MINIMIZATIONS IN THE CYCLE 43 OF ARPEGE/IFS.

YESSAD K. (METEO-FRANCE/CNRM/GMAP/ALGO)

February 23, 2016

Abstract:

This documentation describes the minimizations done in ARPEGE/IFS, for example in the 3DVAR and 4DVAR assimilation. First some algorithmic aspects are recalled about minimizers and solving of linear algebra systems. There is then a specific description of the minimizers employed in ARPEGE/IFS (M1QN3, N1CG1 and CONGRAD) and of the applications using these minimizers.

Résumé:

Cette documentation décrit les minimisations faites dans ARPEGE/IFS, par exemple dans les assimilations 3DVAR et 4DVAR. On rappelle d'abord quelques notions sur les minimiseurs et sur la résolution des systèmes linéaires. On fournit ensuite une description spécifique des minimiseurs utilisés dans ARPEGE/IFS (M1QN3, N1CG1 et CONGRAD) et des applications utilisant ces minimiseurs.

Contents

1 Introduction.

ARPEGE/IFS needs to minimize a cost function for example in the 4DVAR assimilation but also to compute a spectral orography in the climatological files (configuration 923). If the cost function is purely quadratic the minimization is equivalent to solve a linear system of equations.

There are several ranges of ways to solve such problems, some apply to solve small linear systems, some other to solve small or big linear systems, some other to minimize functions which are not necessary quadratic ones:

- Direct algorithms to invert linear systems: they can be used to invert rather small systems (the space dimension has a range of several hundreds but not more):
  - Gauss method.
  - Gauss-Jordan method.
  - LU factorisation (square matrices).
  - Cholesky LL^T factorisation (symmetric positive definite square matrices).
  - QR factorisation (square matrices).
- Pure iterative algorithms to invert linear systems or minimize functions.
  - Jacobi (solves a small linear system).
  - Gauss-Seidel (solves a small linear system).
  - Relaxation (solves a small linear system).
  - Newton and quasi-Newton algorithm (minimization of a function which is not necessary a quadratic one, the dimension of the problem can be large).
  - Relaxation method (minimization, the dimension of the problem can be large).
  - Gradient method (minimization, the dimension of the problem can be large).
- Mixed iterative algorithms to invert linear systems or minimize functions. What we say by "mixed" is that these algorithms are direct algorithms (the maximum number of iterations is equal to the dimension of the problem), but one uses them with a number of iterations significantly lower than the possible maximum, so one uses them as iterative algorithms.
  - Conjugate gradient methods (minimization, the dimension of the problem can be large). There are several versions of this algorithm, some for purely quadratic functions, some other which can be also used for non quadratic functions.
  - Conjugate gradient methods combined with a Lanczos algorithm (minimization, the dimension of the problem can be large).

These algorithms are described in some mathematical books but for each of ones one will recall the principles of them and the range of use in order to have a synthetic overview of them. What is indeed our main concern in this documentation is the sub-list of iterative minimization algorithms which allow to minimize quadratic or non-quadratic functions and to solve big linear systems (the dimension of the problem to solve being in a range around tens or hundreds of thousands).

∗ **Distributed memory:** Some distributed memory features are present in the code and will be briefly described. For convenience one uses some generic appellations.

- Expression "DM-local" for a quantity means "local to the couple of processors (*proca*,*procb*)": each processor has its own value for the quantity. Expression "DM-local computations" means that the computations are made independently in each processor on "DM-local" quantities, leading to results internal to each processor, which can be different from a processor to another one.
- Expression "DM-global" for a quantity means that it has a unique value available in all the processors. Expression "DM-global computations" means that the computations are either made in one processor, then the results are dispatched in all the processors, or the same computations are made in all the processors, leading to the same results in all the processors.
- In a routine description the mention "For distributed memory computations are DM-local" means that all calculations made by this routine are DM-local; the mention "For distributed memory computations are DM-global" means that all calculations made by this routine are DM-global; when no information is provided it means that a part of calculations are DM-local and the other part is DM-global.
- Expression "main" processor currently refers to the processor number 1: (*proca*,*procb*)=(1,1).

∗ **Modifications since cycle 42:**

- No significant change.

2 Some algorithms of minimization and of linear systems solving.

2.1 Purely direct algorithms.

These algorithms generally apply to solve small linear systems. The system which has to be solved writes:

$$\mathtt{A}\mathbf{X} = \mathbf{B} \tag{1}$$

where $\mathtt{A}$ is a $N * N$ matrix, $\mathbf{X}$ is the unknown vector of dimension N and $\mathbf{B}$ is the RHS known vector of dimension N.

2.1.1 Gauss method.

The algorithm involves N_{I} iterations which leads to replace the system $\mathtt{A}\mathbf{X} = \mathbf{B}$ by the system:

$$\mathtt{MA}\mathbf{X} = \mathtt{M}\mathbf{B} \tag{2}$$

where the product [$\mathtt{MA}$] is an upper triangular $N * N$ matrix; this new system can be easily solved. This method can be used for any invertible matrix $\mathtt{A}$. This method requires $N^3/3$ additions, $N^3/3$ multiplications and $N^2/2$ divisions.

2.1.2 Gauss-Jordan method.

This is a variant of the Gauss method but $\mathtt{M}$ is now computed in order to have a product [$\mathtt{MA}$] which is diagonal. This method can be used for example when the storage of $\mathtt{A}^{-1}$ is required.

2.1.3 LU factorisation method.

Matrix $\mathtt{A}$ is factorized into the matricial product $\mathtt{LU}$ where $\mathtt{L}$ is a lower triangular matrix, the diagonal elements of which are all equal to 1, and $\mathtt{U}$ is an upper triangular matrix. This is a particular case of Gauss method where $\mathtt{M} = \mathtt{L}^{-1}$ but one assumes that $\mathtt{L}$ is stored. This method can be used when the same matrix has to be inverted for a large number of unknown vectors $\mathbf{X}$: in this case the matrices $\mathtt{U}$ and $\mathtt{L}$ are first pre-computed and stored. This method can be used for any invertible matrix $\mathtt{A}$ which a minimum amount of "good" properties which are not detailed in this documentation.

2.1.4 Cholesky method.

This is a particular application of the Gauss method when $\mathtt{A}$ is a definite positive symmetric matrix. In this case $\mathtt{A}$ is factorised as $\mathtt{A} = \mathtt{B}\mathtt{B}^T$ with the properties $\mathtt{M} = \mathtt{B}^{-1}$ and $\mathtt{MA} = \mathtt{B}^T$. $\mathtt{B}$ is a real lower triangular matrix. This method requires $N^3/6$ additions, $N^3/6$ multiplications, $N^2/2$ divisions and N square roots extractions.

2.1.5 QR factorisation method.

This is a variant of the Gauss method but $\mathtt{M}$ is chosen to be equal to $\mathtt{Q}^{-1}$ and $\mathtt{A}$ is factorised as $\mathtt{QR}$ where:

- $\mathtt{Q}$ is a unitary matrix.
- $\mathtt{R}$ is an upper triangular matrix.
- $\mathtt{Q}^{-1}$ can be written as the product of $N-1$ Householder matrices: $\mathtt{Q}^{-1} = \mathtt{H}_{N-1}\mathtt{H}_{N-2}...\mathtt{H}_1$.

This method can be used for any matrix $\mathtt{A}$. If $\mathtt{A}$ is an invertible matrix, the factorisation QR is an unique one.

2.2 Iterative algorithms for small linear systems.

$\mathtt{D}$ is the diagonal part of $\mathtt{A}$, $-\mathtt{E}$ (resp. $-\mathtt{F}$) is the lower (resp. upper) triangular part of $\mathtt{A}$. So $\mathtt{A}$ writes $\mathtt{D} - \mathtt{E} - \mathtt{F}$. These iterative methods are used for small linear systems and use the decomposition of matrix $\mathtt{A}$ into its diagonal part, its lower triangular part and its upper triangular part.

2.2.1 Jacobi method.

- $\mathtt{A}$ is written $\mathtt{A} = \mathtt{D} - (\mathtt{E} + \mathtt{F})$.
- An iteration writes: $\mathbf{X}_{k+1} = \mathtt{D}^{-1}(\mathtt{E} + \mathtt{F})\mathbf{X}_k + \mathtt{D}^{-1}\mathbf{B}$.

This method is convergent if the spectral radius (i.e. the maximum of the absolute values of the eigenvalues) of the matrix $[\mathtt{D}^{-1}(\mathtt{E} + \mathtt{F})]$ is lower than 1.

2.2.2 Gauss-Seidel method.

- $\mathtt{A}$ is written $\mathtt{A} = (\mathtt{D} - \mathtt{E}) - \mathtt{F}$.
- An iteration writes: $\mathbf{X}_{k+1} = (\mathtt{D} - \mathtt{E})^{-1}\mathtt{F}\mathbf{X}_k + (\mathtt{D} - \mathtt{E})^{-1}\mathbf{B}$.

This method is convergent if the spectral radius of the matrix $[(\mathtt{D} - \mathtt{E})^{-1}\mathtt{F}]$ is lower than 1.

2.2.3 Relaxation method.

- $\mathtt{A}$ is written:
$$\mathtt{A} = \left(\frac{1}{\omega}\mathtt{D} - \mathtt{E}\right) - \left(\frac{1-\omega}{\omega}\mathtt{D} + \mathtt{F}\right)$$
- An iteration writes:
$$\mathbf{X}_{k+1} = \left(\frac{1}{\omega}\mathtt{D} - \mathtt{E}\right)^{-1}\left(\frac{1-\omega}{\omega}\mathtt{D} + \mathtt{F}\right)\mathbf{X}_k + \left(\frac{1}{\omega}\mathtt{D} - \mathtt{E}\right)^{-1}\mathbf{B}$$

This method is convergent if the spectral radius of the matrix $\left(\frac{1}{\omega}\mathtt{D} - \mathtt{E}\right)^{-1}\left(\frac{1-\omega}{\omega}\mathtt{D} + \mathtt{F}\right)$ is lower than 1. When matrix $\mathtt{A}$ is a definite positive hermitian matrix, this method is convergent if $0 < \omega < 2$ and the best convergence is obtained for a value of ω which is always between 1 and 2.

2.3 Iterative and "mixed" algorithms for large linear systems and minimization problems.

2.3.1 General considerations and denotations.

One tries to do a synthesis of existing algorithms and to explain them briefly, preferably using the same denotations for all these algorithms. Literature provides a large number of books or papers explaining these algorithms, with often different denotations. For example, (Bonnans et al., 1997) gives (in French) a good overview of these algorithms, (IDNEWT) gives (in French) a synthetic overview of Newton and quasi-Newton algorithms, (Shewchuk, 1994) is more oriented towards conjugate gradient methods with good practical examples. The following denotations will be used:

- N is the dimension of the problem.
- N_{I} is the number of iterations required.
- $\mathbf{X}$ is the unknown vector to be found by an iterative algorithm; the successive iterations give $\mathbf{X}_{iter=0}$, $\mathbf{X}_{iter=1}$, ..., $\mathbf{X}_{iter=N_{\mathrm{I}}}$. $\mathbf{X}$ has N components numbered from $X_{i=1}$ to $X_{i=N}$.
- The function to be minimized is $f(\mathbf{X})$ (this is a scalar function).
- f is not necessary a quadratic function. f can be written as a sum of its second-order limited development and of a residual term, around a reference value of $\mathbf{X}$ denoted by $\mathbf{X}_{\mathrm{ref}}$; if $f(\mathbf{X}_{\mathrm{ref}})$ is different from zero we can consider the function F defined by $F(\mathbf{X}) = f(\mathbf{X}) - f(\mathbf{X}_{\mathrm{ref}})$ instead of f; if $\mathbf{X}_{\mathrm{ref}}$ is different from zero we can consider the function F defined by $F(\mathbf{X} - \mathbf{X}_{\mathrm{ref}}) = f(\mathbf{X})$; so we can stick to the case where $f(0) = 0$ and where the second-order limited development is done around $\mathbf{X}_{\mathrm{ref}} = 0$. In this case the second-order limited development writes:
$$f(\mathbf{X}) = J(\mathbf{X}) = - < \mathbf{B}, \mathbf{X} > + 0.5 < \mathtt{A}\mathbf{X}, \mathbf{X} > + \mathit{residual}$$
where:
  - $< \mathbf{X}, \mathbf{Y} >$ is the scalar product of vectors $\mathbf{X}$ and $\mathbf{Y}$. It can be the "canonical" scalar product or some other scalar product.
  - $-\mathbf{B}$ is the vector gradient of f at $\mathbf{X} = 0$; the i-th component of $\mathbf{B}$ contains $-\frac{\partial f}{\partial X_i}(\mathbf{X} = 0)$.
  - Matrix $\mathtt{A}$ contains the second-order derivatives $\frac{\partial^2 f}{\partial X_i X_j}(\mathbf{X} = 0)$ and is the Hessian operator of f at $\mathbf{X} = 0$.
- In the particular case where f is a quadratic function and where $f(0) = 0$, the residual "$\mathit{residual}$" becomes zero, so f, which is also denoted by J in this case, writes:
$$f(\mathbf{X}) = J(\mathbf{X}) = 0.5 < \mathtt{A}\mathbf{X}, \mathbf{X} > - < \mathbf{B}, \mathbf{X} >$$
The Hessian J'' defined by matrix $\mathtt{A}$ is no longer $\mathbf{X}$-dependent. Taking the gradient of this equation writes:
$$\mathbf{g} = [\nabla J](\mathbf{X}) = \mathtt{A}\mathbf{X} - \mathbf{B}$$
A necessary condition of minimum of f is that $\mathbf{X}$ zeroes the gradient of the RHS, i.e.:
$$\mathtt{A}\mathbf{X} - \mathbf{B} = 0$$

The searched solution is still solution of a linear system. But the given solution can give a maximum of f, a minimum of f or a "saddle" value of f so if one wants to be sure that it gives a minimum of f one must assume some additional good properties for f. If the matrix $\mathtt{A}$ is a positive definite one, the searched solution gives a minimum of f.

- Some methods stand to a minimization and can be used for non quadratic functions f, some other methods apply to quadratic functions and assume the equivalence of "minimizing f" and "solving the linear system $\mathtt{A}\mathbf{X} = \mathbf{B}$.
- When solving the linear system $\mathtt{A}\mathbf{X} = \mathbf{B}$, at each iteration:
  - the residual term is $\mathbf{r}_{iter} = \mathtt{A}\mathbf{X}_{iter} - \mathbf{B}$.
  - the increment is $\mathbf{S}_{iter} = \mathbf{X}_{iter+1} - \mathbf{X}_{iter}$.
  - this increment $\mathbf{S}_{iter}$ generally writes as $\rho_{iter}\mathbf{d}_{iter}$ where $\mathbf{d}_{iter}$ is the direction and ρ_{iter} is the step.
  - the gradient increment is $\mathbf{Y}_{iter} = \mathbf{g}_{iter+1} - \mathbf{g}_{iter}$.

2.3.2 Preconditioning.

A good conditioning of matrix A (i.e. the ratio of the absolute values of the extreme eigenvalues $r_{\text{cond}} = \frac{|\lambda_{\min}|}{|\lambda_{\max}|}$ is as close as possible to 1) is required to have a quick convergence of the iterative algorithm.
Left preconditioning: in some cases it is more interesting to solve

$$\mathtt{H}^{-1}\mathtt{A}\mathbf{X} = \mathtt{H}^{-1}\mathtt{B}$$

instead of:

$$\mathtt{A}\mathbf{X} = \mathtt{B}$$

with a "good" matrix $\mathtt{H}$ the properties of which are not always easy to match. We wants at the same time that this matrix $\mathtt{H}$ be:

- easily invertible, and if it is diagonal it would be even better. If $\mathtt{A}$ is a positive definite symmetric matrix, $\mathtt{H}$ must be also a positive definite symmetric matrix.
- its "biggest" eigenvalue (in absolute value) is close to $\lambda_{\max}$, and its "lowest" eigenvalue (in absolute value) is close to $\lambda_{\min}$.
- $\mathtt{H}^{-1}\mathtt{A}$ is not so far from identity and has a very good conditioning (considerably improved compared to the one of $\mathtt{A}$).
- $\mathtt{H}$ can be constant or can be recomputed at each iteration.

Some algorithms to provide a good conditioning will be described later. All algorithms will be described from now on including matrix $\mathtt{H}$.
Remark: an alternative way of using a preconditioner is to write the linear system to solve as follows (right preconditioning):

$$\mathtt{A}\mathtt{H}^{-1}[\mathtt{H}\mathbf{X}] = \mathtt{B}$$

the unknown to find being $\mathtt{H}\mathbf{X}$ instead of $\mathbf{X}$. $\mathtt{H}$ must have the "good properties" listed above.
Right and left preconditioning can be combined, the linear system to solve is rewritten as follows:

$$\mathtt{H}^{-1}\mathtt{A}\mathtt{J}^{-1}[\mathtt{J}\mathbf{X}] = \mathtt{H}^{-1}\mathtt{B}$$

$\mathtt{H}$ and $\mathtt{J}$ are two preconditioners.
One can find information about preconditioning for example in part 3 of (Barrett et al., 1994).
Let us list some examples of ways of computing preconditioning matrices.

- Quasi-Newton preconditioner: the inverse $\mathtt{H}^{-1}$ can be directly computed with the same recurrence method as for the matrix $\mathtt{W}$ used in the Quasi-Newton method. It can be considered as a good approximation of the inverse of $\mathtt{A}$. For more details see for example (Morales and Nocedal, 1999).
- Another example of preconditioner is given in the part 6.5 of ECMWF documentation (TDECDAS).

2.3.3 Newton method.

This algorithm can be used for non-quadratic functions. One assumes that the function to minimize has a minimum in the "region of minimum". At each iteration of the minimization algorithm, one will minimize the second-order limited development of f around the value $\mathbf{X} = \mathbf{X}_{iter}$. This is equivalent to solve a linear system which changes at each iteration.

- One starts from an arbitrary "guess" $\mathbf{X}_0$ of the solution.
- Definition of $\mathtt{A}_0 = J''(\mathbf{X}_0)$.
- Resolution of the linear system $\mathtt{A}_0\mathbf{d}_0 = -\nabla J(\mathbf{X}_0)$ which provides $\mathbf{d}_0$.

- If $\mathbf{d}_0 = 0$ the algorithm ends.
- If $\mathbf{d}_0$ is not zero:
  - Definition of the step ρ_0: see below the general case for iteration "*iter*".
  - Computation of:
$$\mathbf{X}_1 = \mathbf{X}_0 + \rho_0 \mathbf{d}_0$$
- The following iterations write:
  - Definition of $\mathtt{A}_{iter} = J''(\mathbf{X}_{iter})$.
  - Computation of $\mathbf{g}_{iter} = \nabla J(\mathbf{X}_{iter})$: that can be done by an analytic formula or by an iterative formula which writes $\mathbf{g}_{iter} = \mathbf{g}_{iter-1} + \rho_{iter-1}\mathtt{A}_{iter-1}\mathbf{d}_{iter-1}$.
  - Computation of $\mathbf{d}_{iter}$: resolution of the linear system $\mathtt{A}_{iter}\mathbf{d}_{iter} = -\nabla J(\mathbf{X}_{iter})$ which provides $\mathbf{d}_{iter}$.
  - Computation of ρ_{iter}: several methods can be used, the two known ones are the following ones:
    * use of a trust region (this is a rather tricky algorithm, the description of which will not be provided in the current documentation; for more details see for example (IDNEWT)).
    * linear research along the descent direction, using a Wolfe algorithm. Roughly one searches for a value of ρ_{iter} (by dichotomy) which minimizes $J(\mathbf{X}_{iter} + \rho_{iter}\mathbf{d}_{iter})$ (or a polynomial approximation of this function, often cubic). Moreover this value has to be bounded as follows:
$$J(\mathbf{X}_{iter} + \rho_{iter}\mathbf{d}_{iter}) \leq J(\mathbf{X}_{iter}) + \alpha_1 \rho_{iter} < \nabla J(\mathbf{X}_{iter}); \mathbf{d}_{iter} >$$
$$< \nabla J(\mathbf{X}_{iter} + \rho_{iter}\mathbf{d}_{iter}); \mathbf{d}_{iter} > \geq \alpha_2 < \nabla J(\mathbf{X}_{iter}); \mathbf{d}_{iter} >$$
      where $0 < \alpha_1 < 0.5$ and $\alpha_1 < \alpha_2 < 1$.
  - Computation of $\mathbf{X}_{iter+1}$:
$$\mathbf{X}_{iter+1} = \mathbf{X}_{iter} + \rho_{iter}\mathbf{d}_{iter}$$
- Algorithm stops when the gradient $\nabla J(\mathbf{X}_{iter})$ has a "sufficiently" small norm (for more details see section (2.3.11)).
- This algorithm is a purely iterative one (in theory it does not give the exact solution in a finite number of iterations).
- The descent direction is generally not parallel to the gradient.
- Since `A` depends on the iteration, the main shortcoming of this algorithm is that it requires the inversion of a big linear system at each iteration, which can be done in theory by some other methods described in this documentation (Fletcher-Reeves conjugate gradient method for example), but which can become very expensive (in time and memory), so the purely Newton method is generally replaced by cheaper algorithms which replace `A` by an approximation easier to compute and to invert. That provides a class of quasi-Newton methods described below.

2.3.4 Quasi-Newton method.

This algorithm can be used for non-quadratic functions and is a simplification of the Newton method. `A` is replaced by an approximation `H` simpler to compute and to invert. `H` is generally computed by a recurrence. `H` must have some interesting properties:

- At each iteration, `H` must be a good approximation of `A`.
- `H` is easy to compute (generally by an iterative algorithm) and to invert; if `H` is diagonal this is better.

The alternative way is to compute directly a good approximation `W` of $\mathtt{A}^{-1}$. In the literature one finds some methods to compute such operators (`W` can be also used as preconditioners for steepest descent of conjugate gradient methods), the most known is the Broyden-Fletcher-Goldbarb-Shanno (abbreviated into BFGS) algorithm. At each iteration:

- the N-lines and one column matrix $\mathtt{S}_{iter}$ containing the vector $(\mathbf{X}_{iter} - \mathbf{X}_{iter-1})$ is stored.
- the N-lines and one column matrix $\mathtt{Y}_{iter}$ containing the vector $(\mathbf{g}_{iter} - \mathbf{g}_{iter-1})$ is stored.
- a guess $\mathtt{W}_0$ is computed (that can be the identity matrix).
- a recurrence formula gives $\mathtt{W}_1$ to $\mathtt{W}_{iter}$:
$$\mathtt{W}_{iter} = \left(I - \frac{1}{\mathtt{Y}_{iter}^T \mathtt{S}_{iter}} \mathtt{S}_{iter} \mathtt{Y}_{iter}^T\right) \mathtt{W}_{iter-1} \left(I - \frac{1}{\mathtt{Y}_{iter}^T \mathtt{S}_{iter}} \mathtt{S}_{iter} \mathtt{Y}_{iter}^T\right) + \frac{1}{\mathtt{Y}_{iter}^T \mathtt{S}_{iter}} \mathtt{S}_{iter} \mathtt{S}_{iter}^T$$

Since $\mathtt{W}$ is never stored ($N * N$ matrix, too big in memory), all the $\mathtt{S}$ and $\mathtt{Y}$ have to be stored, and all the recurrence from $\mathtt{W}_0$ has to be performed to each iteration (for $\mathtt{S}$ and $\mathtt{Y}$ the total amount of data stored at iteration "*iter*" is $(2iter) * N$). The previous iterative algorithm is applied to vector $-\nabla J(\mathbf{X}_{iter})$ and gives the vector $-\mathtt{W}_{iter}\nabla J(\mathbf{X}_{iter})$. The quasi-Newton method follows the same scheme as the Newton method, excepted that:

$$\mathbf{d}_{iter} = -\mathtt{A}_{iter}^{-1}\nabla J(\mathbf{X}_{iter})$$

is replaced by:

$$\mathbf{d}_{iter} = -\mathtt{W}_{iter}\nabla J(\mathbf{X}_{iter})$$

2.3.5 Limited memory quasi-Newton method.

When the number of iterations N_{I} necessary to find the solution becomes important, the number of pairs $(\mathtt{S}, \mathtt{Y})$ becomes important to be stored, and some algorithms limiting the storage to a "good" subset of pairs $(\mathtt{S}, \mathtt{Y})$ have been developed. N_{M} is the maximal number of pairs kept (for iterations number $N_{\mathrm{M}} + 1$ to N_{I}). The choice of the iterations kept is generally the N_{M} last pairs but it can be something else. Practically a value of N_{M} around 5 is a good choice. One now assumes that, for the iteration *iter*, the selected pairs are $(\mathtt{S}_{(iter,m_1)}, \mathtt{Y}_{(iter,m_1)})$ to $(\mathtt{S}_{(iter,m_{N_{\mathrm{M}}})}, \mathtt{Y}_{(iter,m_{N_{\mathrm{M}}})})$. $m_{N_{\mathrm{M}}}$ is generally (but not necessary) equal to *iter*; m_1 is generally (but not necessary) equal to $iter + 1 - N_{\mathrm{M}}$; the list of m selected can depend of *iter*.
For the iteration *iter*:

- one defines a guess $\mathtt{W}_{(iter,m=m_1)}$ which is generally a diagonal matrix $\mathtt{D}_{iter}$; it can be the identity matrix (this is generally the case for $\mathtt{D}_0$) but some authors have developed better algorithms to compute $\mathtt{D}_{iter}$; for example the scalar initial scaling mode ("SIS" algorithm) yields:

$$\mathtt{D}_{iter} = \frac{\mathtt{Y}_{iter-1}^T\mathtt{S}_{iter-1}}{\mathtt{Y}_{iter-1}^T\mathtt{Y}_{iter-1}}I$$

- the recurrence formula giving $\mathtt{W}_{(iter,m=m_i)}$ knowing $\mathtt{W}_{(iter,m=m_{i-1})}$ writes:

$$\mathtt{W}_{(iter,m=m_i)} = \mathtt{F}_{(iter,m=m_i)}\mathtt{W}_{(iter,m=m_{i-1})}\mathtt{F}_{(iter,m=m_i)} + \frac{1}{\mathtt{Y}^T{}_{(iter,m=m_i)}\mathtt{S}_{(iter,m=m_i)}}\mathtt{S}_{(iter,m=m_i)}\mathtt{S}^T{}_{(iter,m=m_i)}$$

  where $\mathtt{F}$ is:

$$\mathtt{F}_{(iter,m=m_i)} = \left(I - \frac{1}{\mathtt{Y}^T{}_{(iter,m=m_i)}\mathtt{S}_{(iter,m=m_i)}}\mathtt{S}_{(iter,m=m_i)}\mathtt{Y}^T{}_{(iter,m=m_i)}\right)$$

More details are given for example in (IDQN3).

2.3.6 Truncated Newton method.

Looks like the Newton method. The resolution of the linear system $\mathtt{A}_{iter}\mathbf{d}_{iter} = -\nabla J(\mathbf{X}_{iter})$ is still done (without approximation of the Hessian) by an iterative algorithm but the convergence criterion is coarse for lower values of *iter* (so the resolution can be done quickly with a small amount of iterations); when *iter* increases the convergence criterion becomes more severe (in the purely Newton method the convergence criterion does not depend on *iter*). To sum up, the computation of $\mathbf{d}_{iter}$ is relatively cheap for lower values of *iter* and more and more expensive when *iter* increases.

2.3.7 Fletcher-Reeves conjugate gradient method.

This algorithm can be used for quadratic functions. One assumes that the function is a positive definite quadratic one. The aim is to minimize:

$$f(\mathbf{X}) = J(\mathbf{X}) = 0.5 < \mathtt{A}\mathbf{X}, \mathbf{X} > - < \mathbf{B}, \mathbf{X} >$$

and this equivalent to solve the following linear system:

$$\mathtt{H}^{-1}\mathtt{A}\mathbf{X} = \mathtt{H}^{-1}\mathbf{B}$$

- One starts from an arbitrary "guess" $\mathbf{X}_0$ of the solution.
- Definition of $\mathtt{H}_0$.
- Computation of $\mathbf{d}_0 = -\mathtt{H}_0^{-1}\nabla J(\mathbf{X}_0)$.
- If $\mathbf{d}_0 = 0$ the algorithm ends.
- If $\mathbf{d}_0$ is not zero:
  - Definition of:

$$\rho_0 = \frac{< \nabla J(\mathbf{X}_0); -\mathbf{d}_0 >}{< \mathtt{A}\mathbf{d}_0; \mathbf{d}_0 >} = \frac{< \mathtt{H}_0^{-1}\nabla J(\mathbf{X}_0); \nabla J(\mathbf{X}_0) >}{< \mathtt{A}\mathbf{d}_0; \mathbf{d}_0 >}$$

– Computation of:

$$\mathbf{X}_1 = \mathbf{X}_0 + \rho_0 \mathbf{d}_0$$

- The following iterations write:
  - Definition of $\mathtt{H}_{iter}$ and computation of $\mathtt{H}_{iter}^{-1}$.
  - Computation of $\mathbf{g}_{iter} = \nabla J(\mathbf{X}_{iter})$: that can be done by an analytic formula or by an iterative formula which writes $\mathbf{g}_{iter} = \mathbf{g}_{iter-1} + \rho_{iter-1}\mathtt{A}\mathbf{d}_{iter-1}$.
  - Computation of $\mathbf{d}_{iter}$:

$$\mathbf{d}_{iter} = -\mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter}) + \frac{< \mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter}); \nabla J(\mathbf{X}_{iter}) >}{< \mathtt{H}_{iter-1}^{-1}\nabla J(\mathbf{X}_{iter-1}); \nabla J(\mathbf{X}_{iter-1}) >}\mathbf{d}_{iter-1}$$

  - Computation of ρ_{iter}:

$$\rho_{iter} = \frac{< \mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter}); \nabla J(\mathbf{X}_{iter}) >}{< \mathtt{A}\mathbf{d}_{iter}; \mathbf{d}_{iter} >} = \frac{< \nabla J(\mathbf{X}_{iter}); -\mathbf{d}_{iter} >}{< \mathtt{A}\mathbf{d}_{iter}; \mathbf{d}_{iter} >}$$

  - Computation of $\mathbf{X}_{iter+1}$:

$$\mathbf{X}_{iter+1} = \mathbf{X}_{iter} + \rho_{iter}\mathbf{d}_{iter}$$

- Algorithm stops when the gradient $\nabla J(\mathbf{X}_{iter})$ has a "sufficiently" small norm (for more details see section (2.3.11)).
- This algorithm converges in a finite number of iterations, the maximum possible number is N.
- Two different iterations of $\nabla J(\mathbf{X})$ (and also two different iterations of $\mathbf{d}_{iter}$) are "orthogonal" relative to the scalar product defined with matrix $\mathtt{H}^{-1}$.
- The descent direction is not parallel to the gradient.

2.3.8 Polak-Ribière conjugate gradient method.

For purely quadratic functions this algorithm is identical to the Fletcher-Reeves conjugate gradient method. This algorithm is an extension of the Fletcher-Reeves one for non-quadratic functions. When the function to minimize is not a quadratic one the differences with the Fletcher-Reeves algorithm are:

- The new way of computing the direction $\mathbf{d}_{iter}$ is:

$$\mathbf{d}_{iter} = -\mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter}) + \frac{< \mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter}); \nabla J(\mathbf{X}_{iter}) - \nabla J(\mathbf{X}_{iter-1}) >}{< \mathtt{H}_{iter-1}^{-1}\nabla J(\mathbf{X}_{iter-1}); \nabla J(\mathbf{X}_{iter-1}) >}\mathbf{d}_{iter-1}$$

- This algorithm does not converge in a finite number of iterations.
- Two different iterations of $\nabla J(\mathbf{X})$ are no longer necessarily "orthogonal" relative to the scalar product defined with matrix $\mathtt{H}^{-1}$.

2.3.9 Gradient method with optimal step (or steepest descent method).

This algorithm can be used for quadratic functions. One assumes that the function is a positive definite quadratic one. This algorithm looks like the Fletcher-Reeves conjugate gradient method excepted on the following points:

- Computation of $\mathbf{d}_{iter}$:

$$\mathbf{d}_{iter} = -\mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter})$$

- The descent direction is parallel to the gradient and orthogonal to the iso-J.
- This algorithm does not converge in a finite number of iterations.
- Two different iterations of $\nabla J(\mathbf{X})$ are no longer necessary "orthogonal" relative to the scalar product defined with matrix $\mathtt{H}^{-1}$.

2.3.10 Conjugate gradient method combined with a Lanczos algorithm.

This algorithm is a combination of a Fletcher-Reeves conjugate gradient algorithm with a Lanczos algorithm. The main difference with the Fletcher-Reeves algorithm is that $\mathbf{X}_{iter+1}$ is not computed using directly formula $\mathbf{X}_{iter+1} = \mathbf{X}_{iter} + \rho_{iter}\mathbf{d}_{iter}$ but using a more tricky algorithm involving a matrix $\mathtt{T}$ of dimensions $(iter+1)*(iter+1)$, the eigenvalues of which converge towards the eigenvalues of $\mathtt{A}$ when $iter$ increases.
This algorithm can be used for quadratic functions. One assumes that the function is a positive definite quadratic one. The aim is to minimize:

$$f(\mathbf{X}) = J(\mathbf{X}) = 0.5 < \mathtt{A}\mathbf{X}, \mathbf{X} > - < \mathbf{B}, \mathbf{X} >$$

and this equivalent to solve the following linear system:

$$\mathtt{H}^{-1}\mathtt{A}\mathbf{X} = \mathtt{H}^{-1}\mathbf{B}$$

- One starts from an arbitrary "guess" $\mathbf{X}_0$ of the solution.
- Definition of $\mathtt{H}_0$.
- Computation of $\mathbf{d}_0 = -\mathtt{H}_0^{-1}\nabla J(\mathbf{X}_0)$.
- If $\mathbf{d}_0 = 0$ the algorithm ends.
- If $\mathbf{d}_0$ is not zero:
  - Definition of:
    $$\rho_0 = \frac{< \nabla J(\mathbf{X}_0); -\mathbf{d}_0 >}{< \mathtt{A}\mathbf{d}_0; \mathbf{d}_0 >} = \frac{< \mathtt{H}_0^{-1}\nabla J(\mathbf{X}_0); \nabla J(\mathbf{X}_0) >}{< \mathtt{A}\mathbf{d}_0; \mathbf{d}_0 >}$$
    (identical to Fletcher-Reeves conjugate gradient algorithm).
  - Definition of:
    $$\mathbf{q}_0 = \frac{1}{< \mathtt{H}_0^{-1}\nabla J(\mathbf{X}_0); \nabla J(\mathbf{X}_0) >}\nabla J(\mathbf{X}_0)$$
  - Definition of $\mathtt{Q}_1$ which is the N-lines and one column matrix containing vector $\mathbf{q}_0$.
  - Definition of: $\delta_1 = \frac{1}{\rho_0}$.
  - Definition of the $1*1$ matrix $\mathtt{T}_1 = [\delta_0]$.
  - Definition of the $1*1$ matrix:
    $$\mathtt{Z}_1 = -\mathtt{T}_1^{-1}\mathtt{Q}_1^T\mathtt{H}_0^{-1}\nabla J(\mathbf{X}_0)$$
    which is simply the scalar $-\rho_0$.
  - Computation of:
    $$\mathbf{X}_1 = \mathbf{X}_0 + \mathtt{H}_0^{-1}\mathtt{Q}_1\mathtt{Z}_1$$
    which is equivalent to:
    $$\mathbf{X}_1 = \mathbf{X}_0 + \rho_0\mathbf{d}_0$$
    since $\mathtt{Z}_1 = \rho_0$ and $\mathtt{H}_0^{-1}\mathtt{Q}_1 = -\mathbf{d}_0$.
- The following iterations write:
  - Definition of $\mathtt{H}_{iter}$ and computation of $\mathtt{H}_{iter}^{-1}$.
  - Computation of $\mathbf{g}_{iter} = \nabla J(\mathbf{X}_{iter})$: that can be done by an analytic formula or by an iterative formula which writes $\mathbf{g}_{iter} = \mathbf{g}_{iter-1} + \rho_{iter-1}\mathtt{A}\mathbf{d}_{iter-1}$ (identical to Fletcher-Reeves conjugate gradient algorithm).
  - Computation of $\mathbf{d}_{iter}$:
    $$\mathbf{d}_{iter} = -\mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter}) + \frac{< \mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter}); \nabla J(\mathbf{X}_{iter}) >}{< \mathtt{H}_{iter-1}^{-1}\nabla J(\mathbf{X}_{iter-1}); \nabla J(\mathbf{X}_{iter-1}) >}\mathbf{d}_{iter-1}$$
    (identical to Fletcher-Reeves conjugate gradient algorithm).
  - Computation of ρ_{iter}:
    $$\rho_{iter} = \frac{< \mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter}); \nabla J(\mathbf{X}_{iter}) >}{< \mathtt{A}\mathbf{d}_{iter}; \mathbf{d}_{iter} >} = \frac{< \nabla J(\mathbf{X}_{iter}); -\mathbf{d}_{iter} >}{< \mathtt{A}\mathbf{d}_{iter}; \mathbf{d}_{iter} >}$$
    (identical to Fletcher-Reeves conjugate gradient algorithm).
  - Definition of:
    $$\mathbf{q}_{iter} = \frac{1}{< \mathtt{H}_{iter}^{-1}\nabla J(\mathbf{X}_{iter}); \nabla J(\mathbf{X}_{iter}) >}\nabla J(\mathbf{X}_{iter})$$
  - Definition of $\mathtt{Q}_{iter+1}$ which is the N-lines and $iter+1$-columns matrix containing vectors $\mathbf{q}_0$ to $\mathbf{q}_{iter}$.

– Definition of:

$$\delta_{iter+1} = \frac{< \mathtt{H}_{iter}^{-1} \nabla J(\mathbf{X}_{iter}); \nabla J(\mathbf{X}_{iter}) >^2}{< \mathtt{H}_{iter-1}^{-1} \nabla J(\mathbf{X}_{iter-1}); \nabla J(\mathbf{X}_{iter-1}) >^2} \frac{1}{\rho_{iter-1}} + \frac{1}{\rho_{iter}}$$

– Definition of:

$$\gamma_{iter} = -\frac{< \mathtt{H}_{iter}^{-1} \nabla J(\mathbf{X}_{iter}); \nabla J(\mathbf{X}_{iter}) >}{< \mathtt{H}_{iter-1}^{-1} \nabla J(\mathbf{X}_{iter-1}); \nabla J(\mathbf{X}_{iter-1}) >} \frac{1}{\rho_{iter-1}}$$

– Definition of the $(iter+1) * (iter+1)$ matrix $\mathtt{T}_{iter+1}$:

  * $\mathtt{T}_{iter+1}$ is a symmetric tri-diagonal matrix.
  * The main diagonal of $\mathtt{T}_{iter+1}$ contains δ_1 to δ_{iter+1}.
  * The first lower and upper diagonal of $\mathtt{T}_{iter+1}$ contains γ_1 to γ_{iter}.

  Computation of its inverse $\mathtt{T}_{iter+1}^{-1}$.

– Definition of the $iter+1$-lines and one column matrix:

$$\mathtt{Z}_{iter+1} = -\mathtt{T}_{iter+1}^{-1} \mathtt{Q}_{iter+1}^{T} \mathtt{H}_{iter}^{-1} \nabla J(\mathbf{X}_0)$$

– Computation of:

$$\mathbf{X}_{iter+1} = \mathbf{X}_0 + \mathtt{H}_{iter}^{-1} \mathtt{Q}_{iter+1} \mathtt{Z}_{iter+1}$$

- Algorithm stops when the gradient $\nabla J(\mathbf{X}_{iter})$ has a "sufficiently" small norm (for more details see section (2.3.11)).
- This algorithm converges in a finite number of iterations, the maximum possible number is N.
- Two different iterations of $\nabla J(\mathbf{X})$ (and also two different iterations of $\mathbf{d}_{iter}$) are "orthogonal" relative to the scalar product defined with matrix $\mathtt{H}^{-1}$.
- The descent direction is not parallel to the gradient.
- Matrix $\mathtt{T}$ has some good properties:
  – Eigenvalues of $\mathtt{T}_{iter}$ converge towards the eigenvalues of $\mathtt{A}$ when $iter$ increases.
  – Writing $\mathtt{T}_{iter} = \mathtt{P}_{iter}^{-1} \mathtt{D}_{iter} \mathtt{P}_{iter}$ where $\mathtt{D}_{iter}$ is diagonal and $\mathtt{P}_{iter}$ contains the eigenvectors of $\mathtt{T}_{iter}$, the product $\mathtt{Q}_{iter} \mathtt{P}_{iter}$ converges towards the eigenvectors of $\mathtt{A}$.

2.3.11 When to stop the algorithm.

This topic is discussed in chapter 4.2 of (Barrett et al., 1994). The stopping criterion is generally:

$$\frac{< \mathbf{g}_{iter}, \mathbf{g}_{iter} >}{< \mathbf{g}_0, \mathbf{g}_0 >} < \epsilon \tag{3}$$

An additional criterion can be used:

$$< \mathbf{X}_{iter} - \mathbf{X}_{iter-1}, \mathbf{X}_{iter} - \mathbf{X}_{iter-1} >< \epsilon_2 \tag{4}$$

Some additional constraints can be added in some of the minimizers, for example:

- In **N1CG1**: the cost function must not increase, in the contrary the algorithm stops.
- In **CONGRAD**: the eigenvalues of the matrix $\mathtt{T}$ obtained by the Lanczos algorithm must be all ≥ 0, in the contrary the algorithm stops at the last iteration where all the eigenvalues are ≥ 0.
- In **CONGRAD** one checks that matrix $\mathtt{A}$ is positive definite, in the contrary the algorithm stops.
- In **CONGRAD** the algorithm stops when the ratio final gradient / initial gradient is below a predefined threshold, or when the final gradient is below a predefined threshold.

3 Available minimizers in the "arpifs" and "algor" libraries: description and organigrammes.

Codes are in library "algor" excepted for **CONGRAD** which is in the main library "arpifs".

3.1 Minimizer M1QN3 developed at INRIA.

M1QN3 follows a limited memory quasi-Newton method. It can be used for non-quadratic functions.

∗ Organigramme of M1QN3:

```
M1QN3 ->
 * ALLOCATE_CTLVEC
 * M1QN3A ->
   - ALLOCATE_CTLVEC
   - [ YSTBL (not described in detail) ] [commented]
   - [PROSCA]
   - DD ->
      * [PROSCA]
      * [CTONB]
      * [CTCAB]
   - DDS ->
      * [ YSTBL (not described in detail) ] [commented]
      * [PROSCA]
      * [CTONB]
      * [CTCAB]
   - [SIMUL]
   - MLIS0 ->
      * [PROSCA]
      * [SIMUL]
      * ECUBE
   - DEALLOCATE_CTLVEC
 * [PROSCA]
 * DEALLOCATE_CTLVEC
```

∗ Comments about routines involved:

- **M1QN3**: is the main routine; calls **M1QN3A** after having structured the available memory.
- **M1QN3A**: is the body of the minimizer.
- **ALLOCATE_CTLVEC**
  (resp. **DEALLOCATE_CTLVEC**) (in module-routine **CONTROL_VECTORS**): allocation (resp. deallocation) of some "CONTROL_VECTOR"-type arrays.
- [**PROSCA**]: dummy argument generic name of a routine doing a scalar product.
- [**CTONB**] and [**CTCAB**]: dummy argument generic name of routines doing a basis change and its inverse.
- **DD**: computes the descent direction; manages multiplications by the inverse of the preconditioner.
- **DDS**: does the same as **DD** but is used in the case when the pairs (**Y**;**S**) are not completely stored in memory but read on a buffer via the routine **YSTBL**.
- **MLIS0**, **ECUBE**: subroutines realizing the line-search.
- [**SIMUL**]: dummy argument generic name of the simulator.
- **YSTBL**: is used for very large problems (currently commented).

∗ **Additional remarks:** A more detailed documentation of **M1QN3** can be found in (IDQN3). Preconditioning is a left one and can be done by a quasi-Newton preconditioner.

3.2 Minimizer M1QN3R.

M1QN3R follows a limited memory quasi-Newton method. It can be used for non-quadratic functions. This is a version of **M1QN3** without the "CONTROL_VECTOR" structure.

∗ Organigramme of M1QN3R:

```
M1QN3R ->
 * MUPDTSR
 * M1QN3AR ->
   - [PROSCAR]
   - DDR ->
      * [PROSCAR]
      * [CTONBR]
      * [CTCABR]
   - DDSR ->
      * YSTBLR (not described in detail)
      * [PROSCAR]
      * [CTONBR]
      * [CTCABR]
   - [SIMULR]
   - MLISOR ->
      * [PROSCAR]
      * [SIMULR]
      * ECUBER
   - YSTBLR (not described in detail)
 * [PROSCAR]
```

∗ Comments about routines involved:

- **M1QN3R**: is the main routine; calls **M1QN3AR** after having structured the available memory.
- **M1QN3AR**: is the body of the minimizer.
- **MUPDTSR**: returns the number of updates to form the approximate Hessian.
- [**PROSCAR**]: dummy argument generic name of a routine doing a scalar product.
- [**CTONBR**] and [**CTCABR**]: dummy argument generic name of routines doing a basis change and its inverse.
- **DDR**: computes the descent direction; manages multiplications by the inverse of the preconditioner.
- **DDSR**: does the same as **DDR** but is used in the case when the pairs (**Y**;**S**) are not completely stored in memory but read on a buffer via the routine **YSTBLR**.
- **MLIS0R**, **ECUBER**: subroutines realizing the line-search.
- [**SIMULR**]: dummy argument generic name of the simulator.
- **YSTBLR**: is used for very large problems.

∗ **Additional remarks:** A more detailed documentation of **M1QN3R** can be found in (IDQN3). Preconditioning is a left one and can be done by a quasi-Newton preconditioner.

3.3 Minimizer M1QN3_1DV.

M1QN3_1DV follows a limited memory quasi-Newton method. It can be used for non-quadratic functions. This is a version of **M1QN3** adapted for 1DVAR applications.

∗ Organigramme of M1QN3_1DV:

```
M1QN3_1DV ->
 * MUPDTS_1DV
 * M1QN3A_1DV ->
   - [PROSCA]
   - DD_1DV ->
      * [PROSCA]
      * CTONB_1DV
      * CTCAB_1DV
   - DDS_1DV ->
      * YSTBL_1DV
      * [PROSCA]
      * CTONB_1DV
      * CTCAB_1DV
   - ONEDVAR_SIMUL
   - MLIS0_1DV ->
      * [PROSCA]
      * ONEDVAR_SIMUL
      * ECUBE_1DV
 * [PROSCA]
```

∗ Comments about routines involved:

- **M1QN3_1DV**: is the main routine; calls **M1QN3A_1DV** after having structured the available memory.
- **M1QN3A_1DV**: is the body of the minimizer.
- [**PROSCA**]: dummy argument generic name of a routine doing a scalar product.
- **CTONB_1DV** and **CTCAB_1DV**: routines doing a basis change and its inverse.
- **DD_1DV**: computes the descent direction; manages multiplications by the inverse of the preconditioner.
- **DDS_1DV**: does the same as **DD_1DV** but is used in the case when the pairs (**Y**;**S**) are not completely stored in memory but read on a buffer via the routine **YSTBL_1DV**.
- **MLIS0_1DV**, **ECUBE_1DV**: subroutines realizing the line-search.
- **ONEDVAR_SIMUL**: simulator.
- **YSTBL_1DV**: is used for very large problems.

∗ **Additional remarks:** Preconditioning is a left one and can be done by a quasi-Newton preconditioner.

3.4 Minimizer N1CG1.

N1CG1 follows a Fletcher-Reeves conjugate gradient method. It can be used to solve positive definite quadratic functions or to solve linear systems involving a symmetric positive definite matrix.

∗ Organigramme of N1CG1:

```
N1CG1 ->
 * ALLOCATE_CTLVEC
 * [SIMUL]
 * N1CGA
   - DPSEUCLID -> DOT_PRODUCT
   - DBFGSL -> DPSEUCLID -> DOT_PRODUCT
   - [SIMUL]
   - DYSAVE -> DPSEUCLID -> DOT_PRODUCT
 * DEALLOCATE_CTLVEC
```

∗ Comments about routines involved:

- **N1CG1**: is the main routine; calls **N1CG1A**.
- **N1CG1A**: is the body of the minimizer.
- **ALLOCATE_CTLVEC**
  (resp. **DEALLOCATE_CTLVEC**) (in module-routine **CONTROL_VECTORS**): allocation (resp. deallocation) of some "CONTROL_VECTOR"-type arrays.
- **DPSEUCLID** does the "basic" scalar product and calls **DOT_PRODUCT**.
- **DBFGSL**: computes the descent direction; manages multiplications by the inverse of the preconditioner. This routine is nearly identical to the routine **DD** called under **M1QN3**.
- **DYSAVE** is called when asking for preconditioning does the selection and saving of pairs (**Y**;**S**).
- [**SIMUL**]: dummy argument generic name of the simulator.

∗ Additional remarks: This routine can be used with a left quasi-Newton preconditioner; see for example (Morales and Nocedal, 1999) for the description of such a preconditioner.

3.5 Minimizer CONGRAD.

CONGRAD follows a conjugate gradient method combined with a Lanczos algorithm. It can be used to solve positive definite quadratic functions or to solve linear systems involving a symmetric positive definite matrix. Preconditioning is a right one.

∗ Organigramme of CONGRAD:

```
CONGRAD ->
 * ALLOCATE_CTLVEC
 * SIM4D -> (call tree not described)
 * OPK -> (call tree not described)
 * PRECOND
 * PTSV -> [LAPACK]/SPTSV ou [LAPACK]/DPTSV
 * STEQR -> [LAPACK]/SSTEQR ou [LAPACK]/DSTEQR
 * WREVECS ->
   - ALLOCATE_CTLVEC
   - DEALLOCATE_CTLVEC
 * DEALLOCATE_CTLVEC
```

The call to **XFORMEV** has been moved in the routines which call **CONGRAD**.

∗ Organigramme of the setup part computing the preconditioner:

```
CVA1 ->
 * PREPPCM
   - ALLOCATE_CTLVEC
   - DEALLOCATE_CTLVEC
   - ALLGATHER_CVSECTION
   - SET_CVSECTION
```

∗ Comments about routines involved:

- **CONGRAD**: is the main routine and the body of the minimizer.
- **ALLGATHER_CVSECTION** (in module-routine **CONTROL_VECTORS_COMM_MOD**): gather part of the control variable on all processors.
- **ALLOCATE_CTLVEC**
  (resp. **DEALLOCATE_CTLVEC**) (in module-routine **CONTROL_VECTORS**): allocation (resp. deallocation) of some "CONTROL_VECTOR"-type arrays.
- **SIM4D**: is the simulator used in the 4D-VAR (no other simulator is currently allowed here).
- **OPK**: is the simulator used in the calculation of singular vectors.
- **PRECOND**: applies the preconditioner.
- **PREPPCM**: pre-computes the preconditioner.
- **PTSV**: resolution of a real linear system of equations, the matrix of which being symmetric positive definite and tridiagonal.
- **SET_CVSECTION** (in module-routine **CONTROL_VECTORS_COMM_MOD**): set part of the control variable on all processors.
- **STEQR**: computes the eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix.
- **CVA1**: controls variational assimilation job at level 1.
- **WREVECS**: computes and saves eigenvectors.
- **XFORMEV**: transforms eigenvectors towards model variable space.
- The "basic" scalar product is directly done by calls of **DOT_PRODUCT**.

∗ **Additional remarks:** A more detailed description of the algorithm used by **CONGRAD** can be found in (Fisher, 1998) and in the part 6 of ECMWF documentation (TDECDAS). The preconditioning is done by a change of variable inside routine **PRECOND** (solve $\mathtt{AH}^{-1}[\mathtt{HX}] = \mathtt{B}$ instead of $\mathtt{AX} = \mathtt{B}$); calculation of the preconditioner is described in the part 6.5 of ECMWF documentation (TDECDAS). The preconditioning matrix is computed in routine **PREPPCM**; the change of variable is done inside routine **PRECOND**.

4 Configurations of use of minimizations, and simulators.

4.1 Minimization for orography in climatological files (configuration 923).

It uses a non-quadratic function, and a minimization by the quasi-Newton algorithm:

- Minimizer: **M1QN3R**.
- Simulator: **SIMREL**.
- Size of the problem: a local variable **IRZ** computed in **RELSPE**, which is a function of **NSEFRE**.
- Caller: **APLM1G**.

4.2 Minimization in the 4DVAR assimilation.

It uses a quadratic function.

- Minimizer: the minimizers which can be used are **M1QN3**, **N1CG1** or **CONGRAD**.
- Simulator: **SIM4D**. A more detailed description of **SIM4D** can be found in part 4.2.3 of (IDVAR) and in the IFS technical documentation part II (section 2.3).
- Size of the problem: **NSEFRE**.
- Callers: **CFCSENS2OBS**, **CVA2**, **FORECAST_ERROR**.

4.3 Other applications using minimizers.

- **ONEDVAR_RAINTB** calls **M1QN3_1DV** (simulator **ONEDVAR_SIMUL**).
- **NALAN1** calls **CONGRAD** (simulator **SIM4D** or **OPK**).

5 Modules and namelists.

These modules are auto-documented so description of each variable is provided in the code source. We can recall here the most important variables to know for each module:

- **PTRSPOR**: used only in the minimization of orography done in the part 1 of configuration 923.
- **YOMCVA** (control variable).
- **YOMDIM** (dimensioning), in particular NSEFRE.
- **YOMGETMINI**: contains variables which are at least used when getting the pre-conditioner for CONGRAD.
- **YOMIOMI** (warm restart I/O mechanism for minimization). Some of these variables are in namelist **NAMIOMI**.
- **YOMSENS** (variables for sensitivity). Some of these variables are in namelist **NAMSENS**.
- **YOMVAR**: a subset of variables are used in minimizers or in applications calling minimizers.
  - NITER, NITER_MIN, NSIMU, NINFRA.
  - NMIMP, RXMIN (used for M1QN3).
  - LN1CG1, ZEPSNEG, N1IMP, NSELECT, NPRECO, NBFGSB (used for N1CG1).
  - LCONGRAD, NPCVECS, LWRIEVEC, NWRIEVEC, N_DIAGS_CONVERGENCE, N_DIAGS_EIGENVECS, LEVECCNTL, LEVECGRIB, RTOL_CHECK_GRADIENT, LMPCGL, EVBCGL, MCGLVEC, LTWCGL (used for CONGRAD).
  - RCVGE, R_NORM_REDUCTION_ABORT_LEVEL, L_CHECK_CONVERGENCE, L_ABS_CONVERGENCE, L_INFO_CONVERGENCE.
  - L_INFO_CONTENT, N_INFO_CONTENT_METHOD, N_INFO_CONTENT_SEED.
  - and maybe also LSKIPMIN, NUPTRA, NHEVECS.

  Some of them are in namelist **NAMVAR**.
- **YOMVCGL**: global arrays for intermediate results of the forecast error calculation or used in the CONGRAD minimization (preconditioner).
- **CONTROL_VECTORS_COMM_MOD**: contains routines which do operations on "derivated type" variables used in the minimizations (routines in this module seem linked with distributed memory).
- **CONTROL_VECTORS** and **CONTROL_VECTORS_...** (in library "algor"): contains routines which do operations on "derivated type" variables used in the minimizations.

6 Bibliography.

6.1 Publications.

- Barrett R., M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. Van der Vorst, 1994: Templates for the solution of linear systems: building blocks for iterative methods, 2nd edition, Pub. SIAM, Philadelphia. Available on http://www.netlib.org/templates/Templates.html
- Bonnans, J.F., J.C. Gilbert, C. Lemaréchal and C. Sagastizabal, 1997: Optimisation numérique, aspect théorique et pratique. (Collection Mathématiques et Applications, édité chez Springer).
- Ciarlet, P.G., 1982: Introduction à l'analyse numérique matricielle et à l'optimisation. (Collection Mathématiques appliquées pour la maîtrise, édité chez Masson, 279pp).
- Courtier, Ph., C. Freydier, J.F. Geleyn, F. Rabier and M. Rochas, 1991: The ARPEGE project at METEO-FRANCE. ECMWF Seminar Proceedings 9-13 September 1991, Volume II, 193-231.
- Fisher M., 1998: Minimization algorithms for variational data assimilation. Proceedings of ECMWF seminar on recent developments in numerical methods for atmospheric modelling, 7-11 Sept 1998, pp 364-385.
- Fletcher, R., 1993: An overview of unconstrained optimization. Pre-print, 31pp. Postscript file available as http://www.cerfacs.fr/~ wlay/LAY/Publications/Opti/fletcher.ps.gz
- Morales, J.L., and J. Nocedal, 1999: Automatic preconditioning by limited memory quasi-Newton updating. Pre-print, 22pp.
- Shewchuk, J.R., 1994: An introduction to the conjugate gradient method without the agonizing pain. Edition $1\frac{1}{4}$. Note, 64pp, available on http://www.cs.cmu.edu/~ jrs/

6.2 Some internal notes and other ARPEGE notes.

- (TDECDAS) 2015: IFS technical documentation (CY41R1). Part II: data assimilation. Available at "https://software.ecmwf.int/wiki/display/IFS/Official+IFS+Documentation".
- (TDECTEC) 2015: IFS technical documentation (CY41R1). Part VI: technical and computational procedures. Available at "https://software.ecmwf.int/wiki/display/IFS/Official+IFS+Documentation".
- (IDVAR) Fischer, C., and L. Berre, 2007: The variational computations inside ARPEGE/ALADIN: cycle CY32. Internal note, 77pp. Available on the intranet server "http://www.cnrm.meteo.fr/gmapdoc/".
- (IDQN3) Gilbert, J.C., and C. Lemaréchal, 1993: The modules M1QN3 and N1QN3. Version 2.0b. Internal note, 12pp.
- (IDNEWT) Veersé, F., 1996. Quelques algorithmes de minimisation pour l'assimilation variationnelle de données météorologiques. Internal note (slides-sized), 26pp.
- (IDBAS) Yessad, K., 2016: Basics about ARPEGE/IFS, ALADIN and AROME in the cycle 43 of ARPEGE/IFS (internal note).
- (IDEUL) Yessad, K., 2016: Integration of the model equations, and Eulerian dynamics, in the cycle 43 of ARPEGE/IFS (internal note).
- (IDDM) Yessad, K., 2016: Distributed memory features in the cycle 43 of ARPEGE/IFS (internal note).