - |

*Discussion of P7.*

All results are reproduced in table 5.6. The problems with large data disturbance ($b = 0.5$ or $b = 1$) usually caused difficulty to algorithm G. Moreover, for some problems with $n = m$, algorithm G did not find a solution at all. The curve fitting problems with large data disturbance resemble the problems with bad starting values of the parameters. The results of P7 show clearly the differences between the two methods discussed. In case of a bad initial approximation of the parameters, algorithm G may easily diverge, but with an initial guess close to the solution, algorithm G converges faster than the more robust algorithm M.

Algorithms R and F are obviously less efficient than algorithms G and M.

Table 5.6

| b | n | m | $\|f(x)\|$ | number of function evaluations needed | | | |
|---|---|---|---|---|---|---|---|
| | | | | G | M | F | R |
| 0.01 | 5 | 5 | $10^{-10}$ | 9 | 14 | 24 | 23 |
| | 5 | 10 | 0.5937 | 8 | 10 | 23 | 14 |
| | 5 | 15 | 1.4598 | 4 | 8 | 20 | 11 |
| | 10 | 10 | $10^{-10}$ | 15 | 18 | 56 | 42 |
| | 10 | 15 | 1.9722 | 4 | 9 | 21 | 19 |
| | 10 | 20 | 1.1431 | 4 | 8 | 20 | 19 |
| 0.05 | 5 | 5 | $10^{-10}$ | 9 | 15 | 26 | 25 |
| | 5 | 10 | 3.0229 | 4 | 9 | 21 | 14 |
| | 5 | 15 | 7.2875 | 4 | 8 | 19 | 11 |
| | 10 | 10 | $10^{-10}$ | 15 | 28 | 53 | 67 |
| | 10 | 15 | 9.8017 | 6 | 8 | 22 | 19 |
| | 10 | 20 | 5.7156 | 4 | 8 | 18 | 18 |
| 0.1 | 5 | 5 | 1.6790 | – | 18 | 28 | 28 |
| | 5 | 10 | 6.2082 | 6 | 9 | 22 | 12 |
| | 5 | 15 | 14.5454 | 4 | 8 | 21 | 11 |
| | 10 | 10 | $10^{-10}$ | 15 | 18 | 45 | 38 |
| | 10 | 15 | 19.3903 | 7 | 7 | 23 | 19 |
| | 10 | 20 | 11.4315 | 5 | 7 | 18 | 18 |
| 0.2 | 5 | 5 | 7.8174 | – | 15 | 22 | 23 |
| | 5 | 10 | 13.1897 | 24 | 8 | 21 | 14 |
| | 5 | 15 | 28.9611 | 7 | 8 | 20 | 11 |
| | 10 | 10 | $10^{-10}$ | 17 | 18 | 43 | 39 |
| | 10 | 15 | 37.2044 | 6 | 11 | 24 | 18 |
| | 10 | 20 | 22.8625 | 5 | 7 | 19 | 19 |
| 0.5 | 5 | 5 | $10^{-10}$ | – | 12 | 50 | 42 |
| | 5 | 10 | 38.6785 | 24 | 6 | 20 | 12 |
| | 5 | 15 | 71.2177 | 12 | 9 | 31 | 18 |
| | 10 | 10 | $10^{-10}$ | – | 18 | 44 | 53 |
| | 10 | 15 | 63.1456 | 13 | 12 | 36 | 32 |
| | 10 | 20 | 57.1347 | 5 | 7 | 20 | 18 |
| 1 | 5 | 5 | 21.3573 | – | 14 | 68 | 22 |
| | 5 | 10 | 89.3793 | 31 | 8 | 14 | 11 |
| | 5 | 15 | 137.8861 | 14 | 9 | 35 | 15 |
| | 10 | 10 | 3.7004 | – | 29 | 54 | 68 |
| | 10 | 15 | 81.4203 | 7 | 13 | 31 | 28 |
| | 10 | 20 | 114.2509 | 5 | 8 | 20 | 19 |

*Discussion of P8.*

The condition number of the matrix $J^T(x)J(x)$ at the solution was found to be of order $10^9$. Therefore, for none of the algorithms the convergence is quadratic (see table 5.7).

Table 5.7

| $\lVert f(x) \rVert$ | number of function evaluations needed | | | |
|---|---|---|---|---|
| | G | M | F | R |
| .031250 | 5 | 10 | 11 | 13 |
| .013872 | 19 | 11 | 14 | 15 |
| .007392 | 23 | 29 | 61 | 59 |

## 6. CONCLUSIONS

From the results given in section 5 it is obvious that algorithms R and F, which are efficient algorithms for general minimization (BUS [2]), are not efficient for minimizing a sum of squares. For such problems, one had better use an algorithm which is designed to solve nonlinear least squares problems. Furthermore, we may conclude that algorithm G is less reliable for general problems than algorithm M. However, for problems which are known to be well-conditioned and where a good initial approximation of the parameters is known, algorithm G will usually be more efficient than algorithm M.

Therefore, we advise the user to use algorithm M, unless he knows that the matrix $J^T(x)J(x)$ has a relatively small condition number for all x in some convex region containing the solution and the initial guess.

## ACKNOWLEDGEMENTS

We are very grateful to the Working Group on Numerical Algebra at Amsterdam and also to P.W. Hemker, for a large number of valuable suggestions concerning the implementation of the algorithms; to Dr. P.J. van der Houwen for his careful reading of the manuscript and to Th. Gunsing, Mrs. J.W. van Riel-Dijk and D. Zwarst for their efforts to get this report typed and printed.

## REFERENCES

[1] BROWN, K.M., *A quadratically convergent Newton-like method based upon Gaussian elimination*, SIAM. J. Num. An. 6 (1969), 560-569.

[2] BUS, J.C.P., *Minimization of functions of more variables* (Dutch), Mathematical Centre report NR 29/72 (1972).

[3] BUSINGER, P. & G.H. GOLUB, *Linear least squares solutions by Householder transformations*, Num. Math. 7 (1965), 269-276.

[4] DEKKER, T.J., *ALGOL 60 procedures in numerical algebra, part 1*, Mathematical Centre tract 22 (1968).

[5] DOMSELAAR, B. VAN, *A mathematical analysis of the heart-infarct* (Dutch), Mathematical Centre report NN 4/74 (1974).

[6] FLETCHER, R., *A new approach to variable metric algorithms*, Comp. J. 13 (1970), 317-322.

[7] FLETCHER, R. & M.J.D. POWELL, *A rapidly convergent descent method for minimization*, Comp. J. 6 (1963), 163-168.

[8] FREUDENSTEIN, F. & B. ROTH, *Numerical solution of systems of nonlinear equations*, J. ACM. 10 (1963), 550-556.

[9] GOLDSTEIN, A.A. & J.F. PRICE, *An effective algorithm for minimization*, Num. Math. 10 (1967), 184-189.

[10] GOLUB, G.H., *Numerical methods for solving linear least squares problems*, Num. Math. 7 (1965), 206-216.

[11] GOLUB, G.H. & V. PEREYRA, *The differentiation of pseudo inverses and nonlinear least squares problems whose variables separate*, Univ. Central de Venezuela, Publ. 72-05 (1972).

[12] GOLUB, G.H. & C. REINSCH, *Singular value decomposition and least squares solution*, Num. Math. 14 (1970), 403-420.

[13] HARTLEY, H.O., *The modified Gauss-Newton method*, Technometrics 3 (1961), 269-280.

[14] LEVENBERG, K., *A method for the solution of certain non-linear problems in least squares*, Quart. Appl. Math. 2 (1944), 164-168.

[15] MARQUARDT, D.W., *An algorithm for least-squares estimation of nonlinear parameters*, SIAM. J. 11 (1963), 431-441.

[16] *NUMAL, a library of numerical procedures in ALGOL 60*, Vol. 0 up to 8, Mathematical Centre, Amsterdam (1974).

[17] POWELL, M.J.D., *A hybrid method for nonlinear equations.* In: Rabinowitz, P. (ed.), Numerical Methods for nonlinear algebraic equations. Gordon and Breach (1970).

## APPENDIX: ALGOL 60 procedures

In this appendix we give the texts of two ALGOL 60 procedures, *gssnewton* and *marquardt*, which implement algorithms G and M respectively. Before explaining the parameters we should give the following details about procedure marquardt (cf. algorithm M, section 3). The constants $\mu$, $\nu$ and $\omega$ are given values inside the procedure:

$$\mu = 0.01, \qquad \omega = 0.5, \qquad \text{and} \qquad \nu = 10.$$

Furthermore, an upper bound on the value of $\lambda_k$ is imposed which depends on $\|J^T(x_k)J(x_k)\|$ and the machine precision $\varepsilon$, since it makes no sense to use a value of $\lambda_k$ which satisfies

$$\lambda_k \geq \|J^T(x_k)J(x_k)\|/\varepsilon,$$

for in that case $J^T(x_k)J(x_k) + \lambda_k I$ is equal to $\lambda_k I$ if computed with precision $\varepsilon$. An error exit is created if $\lambda_k$ becomes that large, since in our opinion this can only occur if the precision asked for is too high, or the function and/or jacobian matrix are not programmed correctly.

the heading of the procedure marquardt is:

```
    procedure marquardt(m, n, par, rv, jjinv, funct, jacobian, in,
       out); value m, n; integer m, n;
    array par, rv, jjinv, in, out; boolean procedure funct;
    procedure jacobian;
```

the meaning of the formal parameters is:

m: <arithmetic expression>;
the number of equations;

n: <arithmetic expression>;
the number of unknown variables; n should satisfy $n \leq m$;

par: <array identifier>;
array par[1 : n];
the unknown variables of the system;
entry: an approximation to a least squares solution of the system;
exit: the calculated least squares solution;

rv: <array identifier>;
array rv[1 : m];
exit: the residual vector at the calculated solution;

jjinv: <array identifier>;
array jjinv[1 : n, 1 : n];
exit: the inverse of the matrix $J^* \times J$ where J denotes the matrix of partial derivatives drv[i] / dpar[j] (i=1,...,m; j=1,...,n) and $J^*$ denotes the transpose of J.

funct: <procedure identifier>;
the heading of this procedure should be:

boolean procedure funct(m, n, par, rv); value m, n;

integer m, n; array par, rv;

entry: m, n, par;

m, n have the same meaning as in the procedure marquardt;

array par[1:n] contains the current values of the unknowns and should not be altered;

exit: array rv[1 : m];

upon completion of a call of funct, this array rv should contain the residual vector, obtained with the current values of the unknowns;

e.g. in curve fitting problems:

rv[i] := theoretical value f(x[i], par) -
observed value y[i];

after a successful call of funct, the boolean procedure should deliver the value true;

however, if funct delivers the value false, then it is assumed that the current estimates of the unknowns lie outside a feasible region and the process is terminated (see out[1]);

hence, proper programming of funct makes it possible to avoid calculation of a residual vector with values of the unknown variables which make no sense or which even may cause overflow in the computation;

jacobian: <procedure identifier>;

the heading of this procedure should be:

procedure jacobian(m, n, par, rv, jac, locfunct);

value m, n; integer m, n; array par, rv, jac;

procedure locfunct;

entry: m, n, par, rv, locfunct;

for m,n,par see: funct;

rv contains the residual vector obtained with the current values of the unknowns and should not be altered;

a call of locfunct(m,n,par,rv) is equivalent with a call of the user-defined procedure funct(m,n,par,rv), but, in addition, this call is counted to the total number of calls of funct (see out[4]) and, moreover, if funct delivers the value false then the process is terminated;

exit: array jac[1 : m, 1 : n];

upon completion of a call of jacobian, jac should contain the partial derivatives drv[i] / dpar[j], obtained with the current values of the unknown variables given in par[1:n];

it is a prerequisite for the proper operation of the procedure marquardt that the precision of the elements of the matrix jac is at least the precision defined by in[3] and in[4];

in: <array identifier>;

array in[0 : 6];

entry: in this array the user should give some data to control the process;

in[0]: the machine precision;

for the cyber 73 a suitable value is $_{10}$-14;

in[1], in[2] are not used by the procedure marquardt;

in[3], in[4]:

the relative and absolute tolerance for the difference between the euclidean norm of the ultimate and penultimate residual vector; the process is terminated if the improvement of the sum of squares is less than in[3] × (sum of squares) + in[4] × in[4]; these tolerances should be chosen greater than the corresponding errors of the calculated residual vector; note that the euclidean norm of the residual vector is defined as the square root of the sum of squares;

in[5]: the maximum number of calls of funct allowed;

in[6]: a starting value used for the relation between the gradient and the gauss-newton direction (see [2]); if the problem is well conditioned then a suitable value for in[6] will be 0.01; if the problem is ill conditioned then in[6] should be greater, but the value of in[6] should satisfy: in[0] < in[6] $\leq$ 1/in[0];

out: <array identifier>; array out[1 : 7];

exit : in array out some by-products are delivered;

out[1]: this value gives information about the termination of the process;

out[1]=0: normal termination;

out[1]=1: the process has been broken off, because the number of calls of funct exceeded the number given in in[5];

out[1]=2: the process has been broken off, because a call of funct delivered the value false;

out[1]=3: funct became false when called with the initial estimates of par[1:n]; the iteration process was not started and so jjinv[1:n,1:n] can not be used;

out[1]=4: the process has been broken off, because the precision asked for can not be attained; this precision is possibly chosen too high, relative to the precision in which the residual vector is calculated (see in[3]);

out[2]: the euclidean norm of the residual vector calculated with values of the unknowns delivered;

out[3]: the euclidean norm of the residual vector calculated with the initial values of the unknown variables;

out[4]: the number of calls of funct necessary to obtain the calculated result;

out[5]: the total number of iterations performed; note that in each iteration one evaluation of the jacobian matrix had to be made;

out[6]: the improvement of the euclidean norm of the residual vector in the last iteration step;

out[7]: the condition number of $J^* \times J$ , i.e. the ratio of its largest to smallest eigenvalues;

data and results:

if this procedure is used for curve fitting then the relative accuracy in the calculation of the residual vector depends strongly on the errors in the experimental data and this should be reflected in the parameters in[3] and in[4];
the matrix jjinv can be used if some statistical information about the fitted parameters is required; the standard deviation, covariance matrix and correlation matrix may be calculated easily from jjinv ;

procedures used (NUMAL [16]):

mulcol = cp31022,
dupvec = cp31030,
vecvec = cp34010,
matvec = cp34011,
tamvec = cp34012,
mattam = cp34015,
qrisngvaldec = cp34273.

the heading of the procedure gssnewton is:

```
procedure gssnewton(m, n, par, rv, jjinv, funct, jacobian, in,
    out);
value m, n; integer m, n; array par, rv, jjinv, in, out;
```

**boolean** **procedure** funct; **procedure** jacobian;

the meaning of the formal parameters is :

m : <arithmetic expression>;
the number of equations;

n : <arithmetic expression>;
the number of unknowns in the m equations ($n \leq m$);

par : <array identifier>; **array** par[1 : n];
the unknowns of the equations.
entry : an approximation to a least squares solution of the system.
exit : the calulated least squares solution;

rv : <array identifier>; **array** rv[1 : m];
exit : the residual vector of the system at the calculated solution;

jjinv : <array identifier>; **array** jjinv[1 : n,1 : n];
exit : the inverse of the matrix $J^*$ × j, where j is the jacobian matrix at the solution and $J^*$ is J transposed;

funct : <procedure identifier>;
the heading of this procedure should be :

**boolean** **procedure** funct(m, n, par, rv); **value** m, n; **integer** m, n; **array** par, rv;

entry: m, n, par;

m, n have the same meaning as in the procedure gssnewton;

array par[1:n] contains the current values of the unknowns and should not be altered.

exit: array rv[1 : m];

upon completion of a call of funct, this array rv should contain the residual vector, obtained with the current values of the unknowns.

the programmer of funct may decide that some current estimates of the unknowns lie outside a feasible region; in this case funct should deliver the value false and the process is terminated (see out[1]). otherwise funct should deliver the value true;

jacobian : <procedure identifier>;

the heading of this procedure should be :

```
procedure jacobian(m, n, par, rv, jac, locfunct);
value m, n; integer m, n; array par, rv, jac;
procedure locfunct;
```

the meaning of the parameters of jacobian is :

m, n : see gssnewton.

par : <array identifier>; array par[1 : n];

entry : current estimates of the unknowns. these values should not be changed.

rv : <array identifier>; array rv[1 : m];

entry : the residual vector of the system of equations corresponding to the vector of unknowns as given in par.

exit : the entry values.

jac : <array identifier>; array jac[1 : m, 1 : n];

exit : the jacobian matrix at the current estimates given in par, i.e. the matrix of partial derivatives

d(rv)[i] / dpar[j], i = 1(1)m, j = 1(1)n.

locfunct : <procedure identifier>; the heading of this procedure is the same as the heading of funct.

a call of the procedure jacobian should deliver the jacobian matrix evaluated with the current estimates of the unknown variables given in par
in such a way, that the partial derivative
d(rv)[i] / dpar[j] is delivered in jac[i,j], i = 1(1)m, j = 1(1)n.
for the calculation of the derivatives one can use the values of the current estimates of the
unknowns as given in par and the residual vector as given in rv.
one can also use the procedure funct
(parameter of gssnewton) through calls of the procedure locfunct (parameter of jacobian). this parameter of jacobian may be used when the jacobian matrix is approximated using (forward) differences.
an appropriate procedure to this purpose is jacobnmf

( NUMAL [16] ). such a procedure may be used only if the matrix elements are computed sufficiently accurate;

in : <array identifier>; array in[0 : 7];

in this array tolerances and control parameters should be given.

entry :

in[0] : the machine precision. for calculation on the cyber 73 a suitable value is $_{10}-14$.

in[1], in[2] :

relative and absolute tolerance for the step vector (relative to the vector of current estimates in par).

the process is terminated if in some iteration (but not the first) the euclidean norm of the calculated newton step is less than in[1] × norm(par) + in[2]. in[1] should not be chosen smaller than in[0].

in[3] is not used by the procedure gssnewton;

in[4] : absolute tolerance for the euclidean norm of the residual vector. the process is terminated when this norm is less than in[4].

in[5] : the maximum allowed number of function evaluations (i.e. calls of funct).

in[6] : the maximum allowed number of halvings of a calculated newton step vector ( see section 3 ). a suitable value is 15.

in[7] : the maximum allowed number of successive in[6] times halved step vectors. suitable values are 1 and 2;

out : <array identifier>; array out[1 : 9];

in array out information about the termination of the process is delivered.

exit :

out[1] :

the process was terminated because (out[1] = )

1. the norm of the residual vector is small with respect to in[4],
2. the calculated newton step is sufficiently small (see in[1], in[2]),
3. the calculated step was completely damped (halved) in in[7] successive iterations,
4. out[4] exceeds in[5], the maximum allowed number of calls of funct,
5. the jacobian was not full-rank (see out[8]),
6. funct delivered false at a new vector of estimates of the unknowns,
7. funct delivered false in a call from jacobian.

out[2] : the euclidean norm of the last residual vector.

out[3] : the euclidean norm of the initial residual vector.

out[4] : the total number of calls of funct.

out[4] will be less than in[5] + in[6].

out[5] : the total number of iterations.

out[6] : the euclidean norm of the last step vector.

out[7] : iteration number of the last iteration in which the newton step was halved.

out[8], out[9] :

rank and maximum column norm of the jacobian matrix in the last iteration, as delivered by lsqortdec ( NUMAL [16] ) in aux[3] and aux[5].

data and results :

the procedure gssnewton can be used for approximating an exact or a least squares solution of a system of nonlinear equations. when an exact solution is required, the procedure may terminate only with out[1] = 1, and very small values should be assigned to in[1] and in[2]. when a least squares solution is required, positive results of the procedure are signaled by out[1] = 1 or 2. whenever the procedure terminates with out[1] < 5, then the inverse of $J^* \times J$ (see meaning of the parameter jjinv) is delivered in jjinv. in that case the covariance matrix and the standard deviations of the solution can be calculated.

for a curve fitting problem, say :

estimate parameters par[1], ... , par[n] of a function
y = f(x; par[1], ... , par[n]), when a set of data (x[i],y[i]),
i = 1(1)m, has to be fitted,

the following system of m equations in the n unknown parameters par[1], ... , par[n] can be derived :

f(x[i]; par[1], ... , par[n]) - y[i] = 0, i = 1(1)m.

procedures used ( NUMAL [16]):

```
vecvec    = cp34010,
dupvec    = cp31030,
elmvec    = cp34020,
lsqortdec = cp34134,
lsqsol    = cp34131,
lsqinv    = cp34136.
```

source texts :

```
code 34440;
 procedure marquardt(m,n,par,g,v,funct,jacobian,in,out);
 value m,n; integer m,n; array par,g,v,in,out;
 boolean procedure funct; procedure jacobian;
 begin integer maxfe,fe,it,i,j,err;
          real vv,ww,w,mu,res,fpar,fparpres,lambda,lambdamin,
                 p,pw,reltolres,abstolres;
          array em[0:7],val,b,bb,parpres[1:n],jac[1:m,1:n];

          procedure mulcol(l,u,s,t,a,b,x); code 31022;
          procedure dupvec(l,u,s,a,b); code 31030;
          real procedure vecvec(l,u,s,a,b); code 34010;
          real procedure matvec(l,u,s,a,b); code 34011;
 real procedure tamvec(l,u,s,a,b); code 34012;
 real procedure mattam(l,u,s,t,a,b); code 34015;
 integer procedure qrisngvaldec(a,m,n,val,v,em);
 code 34273;


procedure locfunct(m,n,par,g);
integer m,n; array par,g;
begin fe:= fe+1; if fe ≥ maxfe then err:= 1 else
        if ¬ funct(m,n,par,g) then err:= 2;
        if err≠0 then goto exit
end locfunct;


vv:=10; w:=0.5; mu:= 0.01;
ww:=(if in[6]<⏨-7 then ⏨-8 else ⏨-1×in[6]);
em[0]:=em[2]:=em[6]:=in[0]; em[4]:=10×n;
reltolres:=in[3]; abstolres:=in[4]↑2; maxfe:=in[5];
err:= 0; fe:= it:= 1; p:=fpar:= res:= 0;
pw:=-ln(ww×in[0])/2.30;


if ¬ funct(m,n,par,g) then
begin err:= 3; goto escape end;
fpar:= vecvec(1,m,0,g,g); out[3]:=sqrt(fpar);


for it:= 1, it+1 while fpar > abstolres ∧
              res > reltolres×fpar+abstolres do
begin jacobian(m,n,par,g,jac,locfunct);
        i:=qrisngvaldec(jac,m,n,val,v,em);
        if it=1 then
        lambda:= in[6] × vecvec(1,n,0,val,val) else
    if p =0 then lambda:= lambda×w else p:= 0;

    for i:=1 step 1 until n do
    b[i]:=val[i]×tamvec(1,m,i,jac,g);

1:  for i:=1 step 1 until n do
    bb[i]:=b[i]/(val[i]×val[i]+lambda);
    for i:=1 step 1 until n do
    parpres[i]:= par[i] - matvec(1,n,i,v,bb);
    locfunct(m,n,parpres,g);
    fparpres:= vecvec(1,m,0,g,g);
    res:=fpar-fparpres;
    if res < mu × vecvec(1,n,0,b,bb) then
    begin p:= p+1; lambda:= vv × lambda;
       if p=1 then
       begin lambdamin:= ww × vecvec(1,n,0,val,val);
          if lambda<lambdamin then lambda:= lambdamin
       end;
       if p<pw then goto 1 else
       begin err:= 4;
             goto exit
       end;
    end;

    dupvec(1,n,0,par,parpres);
    fpar:=fparpres
end iteration;
```

```
    exit:
         for i:=1 step 1 until n do
         mulcol(1,n,i,i,jac,v,1/(val[i]+in[0]));
         for i:=1 step 1 until n do
         for j:=1 step 1 until i do
         v[i,j]:= v[j,i]:= mattam(1,n,i,j,jac,jac);

         lambda:= lambdamin:= val[1];
         for i:= 2 step 1 until n do
         if val[i]>lambda    then lambda    := val[i] else
         if val[i]<lambdamin then lambdamin:= val[i];

         out[7]:=(lambda/(lambdamin+in[0]))↑2;
         out[2]:=sqrt(fpar);
         out[6]:=sqrt(res+fpar)-out[2];
    escape:
         out[4]:=fe;
         out[5]:=it-1;
         out[1]:=err
    end marquardt;


code 34441;
   procedure gssnewton(m, n, par, rv, jjinv, funct, jacobian,
     in, out);
   value m, n; integer m, n;
   array par, rv, jjinv, in, out;
```

```
boolean procedure funct;
procedure jacobian;

begin integer i, j, inr, mit, text,
    it, itmax, inrmax, tim, feval, fevalmax;
    real rho, res1, res2, rn, reltolpar, abstolpar, abstolres,
       stap, normx;
    boolean conv, testthf, damping on;
    array jac[1:m + 1,1:n], pr, aid, sol[1 : n], fu2[1 : m],
       aux[2 : 5];
    integer array ci[1:n];

    real procedure vecvec(l, u, shift, a, b); code 34010;
    procedure dupvec(l, u, s, a, b); code 31030;
    procedure elmvec(l, u, s, a, b, x); code 34020;
    procedure lsqortdec(a, m, n, aux, aid, ci); code 34134;
    procedure lsqsol(a, m, n, aid, ci, b); code 34131;
    procedure lsqinv(a, n, aid, ci); code 34136;

    boolean procedure loc funct(m, n, par, rv);
    value m, n; integer m, n; array par, rv;
    begin loc funct:= test thf:= funct(m, n, par, rv)
        ∧ test thf; feval:= feval + 1
    end loc funct;


    itmax:= fevalmax:= in[5]; aux[2]:= n × in[0]; tim:= in[7];
    reltolpar:= in[1] ↑ 2; abstolpar:= in[2] ↑ 2;
```

```
abstolres:= in[4] ↑ 2; inrmax:= in[6];
dupvec(1, n, 0, pr, par);
if m < n then
for i:= 1 step 1 until n do jac[m + 1, i]:= 0;
text:= 4; mit:= 0; test thf:= true;
res2:= stap:= out[5]:= out[6]:= out[7]:= 0;
funct(m, n, par, fu2); rn:= vecvec(1, m, 0, fu2, fu2);
out[3]:= sqrt(rn); feval:= 1; damping on:= false;
for it:= 1, it + 1 while it ≤ itmax ∧
     feval < fevalmax do
begin out[5]:= it; jacobian(m, n, par, fu2, jac, locfunct);
  if ¬ test thf then
  begin text:= 7; goto fail end;
  lsqortdec(jac, m, n, aux, aid, ci);
  if aux[3] ≠ n then
  begin text:= 5; go to fail end;
  lsqsol(jac, m, n, aid, ci, fu2); dupvec(1, n, 0, sol, fu2);
  stap:= vecvec(1, n, 0, sol, sol);
  rho:= 2; normx:= vecvec(1, n, 0, par, par);
  if stap > reltolpar × normx + abstolpar
    ∨ it = 1 ∧ stap > 0 then
  begin for inr:= 0, inr + 1
    while if inr = 1 then damping on ∨ res2 ≥ rn
    else ¬ conv ∧ (rn ≤ res1 ∨ res2<res1) do
    begin comment damping stops when
      r0 > r1 ∧ r1 ≤ r2 (best result is x1, r1)
      with  x1 = x0 + i × dx, i:= 1, .5, .25, .125, etc.  ;
      rho:= rho / 2; if inr > 0 then
```

```
        begin res1:= res2; dupvec(1, m, 0, rv, fu2);
            damping on:= inr > 1
        end;
        for i:= 1 step 1 until n do
        pr[i]:= par[i] - sol[i] × rho;
        feval:= feval + 1;
        if ¬ funct(m, n, pr, fu2) then
        begin text:= 6; goto fail end;
        res2:= vecvec(1, m, 0, fu2, fu2); conv:= inr ≥ inrmax
    end damping of step vector;
    if conv then
    begin comment residue constant; mit:= mit + 1;
        if mit < tim then conv:= false
    end else mit:= 0;
    if inr > 1 then
    begin rho:= rho × 2; elmvec(1, n, 0, par, sol, - rho);
        rn:= res1; if inr > 2 then out[7]:= it
    end else
    begin dupvec(1, n, 0, par, pr); rn:= res2;
        dupvec(1, m, 0, rv, fu2)
    end;

    if rn ≤ abstolres then
    begin text:= 1; itmax:= it end else
    if conv ∧ inrmax > 0 then
    begin text:= 3; itmax:= it end
    else dupvec(1, m, 0, fu2, rv)
end iteration with damping and tests else
```

```
       begin text:= 2; rho:= 1; itmax:= it end
    end of iterations;


    lsqinv(jac, n, aid, ci);
    for i:= 1 step 1 until n do
    begin jjinv[i,i]:= jac[i,i];
        for j:= i + 1 step 1 until n do
        jjinv[i,j]:= jjinv[j,i]:= jac[i,j]
    end calculation of inverse matrix of normal equations;
fail :
    out[6]:= sqrt(stap) × rho; out[2]:= sqrt(rn); out[4]:= feval;
    out[1]:= text; out[8]:= aux[3]; out[9]:= aux[5]
end gssnewton;
```