Numerical methods for large-scale minimization

Jean Charles GILBERT
INRIA, Rocquencourt, France

Abstract: *This paper recalls some basic tools for unconstrained minimization problems. It includes a comparison of performance of automatic differentiation codes and methods when they are used to compute a gradient. Some algorithms for minimizing a function with a large number of variables are reviewed and the impact of automatic differentiation on them is discussed.*

Key words: Automatic differentiation, conjugate gradient, large-scale optimization, limited memory quasi-Newton method, partitioned quasi-Newton method, unconstrained optimization.

1 Basic notions

Let us equip $\mathbb{R}^n$ with an inner product $\langle\cdot,\cdot\rangle$ and its associated norm $|\cdot|$. We shall use the following matrix norm on $\mathbb{R}^n$: $||A|| := \sup\{|Au| : |u| = 1\}$.

Given a smooth function $f : \Omega \to \mathbb{R}$, defined on an open set Ω of $\mathbb{R}^n$, the unconstrained minimization problem consists in finding a point $x_* \in \Omega$, such that

$$f(x_*) \leq f(x), \forall x \in N(x_*), \tag{1}$$

where $N(x_*)$ denotes some neighborhood of x_*. A point x_* satisfying the above inequalities is called a *local minimizer* of f. If (1) holds for $N(x_*) = \Omega$, then x_* is called a *global minimizer* of f on Ω. Finding a global minimizer is a hard task and for this subject, we refer the reader to the collections of papers [10, 11, 44]. The algorithms considered in this paper are only able to find local minimizers.

If x_* is a local minimizer, it satisfies the first order optimality conditions:

$$f'(x_*) \cdot h = 0, \forall h \in \mathbb{R}^n,$$

where $f'(x_*)$ is the Fréchet derivative of f at x_*. These conditions can be rewritten as follows:

$$g(x_*) = 0, \tag{2}$$

where $g(x) := \nabla f(x)$ denotes the *gradient* of f at x. It is important to keep in mind that the gradient depends on the inner product chosen on $\mathbb{R}^n$. It is indeed defined by means of the Riesz theorem:

$$\exists\, g(x) \in \mathbb{R}^n, \text{unique} \;:\; \langle g(x), h\rangle = f'(x)\cdot h, \forall h \in \mathbb{R}^n.$$

If the Euclidean inner product is used, $\langle u, v\rangle = \sum_{i=1}^{n} u_i v_i$, then $g_i(x)$ is the i-th partial derivative of f at x. A point x_* satisfying (2) is called a *stationary point* of f. A minimizer is a stationary point and the converse is true for convex function.

Here, we have to be still more modest as to what the algorithms described in this paper can do. In general, they can only find stationary points. This is due to the fact that they will use only the value of f and the value of its gradient. If for some reasons, a local minimizer has to be found (and not only a stationary point) and if there are many stationary points that are not minimizers, something more has to be done. A good idea is to use second order information to locate directions of negative curvature. Along this line, see the papers by Gill and Murray (1974), McCormick (1977), Moré and Sorensen (1979) and Goldfarb (1980).

An important class of methods generating sequences of approximations of a solution is formed by the *descent-direction methods.* They proceed as follows. It is assumed that some approximation x_k is known at the beginning of iteration k, $k \geq 1$ (x_1 has to be given at the beginning of the run). Then, an iteration consists of calculating a new approximation x_{k+1} that is better than the previous one, in the sense that

$$f(x_{k+1}) < f(x_k). \tag{3}$$

To obtain this *descent property*, the first stage consists in choosing a *descent direction* at x_k, i.e., a direction d_k verifying

$$f'(x_k)\cdot d_k < 0. \tag{4}$$

The second stage consists in selecting a positive step-size ρ_k along d_k and taking

$$x_{k+1} = x_k + \rho_k\, d_k$$

such that x_{k+1} satisfies, at least, the descent property (3). This stage is called the *line-search.* Let us summarize this method as follows.

The descent-direction algorithm:

```
1.    choose $x_1 \in \Omega$;
2.    $k := 1$;
3.    until convergence do {
3.1.      choose a descent direction $d_k$;
3.2.      select a positive step-size $\rho_k$ along $d_k$ such that (3) holds;
3.3.      $x_{k+1} := x_k + \rho_k\, d_k$;
3.4.      $k := k + 1$
3.    }
```

The way of calculating the descent direction characterizes a descent-direction method. For example, when $g_k := g(x_k) \neq 0$, an obvious descent direction is:

$$d_k = -g_k. \tag{5}$$

The method that uses this descent direction at each iteration is called the *gradient method* or the *steepest descent method*. More generally, if H_k is a positive definite matrix of order n,

$$d_k = -H_k\, g_k \tag{6}$$

will be a descent direction. Many practical algorithms for minimizing a function have this form of descent direction: the Newton method (near a solution), the BFGS method (or more generally, algorithms in the family of quasi-Newton methods) and even conjugate gradient methods, as we shall see in Section 5.

The descent condition (4) implies that for sufficiently small positive step-size ρ, we have $f(x_k + \rho d_k) < f(x_k)$. Therefore, to satisfy (3), a step-size selection procedure could simply choose a small value for ρ. This is not satisfactory, however, because examples can be given for which taking arbitrarily small positive step-sizes forces a descent algorithm to generate a sequence $\{x_k\}_{k\geq 1}$ that converges to a nonstationary point (*false convergence*). This phenomenon is dangerous and hopefully, it can be avoided in several ways. We mention two of them.

A first idea is to minimize the function $\xi_k(\rho) := f(x_k + \rho d_k)$ for positive ρ. We shall say that a step-size selection procedure realizes an *optimal line-search* if it selects an *optimal step-size*, that is to say a positive step-size ρ_k such that $x_{k+1} = x_k + \rho_k d_k$ satisfies

$$f(x_{k+1}) \leq f(x_k + \bar{\rho}_k\, d_k)\,, \tag{7}$$

where $\bar{\rho}_k$ is the smallest positive stationary point of the function ξ_k. A step-size that is a stationary point of ξ_k will be said to be *exact* and is obtained by a *exact line-search*. Actually, the optimal and exact line-search strategies have two drawbacks. First, the step-size $\bar{\rho}_k$ may not exist at some iteration although problem (1) has a solution. Secondly, finding a stationary point of ξ_k is a nonlinear minimization problem in itself and this cannot be solved exactly, except in particular cases (for instance, when f is quadratic). Therefore, although this procedure is theoretically suitable (see Theorem 1 below), it is useless in practice.

A better choice is to satisfy some inequalities that will guarantee convergence if the descent directions are reasonable. A first inequality assures sufficient decrease of the objective function: for $\omega_1 \in \,]0, 1[$ choose ρ_k such that

$$f(x_{k+1}) \leq f(x_k) + \rho_k\, \omega_1 \langle g_k, d_k\rangle\,. \tag{8}$$

This inequality is always satisfied for small positive ρ_k. In order to avoid too small step-sizes one have to satisfy another inequality like the following:

$$\langle g(x_{k+1}), d_k\rangle \;\geq\; \omega_2 \langle g_k, d_k\rangle \tag{9}$$

with $\omega_2 \in \,]\omega_1, 1[$. Conditions (8) and (9) are called the *Wolfe conditions* and a point x_{k+1} satisfying them is called a *Wolfe point*. It can be proved (see Wolfe (1969)) that if f is bounded below and if $0 < \omega_1 < \omega_2 < 1$, it is always possible to find a Wolfe point. Typical values for the parameters are $\omega_1 = 10^{-4} \ldots 10^{-2}$ and $\omega_2 = 0.5 \ldots 0.99$. In any case, it is healthy to take $\omega_1 < 0.5$, so that the step-size minimizing the quadratic approximation of ξ_k will be accepted by the inequality (8). Calculating a Wolfe point requires a specific algorithm. The one proposed by Lemaréchal (1981) has been proved to find a Wolfe point in a finite number of trials. For some algorithms, we shall need a step-size satisfying the *strong Wolfe conditions*, that is to say, satisfying (8) and an inequality stronger than (9), namely

$$|\langle g(x_{k+1}), d_k\rangle| \leq \omega_2 \, |\langle g_k, d_k\rangle| \,. \tag{10}$$

An algorithm for finding a strong Wolfe point can be found in [2] and [42].

The preceding two line-search techniques allow us to obtain the following result that can be derived from the papers by Zoutendijk (1970) and Wolfe (1969 and 1971). Before stating the theorem, let us specify some notation. We shall denote by θ_k the angle between d_k and $-g_k$, that is

$$\cos\theta_k := -\frac{\langle g_k, d_k\rangle}{|g_k|\,|d_k|} \,.$$

The notation "$\liminf \epsilon_k$" means "$\lim_{k\to\infty} \inf_{1\leq i\leq k} \epsilon_i$".

Theorem 1 *Let f be a differentiable function defined on the open set Ω, with a Lipschitz continuous derivative. Suppose that a descent-direction method generates a sequence of points $\{x_k\}_{k\geq 1}$ in Ω and a sequence of directions $\{d_k\}_{k\geq 1}$ verifying (4). Suppose also that at each iteration the line-search procedure finds either an optimal point satisfying (7) or a Wolfe point satisfying (8) and (9). Suppose finally that $\{f(x_k)\}_{k\geq 1}$ is bounded below. Then, the following conclusions hold:*

(i) $\sum_{k\geq 1} |g_k|^2 \cos^2\theta_k$ *is convergent;*
(ii) $|g_k| \cos\theta_k \to 0$*;*
(iii) *if* $\sum_{k\geq 1} \cos^2\theta_k$ *is divergent, then* $\liminf |g_k| = 0$.

The theorem says nothing on the convergence of the sequence of points $\{x_k\}_{k\geq 1}$. Only the sequence of gradients is concerned, although it is not claimed that g_k converges to zero. Actually, Theorem 1 expresses the contribution of the line-search procedure to force convergence. Now, to make its conclusions useful, some more information concerning the way the descent directions d_k are chosen is needed. For instance, if the angle θ_k remains bounded away from $\pm\pi/2$, then $\cos\theta_k$ remains bounded away from 0 and the conclusion (ii) of Theorem 1 implies that the sequence of gradients converges to zero and therefore that any limit point of $\{x_k\}_{k\geq 1}$ is stationary. This is the case for the gradient method, because $\theta_k = 0, \forall k \geq 1$. This is also the case for the method (6), if the condition numbers $\kappa(H_k)$ of the positive definite matrices H_k remain bounded; that is, if there exists a positive constant $\bar\kappa$ such that

$$\kappa(H_k) := \|H_k\| \, \|H_k^{-1}\| \leq \bar\kappa, \ \forall k \geq 1.$$

Indeed, we have $\cos\theta_k \geq \kappa(H_k)^{-1}$.

These remarks indicate that to be sure to have a convergent algorithm, it is enough to use method (6) with constant matrices H_k (the identity matrix, for instance) or with matrices having bounded condition number. Our task is not finished, however. Indeed, such directions can be very poor descent directions in the sense that the decrease of the objective function may be small along them. Therefore, in the subsequent sections, we shall play the dangerous game of trying to define "good" descent directions by taking information from the function with the risk of violating the conditions of convergence stated in Theorem 1. For some of the methods below, it is still not clear whether they can fail to converge when they aim at minimizing general functions.

2 Automatic differentiation implementations

As we have seen, the gradient $\nabla f(x)$ of the function f to minimize gives precious information on the behavior of f around x. Efficient algorithms always use this information.

Sometimes, the cost of the computation of the gradient has been invoked to motivate users' preference for methods only using the values of the function or to use divided differences to evaluate the gradient. The latter choice is very expensive, since it usually requires n more evaluations of the function. It has also low accuracy. These two reasons should be sufficiently deterrent factors for not using it.

Fortunately, for about a decade, the technique known as the *reverse mode* of automatic differentiation has provided a way of computing the gradient *exactly* (up to the precision of the arithmetic of the computer) at a *relative* cost independent of the number of variables, meaning that

$$\kappa \equiv \frac{\mathcal{T}(f,\nabla f)}{\mathcal{T}(f)} \leq C. \tag{11}$$

Here, $\mathcal{T}(f)$ and $\mathcal{T}(f,\nabla f)$ measure the time to compute f and the pair $(f,\nabla f)$, respectively, and C is a constant independent of n. This constant is small, between 3 and 5, depending on the operations taken in consideration [4, 43, 23]. The method requires that the function to differentiate be described by a computer program. It is also a computer program that will represent the gradient function. This technique is opposed to the *direct mode* of automatic differentiation, which is, in spirit and cost, very similar to the divided difference gradient evaluation. For review papers on the subject, see [23, 24, 16].

The reverse mode goes back at least to Linnainmaa (1976), who introduced it for error propagation analysis, and to Speelpenning (1981), whose interest was precisely the automatic differentiation of functions represented by program. For short (but with a little inaccuracy), the method can be seen as an adjoint method to compute the gradient of f, the lines of the program evaluating f being considered as constraints. More specifically, the method basically deals with programs computing f by a sequence of instructions:

$$\left\{ \begin{array}{l} x_{\mu_k} := \varphi_k(x_{P_k}), \quad \text{for } k = 1,\ldots,K, \\ f := x_N. \end{array} \right. \tag{12}$$

We have denoted by $x_1,\ldots,x_n$ the *independent variables*, i.e. the variables defining the function $f(x) = f(x_1,\ldots,x_n)$. The other *intermediate* $N-n$ *variables* appearing in the program are denoted by $x_{n+1},\ldots,x_N$. Instruction (12) modifies the variable x_{μ_k}, for some $1 \leq \mu_k \leq N$, by means of an *intermediate function* φ_k, which depends on $x_{P_k} \equiv \{x_j : j \in P_k\}$, P_k being some subset of $\{1,\ldots,N\}$. The gradient can be evaluated by introducing variables p_k, $1 \leq k \leq N$ (p_k

is called the *dual* – or *adjoint* – variable to x_k), initialized to

$$p_k := \begin{cases} 0 & \text{for} \quad k = 1, \ldots, N-1 \\ 1 & \text{for} \quad k = N, \end{cases} \tag{13}$$

and evaluated *backwards* by the following *adjoint program*:

$$\left.\begin{array}{l} p_j := p_j + \dfrac{\partial \varphi_k}{\partial x_j} p_{\mu_k}, \quad \forall j \in P_k \backslash \{\mu_k\} \\ p_{\mu_k} := \dfrac{\partial \varphi_k}{\partial x_{\mu_k}} p_{\mu_k} \end{array}\right\}, \quad \text{for} \quad k = K, \ldots, 1. \tag{14}$$

Then, the gradient is obtained by

$$\frac{\partial f}{\partial x_i} = p_i, \quad \text{for} \quad i = 1, \ldots, n. \tag{15}$$

The partial derivatives of φ_k used in (14) have to be evaluated during the "forward sweep" (12), and stored. This delay between the evaluation of $\partial\varphi_k/\partial x_j$ and its use in (14) gives rise to different implementations of the reverse mode.

In the first implementation, the calculations made in (12) are stored in a data structure representing the computational graph of the program. The size of this graph is, as a result, proportional to the number of elementary operations executed to obtain $f(x)$. In other words, the size is proportional to the computation time $\mathcal{T}(f)$. This is not a very desirable feature and constitutes the main drawback of the approach, limiting its use to small or medium size problems. On this point, some promising remedies have been proposed to control the memory requirement. The price to pay is an increase in computation time: see Griewank (1991b), for instance. On the other hand, this approach has a number of nice features: the implementation can be easily done by operator overloading [26], the technique can face programs with intricated structures (recursive subroutines, goto's, ...) and higher order derivatives can also be obtained, using a hardly more complicated algorithm than (13)–(15). Representatives of this approach are JAKEF of Speelpenning (1980) and Hillstrom (1985), PADRE2 of Iri and Kubota (1990) and ADOL-C of Griewank et al. (1991).

The other approach is the *adjoint code* implementation, as used in the meteorology world since its introduction by Ph. Courtier and O. Talagrand – see [60]. The idea is to write a program, following the rules given by formula (14). Here, the structure of the calculation is stored in a program, not in a computational graph. The main advantage of this approach is its sobriety in memory requirement. Indeed, the only pieces to memorize are the values of the variables entering *non linearly* in the intermediate functions φ_k, since those entering linearly no longer appear in (14). From these values, the adjoint code will evaluate the partials $\partial\varphi_k/\partial x_j$ during the backward sweep. Interestingly enough, this implies that, for the differentiation of *linear* functions, nothing has to be stored during the direct calculation (12). This is in complete opposition with the computational graph approach. The approach also offers the possibility to recalculate some variables in the adjoint code, instead of memorizing them. Unfortunately, there is presently no automatic generation of adjoint codes, so that they have to be written by hand.

We now give the results of some tests made in [16], comparing the efficiency of the automatic differentiators JAKEF, PADRE2, ADOL-C with the divided difference approach (DD), as well as the hand-written adjoint code approach (METEO). All tests have been made on a SPARCstation 1. In Tables 1 and 2, label "Kb" refers to the additional memory requirement in Kbytes, label "sec" gives the CPU time (in seconds) for the computation of f and/or ∇f and label "κ" is the quotient in (11).

The first test-problem we consider is U1MT1, a program determining atmospheric pressure and velocity using a meteorological model and measurements collected at various points. The least-square function evaluated in this code depends on $n = 1875$ variables and requires 34.3 Kbytes of memory. Results are presented in Table 1.

Type of calculation	Kb	sec	κ
f alone	—	0.04	—
original f and ∇f	7.5	0.23	5.8
f and ∇f by DD	—	83.86	—
f and ∇f by JAKEF	1142.4	1.19	29.8
f and ∇f by PADRE2	1348.3	1.14	28.5
by ADOL-C:	1362.4	—	—
f with graph	—	4.75	—
f without graph	—	2.24	—
∇f with graph	—	5.50	—
∇f without graph	—	3.14	79.5
∇f by METEO	14.8	0.09	3.3

Table 1: Comparison of performance on U1MT1

The additional memory spaces used by JAKEF, PADRE2 and ADOL-C are approximately identical: it is mainly used to store the computational graph. This is much more than what is necessary in the original program or for the adjoint code method (METEO). For the latter, we chose to store only one additional vector and to recompute in the backward phase (14) all the other variables appearing nonlinearly in the code.

The computing times were obtained by averaging on 10 trials (except for DD!). The CPU time used by the divided difference method is very important: it exceeds n times the one of f, since for some components, the increment dx_i is readapted and the difference recomputed. The times realized by JAKEF and PADRE2 are similar and a bit less than the one of ADOL-C (for the latter, "f with graph" is the time to build the computational graph, "f without graph" and "∇f without graph" are the times to compute f and ∇f using the graph previously built). Note that with ADOL-C, the code is translated from FORTRAN to C++ (using f2c from [13]) and is executed in part in double precision, while only single precision arithmetic is used with the other codes. This may explain the difference in computing time between ADOL-C and the pair JAKEF-PADRE2. The time realized by METEO is really amazing and encouraging. It is less than half the one used by the original code and less than 10 times the one used by JAKEF or PADRE2, despite our choice to recalculate most of the intermediate variables. Note finally that only the method METEO computes the pair $(f, \nabla f)$ at a relative cost not exceeding 5, as expected by the theory.

Another example will highlight the fact that the increase in memory space required by the computational graph implementation, due to the increase in the number of elementary operations for the computation of f, results in a deterioration of the performance of the reverse mode. We consider the following subroutine LOOPS, which computes $f(x) = (\mathtt{ntimes})^3\|x\|^2$ in a laborious way: three nested do-loops ranging from 1 to ntimes are used to obtain $(\mathtt{ntimes})^3$ and $\|x\|^2$ is calculated in the innermost do-loop.

```
   subroutine loops (n,ntimes,x,f)
   integer n,ntimes
   real x(n),f
   integer i,j,k,l
   real r
   f=0.
   do 40 i=1,ntimes
      do 30 j=1,ntimes
         do 20 k=1,ntimes
            do 10 l=1,n
               r=x(l)
               f=f+r*r
10          continue
20       continue
30    continue
40 continue
   return.
```

Results are given in Table 2 for $n = 10$ and ntimes ranging from 5 to 25. We observe that

	f	JAKEF			PADRE2			METEO	
ntimes	sec	Kb	sec	κ	Kb	sec	κ	sec	κ
5	0.01	78.	0.08	8.0	49.	0.04	4.0	0.01	2.0
10	0.03	625.	0.64	21.3	390.	0.38	12.7	0.04	2.3
15	0.10	2109.	2.19	21.9	1318.	1.25	12.5	0.13	2.3
20	0.21	5000.	5.24	25.0	3125.	3.20	15.2	0.32	2.5
25	0.41	9765.	10.41	25.4	6103.	7.21	17.6	0.62	2.5

Table 2: Comparison of performance on LOOPS

for JAKEF and PADRE2, the memory requirement (in Kb) increases as $(\mathtt{ntimes})^3$. We also see that, for JAKEF and PADRE2, the quotient κ deteriorates as **ntimes** increases. This has to be attributed to the computational graph implementation of the reverse mode. On the other hand, it can be shown that, for METEO (at least for our way of writing the adjoint code), κ is less than or equal to 3 when **ntimes** and n are large. This is what we observe in Table 2. These numerical results are very encouraging for the adjoint code approach of the reverse mode of automatic differentiation.

Suppose now that we are given a vector-valued function $F : \mathbb{R}^n \to \mathbb{R}^n$. As we have seen, the reverse mode of automatic differentiation will compute cheaply the gradient $\nabla F_i(x)$ of a component of F, i.e. a line of the Jacobian matrix $\nabla F(x)$. On the other hand, the direct mode can computes cheaply the directional derivative $F'(x) \cdot e_j$, i.e. a column of $\nabla F(x)$. Therefore, it makes sense to ask whether there exists a method, DIAG say, that would compute the n elements of the diagonal of the Jacobian at approximately the same price as a line or a column. The question is important since the diagonal can be a good preconditioner for many numerical algorithms. Apparently, there is no definite answer to this question but it can be shown ([16]) that if such an algorithm existed, it would be possible to compute the product of 2 matrices in $\mathcal{O}(n^2)$ operations. As the best presently known method computes this product in $\mathcal{O}(n^{2.50})$ operations, our result puts some doubt on the existence of DIAG. It also suggests that it is more valuable to know the diagonal of the Jacobian rather than one of its lines or columns.

3 Conjugate gradient methods without optimal line-searches

The conjugate gradient (CG) method was introduced to solve linear systems of equations [28] and later to minimize strongly convex quadratic functions [15]. The basic idea is to generate a set of conjugate directions, i.e., directions that are orthogonal with respect to the inner product induced by the Hessian matrix A of f. As a result, f is minimized on the larger and larger affine subspace spanned by the successive directions, the *Krylov subspace*. When the first direction is $-g_1$, the conjugacy of the directions can be obtained by the following simple formula:

$$d_k = -g_k + \beta_k^{\mathrm{HS}}\, d_{k-1}\,,$$

where β_k^{HS} is the Hestenes-Stiefel beta (see [28]):

$$\beta_k^{\mathrm{HS}} := \frac{\langle y_{k-1}, g_k\rangle}{\langle y_{k-1}, d_{k-1}\rangle}\,.$$

We note $y_k := g_{k+1} - g_k$ and $s_k := x_{k+1} - x_k$. To have conjugacy, it is necessary to do optimal line-searches, which does not pose any difficulty in case of quadratic functions.

```
The linear CG algorithm:
1.    choose a point $x_1 \in \Omega$;
2.    $d_1 := -g_1$; $k := 1$;
3.    repeat {
3.1.      $\rho_k := -\langle g_k, d_k\rangle / \langle A d_k, d_k\rangle$;      /* the optimal step-size */
3.2.      $x_{k+1} := x_k + \rho_k\, d_k$;
3.3.        if convergence then stop;
3.4.        $d_{k+1} := -g_{k+1} + \beta_{k+1}^{\mathrm{HS}}\, d_k$;
3.5.        $k := k+1$
3.    }
```

Usually, real-life large-scale models are not quadratic. The question is therefore to know whether, for minimizing a general nonlinear function, it still makes sense to use a CG-like method defined for some scalar β_k by

$$\begin{cases} d_1 = -g_1 \\ d_k = -g_k + \beta_k\, d_{k-1}\,, \text{ for } k \geq 2\,. \end{cases} \tag{16}$$

Note that for non quadratic functions, it is still possible to define a local notion of conjugacy by using the average Hessian of f:

$$\bar{B}_k = \int_0^1 \nabla^2 f(x_k + ts_k)\, dt\,.$$

Indeed, because $y_k = \bar{B}_k s_k$, equation (16) with $\beta_k = \beta_k^{\mathrm{HS}}$ implies that $\langle \bar{B}_k\, d_k, d_{k+1}\rangle = 0$ for $k \geq 1$. Therefore, d_k and d_{k+1} are conjugate with respect to $\bar{B}_k$. However, it does not seem that this concept is very helpful in the non quadratic case and therefore, on this basis only, it is hard to claim that the HS beta is better than any other beta.

Another point is to know whether the directions defined by (16) are downhill: do we have $\langle g_k, d_k\rangle$ negative? As $\langle g_k, d_k\rangle = -|g_k|^2 + \beta_k\langle d_{k-1}, g_k\rangle$, an obvious way of getting descent directions is to do exact line-searches: $\langle d_{k-1}, g_k\rangle = 0$. We have said that such strategy is not realistic. Hopefully, it can be avoided. The following theorem (see [18, Theorem 3.2]) is an extension of a result of Al-Baali (1985). It shows that provided $|\beta_k|$ is not too large, a line-search procedure realizing the strong Wolfe conditions automatically ensures descent of the direction d_k. Furthermore, global convergence can be proved. Note that this result strongly relies on the fact that the first search direction is $-g_1$. In this theorem, the Fletcher-Reeves beta is used to control the magnitude of β_k. It is defined by (see [15]):

$$\beta_k^{\mathrm{FR}} := \frac{|g_k|^2}{|g_{k-1}|^2}\,.$$

For quadratic functions and optimal line-searches, $\beta_k^{\mathrm{HS}} = \beta_k^{\mathrm{FR}}$.

Theorem 2 *Let f be a twice continuously differentiable function defined on the open set $\Omega = I\!R^n$ and let x_1 be a point in Ω. Suppose that the level set $\{\, x \in \Omega \;:\; f(x) \leq f(x_1)\,\}$ is bounded. If the nonlinear* CG *method (16) with $|\beta_k| \leq \beta_k^{\mathrm{FR}}$ starts from x_1 and uses the strong Wolfe line-search with $0 < \omega_1 < \omega_2 < 1/2$, then the directions d_k are downhill and* $\liminf |g_k| = 0$.

This result could let think that the CG method (16) with $\beta_k = \beta_k^{\mathrm{FR}}$ and the strong Wolfe line-search is a good algorithm. Unfortunately, it is not the case. The point is that this theorem only ensures the global convergence of the method, i.e., the convergence from any starting point, but not its fast convergence. In fact, numerical experiments with the Fletcher-Reeves formula have shown that the method can converge very slowly. A partial clarification of this phenomenon has been given by Powell (1977, 1985) who gave a reason why the Polak-Ribière beta, defined by (see [51]):

$$\beta_k^{\mathrm{PR}} := \frac{\langle y_{k-1}, g_k \rangle}{|g_{k-1}|^2},$$

should by used instead of β_k^{FR} (for exact line-searches, $\beta_k^{\mathrm{HS}} = \beta_k^{\mathrm{PR}}$). Numerical experiments have confirmed the superiority of the PR beta over the FR beta.

The misfortune is that the PR method is not as safe as the FR method! If it can be proved that the method is convergent for convex functions (see [51]), Powell (1984) has given an example of a nonconvex function where the PR method diverges: the generated sequence cycles and each of its limit points is nonstationary! Two globally convergent remedies to this situation have been proposed in [18]. The first one is inspired by Theorem 2. It consists of using the PR beta only when it lies in the interval $[-\beta_k^{\mathrm{FR}}, +\beta_k^{\mathrm{FR}}]$. Specifically,

$$\textbf{Method PR|FR}: \quad \beta_k = \begin{cases} -\beta_k^{\mathrm{FR}} & \text{if } \beta_k^{\mathrm{PR}} < -\beta_k^{\mathrm{FR}} \\ \beta_k^{\mathrm{PR}} & \text{if } |\beta_k^{\mathrm{PR}}| \le \beta_k^{\mathrm{FR}} \\ \beta_k^{\mathrm{FR}} & \text{if } \beta_k^{\mathrm{PR}} > \beta_k^{\mathrm{FR}}. \end{cases}$$

This strategy avoids one of the main disadvantages of the FR method, as discussed in [18].

Another globally convergent remedy consists of restarting the PR method in the opposite direction of the gradient, each time β_k^{PR} is negative. Synthetically, this corresponds to the following choice of β_k:

$$\textbf{Method PR}^+: \quad \beta_k = \max\{0, \beta_k^{\mathrm{PR}}\}.$$

For this method, a specific line-search assuring descent of the directions has to be implemented.

FR	PR|FR	PR	PR$^+$
> 4.07	1.55	1.02	1.00

Table 3: Comparison of the methods FR, PR|FR, PR and PR$^+$.

Table 3 allows us to compare the performance of several CG methods: FR, PR, PR|FR and PR$^+$. The results come from tests made in [18] on a collection of 26 large-scale problems with a number of variables between 100 and 10000. The table gives the total number of function/gradient calls used for all the test-problems divided by the 18792 calls required by method PR$^+$. The sign $>$ means that the method was stopped on some test-problems because the number of function/gradient calls exceeds 9999. The actual number should therefore be greater than the one given in the table. Methods PR and PR$^+$ appear to be the best one, while the FR method is on average more than four times slower than method PR$^+$, but this behavior of the FR method is well known.

4 The Newton and truncated Newton algorithms

4.1 The Newton method

The Newton method for solving nonlinear equations (here $\nabla f(x) = 0$) is well known. It consists in linearizing the equations and solving them. Here, this gives the following *linear* system to solve in d:

$$\nabla^2 f(x_k)\, d + g_k = 0,$$

where $\nabla^2 f(x_k)$ denotes the Hessian matrix of f at x_k. This matrix also depends on the inner product. If $\nabla^2 f(x_k)$ is nonsingular, the solution is

$$d_k^N = -\nabla^2 f(x_k)^{-1} g_k \,. \tag{17}$$

The direction has the form (6) and is a descent direction when $\nabla^2 f(x_k)$ is positive definite, which is the case when x_k is close to "normal" solution.

An attractive feature of Newton's method is its rate of convergence. If the function f is sufficiently smooth and if x_1 is close enough to x_*, then the method generates a *quadratically convergent* sequence $\{x_k\}_{k\geq 1}$, that is

$$||x_{k+1} - x_*|| \leq C\,||x_k - x_*||^2,\ \forall k \geq 1\,,$$

for some positive constant C. This property implies that if x_k is close enough to x_*, its number of correct significant digits grows very quickly. Quadratic convergence is typical of algorithms using second order information, i.e. second derivatives.

If convergence occurs with the Newton method when the initial point x_1 is close to a solution, divergence is often observed if the guess is not good enough. This drawback can be tempered by line-search techniques as described in Section 1 or by a trust region approach (see the excellent survey by Moré (1983)). Other difficulties may occur. For example, when the Hessian is not positive definite, the Newton direction may no longer be downhill. One remedy to this situation is to modify the Hessian in equation (17) to obtain a positive definite matrix: one of the most effective way of doing this is due to Gill and Murray (1974). Another remedy is to solve the linear system (17) within a trust region. A last possibility, which can be extended to large-scale problems, is the *truncated Newton method.*

4.2 The truncated Newton method

Newton's method is difficult to use for large-scale problems because the storage of the full Hessian requires $\mathcal{O}(n^2)$ memory locations and the solution of Newton's equation (17) demands $\mathcal{O}(n^3)$ operations. The truncated Newton method attempts to overcome these two difficuties.

The basic idea is to solve only partially the linear system (17), using an iterative procedure (formed of *inner iterations*, as opposed to the *outer iterations* of Newton's method). For instance, the CG method can be used as long as the possible non definite positiveness of the Hessian is not detected [6]. Then, it can be shown that the resulting direction is a descent direction of f (and a negative curvature direction if the process has been interrupted). Another possibility would be to use a Lanczos tridiagonalization process [47].

These two methods are suitable for large-scale problems, since the only information used from the Hessian is its action on a vector v. There are different possibilities to evaluate the product Hessian $*$ vector (see, for instance, the discussion in [49]), but when the Hessian is dense, it is generally *approximated* by divided differences:

$$\nabla^2 f(x_k)\, v \simeq \frac{\nabla f(x_k + \epsilon v) - \nabla f(x_k)}{\epsilon},$$

where ϵ is a small positive number. This formula costs one gradient evaluation. Now, automatic differentiation offers the possibility to obtain this product *exactly* and cheaply. Indeed, $\nabla^2 f(x)\, v$ is the directional derivative of $x \to \nabla f(x)$ in the direction v. If the gradient has been represented by an adjoint code (reverse mode), the product $\nabla^2 f(x)\, v$ can be evaluated by the direct mode at a cost similar to one gradient evaluation.

5 The quasi-Newton, LBFGS and partitioned quasi-Newton methods

5.1 Quasi-Newton methods

The Newton method requires the calculation of the Hessian matrix of f, which can be in certain cases, either too expensive or impossible. This is precisely the domain where the quasi-Newton (QN) methods can be useful. See the review paper by Dennis and Moré (1977).

The descent direction of a QN method has the form (6), where H_k is a matrix that is updated at each iteration by some formula. Hence, the method generates two sequences: a sequence of points $\{x_k\}_{k\geq 1}$ and a sequence of matrices $\{H_k\}_{k\geq 1}$. At the beginning of the run, the algorithm is given a point x_1 *and* a matrix H_1. Often, H_1 is the identity matrix (we shall give better choices later) or a diagonal matrix, like the diagonal of the Hessian $\nabla^2 f(x_1)$, when it is available.

The general scheme of a QN method is the following.

```
The QN algorithm:
1.    choose $x_1 \in \Omega$ and a positive definite matrix $H_1$;
2.    $k := 1$;
3.    until convergence do {
3.1.      $d_k := -H_k\, g_k$;
3.2.      select a step-size $\rho_k$ by the Wolfe procedure (8) and (9);
3.3.      $x_{k+1} := x_k + \rho_k\, d_k$;
3.4.      calculate $H_{k+1}$;
3.5.      $k := k + 1$
3.    }
```

The only thing to examine in this algorithm is the way the matrix H_k is updated. If we set $B_k = H_k^{-1}$ (we shall see how to maintain H_k positive definite, hence nonsingular), we see that

B_k plays the part of $\nabla^2 f(x_k)$ in Newton's method. On the other hand, with $s_k := x_{k+1} - x_k$ and $y_k := g_{k+1} - g_k$, we have by Taylor's theorem:

$$y_k = \left(\int_0^1 \nabla^2 f(x_k + t s_k)\, dt \right) s_k .$$

These two remarks show that it makes sense to impose the matrix B_{k+1}, which has to be calculated at the end of iteration k, to satisfy:

$$y_k = B_{k+1} s_k .$$

This equation is called the *secant* or *quasi-Newton equation.* If $n \geq 2$, the equation does not determine B_{k+1} completely and some other conditions can be imposed. Generally, they make B_{k+1} keeping information from B_k. There is today a wide consensus that one of the best update formulae for minimization is the BFGS update. To write this formula, it is convenient to use the following *tensor product* of two vectors u and v of $I\!R^n$: $u \otimes v$ is the rank one matrix defined by $(u \otimes v)h = \langle v, h \rangle u\,, \forall h \in I\!R^n$. If the Euclidean inner product is used, $u \otimes v$ is the matrix uv^T. See the appendix in [17] for a derivation of the BFGS formula in this setting, which differs from the usual one by the use of a general inner product. The BFGS *formula* can be written like this:

$$B_{k+1} = B_k + \frac{y_k \otimes y_k}{\langle y_k, s_k \rangle} - \frac{(B_k s_k) \otimes (B_k s_k)}{\langle B_k s_k, s_k \rangle} =: U(B_k,\, y_k,\, s_k) . \tag{18}$$

Note that the inverse matrices H_k can also be updated by using the *inverse* BFGS *formula*:

$$H_{k+1} = \left(I - \frac{s_k \otimes y_k}{\langle y_k, s_k \rangle} \right) H_k \left(I - \frac{y_k \otimes s_k}{\langle y_k, s_k \rangle} \right) + \frac{s_k \otimes s_k}{\langle y_k, s_k \rangle} =: \bar{U}(H_k,\, y_k,\, s_k) . \tag{19}$$

Precisely, if $H_k = B_k^{-1}$, $B_{k+1} = U(B_k,\, y_k,\, s_k)$ and $H_{k+1} = \bar{U}(H_k,\, y_k,\, s_k)$, then $H_{k+1} = B_{k+1}^{-1}$.

Formulae (18) and (19) have the nice property of transmitting the positive definiteness of B_k to B_{k+1} (respectively of H_k to H_{k+1}), if and only if $\langle y_k, s_k \rangle$ is positive. Therefore to have H_k positive definite, it will suffice to take H_1 positive definite, to update the matrices by the inverse BFGS formula and to ensure the positivity of $\langle y_k, s_k \rangle$ at each iteration. Note that for strictly convex functions, $\langle y_k, s_k \rangle$ is automatically positive and that for nonconvex functions, the second inequality (9) of the Wolfe line-search implies the positivity of $\langle y_k, s_k \rangle$. This important remark makes the Wolfe step-size selection procedure the most suitable for the QN framework in optimization. It justifies also Statement 3.2 of the QN algorithm.

For problems having small or medium dimension, the BFGS method is very efficient. It is globally convergent for convex problems (see [52]) and for general functions, when convergence occurs, it is fast, namely *superlinearly convergent.* This means that

$$||x_{k+1} - x_*|| / ||x_k - x_*|| \to 0 .$$

Although, this is less good than the quadratic rate of convergence of the Newton method, it is generally satisfactory in practice.

5.2 The limited memory BFGS method

There are many proposals to adapt QN methods to large-scale problems and many of them fit into the following framework.

Although n may be large, the initial matrix H_1 takes generally little space in memory: it is most commonly a positive multiple of the identity matrix. On the other hand, H_k is formed from H_1 and $k-1$ couples $\{(y_i,\, s_i)\}_{1\leq i<k}$ by using formula (19). Therefore, to take less storage, one can think of storing these elements instead of H_k and computing $H_k\, g_k$ by an appropriate algorithm. Of course, when the number of iterations increases, these pieces of information become more and more cumbersome in memory and we must get rid of some of the couples $(y_i,\, s_i)$. A method will be called an *m-storage* QN *method* if only m of these couples are used to form H_k from a *starting matrix* H_k^0. In this type of method, the inverse update formula (19) is preferable to the direct formula (18), because the inversion of B_k may be problematic.

The most famous limited memory methods can be seen from the preceding point of view: see [17] for the details. For example, the algorithm CONMIN of Shanno and Phua (1976) is a 2-storage QN method (see also [57]). The algorithm of Buckley and LeNir (1983) is an m-storage QN method where m varies with the iteration index. Finally, the method proposed by Nocedal (1980) is also of this type. We describe it in more detail because we feel it is important.

Nocedal proposed to use always the last m couples (y,s), i.e., $\{(y_i,s_i)\}_{k-m\leq i\leq k-1}$, to form the matrix H_k: going from iteration k to iteration $k+1$, the couple (y_{k-m},s_{k-m}) is discarded and cyclically replaced by the new couple (y_k,s_k). Therefore, H_k is always formed from the most recent information and, as a result, it should be more appropriate.

Description of Nocedal's proposal to calculate H_k:

```
1.   choose a positive definite matrix $H_k^0$;
2.   for $i := 0$ to $m-1$ do  $H_k^{i+1} := \bar{U}(H_k^i,\, y_{k-m+i},\, s_{k-m+i})$;
3.   $H_k := H_k^m$;
```

Of course, this scheme is formal: in practice, the matrices H_k^i and H_k are not stored (they are too large). Instead, just H_k^0 and the couples $\{(y_i,s_i)\}_{k-m\leq i\leq k-1}$ are kept in memory and an elegant algorithm, described in [57], is provided to compute $H_k\, g_k$.

Note that $H_k\, g_k$ can be considered as the gradient *in* g_k of the function $g_k \to \varphi(g_k) := \frac{1}{2}\langle H_k g_k, g_k\rangle$. Interestingly enough, this observation has been used in [19] to show that the compact algorithm computing $H_k g_k$ in [45] is in fact an adjoint code (in the sense of automatic differentiation, see Section 2) of a program computing $\varphi(g_k)$. This shows the usefulness of automatic differentiation, in particular its reverse mode by adjoint coding, not only to compute the gradient of a numerical function represented by a program, but also to find new algorithms.

The above description of Nocedal's proposal outlines the need of a choice for the starting matrices H_k^0. Contrary to the standard QN methods where this choice has to be made only once at the beginning of a run, here it has to be made at each iteration. Its role is therefore crucial. One of the aims of the work [17] is to examine several possible starting matrices. From this study, two choices emerge.

A first choice is simply to take a judicious multiple of the identity matrix:

$$H_k^0 = \delta_{k-1} I \, , \quad \delta_{k-1} := \frac{\langle y_{k-1}, s_{k-1} \rangle}{|y_{k-1}|^2} \, . \tag{20}$$

The factor δ_{k-1} is known as the *Oren-Spedicato factor* (see [48]) and aims at giving H_k^0 a good scaling. Some justifications for using this factor are given in [17].

The other possibility is to take a diagonal matrix and to update it at each iteration. In this case, the algorithm generates and stores two sequences, $\{x_k\}_{k \geq 1}$ and $\{H_k^0\}_{k \geq 1}$, and a circular order array for the couples (y_i, s_i). Then H_k is formed by updating m times H_k^0, using the m couples $\{(y_i, s_i)\}_{k-m \leq i \leq k-1}$. At the beginning, $H_2^0 = \delta_1 I$. Then, for $k \geq 2$, the formula for updating the i-th diagonal element of H_k^0 is the following:

$$(H_{k+1}^0)^{(i)} = \left(\frac{\langle H_k^0 y_k, y_k \rangle}{\langle y_k, s_k \rangle (H_k^0)^{(i)}} + \frac{\langle y_k, e_i \rangle^2}{\langle y_k, s_k \rangle} - \frac{\langle H_k^0 y_k, y_k \rangle (\langle s_k, e_i \rangle / (H_k^0)^{(i)})^2}{\langle y_k, s_k \rangle \langle (H_k^0)^{-1} s_k, s_k \rangle} \right)^{-1} , \tag{21}$$

where $\{e_i\}_{1 \leq i \leq n}$ is an orthonormal basis for the given inner product on $\mathbb{R}^n$. It was obtained by scaling a diagonalized version of the BFGS formula and proved to be very effective.

	$m = 2$	$m = 5$	$m = 10$	$m = 20$	$m = 50$
VA14	> 1.92	> 1.92	> 1.92	> 1.92	> 1.92
M1GC3	2.07	1.60	1.46	1.37	1.23
M1QN2	1.49	1.14	1.07	0.97	0.92
M1QN3	1.00	0.85	0.85	0.76	0.74

Table 4: Comparison between the limited memory, CG and BFGS methods.

Table 4 gathers the numerical results from [17], which were obtained with 8 real-life test-problems having a dimension between 34 and 1559 (the average value of n is 692). A comparison is done between the standard BFGS method implemented by Buckley and LeNir (1983), the CG method of Hestenes and Stiefel (1952) implemented in the code VA14 from the Harwell library and three algorithms from the MODULOPT library (INRIA): M1GC3 is the code of Buckley and LeNir (1983), M1QN2 and M1QN3 use Nocedal's proposal with the scaling (20) and (21) respectively. Table t:lbfgs gives the number of function/gradient calls divided by the one required by the standard BFGS algorithm. The number m of updates ranges from 2 to 50. The sign $>$ used for VA14 means that the true number should be larger than the one given because the algorithm failed on one of the test-problems. Observe that the performance of the algorithms increases with m, except for VA14, which is not a variable storage method. Observe also that M1QN2 and M1QN3 are better than the BFGS method as soon as m is greater than 10...20 and 2...5, respectively. This improvement seems mainly due to the scaling made by the initial matrix in these algorithms, which acts as a dynamic preconditioner. An algorithm similar to M1QN2 has been tested by Liu and Nocedal (1988) who report similar results in their comparison with the method of Buckley and LeNir.

The convergence of M1QN2 on strongly convex functions has been proved in [35]. However, because of the counter-examples of Powell (1984), there is some doubt about the possibility of extending this result for nonconvex functions. To make this remark clear, observe that, with exact line-searches, the limited memory methods can be seen as a generalization of the CG method of Hestenes-Stiefel or Polak-Ribière. Indeed, let $m = 1$ and $H_k^0 = I$ in Nocedal's proposal and suppose that exact line-searches are done, i.e., $\langle s_{k-1}, g_k\rangle = 0$. Then, $\langle y_{k-1}, d_{k-1}\rangle = -\langle g_{k-1}, d_{k-1}\rangle = |g_{k-1}|^2$ and using formula (19), we obtain:

$$\begin{aligned} d_k &= -H_k\, g_k \\ &= -\bar{U}(I,\, y_{k-1},\, s_{k-1})\, g_k \\ &= -g_k + \frac{\langle y_{k-1}, g_k\rangle}{\langle y_{k-1}, d_{k-1}\rangle} d_{k-1} \\ &= -g_k + \beta_k^{\mathrm{PR}}\, d_{k-1}\,. \end{aligned}$$

As the PR method with exact line-searches does not converge on nonconvex functions, the limited memory method (at least for $m = 1$ and without scaling) does not converge either.

5.3 The partitioned quasi-Newton methods

The algorithms we have seen so far are suitable for any objective function f. The *partitioned quasi-Newton* (PQN) *method* proposed by Griewank and Toint (1982) is dedicated to the minimization of functions having a particular structure.

A function f is said *partially separable* if it can be written as a sum of p *elements functions* f_i:

$$f(x) = \sum_{i=1}^{p} f_i(x)\,, \quad \forall x \in \Omega$$

and if each of the element functions f_i is invariant on some subspace $E_i \subset I\!R^n$:

$$f_i(x + e) = f_i(x)\,, \quad \forall e \in E_i\,.$$

A typical case is when the subspaces E_i are spanned by some basic vectors of $I\!R^n$, which means that the f_i's only depend on few of the n variables. Note that the E_i's need not be disjoint, hence the name of *partial* separability.

In the PQN method, the Hessian of f, $\nabla^2 f(x) = \sum_{i=1}^{p} \nabla^2 f_i(x)$, is approximated at iteration k by a matrix of the form

$$B_k = \sum_{i=1}^{p} B_k^i\,.$$

The matrix B_k^i approximates the Hessian of the i-th element function and is updated to satisfy an element-wise secant condition:

$$B_{k+1}^i\, s_k = y_k^i := \nabla f_i(x_{k+1}) - \nabla f_i(x_k)\,.$$

This approach has some weak points:

- If the element functions are nonconvex, their Hessian is not positive definite and $\langle y_k^i, s_k \rangle$ is not necessarily positive; therefore the overall approximation is not necessarily positive definite either and may even be singular although the true Hessian is positive definite.
- The preceding point implies that different update formulae (BFGS and SR1) are used, depending on the element function, which is a lack of homogeneity and an additional complexity of the method.
- It requires more effort from the user who has to supply separately the gradient of the element functions.

But it has also important nice features:

- if the dimension of the subspaces E_i is large, the element Hessian are identified very quickly because the B^i's have low rank. This implies that the overall approximation is rapidly correct;
- the overall approximation is updated by a consistent high rank update;
- the notion of partial separability is basis independent, in contrast with the notion of sparsity.

Liu and Nocedal (1988) have made some numerical comparison between the PQN approach and M1QN2. They report excellent results for the PQN method when the dimension n_i of the subspaces E_i is small. If the n_i's are not very small, M1QN2 becomes competitive. The comparison was made on the computing time and not on the number of function/gradient calls, which generally favors the PQN code. This seems due to the iteration time of the PQN method that can be 2...5, sometimes 10...15, times larger than the one of M1QN2.

To conclude this section, let us mention that the PQN method has been implemented by Toint (1983) in the Harwell routine VE08.

6 Further references

We conclude by mentioning some further references that can be useful to deepen the topics addressed in this paper. The books by Gill, Murray and Wright (1981), McCormick (1983), Minoux (1983), also available in English, Dennis and Schnabel (1983) and Fletcher (1987) are classical textbooks, see also Lemaréchal (1989). The following references are collections of papers giving a state of the art: [9], [36], [54], [3], [31], [44], [50], and [12].

References

[1] M. Al-Baali (1985). Descent property and global convergence of the Fletcher-Reeves methods with inexact line search. *IMA Journal of Numerical Analysis*, 5, 121–124.

[2] M. Al-Baali, R. Fletcher (1986). An efficient line search for nonlinear least squares. *Journal of Optimization Theory and Applications*, 48, 359–377.

[3] A. Bachem, M. Grötschel, B. Korte (editors) (1983). *Mathematical Programming. The State of the Art.* Springer-Verlag, Berlin.

[4] W. Baur, V. Strassen (1983). The complexity of partial derivatives. *Theoretical Computer Science*, 22, 317–330.

[5] A. Buckley, A. LeNir (1983). QN-like variable storage conjugate gradients. *Mathematical Programming*, 27, 155–175.

[6] R.S. Dembo, T. Steihaug (1983). Truncated-Newton algorithms for large-scale unconstrained optimization. *Mathematical Programming*, 26, 190–212.

[7] J.E. Dennis, J.J. Moré (1977). Quasi-Newton methods, motivation and theory. *SIAM Review*, 19, 46–89.

[8] J.E. Dennis, R.B. Schnabel (1983). *Numerical Methods for Unconstrained optimization and nonlinear equations.* Prentice-Hall, Englewood Cliffs.

[9] L.C.W. Dixon, E. Spedicato, G.P. Szegö (editors) (1980). *Nonlinear Optimization. Theory and Algorithms.* Birkhäuser, Boston.

[10] L.C.W. Dixon, G.P. Szegö (editors) (1975). *Towards Global Optimization. Vol. 1.* North-Holland, Amsterdam.

[11] L.C.W. Dixon, G.P. Szegö (editors) (1978). *Towards Global Optimization. Vol. 2.* North-Holland, Amsterdam.

[12] S. Dolecki (editor) (1989). *Optimization.* Lecture Notes in Mathematics 1405. Springer-Verlag, Berlin.

[13] S.I. Feldman, D.M. Gay, M.W. Maimone, N.L. Schryer (1990). A Fortran-to-C converter. Computing Science Technical Report 149, AT&T Bell Laboratories, Murray Hill, NJ 07974.

[14] R. Fletcher (1987). *Practical Methods of Optimization* (second edition). John Wiley & Sons, Chichester.

[15] R. Fletcher, C.M. Reeves (1964). Function minimization by conjugate gradients. *The Computer Journal*, 7, 149–154.

[16] J.Ch. Gilbert, G. Le Vey, J. Masse (1991). La différentiation automatique de fonctions représentées par des programmes. Rapport de recherche, INRIA, BP 105, F-78153 Le Chesnay, France.

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

[18] J.Ch. Gilbert, J. Nocedal (1991). Global convergence properties of conjugate gradient methods for optimization. *SIAM Journal on Optimization.* (to appear).

[19] J.Ch. Gilbert, J. Nocedal (1991). (to appear).

[20] P.E. Gill, W. Murray (1974). Newton-type methods for unconstrained and linearly constrained optimization. *Mathematical Programming*, 7, 311–350.

[21] P.E. Gill, W. Murray, M.H. Wright (1981). *Practical Optimization.* Academic Press, New York.

[22] D. Goldfarb (1980). Curvilinear path steplength algorithms for minimization which use directions of negative curvature. *Mathematical Programming*, 18, 31–40.

[23] A. Griewank (1989). On automatic differentiation. In M. Iri, K. Tanabe (editors), *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers.

[24] A. Griewank (1991). The chain rule revisited in scientific computing. *SIAM News*, 24.

[25] A. Griewank (1991). Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Technical Report MCS-P228-0491, Argonne National Laboratory, Argonne, Il 60439.

[26] A. Griewank, D. Juedes, J. Srinivasan (1991). ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. Technical Report MCS-P180-1190, Argonne National Laboratory, Argonne, Il 60439.

[27] A. Griewank, Ph.L. Toint (1982). On the unconstrained optimization of partially separable functions. In M.J.D. Powell (editor), *Nonlinear Optimization 1981*. Academic Press, London.

[28] M.R. Hestenes, E. Stiefel (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49, 409–436.

[29] K.E. Hillstrom (1985). User guide for JAKEF. Technical Memorandum ANL/MCS TM-16, Argonne National Laboratory, Argonne, Il 60439.

[30] M. Iri, K. Kubota (1990). PADRE 2, version 1 – User's manual. Research Memorandum RMI 90-01, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo, Japan.

[31] A. Iserles, M.J.D.Powell (editors) (1987). *The State of the Art in Numerical Analysis*. Clarendon Press, Oxford.

[32] C. Lemaréchal (1981). A view of line-searches. In A. Auslender, W. Oettli, J. Stoer (editors), *Optimization and Optimal Control*, number 30 in Lecture Notes in Control and Information Science, pages 59–78. Springer-Verlag, Heidelberg.

[33] C. Lemaréchal (1989). Méthodes numériques d'optimisation. Collection didactique de l'INRIA, BP 105, 78153 Le Chesnay, France.

[34] S. Linnainmaa (1976). Taylor expansion of the accumulated rounding error. *BIT*, 16, 146–160.

[35] D.C. Liu, J. Nocedal (1988). On the limited memory BFGS method for large scale optimization. *Mathematical Programming* B, 45, 503–520.

[36] O.L. Mangasarian, R.R. Meyer, S.M. Robinson (editors) (1981). *Nonlinear Programming 4*. Academic Press, New York.

[37] G.P. McCormick (1977). A modification of Armijo's step-size rule for negative curvature. *Mathematical Programming*, 13, 111–115.

[38] G.P. McCormick (1983). *Nonlinear Programming. Theory, Algorithms and Applications*. J. Wiley & Sons, New York.

[39] M. Minoux (1983). *Programmation Mathématique. Théorie et Algorithmes*. Dunod, Paris.

[40] J.J. Moré (1983). Recent developments in algorithms and software for trust region methods. In A. Bachem, M. Grötschel, B. Korte (editors), *Mathematical Programming, the State of the Art*, pages 258–287. Springer-Verlag, Berlin.

[41] J.J. Moré, D.C. Sorensen (1979). On the use of directions of negative curvature in a modified Newton method. *Mathematical Programming*, 16, 1–20.

[42] J.J. Moré, D.J. Thuente (1990). On line search algorithms with guaranteed sufficient decrease. Preprint MCS-P153-0590, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439.

[43] J. Morgenstern (1985). How to compute fast a function and all its derivatives, a variation on the theorem of Baur-Strassen. *SIGACT News*, 16, 60–62.

[44] G.L. Nemhauser, A.H.G. Rinnooy Kan, M.J. Todd (editors) (1989). *Optimization*, Tome 1 of *Handbooks in Operations Research and Management Science*. North-Holland.

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

[46] J. Nocedal (1990). The performance of several algorithms for large scale unconstrained optimization. In T.F. Coleman, Y. Li (editors), *Large-Scale Numerical Optimization*, pages 138–151. SIAM.

[47] D.P. O'Leary (1982). A discrete Newton algorithm for minimizing a function of many variables. *Mathematical Programming*, 23, 20–33.

[48] S.S. Oren, E. Spedicato (1976). Optimal conditioning of self-scaling variable metric algorithms. *Mathematical Programming*, 10, 70–90.

[49] M. Overton, T. Schlick (1987). A powerful truncated Newton method for potential energy minimization. *Journal of Computational Chemistry*, 8, 1025–1039.

[50] J.-P. Penot (editor) (1989). *New Methods in Optimization and their Industrial Uses.* International Series of Numerical Mathematics 87. Birkhäuser Verlag, Basel.

[51] E. Polak, G. Ribière (1969). Note sur la convergence de méthodes de directions conjuguées. *Revue française d'Informatique et de Recherche Opérationnelle (RIRO)*, 16, 35–43.

[52] M.J.D. Powell (1976). Some global convergence properties of a variable metric algorithm for minimization without exact line searches. In R.W. Cottle, C.E. Lemke (editors), *Nonlinear Programming*, number 9 in SIAM-AMS Proceedings. American Mathematical Society, Providence, RI.

[53] M.J.D. Powell (1977). Restart procedures of the conjugate gradient method. *Mathematical Programming*, 12, 241–254.

[54] M.J.D. Powell (editor) (1982). *Nonlinear Optimization 1981.* Academic Press, London.

[55] M.J.D. Powell (1984). Nonconvex minimization calculations and the conjugate gradient method. In *Lecture Notes in Mathematics 1066*, pages 122–141. Springer-Verlag, Berlin.

[56] M.J.D. Powell (1985). Convergence properties of algorithms for nonlinear optimization. Numerical Analysis Report 1985/NA1, DAMTP, Silver Street, Cambridge CB3 9EW, England.

[57] D.F. Shanno (1978). Conjugate gradient methods with inexact searches. *Mathematics of Operations Research*, 3, 244–256.

[58] D.F. Shanno, K.H. Phua (1976). Algorithm 500, minimization of unconstrained multivariate functions. *ACM Transactions on Mathematical Software*, 2, 87–94.

[59] B. Speelpenning (1980). *Compiling fast partial derivatives of functions given by algorithms.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

[60] O. Talagrand (1991). The use of adjoint equations in numerical modelling of the atmospheric circulation. Presented at the "1991 SIAM Worshop on Automatic Differentiation of Algorithms: Theory, Implementation and Application", January 6-8, 1991, Breckenridge, Colorado.

[61] Ph.L. Toint (1983). VE08AD, a routine for partially separable optimization with bounded variables. Technical report, Harwell Subroutine Library, A.E.R.E. (UK).

[62] P. Wolfe (1969). Convergence conditions for ascent methods. *SIAM Review*, 11, 226–235.

[63] P. Wolfe (1971). Convergence conditions for ascent methods II: some corrections. *SIAM Review*, 13, 185–188.

[64] G. Zoutendijk (1970). Nonlinear programming, computational methods. In J. Abadie (editor), *Integer and Nonlinear Programming*, pages 37–86. North-Holland, Amsterdam.