ELSEVIER

Computer Physics Communications 109 (1998) 227–249

Computer Physics Communications

MERLIN–3.0
A multidimensional optimization environment

D.G. Papageorgiou [a], I.N. Demetropoulos [a], I.E. Lagaris [b,1]

[a] *Department of Chemistry, University of Ioannina, P.O. Box 1186, GR 45110 Ioannina, Greece*

[b] *Department of Computer Science, University of Ioannina, P.O. Box 1186, GR 45110 Ioannina, Greece*

Received 8 December 1997

Abstract

We present an optimization environment for multidimensional continuous functions. Robust and powerful algorithms are used that guarantee its effectiveness. The environment offers programmability and ease of use by providing a specialized operating system and a control language that can be used to create successful optimization strategies. We report on several applications where this software has been successfully used. © 1998 Elsevier Science B.V.

Keywords: Optimization; Modeling; Curve-fitting; Neural network training

PROGRAM SUMMARY

Title of program: MERLIN–3.0

Catalogue identifier: ADHQ

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland

Computer for which the program is designed and others on which it is operable:
Computers: Designed to be portable. Developed on a Sun SPARCstation 5. Tested on Sun Classic (SunOS 4.1.3C), Sun Ultra–2 (SunOS 5.5.1), SGI Challenge-M (IRIX 6.2), CD4680 (EP/IX 1.4.3), Intel-based PCs (Linux 2.0.18 and MS-Windows 95), Macintosh (MacOS 7.5); *Installations:* University of Ioannina, Greece

Programming language used: ANSI Fortran-77

Memory required to execute with typical data: Approximately $\max\{n(n+11), nm\} + 4m + 27n + v$ words, where n is the number of variables, m is the number of squared terms, and v is the size of the object code for $n = 1, m = 1$. For the Sun SPARCstation 5, $v = 512$ K.

No. of bits in a word: 32

No. of processors used: 1

Has the code been vectorised or parallelized?: No

No. of bytes in distributed program, including test data, etc.: 1113284

Distribution format: uuencoded compressed tar file

Keywords: optimization, modeling, curve-fitting, neural network training

[1] Corresponding author. Email: lagaris@cs.uoi.gr.

PII S0010-4655(98)00005-8

Nature of physical problem
Many problems in Physics, Chemistry, Engineering and in other disciplines are frequently reduced to minimizing a function of many variables. As examples we refer to systems of nonlinear equations, to modeling, to variational methods, to curve fitting and to the training of neural networks.

Method of solution
MERLIN provides a programmable environment that makes the whole process of minimizing multidimensional functions with bound constraints, flexible and efficient. Ten algorithms and a strategy are implemented. Two of them use only function values, while the rest use gradient information as well. One algorithm is specific for functions that can be cast in a sum of squares form.

Restrictions on the complexity of the problem
The only restriction is set by the available memory of the hardware configuration.

Typical running time
Depending on the objective function. The test run took 1.55 seconds on a Sun SPARCstation 5.

Unusual features of the program
The source code can be customized in regard to the required precision (single or double), to the maximum number of variables and to the maximum number of the squared terms, via a provided installer program. MERLIN can be easily extended by the user through a predesigned plug-in mechanism. Additional documentation is provided in the user manual (147 pages) that accompanies the distributed program.

LONG WRITE-UP

1. Introduction

1.1. What kind of problems MERLIN *handles*

Multidimensional minimization is a common procedure needed in many fields. A variety of problems in engineering, physics, chemistry, etc., are frequently reduced to ones of minimizing a function of many variables. For instance we refer to systems of nonlinear equations, to variational methods, to curve fitting and to the training of neural networks. Minimizing a multidimensional function faces a lot of difficulties. There is no single method that can tackle all problems in a satisfactory way. It has been realized that one needs a strategy, combining different methods, to efficiently handle a wide spectrum of problems. The presence of constraints, even of simple ones, enhances the difficulty. In addition most algorithms require evaluation of the gradient. This imposes additional problems since it is not always straightforward to code it. Hence one resorts to approximating the derivatives using differencing, that costs in computing time as well as in accuracy.

MERLIN is an integrated environment designed to solve optimization problems. It is devised to be easy-to-use, and implemented so as to be portable among different platforms. Another feature is that MERLIN is *open*, i.e. a *plug-in* mechanism is provided so that others can easily embed their own code modules. MERLIN handles the following category of problems:
Find a local minimum of the function

$$f(\boldsymbol{x}), \quad \boldsymbol{x} \in R^N \; \boldsymbol{x} = (x_1, x_2, \ldots, x_N)^{\mathrm{T}}$$

under the conditions

$$x_i \in [l_i, u_i] \quad \text{for } i = 1, 2, \ldots, N.$$

Special merit has been taken for problems where the objective function can be written as a sum of squares, i.e.

$$f(\boldsymbol{x}) = \sum_{i=1}^{M} f_i^2(\boldsymbol{x}).$$

This form is particularly suited when one needs to fit data points using a model function. One then minimizes the chisquare which is of the above form. In this case MERLIN can calculate parabolic estimates of the confidence intervals for the model parameters as well as partial covariance matrices.

MERLIN can be used both interactively and in batch. Interactively the user drives MERLIN by entering commands through the keyboard. In batch MERLIN reads commands from an input command file. Interactively MERLIN is tolerant to errors in input and issues appropriate warning messages, while in batch aborts. There are various commands at the user's disposal that either invoke minimization algorithms or perform other auxiliary operations.

MERLIN is programmable. Its programming language MCL (Merlin Control Language), is a high level, easy to learn language and is described in a separate article [1]. The MCL compiler takes as input a strategy (coded in MCL) and produces as output a file that contains commands that can steer MERLIN appropriately. MERLIN and MCL are both written in ANSI Fortran-77 to guarantee portability.

1.2. Conventions

1.2.1. Typing

- Lower case boldface letters ($\boldsymbol{x}$, $\boldsymbol{g}$, etc.) stand for vectors in the N-dimensional space.
- Upper case boldface letters ($\boldsymbol{G}$, $\boldsymbol{H}$, etc.) stand for $N \times N$ matrices.
- MERLIN commands, Fortran code and I/O data are printed using a monospaced font.
- ⟨cr⟩ denotes keying the "carriage return".
- ... implies that preceding symbols may be repeated.

1.2.2. Symbols

- N, n The number of parameters (dimensionality of the problem).
- $f(\boldsymbol{x})$ The objective function.
- x_i The ith component of $\boldsymbol{x}$.
- $f_k(\boldsymbol{x})$ The kth term entering in the calculation of the objective function when it has the form $\sum_{k=1}^{M} f_k^2(\boldsymbol{x})$.
- M The number of the squared terms involved in the above sum.
- $\boldsymbol{g}$ The gradient vector of the objective function $\nabla f(\boldsymbol{x})$.
- $\boldsymbol{J}$ The Jacobian matrix with $J_{ij} = \partial f_i / \partial x_j$.
- $\boldsymbol{G}$ The Hessian matrix with $G_{ij} = \partial^2 f / \partial x_i \, \partial x_j$.
- $\boldsymbol{B}$ An approximation to $\boldsymbol{G}$.
- $\boldsymbol{H}$ An approximation to $\boldsymbol{G}^{-1}$.
- l_i, u_i The lower and upper bounds for x_i.

2. The MERLIN operating system

MERLIN provides an operating system in order to render the whole process efficient, flexible and programmable. MOS has a command interpreter as a front end that accepts the input commands and instructs MERLIN to take an appropriate action. Like most operating systems, MOS supports command aliasing, as well as argumentary command packages called *macros*. In addition, there is the Merlin Control Language (MCL) through which one can devise intelligent minimization strategies. The MCL compiler is a separate software package built specifically for this purpose.

2.1. Input

The underlying mechanism for input is what we call *one line input*. When a line of input is entered, it is accepted by MOS as a character string. This string is being parsed and lexically analyzed in order to be interpreted. The interpretation of the input line depends on the preceding command, since different commands expect different kind of input. There are commands that require no input, commands that require a simple and rather obvious type of input and commands that require specialized and rather involved type of input. To ease the operation, the commands that require complex input have been implemented via a self-explanatory mechanism, that issues a reminder to the user. This is what is called *panel mechanism*. A command's panel is a table consisting of indexed keywords that correspond to the required input parameters, the current parameter values, a very short description of their function and a menu of possible values. Each panel is followed by a prompt to enter the appropriate input which is to be handled by the one line input mechanism. Panels can be deactivated and reactivated at will, via corresponding MOS commands.

2.2. Output

During operation MERLIN outputs standard information (normal output) and error messages (error output). MOS provides several ways to control the amount of issued output. In addition it supports output redirection to a file via a Unix like syntax. The output of a whole set of consecutive commands can be redirected as well.

2.3. Help

An on-line help facility is available in MERLIN. The user can request help for any MERLIN command or panel parameter. The help texts describe the command's function, its proper syntax and include examples and pointers to related information.

2.4. Commands

MOS has a large repertoire of commands. These can be classified into categories according to their action. In order to present the general idea we will mention the most essential categories. Note that commands can be entered in either lower or upper case invariably.

- Commands that manipulate the attributes of the minimization parameters. The ith minimization parameter has the following attributes:
  - Its current value x_i.
  - A unique symbolic name, up to 10 characters long, that may be set by the user.
  - An indication whether this parameter is fixed. This is referred to as the fix status of the parameter. The minimization routines do not alter the value of fixed parameters.
  - An indication whether there exist lower and/or upper bounds for this parameter. This is referred to as the margin status of the parameter. The minimization routines make sure they never evaluate the function outside the allowed bounds.
  - The lower bound l_i, if it exists.
  - The upper bound u_i, if it exists.

  A few illustrative specific examples may be useful:
  - `point` *index value* ...
    Assigns values to the minimization parameters.
    Example: `point 1 1.2 2 4.7`

- `fix` *index* ...
  Declares that a variable is fixed.
  Example: `fix 1 7 4`
- `loose` *index* ...
  Declares that a variable is no longer fixed.
  Example: `loose 2 3 6`
- `lmargin` *index value* ...
  Sets the lower bounds.
  Example: `lmargin 2 2.7`
- `ldemargin` *index* ...
  Removes the lower bound from a variable.
  Example: `ldemargin 2`
- `godfather` *index name* ...
  Assigns symbolic names to the minimization parameters.
  Example: `godfather 1 rho 2 sigma`
  One can then use these names in place of the parameter indices:
  `point rho 10.1 sigma 0.11`

- Commands that invoke one of the coded minimization algorithms, for example: `simplex`, `congra`, `bfgs`. All these commands are equipped with panels. As an example we list below the input that invokes the BFGS algorithm with a maximum allowed 1000 calls to the objective function.

```
bfgs <cr>
noc 1000 <cr>
```

- Commands that set the options for the various operation modes. These modes determine the overall behavior of MERLIN. To be more specific, there is the *printout* mode that determines the amount of output issued by MERLIN. The printout mode is set by the commands `fullprint`, `halfprint` and `noprint`, corresponding to full output, only error messages and no output at all. As another example consider the *gradient* mode. The gradient mode is set by the commands `fast`, `quad`, `numer`, `anal` and `mixed`, corresponding to calculating all gradient components using forward differences, two point central differences, six point central differences, user supplied code, or any of the above choices for different components of the gradient.
- Commands that issue information about the current state of both the optimization process and the system. For example, the command `shortdis` displays the attributes of the current minimization parameters, the corresponding value of the objective function and the number of function, gradient, Hessian and Jacobian calls spent so far. Also the command `modedis` displays the currently chosen options for the various modes.
- Commands that perform file manipulation operations. For instance, the command `delete` that erases a file or the command `memo` that appends the current attributes of the minimization parameters to a file, etc.
- Commands that deal with the construction of macro packages, aliasing and other utilities.

2.5. *Extensions*

MOS is an *open* operating system in the sense that it allows for extensions, called plug-ins, and provides a mechanism to ease their integration into the environment. If the plug-ins are coded according to certain guidelines, they become automatically recognized by the MCL compiler as well. In addition a number of useful routines, called glue-routines, have been devised that enable the programmer to easily access all the important MERLIN variables, arrays and common blocks. The programmer may also take advantage of the panel mechanism in his code, by calling a provided routine and filling in a text file template.

2.6. Installation–customization

MERLIN is distributed in a form which is not directly compilable. It comes with an installation program that takes the distributed files as input and produces a standard ANSI Fortran-77 source code. The installer sets several required dimensions, some constants as well as the desired precision (single or double). The MERLIN command `limits` displays the values of these installation parameters. MERLIN has also some hard-coded defaults. Most of these can be modified without changing the source code. The customization mechanism is facilitated by editing a configuration text file (`CONFIG`) that is read during MERLIN startup. Also the default panel parameters can be customized by editing a corresponding Panel Description text file (`PDESC`).

3. Algorithms

MERLIN contains implementations of powerful minimization algorithms. In particular there are two direct methods (Roll and Simplex) that use no derivative information. These are appropriate for small problems, when the objective function is subject to noise or when derivatives cannot be calculated. From the conjugate gradient methods three algorithms are chosen: the Fletcher–Reeves, the Polak–Ribiere and the Generalized Polak–Ribiere. From the quasi-Newton family the DFP method and several versions of the BFGS method are coded. For the special case when the objective function is a sum of squares, an efficient Levenberg–Marquardt method is included.

The algorithms implemented in MERLIN are described in the following sections. Wherever appropriate we use a Fortran like form for the sake of clarity.

3.1. Direct methods

3.1.1. The Roll method

This method [2] belongs to the class of pattern search methods. It proceeds by exploring the local topology of the objective function and taking proper steps along each direction separately. In that it resembles the obvious (and ad hoc) alternating variables method [6]. When, however, the correlation among the variables becomes important, this procedure cannot proceed further. In order to cure this problem the method performs a line search along a properly formed direction, after each iteration over all the variables.

Let $\boldsymbol{x}^c = (x_1^c, x_2^c, \ldots, x_n^c)^{\mathrm{T}}$ be the current point and $f_c = f(\boldsymbol{x}^c)$. Let also s_i be a step associated with each free variable x_i. On each direction i the algorithm executes the following steps:

(1) Pick a trial point $x_j^t = x_j^c$ for all $j \neq i$ and $x_i^t = x_i^c + s_i$.
(2) Calculate $f_+ = f(\boldsymbol{x}^t)$.
(3) If $f_+ < f_c$ set $\boldsymbol{x}^c = \boldsymbol{x}^t$, $f_c = f_+$ and $s_i = as_i$. Then, go to step (8).
(4) If $f_+ \geq f_c$, pick another trial point as $x_j^t = x_j^c$ for all $j \neq i$ and $x_i^t = x_i^c - s_i$.
(5) Calculate $f_- = f(\boldsymbol{x}^t)$.
(6) If $f_- < f_c$, set $\boldsymbol{x}^c = \boldsymbol{x}^t$, $f_c = f_-$ and $s_i = -as_i$. Then, go to step (8).
(7) If $f_- \geq f_c$, calculate an appropriate step by $s_i = -\frac{1}{2}\frac{(f_+ - f_-)}{(f_+ + f_- - 2f_c)}s_i$.
(8) Proceed from step (1) for the next value of i.

In the above, $a > 1$ is a user-set factor. After looping over all variables, a line search is performed in the direction $\boldsymbol{s} = (s_1, s_2, \ldots, s_n)^{\mathrm{T}}$. The above procedure is repeated until a termination criterion applies.

3.1.2. The Simplex method

This method belongs to the class of the direct search methods for nonlinear optimization. It should not be confused with the well-known simplex method of linear programming. Originally this algorithm was designed by Spendley et al. [3] and was refined later by Nelder and Mead [4,5]. A simplex (or polytope) in $\mathbb{R}^n$ is a

construct with $(n+1)$ vertices defining a volume element. For instance, in two dimensions the simplex is a triangle, in three dimensions it is a tetrahedron, and so on so forth. The input to the algorithm, apart from a few parameters of minor importance, is an initial simplex. The algorithm brings the simplex in the area of a minimum, adapts it to the local geometry, and finally shrinks it around the minimizer. It is a derivative-free, iterative method that proceeds towards the minimum using a population of $n+1$ points (the simplex vertices) and hence it is expected to be tolerant to noise, in spite of its deterministic nature. The method executes the following steps (simplex vertices are denoted by $\boldsymbol{w}_i$):

(1) Examine the termination criteria to decide whether to stop or not.
(2) Number the simplex vertices $\boldsymbol{w}_i$ so that the sequence $f_i = f(\boldsymbol{w}_i)$ is sorted in ascending order.
(3) Calculate the centroid of the first n vertices: $\boldsymbol{c} = \frac{1}{n}\sum_{i=0}^{n-1} \boldsymbol{w}_i$.
(4) Invert the "worst" vertex $\boldsymbol{w}_n$ as $\boldsymbol{r} = \boldsymbol{c} + \alpha(\boldsymbol{c} - \boldsymbol{w}_n)$ (usually $\alpha = 1$).
(5) If $f_0 \le f(\boldsymbol{r}) \le f_{n-1}$ then
    set $\boldsymbol{w}_n = \boldsymbol{r}$, $f_n = f(\boldsymbol{r})$, and go to step (1)
    endif
(6) If $f(\boldsymbol{r}) < f_0$ then
    expand as $\boldsymbol{e} = \boldsymbol{c} + \gamma(\boldsymbol{r} - \boldsymbol{c})$ ($\gamma > 1$, usually $\gamma = 2$)
    If $f(\boldsymbol{e}) < f(\boldsymbol{r})$ then
      set $\boldsymbol{w}_n = \boldsymbol{e}$, $f_n = f(\boldsymbol{e})$
    else
      set $\boldsymbol{w}_n = \boldsymbol{r}$, $f_n = f(\boldsymbol{r})$
    endif
    go to step (1)
    endif
(7) If $f(\boldsymbol{r}) \ge f_{n-1}$ then
    If $f(\boldsymbol{r}) \ge f_n$ then
      contract as $\boldsymbol{k} = \boldsymbol{c} + \beta(\boldsymbol{w}_n - \boldsymbol{c})$ ($\beta < 1$, usually $\beta = \frac{1}{2}$)
    else
      contract as $\boldsymbol{k} = \boldsymbol{c} + \beta(\boldsymbol{r} - \boldsymbol{c})$
    endif
    If $f(\boldsymbol{k}) < \min\{f(\boldsymbol{r}), f_n\}$, then
      set $\boldsymbol{w}_n = \boldsymbol{k}$, $f_n = f(\boldsymbol{k})$
    else
      shrink the whole polytope as
      set $\boldsymbol{w}_i = \frac{1}{2}(\boldsymbol{w}_0 + \boldsymbol{w}_i)$, $f_i = f(\boldsymbol{w}_i)$ for $i = 1, 2, \ldots, n$
    endif
    go to step (1)
    endif

The initial simplex may be constructed in various ways. We have implemented two. One approach is to pick for the first vertex the current point and the rest of the vertices by line searches originating at the first and heading along each of the n directions. The second approach picks again for the first vertex the current point and generates the rest by taking a single step along each of the n directions.

3.2. *Quasi-Newton methods*

The backbone algorithm for the quasi-Newton methods [6–8] is presented below. Detailed descriptions for the several versions follow. At the start of the kth iteration a point $\boldsymbol{x}^{(k)}$, the gradient $\boldsymbol{g}^{(k)}$ and an approximation $\boldsymbol{B}^{(k)}$ to the Hessian matrix $\boldsymbol{G}^{(k)}$ are available. The following steps are then executed:

(1) Check the termination criteria in order to stop or not.

(2) Solve $B^{(k)}s^{(k)} = -g^{(k)}$ for $s^{(k)}$.
(3) Select a new "better" point $x^{(k+1)}$.
(4) Calculate the gradient vector $g^{(k+1)}$.
(5) Update $B^{(k)}$ to $B^{(k+1)}$ using a quasi-Newton formula.

Steps (3) and (5) above need further description. To obtain a new point either a line search is performed along the quasi-Newton direction (commands `bfgs`, `tolmin`, and `dfp`), or a trust region strategy is applied using the *double dogleg* technique [9,10] (command `trust`).

The line search determines a value $\lambda = \lambda^*$ so as to reduce the value of the function $f(x^{(k)}+\lambda s^{(k)})$, according to the so-called Wolfe–Powell [11–13] criteria. The new point is then taken to be $x^{(k+1)} = x^{(k)} + \lambda^* s^{(k)}$.

The trust region strategy minimizes with respect to $h^{(k)}$ the quadratic form

$$q(h^{(k)}) = f(x^{(k)}) + g^{(k)T}h^{(k)} + \tfrac{1}{2}h^{(k)T}B^{(k)}h^{(k)},$$

subject to $\|h^{(k)}\| \leq R^{(k)}$, where $R^{(k)}$ is a properly chosen radius (the trust region radius) so that the quadratic approximation is reliable. In this case the candidate point $x^{(k)} + h^{(k)}$ is either accepted, if it corresponds to a lower value, or rejected otherwise. The trust region radius is then updated to $R^{(k+1)}$ in order to make the quadratic approximation more trustworthy at the next iteration. Algorithmic details are given in Sections 3.5.1 and 3.5.2.

The updates used in step (5) are the BFGS [14–17] update (commands `bfgs`, `tolmin` and `trust`) and the DFP [18,19] update (command `dfp`). Using the definitions: $\delta^{(k)} = x^{(k+1)} - x^{(k)}$ and $\gamma^{(k)} = g^{(k+1)} - g^{(k)}$, we can write down the update formulas (the iteration superscript (k) is dropped on the right-hand side). The BFGS update is

$$B^{(k+1)} = B + \frac{\gamma\gamma^{\mathrm{T}}}{\delta^{\mathrm{T}}\gamma} - \frac{B\delta\delta^{\mathrm{T}}B}{\delta^{\mathrm{T}}B\delta}.$$

The DFP update is

$$B^{(k+1)} = B + \left(1 + \frac{\delta^{\mathrm{T}}B\delta}{\delta^{\mathrm{T}}\gamma}\right)\frac{\gamma\gamma^{\mathrm{T}}}{\delta^{\mathrm{T}}\gamma} - \frac{\gamma\delta^{\mathrm{T}}B + B\delta\gamma^{\mathrm{T}}}{\delta^{\mathrm{T}}\gamma}.$$

These updates are implemented using factorizations for the B matrix: the Choleski $B = LL^{\mathrm{T}}$ factorization [7] (commands `dfp`, `bfgs` and `trust`), and the Goldfarb–Idnani [20] $Z^{\mathrm{T}}BZ = I$ factorization (command `tolmin`).

3.2.1. The Choleski factorization

Let $B^{(k)} = LL^{\mathrm{T}}$ and $B^{(k+1)} = L_+L_+^{\mathrm{T}}$, where the lower triangular matrix L_+ is calculated from a matrix J_+ obeying $B^{(k+1)} = J_+J_+^{\mathrm{T}}$ by QR decomposition: $J_+^{\mathrm{T}} = Q_+L_+^{\mathrm{T}}$ where $\mathbf{Q}_+$ is an orthogonal matrix. Obviously we have

$$B^{(k+1)} = J_+J_+^{\mathrm{T}} = L_+Q_+^{\mathrm{T}}Q_+L_+^{\mathrm{T}} = L_+L_+^{\mathrm{T}}.$$

This J_+ matrix is given for the BFGS case as

$$J_+ = L + \frac{(\gamma - Lv)v^{\mathrm{T}}}{v^{\mathrm{T}}v},$$

where $v = aL^{\mathrm{T}}\delta$ and $a^2 = \delta^{\mathrm{T}}\gamma/\delta^{\mathrm{T}}B\delta$, while for the DFP case it is given as

$$J_+ = L - \frac{\gamma(\delta^{\mathrm{T}}L - w^{\mathrm{T}})}{\delta^{\mathrm{T}}\gamma},$$

where w is given by $Lw = \beta\gamma$ and $\beta^2 = \delta^{\mathrm{T}}\gamma/\gamma^{\mathrm{T}}B^{-1}\gamma$.

3.2.2. The Goldfarb–Idnani factorization

If $\boldsymbol{Z}^{(k)T}\boldsymbol{B}^{(k)}\boldsymbol{Z}^{(k)} = \boldsymbol{I}$ and $\boldsymbol{B}^{(k+1)}$ is given by the BFGS update, we want to find a matrix $\boldsymbol{Z}^{(k+1)}$ such that $\boldsymbol{Z}^{(k+1)T}\boldsymbol{B}^{(k+1)}\boldsymbol{Z}^{(k+1)} = \boldsymbol{I}$. If the first column of $\boldsymbol{Z}^{(k)}$ is parallel to $\boldsymbol{\delta}^{(k)}$, then the matrix with columns

$$\boldsymbol{Z}_1^{(k+1)} = \sqrt{\boldsymbol{\delta}^{(k)T}\boldsymbol{\gamma}^{(k)}}\,\boldsymbol{\delta}^{(k)},$$

$$\boldsymbol{Z}_j^{(k+1)} = \boldsymbol{Z}_j^{(k)} - \frac{\boldsymbol{Z}_j^{(k)T}\boldsymbol{\gamma}^{(k)}}{\boldsymbol{\delta}^{(k)T}\boldsymbol{\gamma}^{(k)}}\boldsymbol{\delta}^{(k)} \quad \text{for } j = 2,3,\ldots,n,$$

is the required $\boldsymbol{Z}^{(k+1)}$ matrix. Since, however, $\widetilde{\boldsymbol{Z}}^{(k)} = \boldsymbol{Z}^{(k)}\boldsymbol{Q}^{(k)}$ with $\boldsymbol{Q}^{(k)}$ any orthogonal matrix also satisfies $\widetilde{\boldsymbol{Z}}^{(k)T}\boldsymbol{B}^{(k)}\widetilde{\boldsymbol{Z}}^{(k)} = \boldsymbol{I}$, one can construct a $\boldsymbol{Q}^{(k)}$ matrix such that $\widetilde{\boldsymbol{Z}}_1^{(k)}$ is parallel to $\boldsymbol{\delta}^{(k)}$ and then apply the above procedure on the $\widetilde{\boldsymbol{Z}}^{(k)}$ matrix to get $\boldsymbol{Z}^{(k+1)}$. This factorization is the one preferred by Powell [21] and used in [22].

3.3. Conjugate gradient methods

The conjugate gradient methods are economical in computer memory since they require only a few arrays of N-elements each. The backbone algorithm for the Fletcher–Reeves [23] and Polak–Ribiere [24] methods is described below,

Initially at the provided point $\boldsymbol{x}^{(1)}$ we set $\boldsymbol{s}^{(1)} = -\boldsymbol{g}^{(1)}$. The kth iteration consists of the following steps:

(1) Perform a line search along $\boldsymbol{s}^{(k)}$ and obtain $\boldsymbol{x}^{(k+1)}$.
(2) Check the termination criteria in order to stop or not.
(3) Calculate the gradient vector $\boldsymbol{g}^{(k+1)}$.
(4) Calculate a scalar $\beta^{(k)}$ using one of the following prescriptions:
   (a) Fletcher–Reeves: $\beta^{(k)} = \frac{\boldsymbol{g}^{(k+1)\mathrm{T}}\boldsymbol{g}^{(k+1)}}{\boldsymbol{g}^{(k)\mathrm{T}}\boldsymbol{g}^{(k)}}$.
   (b) Polak–Ribiere: $\beta^{(k)} = \frac{(\boldsymbol{g}^{(k+1)}-\boldsymbol{g}^{(k)})^{\mathrm{T}}\boldsymbol{g}^{(k+1)}}{\boldsymbol{g}^{(k)\mathrm{T}}\boldsymbol{g}^{(k)}}$.
(5) Calculate a new search direction as $\boldsymbol{s}^{(k+1)} = -\boldsymbol{g}^{(k+1)} + \beta^{(k)}\boldsymbol{s}^{(k)}$.

The generalized Polak–Ribiere [25] method is as follows:

(1) Perform a line search along $\boldsymbol{s}^{(k)}$ and obtain $\boldsymbol{x}^{(k+1)}$.
(2) Check the termination criteria in order to stop or not.
(3) If $k+1 > n > 2$ restart the procedure.
(4) Calculate $\boldsymbol{d}^{(k)} = \boldsymbol{s}^{(k)} - \frac{\boldsymbol{g}^{(k+1)\mathrm{T}}\boldsymbol{s}^{(k)}}{\boldsymbol{g}^{(k+1)\mathrm{T}}\boldsymbol{g}^{(k+1)}}\boldsymbol{g}^{(k+1)}$.
(5) Choose $\delta = \min\left\{1, \sqrt{\frac{\eta}{\boldsymbol{d}^{(k)\mathrm{T}}\boldsymbol{d}^{(k)}}}\right\}$, η being the machine's accuracy.
(6) Calculate a new search direction as $\boldsymbol{s}^{(k+1)} = -\boldsymbol{g}^{(k+1)} + \beta^{(k)}\boldsymbol{d}^{(k)}$, where $\beta^{(k)} = -\boldsymbol{g}^{(k+1)^{\mathrm{T}}}\frac{\boldsymbol{g}^{(k+1)}-\boldsymbol{g}'^{(k)}}{\boldsymbol{g}'^{(k)\mathrm{T}}\boldsymbol{d}^{(k)}}$ and $\boldsymbol{g}'^{(k)} = \boldsymbol{g}(\boldsymbol{x}^{(k+1)} - \delta\boldsymbol{d}^{(k)})$.

3.4. Levenberg–Marquardt method for sum of squares

For the case where the objective function is a sum of squares, i.e.

$$f(\boldsymbol{x}) = \sum_{i=1}^{M} f_i^2(\boldsymbol{x}) \equiv \boldsymbol{r}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{r}(\boldsymbol{x}),$$

with $\boldsymbol{r}^{\mathrm{T}}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_M(\boldsymbol{x}))^{\mathrm{T}}$, a special method has been proposed first by Levenberg [26] and later on by Marquardt [27]. Let $\boldsymbol{J}$ be the Jacobian matrix $\partial f_i(\boldsymbol{x})/\partial x_j$ and let $\boldsymbol{D}$ be a diagonal matrix. The quadratic approximation to $f(\boldsymbol{x}+\boldsymbol{h})$ is given by

$$q(\boldsymbol{h}) = f(\boldsymbol{x}) + \boldsymbol{g}^{\mathrm{T}}(\boldsymbol{x})\boldsymbol{h} + \tfrac{1}{2}\boldsymbol{h}^{\mathrm{T}}\boldsymbol{B}(\boldsymbol{x})\boldsymbol{h}$$

and is being minimized under the condition $\|\boldsymbol{D}\boldsymbol{h}\| \leq R$, where R is the radius of the trust region. The trust region in this case is a hyper-ellipsoid with semi-axis lengths $R/\boldsymbol{D}_{ii}$. Since $\boldsymbol{g} = 2\boldsymbol{J}^{\mathrm{T}}\boldsymbol{r}$ and if the Gauss–Newton approximation is made, i.e. $\boldsymbol{B}(\boldsymbol{x}) \approx 2\boldsymbol{J}^{\mathrm{T}}\boldsymbol{J}$, we get using the Lagrange multiplier procedure,

$$[\boldsymbol{J}^{\mathrm{T}}\boldsymbol{J} + \lambda\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}]\boldsymbol{h} = -\boldsymbol{J}^{\mathrm{T}}\boldsymbol{r} \quad \text{or} \quad \boldsymbol{h}(\lambda) = -[\boldsymbol{J}^{\mathrm{T}}\boldsymbol{J} + \lambda\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}]^{-1}\boldsymbol{J}^{\mathrm{T}}\boldsymbol{r}.$$

Initially we set $\boldsymbol{D}_{ii}^{(0)} = \|\partial \boldsymbol{r}(\boldsymbol{x}^{(0)})/\partial x_i\| \ \ \forall\, i = 1, 2, \ldots, N$. The kth iteration of the algorithm is as

(1) If $\|\boldsymbol{D}^{(k)}\boldsymbol{h}^{(k)}(0)\| \leq R^{(k)}$ then
  set $\boldsymbol{\delta}^{(k)} = \boldsymbol{h}^{(k)}(0)$
  else
  find a $\lambda^{(k)} > 0$ such that $\|\boldsymbol{D}^{(k)}\boldsymbol{h}^{(k)}(\lambda^{(k)})\| = R^{(k)}$
  set $\boldsymbol{\delta}^{(k)} = \boldsymbol{h}^{(k)}(\lambda^{(k)})$
  endif
  If $f(\boldsymbol{x}^{(k)} + \boldsymbol{\delta}^{(k)}) < f(\boldsymbol{x}^{(k)})$ then
  set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{\delta}^{(k)}$, and calculate $\boldsymbol{J}^{(k+1)}$
  else
  set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)}$ and $\boldsymbol{J}^{(k+1)} = \boldsymbol{J}^{(k)}$
  endif

(2) Update $R^{(k)}$ to $R^{(k+1)}$ (the algorithm is similar to the one described in the next section for the dogleg case).

(3) Choose $\boldsymbol{D}^{(k+1)}$ as $\boldsymbol{D}_{ii}^{(k+1)} = \max\left\{\boldsymbol{D}_{ii}^{(k)}, \|\frac{\partial \boldsymbol{r}(\boldsymbol{x}^{(k+1)})}{\partial x_i}\|\right\} \ \ \forall\, i = 1, 2, \ldots, N.$

A very robust implementation of the above is described by Moré [28].

3.5. Selection techniques

3.5.1. The dogleg technique

Given a quadratic model

$$f(\boldsymbol{x}+\boldsymbol{h}) \approx q(\boldsymbol{h}) = f(\boldsymbol{x}) + \boldsymbol{h}^{\mathrm{T}}\nabla f(\boldsymbol{x}) + \tfrac{1}{2}\boldsymbol{h}^{\mathrm{T}}\nabla^2 f(\boldsymbol{x})\boldsymbol{h} = f(\boldsymbol{x}) + \boldsymbol{h}^{\mathrm{T}}\boldsymbol{g} + \tfrac{1}{2}\boldsymbol{h}^{\mathrm{T}}\boldsymbol{G}\boldsymbol{h},$$

the problem

$$\min_{\boldsymbol{h}}\{q(\boldsymbol{h})\} \text{ subject to } \|\boldsymbol{h}\| \leq R$$

is solved approximately by the following technique termed by Powell as the *dogleg* method [9]. Two points are calculated. The Cauchy point $\boldsymbol{x}_c = \boldsymbol{x} + \boldsymbol{h}_c$ and the Newton point $\boldsymbol{x}_N = \boldsymbol{x} + \boldsymbol{h}_N$. The Cauchy point is the minimum along the gradient direction, i.e. $\boldsymbol{h}_c = -\lambda\boldsymbol{g}$ with $\lambda = \boldsymbol{g}^{\mathrm{T}}\boldsymbol{g}/\boldsymbol{g}^{\mathrm{T}}\boldsymbol{G}\boldsymbol{g}$, while the Newton step is given by $\boldsymbol{h}_N = -\boldsymbol{G}^{-1}\boldsymbol{g}$. If $\|\boldsymbol{h}_N\| \leq R$ the Newton point is taken as the next trial iterate. Otherwise the first point where the piecewise linear trajectory $\boldsymbol{x} \to \boldsymbol{x}_c \to \boldsymbol{x}_N$ intersects the sphere of radius R centered at $\boldsymbol{x}$ is taken as the next trial iterate.

Dennis and Mei [10] proposed a similar procedure termed *double dogleg*, that defines another point $\boldsymbol{x}_D = \boldsymbol{x} + \boldsymbol{h}_D$ with $\boldsymbol{h}_D = \zeta\boldsymbol{h}_N$, $\zeta = 0.8\gamma + 0.2$, $\gamma = (\boldsymbol{g}^{\mathrm{T}}\boldsymbol{g})^2/[(\boldsymbol{g}^{\mathrm{T}}\boldsymbol{G}\boldsymbol{g})(\boldsymbol{g}^{\mathrm{T}}\boldsymbol{G}^{-1}\boldsymbol{g})]$ and a modified trajectory $\boldsymbol{x} \to \boldsymbol{x}_c \to \boldsymbol{x}_D \to \boldsymbol{x}_N$. The updating scheme of the trust region radius is given below:

(1) Calculate the ratio of the actual to the expected reduction $r^{(k)} = (f^{(k)} - f^{(k+1)})/(f^{(k)} - q(\boldsymbol{h}^{(k)}))$, where $f^{(k)}$ stands for $f(\boldsymbol{x}^{(k)})$ and $f^{(k+1)} = f(\boldsymbol{x}^{(k)} + \boldsymbol{h}^{(k)})$.
(2) Accept or reject the trial point according to
If $r^{(k)} \leq 0$ then
$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)}$, $f^{(k+1)} = f^{(k)}$
else
$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{h}^{(k)}$
endif
(3) If $r^{(k)} < 0.25$ then
$R^{(k+1)} = \|\boldsymbol{h}^{(k)}\|/4$
else if $r^{(k)} > 0.75$ and $\|\boldsymbol{h}^{(k)}\| = R^{(k)}$ then
$R^{(k+1)} = 2R^{(k)}$
else
$R^{(k+1)} = R^{(k)}$
endif

3.5.2. Line search for descent methods

Line searches are used in quasi-Newton and conjugate gradient methods. The idea of a line search algorithm is simple: given a descent direction $\boldsymbol{s}^{(k)}$, we take a step $\lambda^{(k)}$ in that direction that yields an acceptable next iterate. For convenience we denote $f(\lambda) \equiv f(\boldsymbol{x}^{(k)} + \lambda^{(k)}\boldsymbol{s}^{(k)})$ and $f'(\lambda) \equiv \boldsymbol{s}^{(k)\mathrm{T}}\boldsymbol{g}^{(k)}(\boldsymbol{x}^{(k)} + \lambda^{(k)}\boldsymbol{s}^{(k)})$. Descent methods are known to converge [11–13] when λ is chosen to satisfy the *weak Wolfe–Powell* conditions,

$$f(\lambda) \leq f(0) + \lambda\rho f'(0) \tag{1}$$

and

$$f'(\lambda) \geq \sigma f'(0)\,, \tag{2}$$

where $\rho \in (0, \frac{1}{2})$ and $\sigma \in (\rho, 1)$. In practice, we prefer to use the more stringent test

$$|f'(\lambda)| \leq -\sigma f'(0) \tag{3}$$

in place of Eq. (2), which along with Eq. (1) are called the *strong Wolfe–Powell* conditions.

Our line search algorithm uses a sectioning scheme, that mainly follows Al-Baali and Fletcher [30]. In the sectioning scheme, sequences a_j, b_j, λ_j are generated. a_j is always the current best point (least f) that satisfies Eq. (1) but neither Eq. (2) nor Eq. (3). λ_j is the current trial point. b_j either fails to satisfy Eq. (1), or $f(b_j) \geq f(a_j)$, or both. However, the interval (a_j, b_j) will always bracket either an interval of acceptable points, or points for which $f(\lambda) \leq \overline{f}$, with $\overline{f}$ being a lower bound on f.

The line search is initialized with $a_1 = 0$, $b_1 = \infty$, $\overline{f} \leq f(0)$ and an estimation for $\lambda_1 > 0$. The jth iteration is given below:

(1) Evaluate $f(\lambda_j)$.
(2) If $f(\lambda_j) \leq \overline{f}$ then terminate.
(3) If $f(\lambda_j) > f(0) + \lambda_j\rho f'(0)$ or $f(\lambda_j) \geq f(a_j)$ then
choose $\lambda_{j+1} \in T(a_j, \lambda_j)$ using either a quadratic interpolating $f(a_j)$, $f'(a_j)$ and $f(\lambda_j)$,
or a cubic interpolating $f(a_j)$, $f'(a_j)$, $f(\lambda_j)$ and $f(b_j)$
set $a_{j+1} = a_j$, $b_{j+1} = \lambda_j$
else
evaluate $f'(\lambda_j)$
test for termination; for the weak Wolfe–Powell conditions use Eq. (2), otherwise use Eq. (3)
set $a_{j+1} = \lambda_j$

If $(b_j - a_j) f'(\lambda_j) < 0$ then
  choose $\lambda_{j+1} \in E(a_j, \lambda_j, b_j)$ using a cubic interpolating either $f(a_j)$, $f'(a_j)$, $f(\lambda_j)$ $f'(\lambda_j)$, or $f(a_j)$, $f(b_j)$, $f(\lambda_j)$, $f'(\lambda_j)$
  set $b_{j+1} = b_j$
else
  choose $\lambda_{j+1} \in T(a_j, \lambda_j)$ using a cubic that interpolates $f(a_j)$, $f'(a_j)$, $f(\lambda_j)$, $f'(\lambda_j)$
  set $b_{j+1} = a_j$
endif
endif

When interpolating, we use the truncation scheme defined by

$$T(a,b) = \begin{cases} [a + \tau_1(b-a), b - \tau_2(b-a)] & \text{if } a < b, \\ [b + \tau_2(a-b), a - \tau_1(a-b)] & \text{if } b < a, \end{cases} \tag{4}$$

where $0 < \tau_1 \leq \tau_2 \leq \frac{1}{2}$. When extrapolating, we define

$$E(a,\lambda,b) = \begin{cases} [\min(\tau_3, \lambda), \min(\tau_4, \lambda)] & \text{if } a < \lambda < b = \infty, \\ [\lambda + \tau_5(b-a), b - \tau_6(b-a)] & \text{if } a < \lambda < b, \\ [b + \tau_6(a-b), \lambda - \tau_5(a-b)] & \text{if } b < \lambda < a, \end{cases} \tag{5}$$

where $1 < \tau_3 \leq \tau_4$ and $0 < \tau_5 \leq \tau_6 \leq \frac{1}{2}$.

3.5.3. One-dimensional minimization

A one-dimensional minimization procedure is needed by the Roll and simplex methods. Given an interval $[a_1, b_1]$ that brackets a minimum, we use Brent's method [29] that locates the minimum λ, within a prescribed tolerance ε. At every iteration j the method keeps track of six points a_j, b_j, u_j, v_j, w_j and λ_j, not necessarily distinct. A minimum always lies in the interval $[a_j, b_j]$. λ_j is the point with the least value of f. w_j is the point with the next lowest value of f. v_j is the previous value of w_j, and u_j is the last point at which f has been evaluated. Initially $v_1 = w_1 = \lambda_1 = a_1 + \frac{3-\sqrt{5}}{2}(b_1 - a_1)$. The jth iteration is described below.

(1) Test for termination. If $\max(\lambda_j - a_j, b_j - \lambda_j) \leq 2\varepsilon$ then return with λ_j as the approximate position of the minimum.
(2) Calculate p, q so that $\lambda_j + p/q$ is the turning point of the parabola passing through the points $(v_j, f(v_j))$, $(w_j, f(w_j))$ and $(\lambda_j, f(\lambda_j))$.
(3) Calculate the new point u_{j+1}: let e be the value of p/q at the second-last cycle. If $|e| \leq \varepsilon$, $q = 0$, $\lambda_j + p/q \notin (a,b)$ or $|p/q| \geq |e|/2$, then take a golden section step, otherwise u_{j+1} is taken to be $\lambda_j + p/q$, except that the distances $|u_{j+1} - \lambda_j|$, $u_{j+1} - a_j$ and $b_j - u_{j+1}$ must be at least ε.
(4) Evaluate f at the new point u_{j+1}.
(5) Update the points a_j, b_j, v_j, w_j and λ_j as necessary.

3.6. Termination criteria

Termination criteria are conditions that control the termination of the algorithm. In our case they contain convergence as well as other conditions that serve practical issues. Different methods use different criteria for termination. However there are some criteria shared by all methods. These are:

(1) The number of calls to the objective function has exceeded a preset limit.
(2) The value of the objective function surpassed a preset target value.

The quasi-Newton and the conjugate gradient methods use in addition the following:

(1) The reduction rate of the objective function is slower than specified.
(2) The number of iterations has exceeded a preset limit.

(3) The absolutely maximum relative gradient component is less than a preset limit.
(4) The relative change of each of the x_i's is absolutely lower than a preset limit.

The Roll method terminates if the reduction rate of the objective function is for a number of consecutive iterations, less than a specified minimum value. For the Simplex method the termination criterion relies on comparing a measure of the polytope's "error" to a preset small positive number. Specifically, the algorithm terminates if

$$\frac{1}{n+1}\sum_{i=0}^{n}|f_i-\bar{f}|\leq\epsilon,$$

where

$$\bar{f}=\frac{1}{n+1}\sum_{i=0}^{n}f_i.$$

For the Levenberg–Marquardt method three tests are used in addition:
(1) The reduction rate of the objective function is slower than specified.
(2) The change $\|\boldsymbol{h}\|/\|\boldsymbol{Dx}\|$ is lower than a preset limit.
(3) The quantity $\max_i\{\boldsymbol{J}_i^{\mathrm{T}}\boldsymbol{r}/\|\boldsymbol{J}_i\|\|\boldsymbol{r}\|\}$ is less than a preset limit, where $\boldsymbol{J}_i$ is the ith column of the Jacobian matrix.

3.7. Simple bound constraints

The treatment of the simple bound constraints $l_i\leq x_i\leq u_i$ is different for each method.

- All methods that use a line search to select the new point (`bfgs`, `congra`, `dfp` and `tolmin`), handle the bounds by calculating the feasible segment on the search direction. The line search is then performed in the feasible domain. If this segment is very short, then the direction of search is modified so as to become parallel to the limiting constraint boundary.
- In the Simplex method, all operations (reflection, expansion, contraction) are performed by first defining a direction and then selecting a point on this direction. Again the feasible segment is determined and the point is constrained inside this segment.
- In the Roll method, if a variable steps beyond a bound, it is reset to the middle of the distance between the original position and the limiting bound.
- In the Levenberg–Marquardt method, where code from the MINPACK-1 is used (command `leve`), as well as in the trust region implementation of the BFGS method (command `trust`), the following variable transformations are used:
  - if only a lower bound exists, i.e. $l_i\leq x_i$, then $x_i=l_i+y_i^2$;
  - if only an upper bound exists, i.e. $x_i\leq u_i$ then $x_i=u_i-y_i^2$;
  - if both bounds exist, i.e. $l_i\leq x_i\leq u_i$, then $x_i=(u_i-l_i)e^{-y_i^2}+l_i$.

4. User written programs

MERLIN offers an optimization environment, supplies several utility as well as minimization tools, so that the user can choose whatever is convenient and effective each time. The programs the user may have to write are the following:

(1) The main program.
(2) The objective function in a general form, or in a sum of squares form.

(3) The gradient vector (optional).
(4) The Jacobian matrix (optional).
(5) The Hessian matrix (optional).

In what follows we give some examples for the user supplied modules. These examples are in single precision and should be used with the single precision installation. Double precision user modules should be used in conjunction with the double precision installation.

4.1. The main program

The role of the main program is to initiate MERLIN's execution. The user may write his own main program or may use the sample provided below modified appropriately to meet his needs. To invoke MERLIN one should make a call to the subprogram:

```
SUBROUTINE MERLIN ( N, M, VERSIM, MAXW, IQUIT )
```

where

- `N` (input) is the dimensionality of the problem.
- `M` (input) is the number of the squared terms (useful for the sum of squares form)
- `VERSIM` is a work-space array dimensioned as `VERSIM(MAXW)`.
- `MAXW` (input) = $\max\{N(N+11), NM\}$.
- `IQUIT` (output) is an output flag specified by the user at run-time, or set by MOS to indicate an error condition.

We list a sample main program for illustration purposes.

```
      PROGRAM MASTER
* Maximum dimensionality handled.
      PARAMETER (MXV = 200)
* Maximum number of squared terms.
      PARAMETER (MXT = 1000)
* Storage required: MAX( MXV*MXT, MXV*(MXV+11) )
      PARAMETER (MAXW = MXT*MXV)
      DIMENSION VERSIM (MAXW)
      WRITE (*,*) 'Enter number of variables, number of squared terms'
      READ (*,*) N, M
* Now invoke the Merlin system.
      CALL MERLIN(N,M,VERSIM,MAXW,IQUIT)
      END
```

4.2. The objective function. General form

This must be written as a function subprogram:

```
      FUNCTION FUNMIN(X,N)
      DIMENSION X(N)
```

- `X` (input) is an array holding the values of the parameters x_i.
- `N` (input) is the dimensionality of the problem.
- `FUNMIN` (output) upon return assumes the value of the objective function $f(\boldsymbol{x})$.

An example for the Rosenbrock test function is given below:

$$f(\boldsymbol{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

```
FUNCTION FUNMIN(X,N)
DIMENSION X(N)
FUNMIN = 100*(X(2)-X(1)**2)**2 + (1-X(1))**2
END
```

4.3. The objective function. Sum of squares form

This must be written as a subroutine subprogram:

```
SUBROUTINE SUBSUM ( M, N, X, F )
DIMENSION X(N), F(M)
```

- M (input) is the number of the squared terms.
- N (input) is the dimensionality of the problem.
- X (input) is an array holding the values of the parameters x_i.
- F (output) is an array holding the values of the terms $f_i(\boldsymbol{x})$.

An example is given below again for the Rosenbrock test function with $f_1 = 10(x_2 - x_1^2)$ and $f_2 = 1 - x_1$.

```
SUBROUTINE SUBSUM(M,N,X,F)
DIMENSION X(N),F(M)
F(1) = 10*(X(2)-X(1)**2)
F(2) = 1-X(1)
END
```

4.4. The gradient vector

This must be written as a subroutine subprogram:

```
SUBROUTINE GRANAL(N,X,GRAD)
DIMENSION X(N),GRAD(N)
```

- N (input) is the dimensionality of the problem.
- X (input) is an array holding the values of the parameters x_i.
- GRAD (output) is an array holding the values of the gradient components $g_i(\boldsymbol{x})$.

An example is given below for the gradient of the Rosenbrock function.

```
SUBROUTINE GRANAL(N,X,GRAD)
DIMENSION X(N),GRAD(N)
A = X(2)-X(1)**2
GRAD(1) = -400*A*X(1) - 2*(1-X(1))
GRAD(2) =  200*A
END
```

Note that in earlier versions of MERLIN this routine had a different calling sequence.

4.5. The Jacobian matrix

This must be written as a subroutine subprogram:

```
SUBROUTINE JANAL ( M, N, X, FJ, LD )
DIMENSION X(N), FJ(LD,N)
```

- M (input) is the number of the squared terms.
- N (input) is the dimensionality of the problem.

- X (input) is an array holding the values of the parameters x_i.
- FJ (output) is an array holding the values: FJ(I,J) $= \partial f_i / \partial x_j$.
- LD (input) is the leading dimension of the matrix FJ used by MERLIN to store the Jacobian.

An example is given below for the Jacobian of the Rosenbrock function.

```
SUBROUTINE JANAL ( M, N, X, FJ, LD )
DIMENSION X(N), FJ(LD,N)
FJ(1,1) = -20*X(1)
FJ(1,2) =  10
FJ(2,1) =  -1
FJ(2,2) =   0
END
```

4.6. The Hessian matrix

This must be written as a subroutine subprogram:

```
SUBROUTINE HANAL ( H, LD, N, X )
DIMENSION H(LD,N), X(N)
```

- H (output) is an array holding the values of the Hessian elements $G_{ij}(\boldsymbol{x})$.
- LD (input) is the leading dimension of the matrix H used by MERLIN to store the Hessian.
- N (input) is the dimensionality of the problem.
- X (input) is an array holding the values of the parameters x_i.

Note that *only the lower triangular part of* H *must be filled in.* The rest of the Hessian matrix is completed by MERLIN, using symmetry. An example is given below for the Hessian of the Rosenbrock function.

```
SUBROUTINE HANAL(H,LD,N,X)
DIMENSION H(LD,N), X(N)
H(1,1) = 2 + 1200*X(1)**2 - 400*X(2)
H(2,1) = -400*X(1)
H(2,2) = 200
END
```

4.7. Important note

The user must always construct one of the FUNCTION FUNMIN, or the SUBROUTINE SUBSUM subprograms. However, a dummy routine must be provided for the one that is left out, since many linkers will not create the executable file otherwise. We list below examples for the dummy subprograms.

```
FUNCTION FUNMIN(X,N)
DIMENSION X(N)
END
```

```
SUBROUTINE SUBSUM(M,N,X,F)
DIMENSION  X(N),F(M)
END
```

The same action must be taken when the user does not wish to code the SUBROUTINE GRANAL, the SUBROUTINE JANAL and the SUBROUTINE HANAL subprograms. The dummy routines should read as

```
SUBROUTINE GRANAL(N,X,GRAD)
```

```
DIMENSION X(N),GRAD(N)
END
SUBROUTINE JANAL ( M, N, X, FJ, LD )
DIMENSION X(N), FJ(LD,N)
END
SUBROUTINE HANAL(H,LD,N,X)
DIMENSION H(LD,N), X(N)
END
```

5. About the software

The present package MERLIN/MCL version 3.0 has evolved from earlier versions. The first published version dates back to 1987 [2]. The first programmable version along with the definition of the MCL programming language appeared in 1989 [31,32], and a double precision version in 1990 [33]. Since then there were numerous unpublished revisions, in an effort to improve and enhance the whole environment. Note that the MERLIN package in its 1989 version was around 10000 lines of code long, while the current edition is about four times longer. The current version is almost entirely rewritten and has been tested in various fields, especially in molecular physics and in particular in conjunction with the molecular mechanics method [34,35]. We have also used this software extensively for training neural networks in various circumstances [36–40]. Users of the older MERLIN/MCL packages will be able to work immediately with the present software, since we made every effort to maintain the old style and philosophy of operation.

At this point we would like to acknowledge the incorporation of software written by others.

(1) **MINPACK-1**
We have used parts of the MINPACK [41] software (subroutines LMDIF, FDJAC2, LMPAR, QRFAC, QRSOLV and ENORM) to treat the special case of the "sum of squares" objective function. The related command `leve` invokes the above-mentioned routines.

(2) **TOLMIN**
We have incorporated the TOLMIN package [22] written by M.J.D. Powell, that is based on [21]. This software is capable of treating general linear constraints in addition to simple bounds. However, for reasons of uniformity with the rest of the environment, this feature has been disabled.

(3) **RANLUX**
We have incorporated the random number generator RANLUX, due to F. James [42].

6. Description of the test run

The test run was performed in a Sun SPARCstation 5 in double precision, using Rosenbrock's function. The test run input consists of 17 lines.

Lines 1–6 set an initial point, and assign symbolic names, lower and upper bounds for parameters. Line 6 sets a title for the current session. Line 7 displays the parameters and their attributes, while line 9 invokes the Roll method with a maximum of 20 function calls. Line 9 instructs MERLIN to use the user supplied `SUBROUTINE GRANAL` to obtain the gradient vector. Lines 10–11 calculate the Hessian matrix (using the user supplied `SUBROUTINE HANAL`) and its Choleski decomposition. Line 12 invokes the BFGS method using the previously decomposed Hessian as an initial approximation and a maximum of 10 function calls. Lines 13–14 invoke the Levenberg–Marquardt method using the user supplied `SUBROUTINE JANAL` to estimate the Jacobian. Finally, lines 15–16 display the current values of the parameters and terminate execution.

Acknowledgements

We would like to thank the authors of MINPACK [41] and in particular Dr. J. More with whom we have corresponded for allowing us to embed code from MINPACK-1 in our package. We also would like to thank Prof. Powell for his kind permission that made possible the integration of TOLMIN in the MERLIN environment and for his many useful suggestions during his visits to the University of Ioannina. The contribution of Mr. P. Kiriakakis to the PostScript graph module is gratefully acknowledged.

This work has been partially supported by the Greek Government through the General Secretariat of Research and Technology, under contracts ΠΕΝΕΔ 89ΕΔ53 and ΠΕΝΕΔ 91ΕΔ959.

References

[1] D.G. Papageorgiou, I.N. Demetropoulos, I.E. Lagaris, The Merlin Control Language for strategic optimization, Comput. Phys. Commun. 109 (1998) 250–275, succeeding article.
[2] G.A. Evangelakis, J.P. Rizos, I.E. Lagaris, I.N. Demetropoulos, Merlin – A portable system for multidimensional minimization, Comput. Phys. Commun. 46 (1987) 401–415.
[3] W. Spendley, G.R. Hext, F.R. Himsworth, Sequential application of simplex designs of optimization and evolutionary operations, Technometrics 4 (1962) 441–461.
[4] J.A. Nelder, R. Mead, A simplex method for function minimization, Comput. J. 7 (1964) 308–313.
[5] J.A. Nelder, R. Mead, A simplex method for function minimization, Comput. J. (Errata) 8 (1965) 27.
[6] R. Fletcher, Practical Methods of Optimization (Wiley, New York, 1987).
[7] J.E. Dennis, Jr., Robert B. Schnabel, Numerical Methods for Unconstrained Optimization, Nonlinear equations (Prentice-Hall, Englewoods Cliffs, NJ, 1983).
[8] P.E. Gill, W. Murray, M.H. Wright, Practical Optimization (Academic Press, New York, 1981).
[9] M.J.D. Powell, Rank one methods for unconstrained optimization, in: Nonlinear and Integer Programming, J. Abadie, ed. (North-Holland, Amsterdam, 1970).
[10] J.E. Dennis, Jr., H.H. Mei, Two new unconstrained optimization algorithms which use function and gradient values, J. Optim. Theory & Appl. 28 (1979) 453–482.
[11] A.A. Goldstein, On steepest descent, SIAM J. Control 3 (1965) 147–151.
[12] P. Wolfe, Convergence conditions for ascent methods, SIAM Rev. 11 (1969) 226–235.
[13] M.J.D. Powell, A view of unconstrained optimization, in: Optimization in Action, L.C.W. Dixonm, ed. (Academic Press, New York, 1976).
[14] R. Fletcher, A new approach to variable metric algorithms, Comput. J. 13 (1970) 317–322.
[15] D. Goldfarb, A family of variable metric methods derived by variational means, Math. Comput. 24 (1970) 23–26.
[16] C.G. Broyden, The convergence of a class of double rank minimization algorithms. 2. The new algorithm, J. Instit. Math. & Appl. 6 (1970) 222–231.
[17] D.F. Shanno, Conditioning of quasi-Newton methods for function minimizations, Math. Comput. 24 (1970) 641–656.
[18] W.C. Davidon, Variable metric method for minimization, AEC Research Development Report, ANL-5990 (1959).
[19] R. Fletcher, M.J.D. Powell, A rapidly convergent descent method for minimization, Comput. J. 6 (1963) 163–168.
[20] D. Goldfarb, A. Idnani, A numerically stable dual method for solving strictly convex quadratic programs, Math. Program. 27 (1983) 1–33.
[21] M.J.D. Powell, A tolerant algorithm for linearly constrained optimization calculations, Math. Program. 45 (1989) 547.
[22] M.J.D. Powell, TOLMIN: A Fortran package for linearly constrained optimization calculations, Report DAMTP/1989/NA2, University of Cambridge.
[23] R. Fletcher, C. Reeves, Function optimization by conjugate gradients, Comput. J. 7 (1964) 149–154.
[24] E. Polak, G. Ribiere, Note Sur la Convergence des Methodes de Directions Conjuguees, Rev. Fr. Inf. & Rech. Oper. (France) 16 (1969) 35–43.
[25] K.M. Khoda, Y. Liu, C. Storey, Generalized Polak–Ribiere algorithm, J. Optim. Th. & Appl. 75 (1992) 345–354.
[26] K. Levenberg, A method for the solution of certain problems in least squares, Q. Appl. Math. 2 (1944) 164–168.
[27] D.W. Marquardt, An algorithm for least squares estimation of nonlinear parameters, SIAM J. Indust. & Appl. Math. 11 (1963) 431–441.
[28] J.J. Moré, The Levenberg–Marquardt algorithm: implementation, theory, in: Numerical Analysis, G.A. Watson, ed., Lecture Notes in Mathematics, Vol. 630 (Springer, Berlin, 1977) pp. 105–116.
[29] R.P. Brent, Algorithms for Minimization without Derivatives (Prentice Hall, Englewood Cliffs, NJ, 1973).

[30] M. Al-Baali, R. Fletcher, An efficient line search for nonlinear least squares, J. Optim. Th. & Applic. 48 (1986) 359–378.
[31] D.G. Papageorgiou, C.S. Chassapis, I.E. Lagaris, MERLIN-2.0 – Enhanced, programmable version, Comput. Phys. Commun. 52 (1989) 241–247.
[32] C.S. Chassapis, D.G. Papageorgiou, I.E. Lagaris, MCL – Optimization oriented programming language, Comput. Phys. Commun. 52 (1989) 223–239.
[33] D.G. Papageorgiou, I.E. Lagaris, MERLIN-2.1, Double Precision, Comput. Phys. Commun. 58 (1990) 119–125.
[34] D.G. Papageorgiou, Locating Local, Global Minima of Nonlinear Functions. Application to Molecular Systems of Biological Interest, PhD Thesis (Dept. of Chemistry, University of Ioannina, Greece, 1997) [In Greek].
[35] D.G. Papageorgiou, I.N. Demetropoulos, I.E. Lagaris, P.T. Papadimitriou, How many conformers of the 1,2,3-Propanetriol Triacetate are present in gas phase and in aqueous solution?, Tetrahedron 52 (1996) 677–686.
[36] A. Likas, I.E. Lagaris, Training reinforcement neurocontrollers using the polytope algorithm, Preprint 13-96 (Dep. Computer Science, Univ. Ioannina, 1996).
[37] D.A. Karras, I.E. Lagaris, A novel neural network training technique based on a multi algorithm constrained optimization strategy, Preprint 14-96 (Dep. Computer Science, Univ. Ioannina, 1996).
[38] I.E. Lagaris, A. Likas, D.I. Fotiadis, Artificial neural networks for solving ordinary, partial differential equations, Preprint 15-96 (Dep. Computer Science, Univ. Ioannina, 1996).
[39] A. Likas, D.A. Karras, I.E. Lagaris, Neural network training, simulation using a multidimensional optimization system, Int. J. Comput. Math., to appear.
[40] I.E. Lagaris, A. Likas, D.I. Fotiadis, Artificial neural network methods in quantum mechanics, Comput. Phys. Commun. 104 (1997) 1.
[41] Jorge J. Moré, Burton S. Garbow, Kenneth E. Hillstrom, User guide for MINPACK-1, ANL Report ANL-80-74 (1980).
[42] F. James, RANLUX: A Fortran implementation of the high-quality pseudo-random number generator of Luscher, Comput. Phys. Commun. 79 (1994) 111–114.

TEST RUN INPUT

```
2 2
point 1 0.333 2 -999.0
godfather 1 rho 2 sigma
lmargin rho -1000 sigma -2000
rmargin rho 100 sigma 200
title Rosenbrock\'s two-dimensional test function
shortdis
roll noc  20
anal
hessian do c use a
hessian do d
bfgs  noc 10 useh 1
janal
leve noc 20
shortdis
stop
```

TEST RUN OUTPUT

```
 Enter number of variables, number of squared terms:
-------------------------------------------------------------------------------
.........................
.........................                    M E R L I N - 3.0
.........................
.........................            D.G. Papageorgiou, I.E. Lagaris
.........................                 I.N. Demetropoulos
.........................                University of Ioannina
.........................                     G R E E C E
.........................
.........................              Email: merlin@nrt.cs.uoi.gr
.........................            Web: http://nrt.cs.uoi.gr/merlin
.........................
-------------------------------------------------------------------------------
The Merlin help file "HELP" is present.
The panel description file "PDESC" is present.
Use the "help" command to obtain on-line information.
-------------------------------------------------------------------------------
Number of terms:      2
Number of variables: 2
Estimated machine's accuracy: 1.E-15
Merlin uses "SUBROUTINE SUBSUM" as the objective function.

 ....   W A R N I N G   ....
...  Initialize variables  ...

    /\/\/\/\/\/\     Merlin is at your command !!!
POINT

    /\/\/\/\/\/\     Merlin is at your command !!!
GODFATHER

    /\/\/\/\/\/\     Merlin is at your command !!!
LMARGIN

    /\/\/\/\/\/\     Merlin is at your command !!!
RMARGIN

    /\/\/\/\/\/\     Merlin is at your command !!!
TITLE

Title is set to: "Rosenbrock's two-dimensional test function"

    /\/\/\/\/\/\     Merlin is at your command !!!
SHORTDIS

Title: Rosenbrock's two-dimensional test function
-------------------------------------------------------------------------------
Number of evaluations:    Function       Gradient       Hessian       Jacobian
Total:                           1              0             0              0
Since last reset:                1              0             0              0
-------------------------------------------------------------------------------
 Index  Name       Fix       Parameter value       Left margin   Right margin
     1) rho        .......  0.333000000000000          -1000.         100.0
     2) sigma      .......  -999.000000000000          -2000.         200.0
```

```
       Value .......      99822257.2967260

    /\/\/\/\/\/\     Merlin is at your command !!!
ROLL
   Iter: 1   Lower value:     99622535.1189260        Calls: 3 of 20
------------------------------------------------------------------------------
                            \/ -- \/ -- \/  Line-search
   Iter: 1   Lower value:    0.444889000000000        Calls: 11 of 20
------------------------------------------------------------------------------
                            \/ -- \/ -- \/  Line-search
   Iter: 2   Lower value:    0.435470882134324        Calls: 24
------------------------------------------------------------------------------
                ROLL: All function evaluations have been used
------------------------------------------------------------------------------
Function evaluations:   24
Iterations:             2
------------------------------------------------------------------------------

    /\/\/\/\/\/\     Merlin is at your command !!!
ANAL

    /\/\/\/\/\/\     Merlin is at your command !!!
HESSIAN

The Hessian matrix has been calculated. 1 Hessian call was used.

    /\/\/\/\/\/\     Merlin is at your command !!!
HESSIAN

The Choleski factorization of the Hessian matrix has been calculated.

The Hessian matrix was positive definite.

    /\/\/\/\/\/\     Merlin is at your command !!!
BFGS
   Iter: 1   Lower value:    0.323818045962521        Calls: 2 of 10
------------------------------------------------------------------------------
   Iter: 2   Lower value:    0.244359687078232        Calls: 3 of 10
------------------------------------------------------------------------------
   Iter: 3   Lower value:    0.148591410516486        Calls: 4 of 10
------------------------------------------------------------------------------
   Iter: 4   Lower value:    0.107527523391454        Calls: 6 of 10
------------------------------------------------------------------------------
   Iter: 5   Lower value:    9.194458231573392E-02    Calls: 8 of 10
------------------------------------------------------------------------------
   Iter: 6   Lower value:    7.210135854827365E-02    Calls: 9 of 10
------------------------------------------------------------------------------
   Iter: 7   Lower value:    4.289716800159461E-02    Calls: 10 of 10
------------------------------------------------------------------------------
                BFGS: All function evaluations have been used
------------------------------------------------------------------------------
Function evaluations:   10
Gradient evaluations:   8
Iterations:             7
------------------------------------------------------------------------------

    /\/\/\/\/\/\     Merlin is at your command !!!
```

```
JANAL

    /\/\/\/\/\/\     Merlin is at your command !!!
LEVE
   Iter: 1   Lower value:    4.289716800094422E-02    Calls: 1 of 20
-------------------------------------------------------------------------------
   Iter: 2   Lower value:    2.193020559607765E-02    Calls: 3 of 20
-------------------------------------------------------------------------------
   Iter: 3   Lower value:    9.784395597646321E-03    Calls: 4 of 20
-------------------------------------------------------------------------------
   Iter: 4   Lower value:    1.053350333714181E-03    Calls: 5 of 20
-------------------------------------------------------------------------------
   Iter: 5   Lower value:    1.298059206400674E-14    Calls: 6 of 20
-------------------------------------------------------------------------------
   Iter: 6   Lower value:                  0.        Calls: 7 of 20
-------------------------------------------------------------------------------
                  LEVE: The gradient criterion is satisfied
-------------------------------------------------------------------------------
Function evaluations:   8
Jacobian evaluations:   6
Iterations:             6
-------------------------------------------------------------------------------

    /\/\/\/\/\/\     Merlin is at your command !!!
SHORTDIS

Title: Rosenbrock's two-dimensional test function
-------------------------------------------------------------------------------
Number of evaluations:    Function       Gradient       Hessian       Jacobian
Total:                          43              8             1              6
Since last reset:               43              8             1              6
-------------------------------------------------------------------------------
 Index  Name       Fix       Parameter value       Left margin   Right margin
     1) rho        .......   1.00000000000000        -1000.         100.0
     2) sigma      .......   1.00000000000000        -2000.         200.0
        Value .......                  0.

    /\/\/\/\/\/\     Merlin is at your command !!!
STOP
---------------------------- Merlin run has ended ----------------------------
```