

May 18, 2007

Large-scale optimization with the limited-memory BFGS program **LVMINI**

Volker Blobel

Test version

This is a test version! The program LVMINI seems to be a rather efficient minimization program for various fitting problems with large number of parameters and with a small number of parameters as well. Until now there is only limited experience and the program may fail for certain classes of problems. At present a single pure algorithm is used. Users are encouraged to try the code - but please, report bad and good experience to the author.

Efficient standard function minimization algorithms for optimization based on NEWTON steps require to calculate and store, among other arrays, the matrix of second derivatives of the parameters. For a very large number n of parameters of the order of 10^5 this would require a very large memory space, larger than available on a standard PC, while the gradient of the function (vector of first derivatives) requires much less space. Simple steepest-descent minimization algorithms, requiring only the gradient, are rather slowly convergent or even, in the case of large correlations between parameters, not converging at all. The L-BFGS algorithm for large-scale optimization is a matrixless quasi-NEWTON method, which allows to take advantage of the good convergence properties of NEWTON methods without having to store the full matrix of second derivatives. It requires the *calculation of the gradient*, i.e. the first derivatives of the objective function. Memory requirements are modest and should allow the solution of minimization problems with e.g. 10^5 parameters on a standard PC in a reasonable time.

Contents

1	Large-scale optimization	3
2	The NEWTON method and objective functions	3
3	Quasi-NEWTON methods for optimization	5
4	Line-search methods	6
5	The limited-memory algorithm	7
6	The LVMINI program	8
7	Numerical results	12
	References	19
	The LVMINI Manual	20
1.	The auxiliary array AUX	20
2.	Constants of the algorithm	20
3.	Minimization	20
4.	Access to the result	22
5.	Printout	22
6.	Alternative: Minimization with a MINUIT-like function	23
7.	Example: Straight line least-squares fit	23
	The LVMINI mini Manual	25

1 Large-scale optimization

Important scientific and technical problems depend on a very large number of parameters. The increase in cheap computing power with larger memory space during the last years allows to solve optimization problems with a larger number of parameter than before, often with traditional methods, which require a lot of computation and large memory space. The standard NEWTON algorithm for function minimization requires to calculate and store the second-derivative matrix and to perform operations with the matrix. Space and cpu-time consumption increase with the second and the third power of the number n of parameters in standard methods, which were traditionally used for problems with a small number of parameters. This dependence on n means a limitation for traditional methods.

Matrix-less methods should allow the solution of larger problems. The simple *steepest descent* method for minimization requires only the gradient vector, but is not scale-invariant and re-scaling can lead to widely different steepest descent directions. The length of the step is not defined by the method. Convergence may be and almost always is very slow in theory, and numerically convergence sometimes does not occur at all as the iteration stagnates¹. Conjugate gradient methods are useful techniques for solving large linear systems of equations and can also be used for nonlinear optimization problems. Another method, related to the conjugate gradient method, is a matrix-less version of the variable metric method, called limited-memory BFGS, which is discussed in this paper. A test version of a minimization program has been developed, which is applicable for large-scale minimization problems, but also for problems with a small number of parameters. The usage of this program LVMINI can help to gain experience in the large-scale optimization.

The method is a variant of the NEWTON method (section 2), and of Quasi-NEWTON methods (section 3). Iterative NEWTON methods in general can be improved by the use of efficient line-search methods (section 4). The limited-memory version of Quasi-NEWTON methods (section 5) is the basis of the program LVMINI, described in section 6. The program allows to find optimal parameters in large-scale optimization problems without constraints, and, for problems on a smaller scale, the covariance matrix in addition. The program requires the accurate calculation of the gradient of the objective function.

Future versions of the program may include the possibility of linear equality constraints, using the elimination method, and also the possibility of numerical gradient calculation.

A modern textbook for these methods and in general on optimization methods is by J. Nocedal and S.J. Wright[1]. Several formulas shown in this report and algorithms used in the program LVMINI are taken from the book.

2 The NEWTON method and objective functions

In optimization problems an objective function $F(\mathbf{x})$ depending on a parameter vector \mathbf{x} is constructed, based on statistical considerations, for example by the method of least squares or the maximum likelihood method. Optimization requires to find the minimum of the function with respect to the parameter n -vector \mathbf{x} :

$$\min_{\mathbf{x}} F(\mathbf{x}), \quad \mathbf{x} \in \mathcal{R}^n .$$

The standard NEWTON method of minimization is based on a sequence of *quadratic models* of the objective function with the iterative determination of parameter values

$$\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_k \dots ,$$

starting from the initial parameter value \mathbf{x}_0 . The quadratic model $M(\mathbf{d}) \approx F(\mathbf{x}_k + \mathbf{d})$ of the objective function $F(\mathbf{x})$ defined in iteration k at the current parameter value \mathbf{x}_k

$$M_k(\mathbf{d}) = F_k + \nabla F_k^T \mathbf{d} + \frac{1}{2} \mathbf{d}^T \mathbf{C}_k \mathbf{d} \tag{1}$$

¹This behaviour is often misinterpreted and assumed to indicate convergence.

is a quadratic function, which depends on the gradient vector ∇F_k^T and on the Hessian $\nabla^2 F_k \equiv \mathbf{C}_k$, the matrix of second derivatives of the objective function. The quadratic model function $M(\mathbf{d})$ is minimized by a step vector \mathbf{d}_k from the solution of the matrix equation

$$\mathbf{C}_k \mathbf{d}_k = -\nabla F_k \quad (2)$$

for the step \mathbf{d}_k . The NEWTON direction is only guaranteed to be useful in a line-search context if the Hessian \mathbf{C}_k is positive-definite; otherwise the direction may be an *ascent* direction. The NEWTON method can be regarded as a steepest descent method, *scaled* by the metrik of the matrix \mathbf{C}_k , and for a suitable matrix \mathbf{C}_k the convergence with line-search is, compared to steepest descent, significantly faster, often with quadratic convergence rate.

The solution of this equation with matrix inversion

$$\mathbf{d}_k = -\mathbf{B}_k \nabla F_k^T \quad \text{with } \mathbf{B}_k = \mathbf{C}_k^{-1}$$

is not the only solution method. If the objective function is a negative log-likelihood function, then the inverse Hessian \mathbf{B} is (after convergence) the covariance matrix \mathbf{V} of the parameters \mathbf{x} .

For a linear least squares problem, where the objective function is in fact a quadratic function, the solution is obtained in one step, if the gradient and the (non-singular) Hessian are accurately calculated. In other cases, where e.g. the Hessian is not accurately calculated or in non-linear least squares problem, iterations are necessary. The iterations are continued until convergence with gradient $\nabla F_k \rightarrow \mathbf{0}$.

The convergence behaviour of an iterative solution is improved by *line-search* algorithms. In *line search* algorithms the vector \mathbf{d}_k is used as step vector in the minimization of the function $\phi(\alpha_k) \equiv F(\mathbf{x}_k + \alpha_k \mathbf{d}_k)$, depending on a single parameter α_k , and the after minimization of $\phi(\alpha_k)$ the new value of the parameter vector is $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$.

Objective functions in statistical applications

Optimization in statistical applications is usually based on the Maximum Likelihood principle. The maximum of the Likelihood function is determined, where the Likelihood function is a product of probability or probability density values. In practice and also for theoretical reasons the negative logarithm of the Likelihood function (log-likelihood function) is used. The inverse of the matrix of second derivative of this function, which is used in the NEWTON method, is a good approximation to the covariance matrix of parameters. The second=derivative matrix is usually, especially in the case of high statistic, almost constant in a region around the minimum.

Least-squares objective functions. In the case of least squares the equations have a certain structure, which often allows to calculate the gradient and the Hessian analytically, at least to a good approximation. Given the measured values m_i , with standard deviations σ_i assigned, the objective function $F(\mathbf{x})$ is given by the sum

$$F(\mathbf{x}) = \frac{1}{2} \sum_i \frac{1}{\sigma_i^2} (f_i - m_i)^2, \quad (3)$$

where the function values f_i represent the parametrized expectation for the measured values m_i . $F(\mathbf{x})$ is the negative log-likelihood function for normal distributed data (note the factor 1/2 in front of the sum, often called a “ χ^2 function” by physicists). The gradient and the Hessian of the objective function $F(\mathbf{x})$ are given by the following sums:

$$(\nabla F)_j = \sum_i \frac{1}{\sigma_i^2} (f_i - m_i) \frac{\partial f_i}{\partial x_j} \quad (4)$$

$$(\nabla^2 F)_{jk} = \sum_i \frac{1}{\sigma_i^2} \frac{\partial f_i}{\partial x_j} \frac{\partial f_i}{\partial x_k} + \sum_i \frac{1}{\sigma_i^2} (f_i - m_i) \frac{\partial^2 f_i}{\partial x_j \partial x_k}. \quad (5)$$

The value of the sum (i.e. $2 \times F(\mathbf{x}^*)$), at the minimum should follow (for the correct parametrized expectation) a χ^2 distribution; the number of degrees of freedom is equal to the number of data points minus the number of parameters.

The gradient ∇F is easily calculated, if it is possible to calculate the first partial derivatives of the function values f_i analytically. In addition the first part of the Hessian can be calculated, using the same data; the first term is usually, at least in the neighbourhood of the solution, more important than the second part. The second part requires the second partial derivatives of the function f_i ; it vanishes for a linear function and becomes small for small residuals $(f_i - m_i)$, i.e. in the neighbourhood of the solution; in practice it is often neglected, and in the literature it is even recommended not to include this term. The approximate matrix (without the second part) is called the GAUSS-NEWTON matrix. An inaccurate or *truncated* Hessian may still be used with success in the NEWTON method.

Maximum-likelihood function for histogram fits. The bin content m_i (counts) follows a Poisson distribution with mean value equal to the expectation value. If the function values f_i represents the parametrized expectation, the objective function in analogy to the least-squares function is given by

$$F(\mathbf{x}) = \sum_i \begin{cases} f_i - m_i + m_i \ln \left(\frac{m_i}{f_i} \right) & \text{if } m_i > 0, \\ f_i & \text{if } m_i = 0. \end{cases} \quad (6)$$

This expression, which includes fixed contributions from the measured values m_i , has a value close to the corresponding least squares function and will become equal to that function in the limit of large values m_i and f_i . The gradient and the Hessian (neglecting terms proportional to second derivatives of the expectation f_i) are:

$$(\nabla F)_j = \sum_i \frac{\partial f_i}{\partial x_j} \left(1 - \frac{m_i}{f_i} \right) \quad (\nabla^2 F)_{jk} = \sum_i \frac{\partial f_i}{\partial x_j} \frac{\partial f_i}{\partial x_k} \frac{m_i}{f_i^2} \quad (7)$$

For large values m_i and f_i the fitted parameters will be almost identical to the least squares case, if σ_i^2 is replaced by f_i , the variance of the Poisson distribution. For small numbers m_i and perhaps values zero the Poisson-based objective function is preferred.

3 Quasi-NEWTON methods for optimization

For an n -vector \mathbf{x} the amount of computation and the required space in the standard NEWTON method with an n -by- n matrix increases at least with the square of the dimension parameter n . Furthermore the evaluation of the second derivative matrix of the objective function may be rather difficult. Quasi-NEWTON methods require only the repeated evaluation of the gradient of the objective function. From the changes in gradients a model of the objective function is constructed which is accurate enough to give superlinear convergence to the solution. The first quasi-NEWTON algorithm was developed in the mid 1950's by W.C. Davidon (Argonne National Laboratory) [2]. In the year 1958 the original paper was not accepted for publication, but in the following years numerous variants were proposed and many papers were published on quasi-NEWTON methods. The most popular quasi-NEWTON algorithm is the so-called BFGS method, named for its discoverers (in the year 1970) C.G. Broyden, R. Fletcher, D. Goldfarb and D.F. Shanno.

The quasi-NEWTON BFGS algorithm starts from some approximation \mathbf{C}_0 of the Hessian, often the unit matrix or a scaled unit matrix. A step \mathbf{d}_k is calculated from the gradient and improved by line-search methods. The update from a matrix \mathbf{C}_k to a matrix \mathbf{C}_{k+1} uses the difference vectors in parameter space and the gradient differences in subsequent steps. The vectors

$$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k \quad \mathbf{y}_k = \nabla F_{k+1} - \nabla F_k \quad (8)$$

are defined and the requirement for the update $\mathbf{C}_{k+1} = \mathbf{C}_k + \Delta \mathbf{C}$ can be expressed by

$$\mathbf{C}_{k+1} \mathbf{s}_k = \mathbf{y}_k, \quad (9)$$

which is called the *secant equation*. With this requirement the updated model function becomes

$$M_{k+1}(\mathbf{d}) = F_{k+1} + \nabla F_{k+1}^T \mathbf{d} + \frac{1}{2} \mathbf{d}^T \mathbf{C}_{k+1} \mathbf{d}$$

and, after the increase $k := k + 1$, a new step \mathbf{d}_k can be calculated by the solution of the linear equation $\mathbf{C}_k \mathbf{d}_k = -\nabla F_k$.

An interesting property of the method is that, instead of the condition on the update of the Hessian \mathbf{C} , an equivalent condition can be directly used for the inverse \mathbf{B} of the Hessian:

$$\mathbf{B}_{k+1} \mathbf{y}_k = \mathbf{s}_k \quad (10)$$

(with \mathbf{y}_k and \mathbf{s}_k exchanged). In this case a new step \mathbf{d}_k is calculated by a simple matrix-vector product:

$$\mathbf{d}_k = -\mathbf{B}_k \nabla F_k. \quad (11)$$

The BFGS updating formula for the inverse of the Hessian is

$$\mathbf{B}_{k+1} = (\mathbf{I} - \rho_k \mathbf{s}_k \mathbf{y}_k^T) \mathbf{B}_k (\mathbf{I} - \rho_k \mathbf{s}_k \mathbf{y}_k^T) + \rho_k \mathbf{s}_k \mathbf{s}_k^T \quad \text{with } \rho_k = \frac{1}{\mathbf{y}_k^T \mathbf{s}_k}. \quad (12)$$

The BFGS update formula is generally assumed to be the best quasi-NEWTON update formula. It is guaranteed to generate positive definite matrices as long as the initial matrix \mathbf{B}_0 is positive definite and $\mathbf{s}_k^T \mathbf{y}_k > 0$; the last condition is ensured by the strong Wolfe condition of the line-search.

4 Line-search methods

Line-search methods are an important component of a minimization algorithm. The accurate minimization of the univariate function

$$\phi(\alpha) \equiv F(\mathbf{x}_k + \alpha \mathbf{d}_k)$$

would generally require a large number of function (and gradient) evaluations. Instead practical strategies perform an *inexact* line-search to determine a step length parameter α at minimal cost, that achieves *adequate* reduction of the function F such that the algorithm makes reasonable progress in minimization. The strong Wolfe conditions require α to satisfy the two inequalities

$$F(\mathbf{x}_k + \alpha \mathbf{d}_k) \leq F(\mathbf{x}_k) + c_1 \alpha \nabla F_k^T \mathbf{d}_k \quad (13)$$

$$|\nabla F(\mathbf{x}_k + \alpha \mathbf{d}_k)^T \mathbf{d}_k| \leq c_2 |\nabla F_k^T \mathbf{d}_k| \quad (14)$$

with $0 < c_1 < c_2 < 1$. These two conditions for an inexact line-search should give

- *sufficient decrease* in the objective function F (first inequality (13)) and,
- *sufficient slope reduction* according to the *curvature condition* (second inequality (14)), in order to avoid short steps; the magnitude of the slope of $\phi(\alpha)$ is required to be smaller than c_2 times the magnitude of the initial slope $\phi(0)$.

For a *loose* line-search good parameter values are $c_1 = 10^{-4}$ and $c_2 = 0.9$. A smaller value of c_2 forces accepted value to lie closer to the minimum (e.g. for nonlinear conjugate gradient methods). An efficient line-search algorithm is difficult to code. A good software implementation is available in the public domain [4].

5 The limited-memory algorithm

The quasi-NEWTON algorithm with the BFGS updating formula and with line-search represents an efficient optimization method for many problems. The solution is determined iteratively according to

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k \quad \mathbf{d}_k = -\mathbf{B}_k \nabla F_k \quad k = 0, 1, 2, \dots \quad (15)$$

The approximation of the n -by- n Hessian or its inverse is dense, and storage and computing requirements grow at least with the square of the number n of parameters. For this reason the method described in section 3 can not be applied directly to large-scale optimization problems, where the number of parameters is of the order of $n = 10^4$ or even $> 10^5$.

For large scale optimization a method is needed with the simplicity of steepest descent (no matrix needed) and the power of the NEWTON method, but without the matrix operations. The method of conjugate gradients is a compromise between the two methods: it allows to calculate a sequence of search directions which are conjugate with $\mathbf{d}_i^T \mathbf{C} \mathbf{d}_j = 0$ for $i \neq j$ and can be used for a sequence of one-dimensional minimizations.

Another possibility is to extend the quasi-NEWTON method, suitable for large-scale optimization by avoiding to store the large matrix. This method is related to the method of conjugate gradients. The *limited-memory* BFGS method is based on the observation that the inverse Hessian \mathbf{B}_k is used *only* in the product $\mathbf{d}_k = -\mathbf{B}_k \nabla F_k$ of equation (15) for the calculation of the next step \mathbf{d}_k . The matrix \mathbf{B}_k is completely defined by the initial matrix \mathbf{B}_0 and the vector pairs $\{\mathbf{s}_1, \mathbf{y}_1\}, \{\mathbf{s}_2, \mathbf{y}_2\} \dots \{\mathbf{s}_k, \mathbf{y}_k\}$. It is possible to calculate the new step \mathbf{d}_k with an explicit use only of the k vector pairs, as shown by J. Nocedal[3]. The initial approximation matrix \mathbf{B}_0 is e.g. a diagonal matrix. Modified versions of \mathbf{B}_k are stored implicitly by storing up to m of the vector pairs $\{\mathbf{s}_k, \mathbf{y}_k\}$. The product $\mathbf{B}_k \nabla F_k$ is obtained by performing a sequence of scalar products and vector summation. The oldest pair is deleted in each iteration and replaced by the new pair $\{\mathbf{s}_k, \mathbf{y}_k\}$.

The following two algorithms are from ref. [1] and [3] (algorithms numbered 7.5 and 7.4 in ref. [1]).

Limited-memory algorithm:

Choose starting parameter vector \mathbf{x}_0 , define $m > 0$; $k = 0$

repeat

 Choose approximate inverse Hessian \mathbf{B}_k^0

 Compute $\mathbf{d}_k := -\mathbf{B}_k \nabla F_k$ (see algorithm below)

 Perform line-search to determine α_k in $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{d}_k$

 If $k > m$

 Remove vector pair $\{\mathbf{s}_{k-m}, \mathbf{y}_{k-m}\}$

 Save new pair $\{\mathbf{s}_k, \mathbf{y}_k\}$ with $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{y}_k = \nabla F_{k+1} - \nabla F_k$

 and calculate $\rho_k = 1/(\mathbf{s}_k^T \mathbf{y}_k)$

$k := k + 1$

until convergence

The line-search should be based on the strong Wolfe condition in order to have stable BFGS updating. The step length $\alpha_k = 1$ will be accepted in most iterations. A strategy for the initial step length is to interpolate a parabola to the two function values F_k and F_{k-1} and the slope $\phi'(0) = \nabla F_k^T \mathbf{d}_k$, with the result $\alpha_0 = 2(F_k - F_{k-1})/\phi'(0)$, which will often converge to 1. It is recommended to use the formula $\alpha_0 := \min(1.0, 1.01 \cdot \alpha_0)$.

The algorithm for the computation of the product $\mathbf{B}_k \mathbf{q}$ of the approximate inverse Hessian \mathbf{B}_k with some vector \mathbf{q} is a two-loop recursion. The new step vector is calculated by $\mathbf{d}_k = -\mathbf{B}_k \nabla F_k$. The same algorithm is later used for an error estimate.

Two-loop recursive algorithm: calculation of $B_k \nabla F_k$

```

 $\mathbf{q} := \nabla F_k$ 
for  $i = k - 1, k - 2, \dots, k - m$ 
     $\alpha_i := \rho_i \mathbf{s}_i^T \mathbf{q}$ 
     $\mathbf{q} := \mathbf{q} - \alpha_i \mathbf{y}_i$ 
end(for)
 $\mathbf{r} := B_k^0 \mathbf{q}$ 
for  $i = k - m, k - m + 1, \dots, k - 1$ 
     $\beta := \rho_i \mathbf{y}_i^T \mathbf{r}$ 
     $\mathbf{r} := \mathbf{r} + \mathbf{s}_i (\alpha_i - \beta)$ 
end(for)
stop with result  $B_k \nabla F_k = \mathbf{r}$ 

```

The two-loop recursion requires $4mn$ multiplications and in addition the product $B_k^0 \mathbf{q}$ between the two loops. The initial matrix B_k^0 , which appears in the boxed equation $\mathbf{r} := B_k^0 \mathbf{q}$, can be chosen freely and can vary between iterations, but the efficiency depends on the choice. One possibility for B_k^0 is to set

$$B_k^0 = \gamma_k \mathbf{I} \quad \text{with factor} \quad \gamma_k = \frac{\mathbf{s}_{k-1}^T \mathbf{y}_{k-1}}{\mathbf{y}_{k-1}^T \mathbf{y}_{k-1}}. \quad (16)$$

This scaled unit matrix seems to be rather efficient and can be used without additional information. Another possibility is to define instead a Hessian approximation C_k^0 and to determine the vector \mathbf{r} from the solution of the equation

$$C_k^0 \mathbf{r} = \mathbf{q}. \quad (17)$$

If the number of parameters is small, space is sufficient to store for example a band matrix or even the full Hessian. The choice of B_k^0 (or C_k^0) influences the behavior of the method. For a large number of parameters there will be space only for a diagonal matrix. A good choice would be to use B_k^0 equal to the approximation of the diagonal of the inverse Hessian for a large number of parameters.

6 The LVMINI program

Program code for the application of the limited memory BFGS method is available from Prof. Nocedal's web page in two versions: a version for unconstrained minimization (L-BFGS) and for minimization with bounds on parameters (L-BFGS-B).

Minimization in experimental high-energy physics are usually problems in the context of maximum likelihood fits and allow to use specialized convergence criteria. In addition error estimates or the full covariance matrix of parameters are required. These features are included in the program LVMINI, described here. LVMINI uses *reverse communication*: the calculation of the objective function and of the gradient is done within the users program, and the minimization program is called from the users program. As an alternative the calculation of the objective function and of the gradient can be done in a user subroutine, which is called from the minimization program (MINUIT style).

The limited memory BFGS algorithms from ref. [1] and [3], described in section 5, are used, with a line-search, using a slightly modified version of the code from ref. [4]. The main differences to the L-BFGS code are:

convergence recognition: in the convergence test the function is assumed to be a (negative log-) maximum-likelihood function. Instead of the value of the gradient norm $|\nabla F|$, the estimated distance $\mathbf{d}^T \nabla F$ to the minimum and the actual function change ΔF is used for the recognition of convergence.

round-off error recognition: the precision of the objective function in statistical applications is often limited by round-off errors, which cause problems in the recognition of convergence in the line-search; a special test for round-off errors is introduced, in order to avoid unnecessary function evaluations.

covariance matrix calculation: after convergence of the minimization the covariance matrix is calculated numerically from the gradients.

parameter scaling: without scaling the minimum is often only reached approximately, if only the gradient is calculated, and if parameters are of very different magnitude and precision. Scale parameters are automatically determined for each parameter, which improve the factor γ_k (equation (16)). No scaling is done if the diagonal elements of the Hessian are calculated.

restart option: an automatic restart is added after an error condition in the line-search (e.g. maximum number of function evaluations per iteration reached)

NaN recognition: the line-search program has been modified to recognize function values like NaN (not-a-number) or infinity; these not acceptable values are possible for illegal parameter values during minimization like a negative width-parameter.

Error estimates. A rough estimate of the errors can perhaps be calculated by the following simple method. The two-loop recursive algorithm from section 5 can be used to obtain directly an estimate of the j -th column vector of the covariance matrix by

$$\mathbf{b}_j = \mathbf{B}\mathbf{e}_j, \quad (18)$$

where \mathbf{e}_j is the j -th unit vector (j -th column of the unit matrix). Test have shown that at least the j -th element, which is the j -th diagonal element of the matrix \mathbf{B} , is an order-of-magnitude estimate of the variance of the j -th parameter, i.e. the square root is an estimate of the standard deviation of the j -th parameter. This value is printed in the standard printout of subroutine LVMINI². This estimate of the standard deviation is used as step size h in the numerical Hessian calculation below.

Covariance matrix calculation. The calculation of the covariance matrix is done after convergence by a numerical method. In order to get the j -th column, \mathbf{c}_j , of the Hessian \mathbf{C} , two function and gradient evaluations are done for $\mathbf{x} = \mathbf{x}^* \pm h\mathbf{e}_j$, where \mathbf{x}^* is the found minimum point of $F(\mathbf{x})$ and \mathbf{e}_j is the j -th unit vector:

$$\mathbf{g}_+ = \mathbf{g}(\mathbf{x}^* + h\mathbf{e}_j) \quad \mathbf{g}_- = \mathbf{g}(\mathbf{x}^* - h\mathbf{e}_j). \quad (19)$$

The j -th column vector of the Hessian \mathbf{C} is then calculated from the difference of the gradient vectors by the central formula

$$\mathbf{c}_j = \frac{\mathbf{g}_+ - \mathbf{g}_-}{2h}. \quad (20)$$

The complete matrix \mathbf{C} is inverted to get the covariance matrix $\mathbf{V} = \mathbf{C}^{-1}$. This method is applicable, if the number of parameters n is not big.

Parameter scaling. Parameters are often of very different magnitude and precision, and this may affect the performance of algorithms using gradients only. Scaling improves the efficiency of the algorithm, if the factor γ_k according to equation (16) is used. A scaling factor c_j is determined for each component by the average of the quantities

$$c_j = \frac{1 + |F|}{|\partial F / \partial x_j|}, \quad (21)$$

²Experience should show, whether this rough estimate, which is easily calculated, is in fact realistic.

where the common factor $(1 + |F|)$ for all components takes into account the different magnitude of the first derivative in parameter space. The average is formed before the first pair $\{\mathbf{s}, \mathbf{y}\}$ is stored. Internally the components of the gradient are multiplied by the scale factors c_j , and the components of the parameter vector are divided. No scaling is necessary if the diagonal elements of the second derivatives are provided.

nr of parameters n	nr of vector pairs m	dim. parameter (1) (no cov. matrix)	dim. parameter (2) (with cov. matrix)
10	5	181	256
100	17	4 135	9 385
1 000	29	65 059	567 559
10 000	29	650 059	50 675 059
100 000	29	6 500 059	711 782 763

Table 1: Space requirement for different number of parameters. The column dim. parameter (1) gives the required dimension parameter of the double-precision array `AUX(.)`, if no covariance matrix is requested, and the column dim. parameter (2) gives the number including the covariance matrix; the latter case requires of course also additional function evaluations.

Work space. A double-precision array work-space array is used in the program. The total number MDIM of double precision words for minimization is given by

$$\begin{aligned} \text{MDIM} &= n \times (7 + 2 \times m) + 2 \times m + 1 && \text{minimization} \\ \text{MDIM} &= n \times (9 + 2 \times m) + 2 \times m + 1 + (n \times n + n)/2 && \text{minimization and error calculation} \end{aligned}$$

The last term in the second expression, quadratic in n , is the space to store the symmetric covariance matrix; this term will dominate for larger values of n .

The Table 1 gives for certain number of parameters the necessary dimension of the double-precision array `AUX(.)` for a certain assumption of the number of vector pairs in the limited memory BFGS update. According to the literature larger values of the number m do in general not improve the convergence. As can be seen 6 500 059 double-precision words or 52 Mbyte are sufficient for an optimization problem with 100 000 parameters (without covariance matrix). The space increases quadratically with n , if the covariance matrix is requested, and also the cpu time will be large.

Iterations. The first function (and gradient) evaluation at the start values of the parameters is called Iteration 0. The search direction in iteration 1 is the negative gradient direction:

$$\mathbf{d} = -\frac{1}{\sqrt{\mathbf{g}^T \mathbf{g}}} \mathbf{g} \quad \text{with} \quad \mathbf{g} = \nabla F. \quad (22)$$

A line-search is performed from the start values of the parameters with search vector \mathbf{d} . At the end of iteration 1 the difference vectors (here $k = 1$)

$$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k \quad \mathbf{y}_k = \nabla F_{k+1} - \nabla F_k$$

are evaluated the first time, and are available for the next iterations.

In the following iterations the limited-memory algorithm is used for the calculation of the step vector. In each iteration a line-search is performed, starting at the best (lowest) function value of the previous iteration. The search vector \mathbf{d} is calculated by the two-loop recursion. The approximate inverse Hessian is equal to the scaled unit matrix

$$B_k^0 \equiv \gamma \mathbf{I} \quad \gamma = \frac{\mathbf{y}^T \mathbf{s}}{\mathbf{y}^T \mathbf{y}}, \quad (23)$$

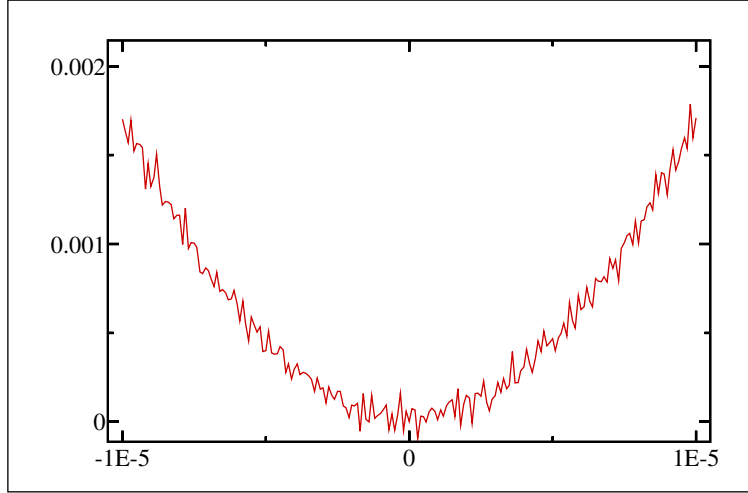


Figure 1: The non-smooth behavior of the objective function in a straight-line fit, shown as a function of (small) variations of the slope-parameter

if no diagonal matrix is provided by the user, and

$$B_k^0 \equiv D^{-1}, \quad (24)$$

if the matrix D is provided; the matrix D is the diagonal matrix of the diagonal elements of the Hessian C . The initial factor α is usually taken as $\alpha = 1$; often the line-search is finished already with the first function evaluation at $\mathbf{x} + 1 \cdot \mathbf{d}$ with a sufficient decrease of the function. The expected function decrease in the iteration (in the ideal case) can be calculated by

$$\Delta F_{\text{expected}} = \frac{1}{2} \mathbf{g}^T \mathbf{d}. \quad (25)$$

This can be compared with the actual function decrease

$$\Delta F_{\text{actual}} = F_{k-1} - F_k. \quad (26)$$

The ratio of ΔF_{actual} to $\Delta F_{\text{expected}}$ should be around the value one near the function minimum. The initial factor α in the next iteration is taken as

$$\alpha = \min \left(1, 1.01 \cdot \frac{\Delta F_{\text{actual}}}{\Delta F_{\text{expected}}} \right), \quad (27)$$

usually $\alpha = 1$.

Convergence and round-off error recognition. Due to round-off errors the function F may not be smooth in the neighbourhood of the minimum of the function, where the gradient ∇F is small. This non-smooth behavior is demonstrated in Figure 1, and it can create problems in the convergence phase of a minimization program. A short part of the code used to produce the dependence in the figure is

```
DO I=1,100
  FIT=PAR(1)+PAR(2)*X(I)
  WT=1.0/DY(I)**2
  ...
  FSUM=FSUM+WT*(FIT-Y(I))**2
END DO
```

In this code `FSUM` the function value is in double precision, and the `PAR`-array is double precision too, while all other variable are in single precision, including the variable `FIT`. If now the variable

FIT is declared as `DOUBLE PRECISION`, the function dependence becomes smooth. This example shows that as far as possible the calculation of the function values has to be done with high accuracy, i.e. with `DOUBLE PRECISION` variables.

The function difference between two points in the parameter space is especially affected by the round-off errors and the first (sufficient decrease) inequality (13) of the line-search conditions can not be satisfied³. The (analytic) derivative is less affected by round-off errors, and the second (curvature condition) inequality (14) is often satisfied already at the standard first step with $\alpha = 1$.

In order to detect large round-off errors, the mean value and the standard deviation of the function-value difference to the start value are determined during the line-search, and the largest of the two (absolute) values, called ΔF_{rd} , is determined. A typical value of ΔF_{rd} due to round-off is below 10^{-3} ; for an objective function in statistical applications this value means insignificant function differences. The strategy is to ignore in the line-search test the first inequality (13) for values $\Delta F_{\text{rd}} < 10^{-3}$.

Convergence recognition is done using the function differences mentioned above:

$$\Delta F_{\text{max}} < \epsilon \quad \text{with} \quad \Delta F_{\text{max}} = \max(\Delta F_{\text{expected}}, \Delta F_{\text{actual}}) . \quad (28)$$

The default value of ϵ is 10^{-4} , adequate to objective functions in statistical applications.

7 Numerical results

A few numerical results are shown here from `LVMINI`; some results are compared to results from the standard and well-documented fit program `MINUIT` from CERN.

`LVMINI` follows a single algorithm, the limited memory version of the BFGS update method for the inverse of the Hessian, combined with a line-search algorithm with the strong Wolfe conditions. The function value and the gradient vector have to be calculated in the user code. Scaling factors to compensate for the different order of magnitude and precision of different parameters are determined during the first function evaluations. The user has to specify the number of vector pairs, and may also require the calculation of the covariance matrix (which takes a lot of space and extra cpu time); there are no further options. There is an attempt to estimate the parameter errors without extra cpu time; the reliability of these estimates is not yet known.

`MINUIT` is the standard fit program[5] for difficult problems in High Energy Physics. From the Foreword: “It is especially suited to handle difficult problems, including those which may require guidance to find the correct solution.” `MINUIT` offers the user a choice of several minimization algorithms and it offers several tools to analyze the parameter errors. It allows several quality levels of strategies. The most efficient and complete single method `MIGRAD`, recommended for general functions, is a variable-metric method with inexact line-searches; thus nominally it is almost identical in the first iterations to the `LVMINI` method. The option `MINIMIZE` in `MINUIT` is identical to `MIGRAD`, but switches to the `SIMPLEX` minimization if `MIGRAD` fails and then back to `MIGRAD`. By default first derivatives are calculated numerically. This feature simplifies the application considerably. Optionally the user can supply e.g. analytical derivatives, but this is usually not done, and in fact Minuit does not even make the best use of analytical first derivatives when they are supplied by the user. Accurate numerical differentiation is difficult and takes cpu time, because the function value has to be determined several times for a single derivative, and the derivative value can be affected by round-off errors. For large scale optimization problems analytical derivatives seem to be essential. `MINUIT` performs a lot of checks during minimization and is able to detect bugs in the user code.

There was a limit of 50 (variable parameters) in the F77 version. The C++ version has no limit on the number of parameters: “There is no protection against an upper limit on the number of

³The numerical calculation of the first derivative of the function F becomes rather difficult, due to the round-off errors, which may introduce an error larger than the derivative itself.

parameters, however the “technological” limitations of MINUIT can be seen around a maximum of 15 free parameters at a time.” The F77 version (Release 96.03) is used in the comparison.

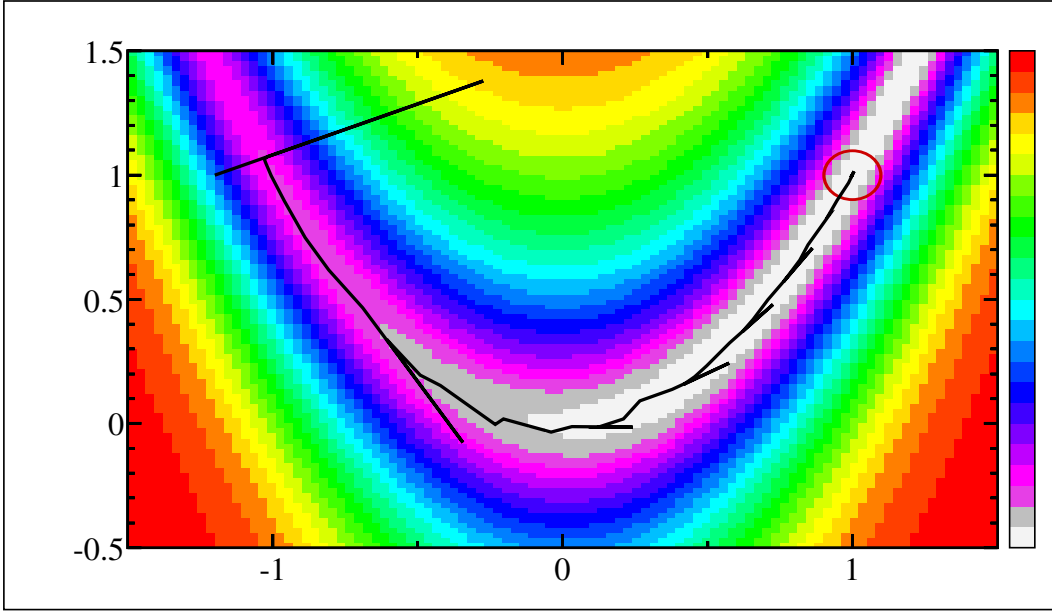


Figure 2: The contour of the Rosenbrock function and the minimization path of the LVMINI program for the standard start point to the minimum at the point (1, 1).

Method	$\mathbf{x}_0 = (-1, 1.2)$	$\mathbf{x}_0 \in \{-10 \dots +10\}$
Steepest descent	> 5000	
LVMINI	61	25 ... 129 71 ± 22
MINUIT, gradient	115	40 ... 252 144 ± 53
MINUIT, no gradient	211	68 ... 504 286 ± 96

Table 2: Number of function evaluations in the minimization of the Rosenbrock function. The last column show the results for 1000 random start positions with both components x_1 and x_2 uniform $\in \{-10 \dots +10\}$. The final function value is in all cases $< 10^{-8}$, except for the method of steepest descent, which requires an extremely large number of function evaluations.

Rosenbrock function

A standard mathematical function for test of minimization methods is the 2-parameter Rosenbrock function. The Rosenbrock function

$$F(x) = 100(x_2 - x_1^2) + (1 - x_1)^2$$

has a unique minimum with function value of zero at the point $x_1 = 1, x_2 = 1$ at the base of a banana-shaped valley. Standard start point usually used in tests is $x_1 = -1.2, x_2 = 1.0$. The Figure 2 shows the contours of the function and the minimization path of the LVMINI program from the standard start point to the minimum at the point (1, 1). Results from the minimization are shown in Table 2. It should be noted, that the behaviour of this mathematical function is not typical for a fit problem.

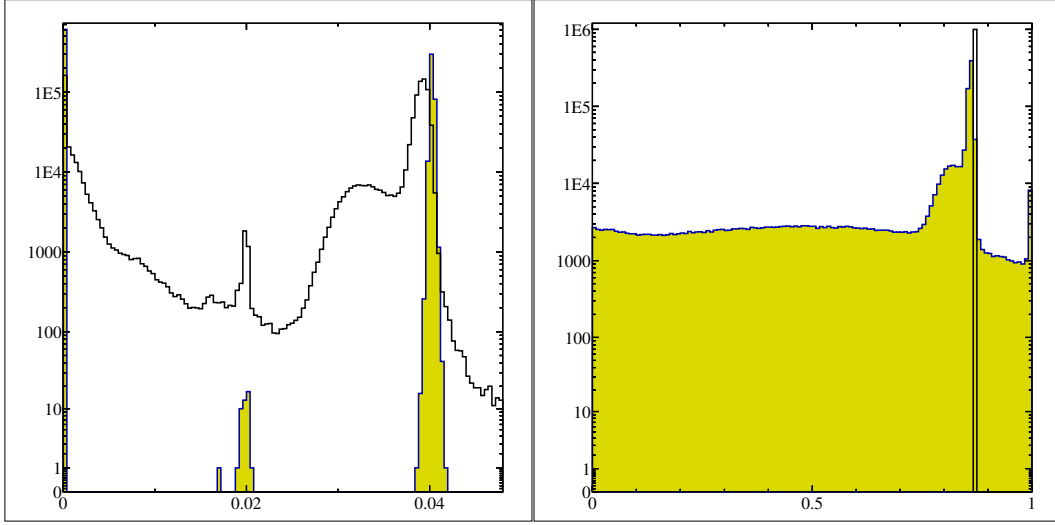


Figure 3: Results from the straight-line fits. The left figure shows the parabolic error after the MINIMIZE option as open histogram, and shaded (yellow) the parabolic error after the HESSE and MINOS options. The error is in fact always 0.040302 (LVMINI result). The right figure shows the MINUIT result for the global correlation coefficient after the HESSE and MINOS options; the narrow line corresponds to the LVMINI result $\rho = 0.868$.

Straight-line fit

Least square straight-line fits and polynomials fits are linear problems, which should in practice not be performed with programs like LVMINI and MINUIT; correlations between parameters are often quite large. Especially the construction of orthogonal polynomials with the Forsythe recurrence relation is a superior method, which automatically includes efficient statistical tests for the necessary degree in polynomial fits. Nevertheless one can test the performance of the LVMINI and MINUIT with such a problem.

A straight line $y = a_1 + a_2x$ is assumed with parameter values $a_1 = 1.0$ and $a_2 = 1.0$ for 100 x -values $x_i = i$. Measured values are simulated by MC with a standard deviation $\sigma_i = 0.2$ and $\sigma_i = 0.1$; deviations to the ideal y -values are simulated using the Gaussian distribution. 10^6 measurements are simulated and fitted, using as start values $a_1 = 0$ and $a_2 = 0$. The results are shown in Table 3.

The calculation of the diagonal of the Hesssian in addition results in a small improvement for LVMINI. The fits with supplied gradient can be directly compared between LVMINI and MINUIT. Since the ideal straight-line is identical for all cases, the parameter errors should be almost identical for all cases; this is the case for LVMINI, also for the global correlation coefficient, that has almost the same value of $\rho_{\text{global}} = 86.8\%$ in all cases. In the MINUIT minimization with option MINIMIZE however the errors and the correlation coefficients are of quite different magnitude. Therefore the MINUIT minimization has been repeated with the sequence MINIMIZE HESSE MINOS of options, which requires some additional function evaluations; results are shon in Figure 3.

Gaussian histogram peak

Using random numbers histograms with a Gaussian peak are generated. The histograms are fitted with a 4-parameter function

$$F(x; \mathbf{a}) = a_1 a_3 \frac{\Delta x}{\sqrt{2\pi}} \exp \left\{ -\frac{1}{2} a_3^2 (x - a_2)^2 \right\} + a_4 ,$$

Method	straight-line fit ($\sigma = 0.2$)	straight-line fit ($\sigma = 0.1$)
LVMINI, gradient, diagonal	13.7 ± 2.6 11 ... 26	14.2 ± 2.9 11 ... 27
LVMINI, gradient	15.8 ± 3.2 11 ... 26	16.1 ± 3.2 11 ... 28
MINUIT, gradient	69.7 ± 36.5 30 ... 190	84.1 ± 37.2 34 ... 197
MINUIT, no gradient	131.2 ± 27.8 38 ... 315	127.6 ± 27.2 42 ... 267
MINUIT, gradient, Minos	159.4 ± 42.3 68 ... 512	177.9 ± 43.8 70 ... 547
MINUIT, no gradient, Minos	392.1 ± 79.9 105 ... 4294	402.5 ± 80.2 133 ... 4233

Table 3: The properties of the minimization programs, from 10^6 simulated straight lines.

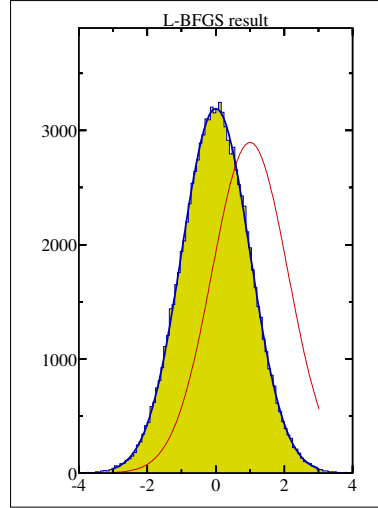


Figure 4: Histogramm of a Gaussian peak. The red curve corresponds to the (bad) initial parameter values, with the peak position displaced by one standard deviation. The fitted curve is also shown.

where a_1 = number of entries in the peak, a_2 = mean value, $a_3 = 1/\text{standard deviation } \sigma$, and a_4 is term describing a (constant) background. In the fit the function is averaged over the histogram bin Δx . Start values for the fit are from a simple mean value calculation, with mean value shifted by 1 standard deviation, to simulate bad start conditions. Figure 4 shows the histogram, the curve corresponding to the initial values and the fitted curve. Average values for the number of function evaluations are given in Table 4.

The errors calculated by LVMINI are as accurate as MINOS errors (in the MINOS option negative and positive errors are determined by an analysis of the shape of the objective function; in this example the two values differ by less than a % and the average agrees with the LVMINI error). The Table 5 shows a detailed comparison of the calculated errors from LVMINI and MINUIT.

	LVMINI	MINUIT	MINUIT-MINOS
no gradient	-	170	638
gradient	14 + 8	118	289
+ diagonal	13 + 8		

Table 4: The number of function evaluations for the Gaussian histogram peak fit. In LVMINI 8 function evaluations are necessary for the error calculation. The first MINUIT column is related to the `minimize` option, where the errors are not yet precise. The second MINUIT column includes the MINOS option for the detailed error analysis.

	LVMINI	MINUIT-Migrad	Hesse,Minos	MINUIT-Migrad	Hesse,Minos
no gradient				160	656
gradient	15 + 8	106	213		
+ diagonal	13 + 8				
error a_1	402.47	403.92	402.47	425.06	402.48
error a_2	$10^{-3} \times 3.220$	$10^{-3} \times 2.999$	$10^{-3} \times 3.217$	$10^{-3} \times 3.385$	$10^{-3} \times 3.217$
error a_3	$10^{-3} \times 3.743$	$10^{-3} \times 3.749$	$10^{-3} \times 3.743$	$10^{-3} \times 2.971$	$10^{-3} \times 3.743$
error a_4	3.189	3.196	3.189	2.977	3.189

Table 5: Number of function evaluations and results for parameters errors for the fit of a Gaussian. The error values in the column `Hesse,Minos` are the mean values of the positive and negative `Minos` error, which differ by less than 0.1 %.

LVMINI printout

Below the printout of the program LVMINI is shown for one example of the fit of a Gaussian histogram peak. The different parts of the printout commented here can also be printed separately after convergence using the sunroutine `LVMFRT`.

The first part shows the initialization parameters before the first function evaluation. The necessary dimension of the AUX array is printed (this has to be checked by the user with the call `LVMDIM` before). The initial parameter values are printed

```
Limited-memory BFGS routine LVMINI initialized=====
with N = 4  dimension parameter
  M = 4  number of history pairs
  NF =      224  max number of function calls
  eps = 0.100E-03  convergence parameter
  c_1 = 0.100E-03  first Wolfe parameter
  c_2 = 0.900      second Wolfe parameter
Diagonal of Hessian provided
Requires AUX array with at least 87 words

Initial parameter values:
par_i      value      par_i      value      par_i      value
  1      100993.        2      1.00062        3      0.898039
  4       0.00000
```

The table below gives information about the iteration until convergence. `Iter` is the iteration number, `Nfcn` the number of function evaluations and `Npair` the number of stored pairs, together with the function value `Fcn value`. `dFex` is the expected function change during the iteration, and `dFac/dFex` is the ratio of the actual function change to the expected function change. This ratio should be close to one, at least in later iterations. A negative value of `dFac/dFex` indicates a worse function value. The value of the factor α in the line-search is printed under `fstep`. The search direction often has a large angle close to 90° to the negative gradient direction (direction of steepest descent). The difference to 90° is printed under `degr`. An angle of exactly 90° does

not correspond to an ascent direction and has to be avoided. If the angle is below 0.0001° , the search direction is modified to a larger angle.

Iter	Nfcn	Npair	Fcn_value	dFex	dFac/dFex	fstep	degr
0	1	0	73771.8229				
1	2	0	2938.3622	0.91E+05	0.777	1.000	
2	3	1	2973.7664	0.48E+04	-0.007	1.000	0.0022
	4	1	166.2770		0.577	0.517	
3	5	2	97.9994	0.51E+02	1.329	1.000	0.0024
4	6	3	52.6833	0.39E+02	1.163	1.000	0.0056
5	7	4	44.4320	0.23E+02	0.366	1.000	0.0014
6	8	4	33.3587	0.15E+02	0.733	0.551	0.0055
7	9	4	31.4648	0.25E+01	0.769	1.000	0.0011
8	10	4	31.2924	0.25E+00	0.690	1.000	0.0013
9	11	4	31.2640	0.31E-01	0.911	1.000	0.0016
10	12	4	31.2633	0.65E-03	1.048	1.000	0.0015
11	13	4	31.2632	0.61E-04	1.485	1.000	0.0015

The two function values of the calculation of the covarinace matrix with numerical methods are printed, if the covariance matrix is requested. The calculation needs $2n$ additional function evaluations. The final function value is printed with the total number of function evaluations.

derivative calculation:

par_i	step(+/-)	secder	dF_minus	dF_plus
1	0.42E+03	0.10E-04	0.89E+00	0.88E+00
2	0.33E-02	0.97E+05	0.51E+00	0.52E+00
3	0.36E-02	0.17E+06	0.12E+01	0.11E+01
4	3.2	0.29E+00	0.15E+01	0.15E+01

4-parameter function value = 31.2631986 after 21 function evaluations

Approximate errors can be calculated directly from the stored pairs $\{\mathbf{s}, \mathbf{y}\}$; ist is not clear, wether these values, which require no extra function evaluations, are meaningful.

Optimization results with approximate errors:

par_i	value	error	par_i	value	error
1	99661.9	+- 420.	2	0.104162E-02	+- 0.325E-02
3	1.00176	+- 0.364E-02	4	3.37565	+- 3.23

The printout below gives the final parameter values together with the errors from the numerical calculation, the global correlation coefficients (maximum correlation to linear combinations of all other parameters) and finally the matrix of correlation parameters between parameter pairs. The errors from the numerical calculation seem to be very reliable.

Optimization results:

par_i	value	error	glcorr
1	99661.9	+- 402.471	0.618
2	0.104162E-02	+- 0.321740E-02	0.004
3	1.00176	+- 0.374300E-02	0.765
4	3.37565	+- 3.18893	0.816

Correlation coefficients [per mille] between parameters i and j:

par_i \ j=	1	2	3	4
1	1.000			
2	0.000	1.000		
3	-0.483	0.003	1.000	
4	-0.618	0.000	0.765	1.000

Limited-memory BFGS routine ending =====

Five-Peak histogram

An example with 20 fitted parameters is given by the fit of five peaks plus linear background to the histogram shown in Figure 5. The shape of the peaks is non-Gaussian and Students density

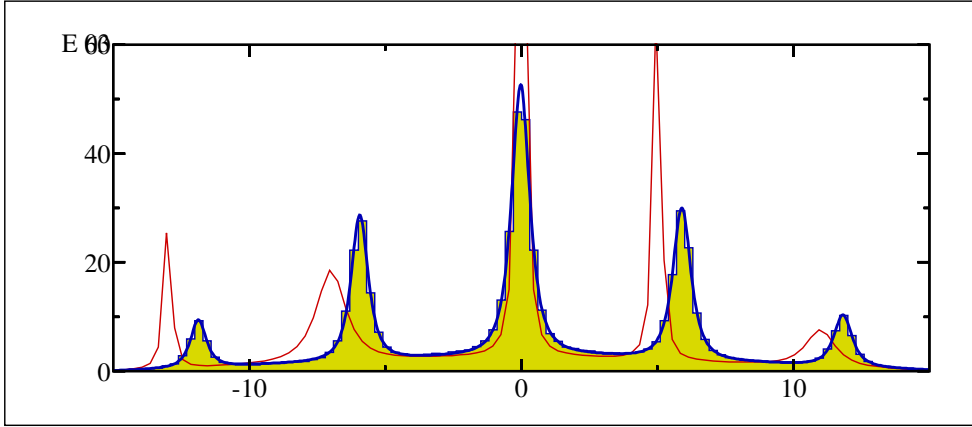


Figure 5: Histogram and fit of five peaks. The peaks are described by Students distribution.

Initial values		LVMINI
good	gradient	233 + 40
	+ diagonal	28 + 40
bad	gradient	257 + 40
	+ diagonal	68 + 40

Table 6: Number of function evaluations in LVMINI for the histogram of Figure 5. The curve corresponding to the initial values labelled as *bad* is shown as a red curve in Figure 5.

has been used to describe the peaks with a fitted value of the number of degrees of freedom. The result given in Table 6 shows that information about the diagonal elements of the Hessian improves the performance of the algorithm.

Numerical experience

The tests of LVMINI with a small number (up to 20) of parameters shows, that the algorithm is rather efficient. The calculation of the diagonal elements of the Hessian, which for the two objective functions discussed in the first section is rather simple, improves the performance sometimes by about 10 %, sometimes by a large factor. Therefore this additional calculation is recommended. The error calculation is rather reliable according to the few tests made. The convergence behavior in the case of bad initial parameter values seems to be not bad.

Tests with a large number of parameters are still in progress. An alignment fit with 200 parameters and a neural net training with 280 parameters worked well. The program seems to work also for $n =$ several 100 000 parameters, but the test will continue.

References

- [1] J. Nocedal and S.J. Wright, *Numerical Optimization*, Springer Series in Operations Research, Springer (1999)
- [2] W.C. Davidon, *Variable metric method for minimization*, manuscript (1958), finally published SIAM J. Optimization **1** (1991) pp. 1-17.
- [3] J. Nocedal, *Updating quasi-Newton matrices with limited storage*, Mathematics of Computation **35** (1980) pp.773-782
- [4] J.J.Moré and D.J. Thuente, *Line search algorithms with guaranteed sufficient decrease*, ACM Transactions on Mathematical Software **20** (1994), pp. 286 – 307
- [5] F. James and M. Roos, MINUIT, Function Minimization and Error Analysis, Reference Manual, CERN Program Library Long Writeup D506 (1994)
F. James and M. Winkler, MINUIT Users Guide (C++ Version), CERN (2004)
- [6] D.C. Liu and J. Nocedal, *On the Limited memory BFGS Method for Large Scale Optimization*, Mathematical Programming **45**, pp. 503 – 528

1. The auxiliary array AUX

The optimization program for minimization requires, in addition to the double-precision parameter vector \mathbf{x} , a *double-precision* auxiliary array `AUX(.)` of a size, which, for minimization, depends only linearly on the number of parameters n of the optimization problem. Additional words ($\propto n^2$), are required, if an error calculation is required after minimization. The array size depends also on the number m of stored vector pairs $\{s_k, y_k\}$. Recommended values of m are in the range $6 \dots 29$ (with $m \leq n$).

The function `LVMDIM(NPAR,MVEC)` returns the required dimension parameter `MDIM` of the auxiliary array `AUX(.)` of double precision words. The array `AUX(.)` is one argument in the minimization calls; after optimization it will contain the results.

`MDIM = LVMDIM(NPAR,MVEC)`

Purpose: calculation of the required number `MDIM` of double-precision words of the auxiliary array `AUX(.)`.

NPAR : integer, specifying the number of parameters: $n \equiv \text{NPAR}$

MVEC : integer, specifying the number of update vectors to be used in the BFGS updates: $m \equiv \text{MVEC}$. Recommended is a number with $6 \leq \text{MVEC} \leq 29$. The argument `MVEC` has to be given with a minus sign, if calculation of the covariance matrix is required.

2. Constants of the algorithm

Three constants are used in the algorithm, which can be changed by the user. The default values should be applicable in most cases.

ϵ : the constant used to recognize convergence; convergence is assumed, if the estimated and the actual decrease of the objective function is less than ϵ ; the default value is 10^{-4} ;

c_1, c_2 : the two constants of the strong Wolfe condition in the line search; the default values are $c_1 = 10^{-4}$ and $c_2 = 0.9$.

`CALL LVMEPS(EPS,WLF1,WLF2)`

Purpose: optional to assign new or default values to the constants of the algorithm. For an argument ≤ 0 the default value is stored. The values of the constants remain valid until a change by this call.

EPS : single precision floating point number, specifying the constant ϵ .

WLF1 : single precision floating point number, specifying the factor c_1 of the strong Wolfe conditions;

WLF2 : single precision floating point number, specifying the factor c_2 of the strong Wolfe conditions.

3. Minimization

The minimization of the objective function requires the repeated function evaluation and gradient calculation. In the standard mode these calculations are done in the user program and environment (an alternative is described in the next section).

The following *double-precision* variables and arrays have to be declared:

F : value of the objective function
X(NPAR) : array of parameter values
AUX(NAUX) : auxiliary array

Minimization is initialized by a call to LVMIN with specification of the necessary minimization parameters and constants.

CALL LVMINI(NPAR,MVEC,NFCN, AUX)

Purpose: initialization of the minimization routine

NPAR : integer, specifying the number of parameters: $n \equiv \text{NPAR}$. By default the minimization will run without any printout. If printout is wanted, a minus sign has to be inserted before NPAR.

MVEC : integer, specifying the number of update vectors to be used in the BFGS updates: $m \equiv \text{MVEC}$. Recommended is a number with $6 \leq \text{MVEC} \leq 29$ (with $m \leq n$). By default only minimization is done without extra error calculation. If error calculation is required, a minus sign has to be inserted before MVEC. Error calculation requires a larger dimension of the auxiliary array AUX(.).

NFCN : integer, the allowed maximum number of function evaluations. The program will stop before convergence if this number is reached. A default value is used if this argument is zero.

AUX : double-precision array AUX(NAUX) of sufficient length.

Initial parameter values have to assigned to the elements of the *double-precision* parameter vector X(NPAR). Neither step sizes nor further information is necessary (certain scale factors are determined automatically during minimization).

The minimization is done, in a loop, by repeated calculation of the *double-precision* function value F of the objective function and of the derivatives in the *double-precision* AUX, for the parameter X(.) values returned by the subroutine LVMFUN. A code fragment is shown below (see also the appendix).

```

CALL LVMINI(NPAR,MVEC,NFCN, AUX)
X(1)      = ... (initial value)
...
X(NPAR) = ... (initial value)

*   calculation of F and the gradient in a loop
10  F          = ... value of the objective function
*           gradient of the objective function:
AUX(1)      = ... value of first derivative of F w.r.t. X(1)
...
AUX(NPAR) = ... value of first derivative of F w.r.t. X(NPAR)

*   optional:
AUX(NPAR+1) = ... value of second derivative of F w.r.t. X(1)
...
AUX(NPAR+NPAR) = ... value of second derivative of F w.r.t. X(NPAR)

*
CALL LVMFUN(X,F,IRET,AUX)
IF(IRET.LT.0) GOTO 10

```

In addition to the gradient vector the user may calculate the diagonal elements of the second derivative matrix and store the values in the array elements AUX(NPAR+1) ...AUX(NPAR+I) ...AUX(NPAR+NPAR). As explained below this calculation is often rather simple and can improve the minimization. The program will automatically detect the presence of these values and will use them in the minimization. Elements AUX(1) to AUX(NPAR+NPAR) are reset to zero at the return of subroutine LVMFUN.

CALL LVMFUN(X,F,IRET,AUX)

Purpose: preparation of the next point in parameter space and recognition of convergence; called after function and gradient calculation.

X : array of parameter values (not to be changed in the user program)

F : value of the objective function; has to be calculated in the user program for $IRET < 0$.

IRET : integer return flag; the loop has to be continued for $IRET < 0$. Values $IRET \geq 0$ mean successful minimization or error.

AUX : double-precision array **AUX(NAUX)**. The user has to calculate the gradient and can, optionally, calculate the diagonal elements of the second derivative matrix. Derivatives have to be stored in the array **AUX(NAUX)**.

For $IRET \geq 0$ the loop is left.

4. Access to the result

IND = LVMIND(IARG)

Purpose: all results are after minimization in the auxiliary array **AUX(NAUX)**. The returned integer can be used to access the information in this array.

IARG=-1: the number of parameters **NPARG**;

IARG=-2: the number of function evaluations;

IARG= 0: index of the final function value: **AUX(IND)**;

IARG= 1: index of the final parameter values: **AUX(IND+I)**, $I=1, NPARG$;

IARG= 2: index of the approximate parameter errors: **AUX(IND+I)**, $I=1, NPARG$. these errors may only be correct in the order of magnitude. They are calculated without extra function evaluations and are available without explicit error calculation.

IARG= 3: index of the parameter errors (or zero): **AUX(IND+I)**, $I=1, NPARG$.

IARG= 4: index of the global correlation coefficients (or zero): **AUX(IND+I)**, $I=1, NPARG$.

IARG= 5: index of the covariance matrix (or zero): **AUX(IND+I)**, $I=1, (NPARG*NPARG+NPARG)/2$.

5. Printout

The minimization in **LVMINI/LVMFUN** is by default done silently (without any printout). Printout to the standard print unit can be switched on by inserting a minus sign in front of **NPARG** in the initialization call **LVMINI**. After the minimization the result can be printed from the array **AUX** by the following call.

CALL LVMPRT(LUP,AUX,IARG)

Purpose: The required information from the minimization is printed on unit **LUP**.

LUP : unit for printing (values 0 and 6 are standard print unit).

AUX : auxiliary array

IARG= 0: the function value is printed.

IARG= 1: parameter values are printed.

IARG= 2: parameter values and approximate errors are printed.

IARG= 3: parameter values and errors are printed.

IARG= 4: parameter values, global correlations and errors are printed.

IARG= 5: the matrix of correlation coefficients is printed (from the covariance matrix).

IARG= 6: information from arguments 0, 2 4 and 5 is printed.

6. Alternative: Minimization with a MINUIT-like function

The minimization system MINUIT requires to supply a function for the calculation of the objective function. Optionally MINUIT accepts the calculation of the gradient vector. Minimization with the limited memory BFGS method can alternatively done with a MINUIT-like function, if it allows the calculation of the gradient vector.

CALL LVMIDI(NPAR,X,FUNCT, AUX)

Purpose: minimization with repeated calls of the function FUNCT, which has to be declared as EXTERNAL in the calling program. A small number NPAR of parameters is assumed and the parameter MVEC is specified automatically. Error calculation is requested automatically.

NPAR : integer, specifying the number of parameters: $n \equiv \text{NPAR}$. By default the minimization will run without any printout. If printout is wanted, a minus sign has to be inserted before NPAR.

X : double-precision array of parameter values (not to be changed in the user program)

FUNCT: MINUIT-like function, called always with argument =2 (i.e. gradient calculation required); see MINUIT manual.

AUX : double-precision array AUX(NAUX)

The complete code of this subroutine is show below.

```
      SUBROUTINE LVMIDI(NPAR,X,FUNCT, AUX)
      DOUBLE PRECISION X(*),FSUM,AUX(*)
      SAVE
      EXTERNAL FUNCT
      *
      ...
      MVEC=MIN(ABS(NPAR),6)           ! number of delta pairs
      CALL LVMINI(NPAR,-MVEC,0, AUX)   ! initialization
      *
      function and gradient evaluation
      10 CALL FUNCT(ABS(NPAR),AUX,FSUM,X,2) ! last arg always 2
      CALL LVMFUN(X,FSUM,IRET,AUX)
      IF(IRET.LT.0)      GOTO 10
      END
```

7. Example: Straight line least-squares fit

The code for a straight-line least-squares fit with LVMINI is shown below. It is assumed hat the data are stored in the common SLDATA.

```

PARAMETER (NAUX=30)
DOUBLE PRECISION F,P(2),AUX(NAUX) ! parameter array is P(.)
* data for the straight-line fit are in the common /SLDATA/
COMMON/SLDATA/NDATA,X(100),Y(100),DY(100)
DOUBLE PRECISION DER(2) ! for derivatives of fit fuunction
NPAR=2
MVEC=-2
* test on array size
IF(LVMDIM(-NPAR,MVEC).GT.NAUX) STOP 'Array too small'
* initialization
CALL LVMINI(NPAR,MVEC,0, AUX)
* start values for parameters
P(1)=0.0D0
P(2)=0.0D0
* calculation of F and gradient in a loop
10 F=0.0D0 ! reset F (AUX is reset in LMVMIN/LMVEX)
DO I=1,NDATA
    FIT=P(1)+P(2)*X(I) ! straight-line function value
    DER(1)=1.0D0 ! derivative of FIT wrt P(1)
    DER(2)=X(I) ! derivative of FIT wrt P(2)
    WEIGHT=1.0D0/DY(I)**2 ! least-squares weight
    DO J=1,NPAR
        AUX(J)=AUX(J)+WEIGHT*DER(J)*(FIT-Y(I))
    END DO
    F=F+WEIGHT*((FIT-Y(I))**2)
END DO
F=0.5D0*F
* calculation finished - get next parameter values
CALL LVMFUN(P,F,IRET,AUX)
IF(IRET.LT.0) GOTO 10

```


The LVMINI_{mini} Manual

Explanation of arguments:

NPARG :	the number of parameters (integer): $n \equiv \text{NPARG}$. By default the minimization will run without any printout. If printout is wanted, a minus sign has to be inserted before NPARG.
PAR :	double-precision array of parameter values (not to be changed in the user program)
F :	double-precision value of the objective function; has to be calculated in the user program before CALL LVMFUN.
AUX :	double-precision auxiliary array AUX(NAUX) with sufficient dimension (see =LVMDIM). Before CALL LVMFUN the user has to calculate the gradient components in AUX(1) ... AUX(NPARG); optionally, the user can calculate the diagonal elements of the second derivative matrix in AUX(NPARG+1) ... AUX(NPARG+NPARG). At the end the array contains the results, which can be extracted using indices from IND=LVIND.
FCN :	MINUIT-like subroutine, to be used in the special call LVMIDI. The name of the subroutien has to be declared as external in the calling program.
MVEC :	the number of update vectors (integer) to be used in the BFGS updates: $m \equiv \text{MVEC}$. Recommended is a number with $6 \leq \text{MVEC} \leq 27$. By default only minimization is done without extra error calculation. If error calculation is required, a minus sign has to be inserted before MVEC. Error calculation requires a larger dimension of the auxiliary array AUX(.).
EPS :	single-precision floating point number, specifying the constant ϵ . If EPS=0.0, the default value 10^{-4} is used.
WLF1 :	single-precision floating point number, specifying the factor c_1 of the strong Wolfe conditions; if WLF1=0.0, the default value 10^{-4} is used.
WLF2 :	single-precision floating point number, specifying the factor c_2 of the strong Wolfe conditions; if WLF2=0.0, the default value 0.9 is used.
NFCN :	integer, the allowed maximum number of function evaluations. The program will return if this number is reached before convergence. If NFCN=0, a default value is used: $\text{NFCN} = 100 + 10 \times \text{NPARG}$.
IRET :	integer return flag; the loop has to be continued for $\text{IRET} < 0$. Values $\text{IRET} \geq 0$ mean successful minimization or error.
LUP :	print-unit (integer) for LVMPT.

Coding template:

```

DOUBLE PRECISION F,PAR(.),AUX(NAUX)
NDIM=LVMDIM(NPARG,+MVEC)      ! optional; NDIM =< NAUX
CALL LVMEPS(EPS,WLF1,WLF2) ! optional
PAR(1) = ... (initial value)
...
PAR(NPARG) = ... (initial value)

*           -NPARG      ... for printout during minimization
CALL LVMINI(+NPARG,+MVEC,NFCN, AUX) ! initialization
*           -MVEC      ... for error calculation
* calculation of F and the gradient in a loop
10 F        = ... value of the objective function
*           gradient of the objective function:
AUX(1)     = ... value of first derivative of F w.r.t. PAR(1)
...
AUX(NPARG) = ... value of first derivative of F w.r.t. PAR(NPARG)
* optional:
AUX(NPARG+1) = ... value of second derivative of F w.r.t. PAR(1)
...
AUX(NPARG+NPARG) = ... value of second derivative of F w.r.t. PAR(NPARG)
*
CALL LVMFUN(PAR,F,IRET,AUX) ! next X or finished
IF(IRET.LT.0) GOTO 10

*alternative: replace all lines after PAR(NPARG) = ... by
CALL LVMIDI(+NPARG,PAR,FCN, AUX) ! with Minuit-like function: EXTERNAL FCN

```

Access to result and print-out:

IND = LVMIND(IARG)	IARG	CALL LVMPRT(LUP,AUX,IARG)
-----	----	-----
nr of function evals	-2	
nr of parameters	-1	
index for final function value	0	x function value
index for parameter	1	parameter values
index for appr. errors	2	x parameters and appr. errors
index for errors	3	parameters and errors
index for global correl.	4	x parameters, glob. corr. and errors
index for cov. matrix	5	x correlation coefficients
	6	equiv. 0, 2, 4, 5

Program file. The LVMINI program package and a small test program, a makefile and the manual is in the file `lvmini.tgz`. This file should be stored in a fresh directory, unpacked by

```
tar -xzf lvmini.tgz
```

and the test program is compiled and linked by

```
make
```

and can be executed by

```
./lvtest
```

(Rosenbrock function minimization and two straight-line fits). Corresponding MINUIT code is included, but commented out.