HAL open science

The module M1QN3

Jean Charles Gilbert, Claude Lemaréchal

▶ **To cite this version:**

Jean Charles Gilbert, Claude Lemaréchal. The module M1QN3. INRIA Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France. 2009. hal-04146287

HAL Id: hal-04146287

https://inria.hal.science/hal-04146287

Submitted on 29 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

The module M1QN3

Version 3.3 (October 2009)

Jean Charles Gilbert[†] and Claude Lemaréchal[‡]

1 Goal of the software **1**

2 Description of the method **2**
2.1 Brief description 3
2.2 Algorithmic details 4

3 Usage **5**
3.1 The solver 5
3.2 The simulator 11

4 Implementation remarks **12**
4.1 Calling sequence in direct communication 12
4.2 Calling sequence in reverse communication 13
4.3 More on some arguments 15
4.4 More on some output modes 15
4.5 Cold start and warm restart 16
4.6 Usage for very large scale problems 17

5 Frequently asked questions **19**
5.1 Preconditioning the problem 19
5.2 Dealing with fixed variables 19

References **20**

Index **20**

1 Goal of the software

The module `m1qn3` is a Fortran-77 software that can solve a large scale unconstrained minimization problem of the form:

$$\min \{f(x) : x \in \mathbb{R}^n\},$$

where the function $f : \mathbb{R}^n \to \mathbb{R}$ is supposed to be smooth (continuously differentiable). `M1qn3` has solved problems with n as large as 10^8. The solver requires evaluations of the function f and its gradient g, computed for a given *inner product*,

[†]INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France. Tel: (33) [0]1 39 63 55 24. E-mail: `Jean-Charles.Gilbert@inria.fr`.

[‡]INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 Montbonnot, France. Tel: (33) [0]4 76 61 52 02. E-mail: `Claude.Lemarechal@inria.fr`.

denoted by $\langle \cdot, \cdot \rangle$. The gradients are used to form a limited memory quasi-Newton approximation of the Hessian of f. The decrease in f is enforced by the Wolfe line-search.

The software can be used either with *direct* or *reverse communication* (these terms are used throughout the presentation of the software; the reader not familiar with this terminology can find clarification in sections 4.1 and 4.2).

In principle, `m1qn3` is intended for problems with a large number n of variables, but it can also be suitable for small or medium scale problems. The method used to solve the problem is a limited memory (or variable-storage) quasi-Newton method. This type of methods allows `m1qn3` to take advantage of the storage that is declared available in core memory. However, it cannot really be ascertained that, when more storage is used, the algorithm becomes more efficient and the decrease in the objective function f more effective (see section 4.3). The software has two *scaling modes*: the *Scalar Initial Scaling* mode (or SIS mode) and the *Diagonal Initial Scaling* mode (or DIS mode). The latter should normally give better results than the former, but it requires one additional vector of storage.

The module `m1qn3` includes of the following subroutines:

- `m1qn3`: main subroutine that has to be called and that calls `m1qn3a` after having structured the available memory,
- `m1qn3a`: optimization subroutine,
- `dd`, `dds`: subroutines that compute the descent direction,
- `mlis3`, `ecube`: subroutines realizing the line-search,
- `mupdts`, `ystbl`: special subroutines for very large problems.

On his part, the user has to supply:

- a program that calls the optimization module `m1qn3`,
- (used only in direct communication) a subroutine, called the *simulator*, and named `simul` by `m1qn3`, that computes the value of f and its gradient g (associated with the inner product $\langle \cdot, \cdot \rangle$) at a given point,
- a subroutine, named `prosca` by `m1qn3`, that computes the inner product $\langle u, v \rangle$ of two vectors of u, $v \in \mathbb{R}^n$,
- (used only in DIS mode) a subroutine, named `ctonb` by `m1qn3`, that gives the coordinates `y` in an orthonormal basis (for the inner product `prosca`) of a vector whose coordinates in the canonical basis of $\mathbb{R}^n$ are `x`,
- (used only in DIS mode) a subroutine, named `ctcab` by `m1qn3`, that does the operation converse to `ctonb`: it gives the coordinates `x` in the canonical basis of $\mathbb{R}^n$ of a vector whose coordinates in the orthonormal basis are `y`.

The standard distribution also provides the subroutines `simul_rc`, `euclid`, `ctonbe`, and `ctcabe`. The subroutine `simul_rc` plays the role of `simul` in reverse communication (it is an empty subroutine). The subroutines `euclid`, `ctonbe`, and `ctcabe` can play the role of `prosca`, `ctonb`, and `ctcab` inside `m1qn3`, when the Euclidean (or dot) product is used to compute the gradient of f, i.e., when $\langle u, v \rangle = u^\top v$.

2 Description of the method

2.1 Brief description

In SIS mode, `m1qn3` is an implementation of an algorithm proposed by Nocedal [3], while the DIS mode uses a diagonal preconditioner improving the algorithm. Both scaling modes are based on the same principles, which we now describe.

At each iteration k, $k \geq 1$, a descent direction d_k of f at the current iterate x_k is determined. This direction has the form $d_k = -W_k g_k$, where W_k is the current approximation of the inverse Hessian $\nabla^2 f(x_k)^{-1}$ of f at x_k and $g_k := \nabla f(x_k)$ is the gradient of f at x_k, computed for the inner product $\langle \cdot, \cdot \rangle$. Then, a step-size α_k is determined along the direction d_k by a line-search procedure (`mlis3`). The next iterate has the form: $x_{k+1} = x_k + \alpha_k d_k$.

When $k \geq m+1$, the approximation W_k of the inverse Hessian is obtained by doing m updates of a positive definite diagonal matrix D_k, using the inverse BFGS formula. For this, the following m pairs of vectors are used in order:

$$\{(y_i, s_i) : k - m \leq i \leq k - 1\}, \tag{2.1}$$

where $s_i := x_{i+1} - x_i$ is the step and $y_i := g_{i+1} - g_i$ is the change in the gradient. The matrix W_k is not stored in memory but instead the product $W_k g_k$ is formed from the pairs (2.1) by an efficient appropriate algorithm. The integer parameter m is determined by `m1qn3` in order to use as much as possible the storage declared available in core memory (see the meaning of the argument `ndz` in section 3).

The user has the possibility to improve the conditioning of the problem by using an adapted inner product $\langle \cdot, \cdot \rangle$, the one realized by the subroutine `prosca`. Let us denote by $\| \cdot \| := \langle \cdot, \cdot \rangle^{1/2}$ the *norm* associated with that inner product. Let us insist on the fact that this inner product must be the one used by the simulator to compute the gradient $g(x)$ of f at x. Indeed, for `m1qn3`, the derivative of f at x along the direction $h \in \mathbb{R}^n$ is given by:

$$f'(x) \cdot h = \langle g(x), h \rangle.$$

The BFGS formula used by `m1qn3` is adapted to this inner product (see section 2.2). Very often, the Euclidean (or dot) inner product, defined by

$$\langle x, y \rangle := x^\top y = \sum_{i=1}^{n} x_i y_i, \tag{2.2}$$

is used. In that case, the gradient of f is formed of the partial derivatives $\partial f / \partial x_i$, $i = 1, \ldots, n$.

The way of computing the starting matrix D_k depends on and characterizes the *scaling mode* of `m1qn3`. In the *Scalar Initial Scaling* (SIS) mode, D_k has the form $\delta_{k-1} I$, where I is the identity matrix and

$$\delta_{k-1} := \frac{\langle y_{k-1}, s_{k-1} \rangle}{\|y_{k-1}\|^2} \tag{2.3}$$

aims at giving W_k a good scaling. In the *Diagonal Initial Scaling* (DIS) mode, D_k is a diagonal matrix obtained from D_{k-1} by using a diagonal update formula (see

section 2.2 for the details). The first update (to obtain D_2) uses $\delta_1 I$ as starting matrix. The notion of diagonal matrix is defined with respect to the given inner product and the orthonormal basis $(e_i)_{1\le i\le n}$.

In DIS mode, an implicit reference is made to an orthonormal basis of $\mathbb{R}^n$ for the inner product $\langle\cdot,\cdot\rangle$. Let $(e_i)_{1\le i\le n}$ denote this basis. If $\langle x, y\rangle = x^\top L^\top L y$ with L nonsingular and if $(\hat{e}_i)_{1\le i\le n}$ is the canonical basis of $\mathbb{R}^n$, i.e., $\hat{e}_i = (0, \ldots, 0, 1, 0, \ldots, 0)$ with 1 in the ith position, you can take $e_i = L^{-1}\hat{e}_i$, for $1 \le i \le n$. Luckily, this basis needs not be stored, but you have to provide subroutines that give the coordinates in the $(e_i)_{1\le i\le n}$ basis of a vector whose coordinates in the $(\hat{e}_i)_{1\le i\le n}$ basis are given (subroutine `ctonb`) and vice-versa (subroutine `ctcab`). With the inner product and the orthonormal basis given above, `ctonb` must compute $\mathtt{y} := L\mathtt{x}$ and `ctcab` must compute $\mathtt{x} := L^{-1}\mathtt{y}$. If $\langle\cdot,\cdot\rangle$ is the Euclidean inner product, `ctonb` has just to copy `x` in `y` and `ctcab` will copy `y` in `x`.

If N and L denote respectively the number of iterations and the total number of step-size trials done during a particular run, then the subroutine `prosca` is called $4 + 7N + L$ times in the two scaling modes and the subroutines `ctonb` and `ctcab` are called respectively $3N$ and N times in DIS mode.

The methods and numerical results are given and discussed by Gilbert and Lemaréchal [1]. See also the numerical experiments by Liu and Nocedal [2].

2.2 Algorithmic details

The *BFGS formula* used in the software updates a matrix W to $\hat{W}$ using two vectors y and s in $\mathbb{R}^n$ by:

$$\hat{W} = \left(I - \frac{s\otimes y}{\langle y, s\rangle}\right) W \left(I - \frac{y\otimes s}{\langle y, s\rangle}\right) + \frac{s\otimes s}{\langle y, s\rangle}.$$

In this formula, $u \otimes v$ is the linear operator that to $d \in \mathbb{R}^n$ associates $\langle v, d\rangle u \in \mathbb{R}^n$. This makes the formula adapted to the inner product $\langle\cdot,\cdot\rangle$. We denote it by $\hat{W} = \overline{\mathrm{BFGS}}(W, y, s)$.

When $k \ge m+1$, given the diagonal matrix D_k (its computation is discussed below), the matrix W_k used to form the search direction $d_k = -W_k g_k$ is obtained as follows:

$$W_k^0 := D_k,$$
$$W_k^{i+1} := \overline{\mathrm{BFGS}}(W_k^i, y_{k-m+i}, s_{k-m+i}), \quad \text{for } 0 \le i \le m-1,$$
$$W_k := W_k^m.$$

Let us summarize this scheme by: $W_k := \overline{\mathrm{BFGS}}_{k-m}^{k-1}(D_k)$. When $1 \le k \le m$, the matrix W_k depends on how the optimizer is tuned by `imode(2)`. For a cold start (`imode(2)` $= 0$), $W_1 = I$ and, for $2 \le k \le m$, W_k is obtained by $W_k := \overline{\mathrm{BFGS}}_1^{k-1}(D_k)$. For a warm restart (`imode(2)` $= 1$), W_k is obtained by: $W_k := \overline{\mathrm{BFGS}}_{k-m}^{k-1}(D_k)$, in which (y_i, s_i) with index $i \le 0$ refers to a pair of a previous run, stored in the working zone `dz`.

As mentioned in section 2.1, SIS and DIS modes differ by the way the diagonal matrix D_k is computed. In SIS mode, $D_k = \delta_{k-1} I$ for $k \geq 2$, while in DIS mode, D_k is a diagonal matrix updated by the following formula:

$$D_{k+1}^{(i)} = \left(\frac{\langle D_k y_k, y_k \rangle}{\langle y_k, s_k \rangle D_k^{(i)}} + \frac{\langle y_k, e_i \rangle^2}{\langle y_k, s_k \rangle} - \frac{\langle D_k y_k, y_k \rangle \langle s_k, e_i \rangle^2}{\langle y_k, s_k \rangle \langle D_k^{-1} s_k, s_k \rangle (D_k^{(i)})^2} \right)^{-1}.$$

The subroutine `m1qn3a` uses the subroutine `mlis3` from MODULOPT for the line-search. The step-size α_k is determined to satisfy the following two Wolfe's conditions:

$$f(x_k + \alpha_k d_k) \leq f(x_k) + \omega_1 \alpha_k \langle g_k, d_k \rangle,$$

$$\langle g(x_k + \alpha_k d_k), d_k \rangle \geq \omega_2 \langle g_k, d_k \rangle.$$

It is necessary to have $0 < \omega_1 < 1/2$ and $\omega_1 < \omega_2 < 1$. We took $\omega_1 = 0.0001$ and $\omega_2 = 0.9$. These constants are set as `parameter` (`rm1` for ω_1 and `rm2` for ω_2) in `m1qn3a`.

3 Usage

In the description of the subroutines below, an argument flagged with (I) means that it is an *input* variable, which has to be initialized before calling `m1qn3`; an argument flagged with (O) means that it is an *output* variable, which only has a meaning on return from `m1qn3`; and an argument flagged with (IO) is an *input-output* argument, which has to be initialized and that has a meaning after the call to `m1qn3`. Arguments of the type (O) and (IO) are generally modified by `m1qn3` and therefore **should not be Fortran-77 constants**!

3.1 The solver

Here is the declaration of the solver `m1qn3`.

```
subroutine m1qn3 (simul, prosca, ctonb, ctcab, n, x, f, g,
                  dxmin, df1, epsg, normtype, impres, io,
                  imode, omode, niter, nsim, iz, dz, ndz,
                  reverse, indic, izs, rzs, dzs).
```

`simul`: In *direct communication*, `simul` is the name of the simulator inside `m1qn3`. The *simulator* is the subroutine that computes the value of the function f and the value of its gradient g at the current iterate. When `m1qn3` needs these values, it executes the instruction

```
call simul (...)
```

Hence the actual name of the simulator must be declared `external` in the program calling `m1qn3` and given as actual argument when calling `m1qn3`. For example, if the actual name of the simulator is `mysimul`, `m1qn3` will be called by

```
external mysimul
...
call m1qn3 (mysimul, ...)
```

See section 3.2, for more information on the simulator, in particular on its arguments.

In *reverse communication*, a simulator is not needed, since the user can organize the computation of f and g in an arbitrary manner (this is one of the advantage of reverse communication, see section 4.2). The empty subroutine `simul_rc`, provided in the standard distribution, should be used as actual argument in reverse communication (it is never used actually). The calling sequence is then

```
external simul_rc
...
call m1qn3 (simul_rc, ...)
```

Section 4.2 tells you more on the use of reverse communication.

`prosca`: Calling name of the subroutine that computes the inner product $\langle u, v\rangle$ of two vectors u and v of $\mathbb{R}^n$. This subroutine is supposed to have the following declaration statement:

```
subroutine prosca (n, u, v, ps, izs, rzs, dzs).
```

Its actual name has to be declared `external` in the program calling `m1qn3`. The arguments `n`, `izs`, `rzs` and `dzs` have the same meaning as below. The arguments `u` and `v` are two `double precision` arrays of dimension n. The inner product of these vectors is supposed to be returned in the `double precision` variable `ps`.

`ctonb` (used only in DIS mode): Calling name of the subroutine that gives the coordinates $(y_i)_{1\le i\le n}$ in the basis $(e_i)_{1\le i\le n}$ of a vector whose coordinates in the basis $(\hat{e}_i)_{1\le i\le n}$ are $(x_i)_{1\le i\le n}$ (see section 2.1). This subroutine is supposed to have the following declaration statement:

```
subroutine ctonb (n, x, y, izs, rzs, dzs).
```

Its actual name has to be declared `external` in the program calling `m1qn3`. The arguments `n`, `izs`, `rzs` and `dzs` have the same meaning as below. The arguments `x` and `y` are two `double precision` arrays of dimension n.

`ctcab` (used only in DIS mode): Calling name of the subroutine that does the operation reverse to the one done by `ctonb`, so that calling `ctonb` and next `ctcab` will not change the value of `x`: it gives the coordinates $(x_i)_{1\le i\le n}$ in the basis $(\hat{e}_i)_{1\le i\le n}$ of a vector whose coordinates in the basis $(e_i)_{1\le i\le n}$ are $(y_i)_{1\le i\le n}$. It is supposed to have the following declaration statement:

```
subroutine ctcab (n, y, x, izs, rzs, dzs).
```

Its actual name has to be declared `external` in the program calling `m1qn3`. The arguments are similar to those of `ctonb`.

`n` (I): Positive `integer` variable. It gives the dimension n of the problem.

`x` (IO): `Double precision` array of dimension n. On entry, it is supposed to be the value x_1 of the initial point. On return, it is the value of the final point calculated by `m1qn3`.

`f` (IO): `Double precision` variable. On entry, it is supposed to be the value of the function f at the initial point x_1. This value will be obtained by calling the simulator `simul` before calling `m1qn3`. On return, it is the value of f at the final point.

`g` (IO): `Double precision` array of dimension n. On entry, it is supposed to be the value of the gradient of f at the initial point x_1. This value will be obtained by calling the simulator `simul` before calling `m1qn3`. On return with `omode` $= 1$ (see below), it is the value of the gradient of f at the final point. For other output modes, the value of g is undetermined.

`dxmin` (I): Positive `double precision` variable. This argument gives the resolution in x for the l_∞ norm: two points whose distance in the *sup*-norm is less than `dxmin` will be considered as indistinguishable by the optimizer (in fact by `mlis3`).

`df1` (I): Positive `double precision` variable. This argument gives an estimation of the expected decrease in f during the first iteration. See section 4.3 for more details.

`epsg` (IO): `Double precision` variable, with value in $]0,1[$ on entry. On entry, `epsg` gives the precision of the stopping criterion that is based on the norm of the gradient. Indeed, the optimizer considers that the convergence has been obtained at x_k and stops with `omode` $= 1$, if

$$\epsilon_k := \frac{\|g_k\|}{\|g_1\|} < \texttt{epsg},$$

where, g_1 and g_k are the gradients of f at the initial and current points respectively and $\|\cdot\|$ is the *norm* specified by the argument `normtype`. If such is the case, `epsg` $= \epsilon_k$ on return.

`normtype` (I): Variable of type `character*3` that specifies the *norm* $\|\cdot\|$ that is used to test optimality (see the argument `epsg`). It can be one of the following strings:

– '`two`' denotes the Euclidean or ℓ_2 norm ($\|v\| = (\sum_i v_i^2)^{1/2}$),
– '`sup`' denotes the sup or ℓ_∞ norm ($\|v\| = \max_i |v_i|$),
– '`dfn`' denotes the norm $\|\cdot\|$ associated with the scalar product defined in the user-supplied subroutine `prosca` ($\|v\| = \langle v, v\rangle^{1/2}$).

It is this norm that is used to print the gradient norms.

`impres` (I): `Integer` variable that controls the outputs on channel `io`.

$= 0$: No print.

≥ 1: Initial and final printouts, error messages.

≥ 3: One line of printout per iteration that gives: the index k of the iteration going from the point x_k to the point x_{k+1}, the number of time the simulator has been called, the value $f(x_k)$ of the objective function and the directional derivative $\langle g_k, d_k \rangle$.

≥ 4: Printouts from `mlis3` during the line-search: see the write-up on `mlis3` in MODULOPT library.

≥ 5: Some more printouts at the end of iteration k: information on the matrix update (the factor δ_k for the SIS mode, see (2.3); the fitting factor and the look of the diagonal initial scaling matrix D_k for the DIS mode), the value ϵ_{k+1} of the stopping criterion, and the angle between $-g_k$ and the search direction d_k.

`io` (I): `Integer` variable that will be taken as the channel number for the outputs, i.e., outputs are written by:

```
write (io,...) ...
```

`imode` (I): `Integer` array of dimension 3 that specifies the input mode of `m1qn3` and tunes its behavior. The following values are meaningful.

`imode(1)` determines the *scaling mode* of `m1qn3`.

`M1qn3` will run in DIS mode if `imode(1)` $= 0$, meaning that the matrix W_k is formed by updating m times a dynamically updated diagonal matrix D_k.

`M1qn3` will run in SIS mode if `imode(1)` $= 1$, meaning that the matrix W_k is formed by updating m times a variable multiple of the identity matrix.

The DIS mode is generally more efficient than the SIS mode and is recommended.

`imode(2)` specifies the *starting mode* of `m1qn3`.

A *cold start* is performed if `imode(2)` $= 0$: the first descent direction is then $-g_1$. If you have not stored the appropriate information in `iz` and `dz` in a previous run, set `imode(2)` to 0.

A *warm start* is performed if `imode(2)` $= 1$: the first descent direction is $-W_1 g_1$, where W_1 is the matrix formed from the (y, s) pairs (and the diagonal matrix, in DIS mode) stored in the `double precision` working zone `dz`. The `integer` working zone `iz` helps `m1qn3` decoding the information in `dz`. This option can only be used if the appropriate data has been stored in `iz` and `dz` in a previous run.

See section 4.5 for more details.

`imode(3)` specifies in direct communication when the simulator has to be called with `indic` $= 1$ or, similarly in reverse communication, when `m1qn3` returns to the calling subroutine with `indic` $= 1$. As explained in section 3.2, when called with `indic` $= 1$, the simulator can do anything but changing the value of its arguments n, x, f, and g (a similar rule must be respected by the calling program in reverse communication). The simulator can take this opportunity to do any regular computation, printout, or plot.

When `imode(3)` $= 0$, the simulator is never called with `indic` $=$

1. When `imode(3)` > 0, the simulator is called with `indic` $= 1$ every `imode(3)` iteration(s), starting at iteration 1. Hence, when `imode(3)` $= 1$, the simulator is called every iteration, which is a way for the simulator of being informed when a new iteration starts (recall that there may be an unpredicted number of simulations per iteration).

`omode` (O): `Integer` variable that specifies the output mode of `m1qn3`. The following values are meaningful.

- $= 0$: The simulator asks to stop by returning the value `indic` $= 0$.
- $= 1$: This is the normal way of stopping for `m1qn3`: the test on the gradient is satisfied (see the meaning of `epsg`).
- $= 2$: One of the input arguments is not well initialized. This can be:
  - `n` ≤ 0, `niter` ≤ 0, `nsim` ≤ 0, `dxmin` ≤ 0.0 or `epsg` $\notin]0, 1[$,
  - `ndz` $< 5n + 1$ (in SIS mode) or `ndz` $< 6n + 1$ (in DIS mode): not enough storage in memory,
  - the contents of `iz` is not correct for a warm restart,
  - the starting point is almost optimal (the norm of the initial gradient is less than 10^{-20}).
- $= 3$: The line-search is blocked on `tmax` $= 10^{20}$ (see section 4.4 and the documentation on `mlis3` in MODULOPT library).
- $= 4$: The maximal number of iterations is reached.
- $= 5$: The maximal number of simulations is reached.
- $= 6$: Stop on `dxmin` during the line-search (see section 4.4).
- $= 7$: Either $\langle g, d\rangle$ is nonnegative or $\langle y, s\rangle$ is nonpositive (see section 4.4).

For additional information and comments, see section 4.

`niter` (IO): Positive `integer` variable. On entry, it gives the maximal number of iterations accepted. On return, `niter` is equal to the number of iterations really done.

`nsim` (IO): Positive `integer` variable. On entry, it gives the maximal number of simulations accepted. On return, `nsim` is equal to the number of simulations really done.

`iz` (IO): `Integer` array of dimension 5. It is the address of a working array for `m1qn3`.

`dz` (IO): `Double precision` array of dimension `ndz`. It is the address of a working array for `m1qn3`.

`ndz` (I): Positive `integer` variable. It gives the dimension of the working area `dz` (see also section 4.3).

In SIS mode, `m1qn3` needs a working area formed of at least 3 vectors of dimension n (d_k, g_k and an auxiliary vector) and it needs for each update one scalar and two vectors: $y\langle y, s\rangle^{-1/2}$ and $s\langle y, s\rangle^{-1/2}$ (see section 2.1). Therefore, if m is the desired number of updates for forming the matrix W_k, it is necessary to have:

$$\texttt{ndz} \geq 3n + m(2n + 1).$$

In fact, the number m of updates is determined by `m1qn3`, which takes

$$m = \left\lfloor \frac{\texttt{ndz} - 3n}{2n+1} \right\rfloor, \tag{3.1}$$

where, $\lfloor \cdot \rfloor$ refers to the floor operator ($\lfloor x \rfloor = i$, when $i \leq x < i+1$ and $i \in \mathbb{N}$). This number has to be greater than or equal to 1. Therefore, if `ndz` is less than $5n+1$, `m1qn3` stops with `omode` $= 2$.

On the other hand, in DIS mode, `m1qn3` needs an additional vector of dimension n for storing D_k. So, take `ndz` $\geq 4n + m(2n+1)$ and $m \geq 1$. Then, `m1qn3` will set:

$$m = \left\lfloor \frac{\texttt{ndz} - 4n}{2n+1} \right\rfloor. \tag{3.2}$$

The above computation of m from `ndz` is irrelevant when `m1qn3` is used as described in section 4.6.

`reverse` (IO): `Integer` variable that is used to specify whether direct or reverse communication is desired and, in the latter case, to communicate with the *call loop* (see section 4.2 and figure 4.2 for the program flow chart on reverse communication).

On entry in `m1qn3`, `reverse`

- < 0: implies that `m1qn3` will stop immediately using the instruction `stop`, to prevent entering an infinite call loop in reverse communication, due to the fact that the calling program has not left the call loop when `m1qn3` returns a negative value,
- $= 0$: indicates that `m1qn3` has to work in *direct communication*,
- $= 1$: indicates that `m1qn3` has to work in *reverse communication*.

On return from `m1qn3` in reverse communication, `reverse`

- < 0: when `m1qn3` has terminated, in which case the call loop must be interrupted,
- $= 1$: the call loop must be pursued.

See sections 4.1 and 4.2 for more details.

`indic` (IO) In reverse communication, `m1qn3` returns to the calling program each time it needs information on the problem (function/gradient values). The calling program must then compute what is required by `m1qn3` (sometimes using a simulator, like in direct communication) and call back the solver in what can be viewed as a *call loop* (see section 4.2 and figure 4.2 for the program flow chart on reverse communication). The parameter `indic` normalizes the communication between `m1qn3` and the part of the program computing function and gradient.

On entry in `m1qn3`, the calling program can send a message to `m1qn3`, using the following values of `indic`:

- < 0: the computation of f and g required by `m1qn3` on its last return was not possible at the given x; in that case, `m1qn3` divides the step-size by 10, before calling the simulator again; this feature can be used when implicit constraints are present, i.e., strict inequality constraints or inequalities that are known to be inactive at the solution; by no way this feature can

handle inequality constraints that are active (satisfied with equality) at the solution;

$= 0$: `m1qn3` has to stop, for example because some events that `m1qn3` cannot understand (not in the field of optimization) has occurred;

> 0: the required computation has been done.

On return from `m1qn3`, the optimization solver sends a message depending on the values of `indic`:

$= 1$: means that the calling program can do anything except changing the values of `indic`, `n`, `x`, `f`, and `g`; this value of `indic` is used by `m1qn3` every `imode(3)` iteration(s), when `imode(3)` > 0, and never, when `imode(3)` $= 0$;

$= 4$: means that the calling program has to compute f and g, to put them in `f` and `g`, and to call back `m1qn3`.

Note that these values of `indic` have the same meanings as those used in the communication with the simulator (see section 3.2).

`izs` (I), `rzs` (I), `dzs` (I) (used only in direct communication): Addresses of `integer`, `real`, and `double precision` working areas respectively. `M1qn3` does not use them but passes their addresses to the subroutines `simul`, `prosca`, `ctonb`, and `ctcab` so that they can be used in these subroutines.

3.2 The simulator

In direct communication, the *simulator* is the subroutine that computes the value of the function f and the value of its gradient g at a given arbitrary point x. `M1qn3` assumes that the subroutine is defined by the following statement:

```
subroutine simul (indic, n, x, f, g, izs, rzs, dzs).
```

The name `simul` of that subroutine can be changed. Its actual name must be declared `external` in the program calling `m1qn3` and must be given as the first argument of `m1qn3` (see section 3.1). Here is a description of the arguments of the simulator.

`indic` (IO) `integer` value that monitors the communication between `m1qn3` and the simulator.

On entry, `m1qn3` sends a message to the simulator by using one of the following values of `indic`:

$= 1$: means that the simulator can do anything except changing the values of `indic`, `n`, `x`, `f`, and `g`; this value of `indic` is used by `m1qn3` every `imode(3)` iteration(s), when `imode(3)` > 0, and never, when `imode(3)` $= 0$;

$= 4$: means that the simulator has to compute f and g, and to put their values in `f` and `g`, respectively.

On return, the simulator can send a message to `m1qn3` by using one of the following values of `indic`:

< 0: the computation of f and g required by `m1qn3` on its last return was not possible at the given x; in that case, `m1qn3` divides the step-size by 10, before calling the simulator again; this feature can be used when implicit constraints are present, i.e., strict inequality constraints or inequalities that are known to be inactive at the solution; by no way this feature can handle inequality constraints that are active (satisfied with equality) at the solution;

$= 0$: `m1qn3` has to stop, for example because some events that `m1qn3` cannot understand (not in the field of optimization) has occurred;

> 0: the required computation has been done.

`x` (I): `double precision` array of dimension n; it is the value of the current iterate (when `indic` $= 1$ on entry) or the point x at which f and g have to be computed (when `indic` $= 4$ on entry);

`f` (O): `double precision` variable that will receive the computed value of $f(x)$ when `indic` $= 4$ on entry.

`g` (O): `double precision` array of dimension n that will receive the computed value of $g(x)$ when `indic` $= 4$ on entry.

`izs`, `rzs`, `dzs` addresses of `integer`, `real`, and `double precision` working areas respectively defined in the program calling `m1qn3`, which does not use them.

4 Implementation remarks

4.1 Calling sequence in direct communication

With a *direct communication* protocol, the relevant part of the code using `m1qn3` as an optimization tool must be organized as in the flow chart of figure 4.1. In this

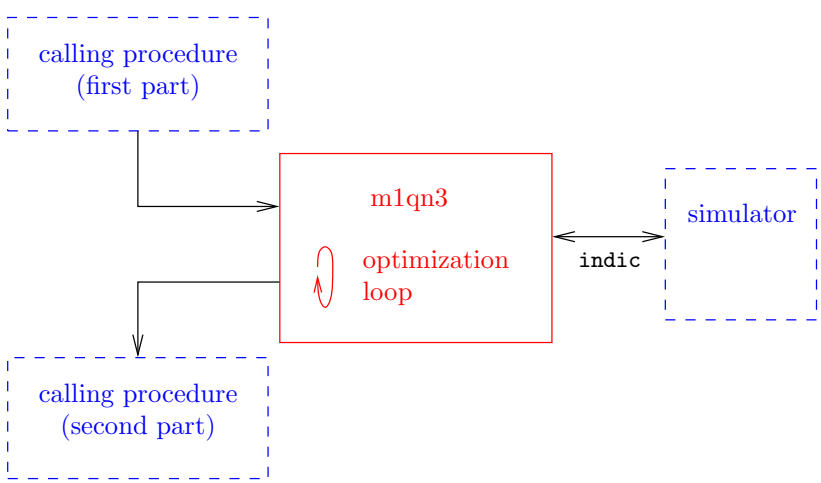

Figure 4.1: Direct communication flow chart

scheme, the user program calls `m1qn3` with `reverse` $= 0$ and recovers the lead only when the optimization routine has completed its job. On the other hand, each time `m1qn3` needs information on the problem to solve, it calls the simulator subroutine. In figure 4.1, for clarity, we have represented by dashed blue boxes the parts of the program that have to be written by the user of the optimization software; the single red box represents `m1qn3`.

The instructions preceding the call to `m1qn3` will include the following items:

1. the declaration in `external` of the following names: the name of the simulator (named `simul` by `m1qn3`), the name of the subroutine doing the inner product (named `prosca` by `m1qn3`) and the name of the subroutines doing the change of basis (named `ctonb` and `ctcab` by `m1qn3`);
2. the calculation of the starting point x_1;
3. the call to the simulator to compute f and g at x_1;
4. the calculation of `df1` and initialization of `dxmin`, `epsg`, `impres`, `io`, `imode`, `niter`, `nsim`, `ndz`, and `reverse` (set to `0`);
5. the reading of values for `iz` and part of `dz` from the appropriate area or device, if a warm restart is performed (`imode`(2) $= 1$);
6. the call to `m1qn3`.

4.2 Calling sequence in reverse communication

With a *reverse communication* protocol, the relevant part of the code using `m1qn3` as an optimization tool must be organized as in the flow chart of figure 4.2. Here also,

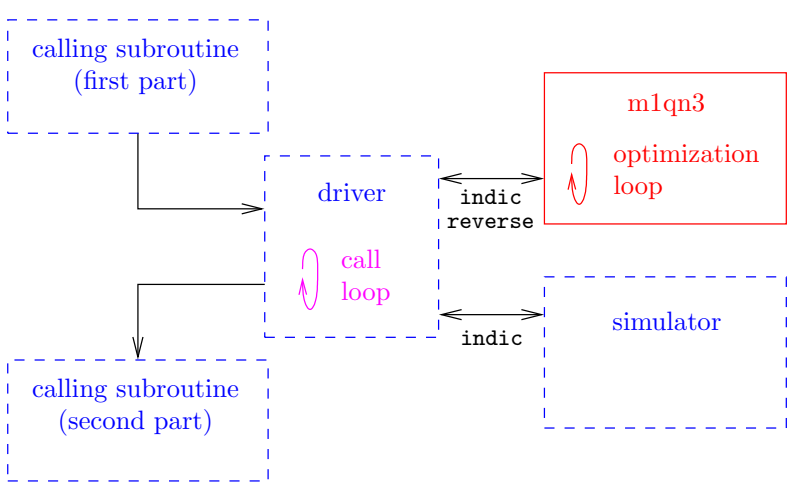

Figure 4.2: Reverse communication flow chart

the dashed blue boxes represent the parts of the program that have to be written by the user of the optimization software, while the single red box represents the `m1qn3` solver. In this scheme, the user program must contain a *call loop*, in a part of the

program sometimes called the *driver*. At each turn of this loop, `m1qn3` is called and is asked to do a single iteration. Once the optimization routine needs information on the problem, it returns to the calling program, requiring the computation of the function and gradient by setting `indic` $= 4$ (and `reverse` $= 1$). Then the user program computes f and g, loops, and calls back `m1qn3`, using also `indic` to inform the optimizer on the realization of the required computation or to ask it to stop[1]. When the run is finished (for some reason specified by `omode`), `m1qn3` returns with `reverse` $= -1$, in which case the call loop must be interrupted (if `m1qn3` is called back with `reverse` < 0, the solver abruptly stops using the instruction `stop` to avoid a possible infinite loop).

The two main advantages of *reverse communication* are that there is no need to write a simulator and that it is easier to call `m1qn3` from a language different from Fortran-77.

The instructions preceding the driver include the following items:

1. the declaration in `external` of the following names: the name of the empty simulator `simul_rc`, the name of the subroutine doing the inner product (named `prosca` by `m1qn3`) and the name of the subroutines doing the change of basis (named `ctonb` and `ctcab` by `m1qn3`);
2. the calculation of the starting point x_1;
3. the computation of f and g at x_1;
4. the calculation of `df1` and initialization of `dxmin`, `epsg`, `impres`, `io`, `imode`, `niter`, `nsim`, and `ndz`;
5. the reading of values for `iz` and part of `dz` from the appropriate area or device, if a warm restart is performed (`imode`(2) $= 1$);
6. setting `reverse` $= 1$ to indicate that reverse commnunication is required.

In Fortran-77, the *call loop* in the driver can be written as follows:

```
      external simul_rc, ...
      ...
      reverse = 1
    1 continue
        call m1qn3 (simul_rc,...,x,f,g,...,reverse,indic,...)
        if (reverse.lt.0) goto 2
        call simul (indic,n,x,f,g)
        goto 1
    2 continue
```

Above, we have assumed that the computation of f and g can be done in a subroutine called `simul`.

[1]When the user decides to stop the iterations, he/she could bypass an additional call to `m1qn3` with `indic` $= 0$. This, however, will spoil a subsequent call to `m1qn3` (for a cold start or a warm restart). It is therefore unadvisable.

4.3 More on some arguments

The value of `df1` is used to obtain an estimate of the step-size at the first iteration, which is taken equal to $2(\texttt{df1})/\|g_1\|^2$. A good value for `df1` can be the total decrease in f up to its minimum or some fraction of this value. As a rule, a too large value is not dangerous: the step-size is reduced by `mlis3` in a few internal iterations. On the other hand, an excessively small value of `df1` could, due to rounding error, force `m1qn3` to stop with `omode` $= 6$ at the first iteration.

If the storage is sufficiently large, it is a good idea to choose `ndz` large enough to have a large number m of updates: see Formulæ (3.1) and (3.2). It seems reasonable to take $m \geq 2$. However, numerical experiments (see [1] and [2]) have shown that it is difficult to take full advantage of a large number of updates, especially since the CPU time is increasing with m. A good compromise should be obtained by taking m between 5 and 10.

4.4 More on some output modes

The output mode `omode` $= 3$ is very unlikely. On the one hand, the matrix W_k is usually well scaled when the updates start with the identity matrix multiplied by the factor given by (2.3). And, on the other hand, as soon as the number of updates is larger than 2 or 3 and $k \geq m$, the BFGS updates also scale W_k. Therefore, the unit step-size is usually accepted by `mlis3`.

The output mode `omode` $= 6$ can have various origins.

- It can come from a mistake in the calculation of the gradient. This is likely to be the case when the solver stops with that output mode after very few iterations.
- If the number of step-size trials at the last iteration is small, this can mean that `dxmin` has been chosen too large. In that case, decreasing `dxmin` should have the clear effect of increasing the number of iterations.
- It can also come from rounding error in the simulator. The precision on the optimal solution that can be reached by the solver depends indeed on the precision of the simulator (the amount of rounding errors made in the computation).

It is worth noting that an inaccurate gradient may lead to a direction of search d_k that is no longer a descent direction of f at x_k. Indeed, `m1qn3` allows itself to build search directions d_k making with $-g_k$ an angle close to $90°$. This fact is essential to have good convergence properties. Now, if the gradient is roughly calculated, the angle between the true negative-gradient and d_k could be larger than $90°$, in which case d_k will not be a descent direction and an output with `omode` $= 6$ will occur. Therefore, it is important to compute the gradient carefully; in particular, a finite-difference gradient is likely to yield less precise solution than a calculated gradient.

The output mode `omode` $= 7$ is very unlikely. It can only be due to rounding error.

4.5 Cold start and warm restart

By setting the input variable `imode` appropriately, you can get a *cold start* (`imode(2)` = 0) or a *warm restart* (`imode(2)` = 1) of `m1qn3`. A cold start of the optimizer is the usual way of using it. For example, the first time you use `m1qn3`, you can only call it with `imode(2)` = 0. Now, when several optimization runs are done sequentially for *similar* problems, it can be useful to use the information stored in the pairs (y, s) and possibly in the preconditioning diagonal matrix D of a previous run to improve the efficiency of the first iterations of the current run. This is what we mean by doing a warm restart.

All the information needed by `m1qn3` to build the preconditioner W_k is contained in the working zones `iz` and `dz`. As we have just said, normally, the relevant information is placed in these vectors by a previous call to `m1qn3`. This can be done in the same run of a main program (sequence of calls to `m1qn3`) or by storing the data on an auxiliary device between two runs. Note that the 5 `integer`s in `iz` are crucial for a correct warm restart, but that only a part of the elements in `dz` are meaningful to this respect, namely the first $2nm$ elements in SIS mode and the first $n(2m+1)$ elements in DIS mode.

For example, the following piece of program

```
epsg = 1.e-4
imode(1) = 0
imode(2) = 0
niter = 100
nsim = 100
call m1qn3 (mysimul,euclid,ctonbe,ctcabe,n,x,f,g,dxmin,
            df1,epsg,normtype,impres,io,imode,omode,niter,
            nsim,iz,dz,ndz,reverse,indic,izs,rzs,dzs)
```

will generate the same iterates as the next piece of program, where the optimization phase is split in two parts, 5 iterations for the first call and 95 for the second, which starts at the situation reached by the first call (`imode(2)` = 1):

```
epsg = 1.e-4
imode(1) = 0
imode(2) = 0
niter = 5
nsim = 100
call m1qn3 (mysimul,euclid,ctonbe,ctcabe,n,x,f,g,dxmin,
            df1,epsg,normtype,impres,io,imode,omode,niter,
            nsim,iz,dz,ndz,reverse,indic,izs,rzs,dzs)
epsg = 1.e-4/epsg
imode(2) = 1
niter = 95
nsim = 100
call m1qn3 (mysimul,euclid,ctonbe,ctcabe,n,x,f,g,dxmin,
            df1,epsg,normtype,impres,io,imode,omode,niter,
```

```
nsim,iz,dz,ndz,reverse,indic,izs,rzs,dzs)
```

In case of reverse communication, the statement `reverse` = 1 must be inserted before the second call to `m1qn3`.

Observe that, for the second run, the required relative gradient norm, specified by `epsg`, has been adapted to the one obtained at the end of the first run (we assume there that `epsg` $\neq 0$ at the end of the first run), so that the two runs (with and without a warm restart) will stop at the same iteration. Now, if the iterates generated in the two runs are identical, it is normal, however, to get non identical printouts since the relative gradients differ.

Note that, in the last case, we have made no restrictions on the number m of updates. In particular, if $m > 5$, the number of pairs stored in `dz` by the first call is less than the number of updates. `M1qn3` can manage this situation correctly. On the other hand, it is imperative that the run of `m1qn3` filling the vectors `iz` and `dz` should be done with the same number of updates and the same scaling mode (SIS or DIS mode) as the current run. Otherwise, `m1qn3` terminates with `omode` = 2 and the error message:

```
inconsistent warm restart
```

4.6 Usage for very large scale problems

Some dispositions have been taken so that, if you have experience with the code and understand what it does, you can adapt it to the optimization of very large scale problems. The idea is to allow you to store yourself the pairs (2.1) on an auxiliary memory (for instance on a disk or any other device) instead of keeping them in core memory. To achieve this goal, you have to write and use new versions of the subroutines `mupdts` and `ystbl` (`dystbl` in double precision), which are provided in the standard distribution. There are two reasons why we do not make this usage of `m1qn3` more friendly. First, it is definitely non standard and it should be useful only in case of a huge number of variables. Secondly, the communication with external device is, in any case, installation dependent, which requires writing appropriate subroutines for this.

Here are the declaration statements of the two subroutines:

```
subroutine mupdts (sscale, inmemo, n, m, ndz),
```

```
subroutine ystbl (store, ybar, sbar, n, j).
```

In `mupdts`, `sscale`, `n` and `ndz` are input arguments and `inmemo` and `m` are output arguments. In `ystbl`, `store`, `n` and `j` are input arguments and `ybar` and `sbar` are input/output arguments. The rest of this section describes what these subroutines should look like.

In the standard distribution, subroutine `mupdts` computes the number `m` of updates from formula (3.1) (if `sscale` = `.true.`, i.e., SIS mode) or (3.2) (if `sscale` = `.false.`, i.e., DIS mode) and informs the optimizer that the pairs (2.1) are to be stored in core memory (`inmemo` = `.true.`). If you want to store yourself the pairs (2.1), the length `ndz` of the working area `dz` will be at least $6n + m$, when calling `m1qn3`. Note that in this case, the number of updates can no longer be deduced from the value of `ndz`. Therefore, your version of `mupdts` will set the value of `m` and will set `inmemo` = `.false.`.

Subroutine `ystbl` is device dependent and will be written accordingly. Its role is to store (when `store` = `.true.`) or to fetch (when `store` = `.false.`) a pair of vectors (y, s) in or from a certain *logical* position j in an auxiliary memory containing m (= number of updates specified by the subroutine `mupdts`) such positions. Position j will always be between 1 and m, inclusive. A pair is always stored before being fetched. If a pair has been stored at position j, `m1qn3` expects that a fetch with the same index j will correspond to the same pair. There is however no link between the logical position specified by the index j and the index i of the pair (y_i, s_i). Note that subroutine `ystbl` is not used when `inmemo` = `.true.`.

Just as an example, suppose that you want to take care yourself of the storage of the pairs (2.1) and to use the core memory as auxiliary memory. Suppose also that $m = 10$ updates are desired. Then subroutine `mupdts` should look like this:

```
      subroutine mupdts (sscale,inmemo,n,m,ndz)
      logical sscale,inmemo
      integer n,m,ndz
      m = 10
      inmemo = .false.
      return
      end
```

On the other hand, if $n = 1000$ and if the 10 pairs of vectors (y, s) are stored in the arrays `yarray` and `sarray`, then subroutine `ystbl` should look like this:

```
      subroutine ystbl (store,ybar,sbar,n,j)
c
      logical store
      integer n,j
      double precision ybar(n),sbar(n)
c
      integer i
      double precision yarray,sarray
      common /ysarray/yarray(1000,10),sarray(1000,10)
c
      if (store) then
        do i = 1,n
          yarray(i,j) = ybar(i)
          sarray(i,j) = sbar(i)
        end do
```

```
      else
        do i = 1,n
          ybar(i) = yarray(i,j)
          sbar(i) = sarray(i,j)
        end do
      end if
c
      return
      end
```

This subroutine stores (when `store` = `.true.`) or fetches (when `store` = `.false.`) a pair of vectors $(y, s) = (\texttt{ybar}, \texttt{sbar})$ of dimension `n` at or from position `j` in the "auxiliary" memory represented by the `common` block `/ysarray/`. Variables `store`, `n` and `j` are input variables and `ybar` and `sbar` are input/output variables, depending on the value of `store`.

For a warm restart (`imode(2)` = 1, see section 4.5), you need to give `m1qn3` correct values in the 5 locations of `iz` and, for the DIS mode, in the first n locations in `dz`, which is the normal place for the diagonal preconditioner (for the SIS mode, the values in `dz` are irrelevant). Appropriate saves will be done accordingly at the end of the previous run.

5 Frequently asked questions

5.1 Preconditioning the problem

5.2 Dealing with fixed variables

Suppose that I have a simulator for an optimization problem defined in the variables $x = (x_1, \ldots, x_n)$ and that I want to solve the same problem, but now with some of the variables fixed. How can I proceed?

We see two possibilities. Below, we have denoted by `simul` the name of the original simulator, defined in all the variables $x_1, \ldots, x_n$.

1. A solution that always works is to write a cover simulator, say `simul_cover`, which is called by `m1qn3` and that calls the original simulator `simul`.

   To be specific, if `nfree` is the number of free variables and if their indices in $\{1, \ldots, n\}$ are given by `free(`i`)`, for $i = 1, \ldots,$ `nfree`, the cover simulator can look like below.

```
      subroutine simul_cover
     &    (indic, nfree, xfree, f, gfree, izs, rzs, dzs)
      ...
      do i = 1,nfree
        x(free(i)) = xfree(i)
      end do
```

```
call simul (indic, n, x, f, g, izs, rzs, dzs)
do i = 1,nfree
  gfree(i) = g(free(i))
end do
return
end
```

2. If the inner product used to compute the gradient (specified by the subroutine `prosca`) is the Euclidean inner product (2.2), then one can proceed by zeroing the components of the gradient corresponding to the fixed variables. Then, `m1qn3` will not modified the fixed variables and the free variables will be modified like in solution 1. This technique is valid in both scaling modes of `m1qn3` (`imode(1)` $= 0$ or 1). Be careful with a warm restart (`imode(2)` $= 1$): the (y, s) pairs stored during the previous run must have been obtained with the same fixed variables.

   To be more specific, if `nfixed` is the number of fixed variables and if their indices in $\{1, \ldots, n\}$ are given by `fixed(`i`)`, for $i = 1, \ldots,$ `nfixed`, the modifications to bring to the original simulator `simul` could be as below.

```
subroutine simul (indic, n, x, f, g, izs, rzs, dzs)
...
do i = 1,nfixed
  g(fixed(i)) = 0.d0
end do
return
end
```

References

[1] J.Ch. Gilbert, C. Lemaréchal (1989). Some numerical experiments with variable storage quasi-Newton algorithms. *Mathematical Programming*, 45, 407-435. 4, 15

[2] D.C. Liu, J. Nocedal (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45, 503-528. 4, 15

[3] J. Nocedal (1980). Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35/151, 773-782. 3

Index

argument
- `ctcab`, 6
- `ctonb`, 6
- `df1`, 7
- `dxmin`, 7
- `dz`, 9
- `dzs`, 11, 12
- `epsg`, 7
- `f`, 7, 12
- `g`, 7, 12

imode, 8
impres, 7
inmemo, 18
io, 8
iz, 9
izs, 11, 12
m, 18
n, 7
ndz, 9, 18
niter, 9
normtype, 7
nsim, 9
omode, 9
prosca, 6
reverse, 10
rzs, 11, 12
simul, 5
sscale, 18
store, 18
x, 7, 12

BFGS formula, 3, **4**

call loop, 10, **13**, 14
cold start, 4, 8, **16–17**
communication
direct, 5, 10, 12–13
reverse, 6, 10, 13–14

DIS, *see* scaling mode

fixed variable, 19–20
frequently asked questions (FAQ), 19–20

I (input argument), 5
inner product, 1–4
Euclidean, **3**, 20
prosca, **6**, 13, 14
IO (input-output argument), 5

norm, 3, 7

O (output argument), 5

scaling mode, 2, **3**, 8
diagonal initial scaling (DIS), 2, **3**
scalar initial scaling (SIS), 2, **3**
simulator, 5, 11, **11–12**
SIS, *see* scaling mode
stop, 10
subroutine
ctcab, 2, **4**, **6–7**
ctcabe, 2
ctonb, 2, **4**, **6**
ctonbe, 2
dd, 2
dds, 2
ecube, 2
euclid, 2
m1qn3a, 2
mlis3, 2
mupdts, 2, **17–19**
prosca, 2
simul, 2, 11, 20
simul_cover, 19
simul_rc, 2, 6
ystbl, 2, **17–19**

warm restart, 4, 8, 9, 13, 14, **16–17**, 19, 20