Outils numériques pour l'optimisation

Jérôme Malick

École OCET - Roscoff Février 2009

Ces notes sont un support pour mon intervention à l'École Optimisation et Contrôle des Écoulements et Transferts en février 2009 à Roscoff. Ce cours est une introduction à l'optimisation différentiable sans contraintes. Le problème est de calculer le minimum d'une fonction de classe C^2 sur l'espace $\mathbb{R}^n$. L'objectif est de présenter les idées de bases des méthodes numériques pour résoudre ce problème. Nous insisterons sur:

- la structure d'un programme d'optimisation
- les statégies de calcul des itérés suivants
- les techniques efficaces en pratique (convergence rapide, gestion de la grande taille...)

Ces notes s'inspirent du livre:

> Numerical Optimization - Theorical and Practical Aspects
> F. Bonnans, J.Ch. Gilbert, C. Lemaréchal, C. Sagastizábal
> Universitext, Springer Verlag

On trouve en particulier dans le premier chapitre de ce livre les démonstrations des résultats énoncés ici ainsi que des références bibliographiques et historiques. Les chapitres suivants de ce livre traitent en détails les domaines que nous n'abordons pas: optimisation non-différentiable et optimisation différentiable avec contraintes

Contents

1 Introduction, motivations

1.1 General principles and overview

We focus in this course on *unconstrained, differentiable optimization*. This sounds rather restrictive but this still permits:
– to introduce general ideas of numerical optimization (valid in constrained or nonsmooth optimization).
– to present important algorithms used in real-life applications problems.
We aim at computing the minimum value of a function, and (if any) a point achieving this minimim; this is written

$$\min \quad f(x)\,, \qquad x \in \mathbb{R}^n, \tag{1}$$

with f is a smooth function (say, C^2 to fix ideas). Beginning at an initial point x_0, optimization algorithms generate a sequence of iterates $(x_k)_k$ that terminate when either we have convergence (see details below), or no more progress can be made. In deciding how to move from one iterate x_k to the next, the algorithms use information about the function f at x_k (and possibly from previous iterates too).

Knowledge on the function. The only information we know about f is the results of computations made in a *black box* (subprogram), independent of the selected algorithm. This subprogram can be called *simulator*, since it simulates the behaviour of the problem under the action of the decision variables (optimal or not). In the context of this note, we assume that the simulators give at any point $x \in \mathbb{R}^n$ the value $f(x) \in \mathbb{R}$ and the gradient $\nabla f(x) \in \mathbb{R}^n$, and in some cases the Hessian $\nabla^2 f(x)$ as well. Throughout this note, we will use the notation g for the gradient ∇f; and most of the time, g_k will stand for $g(x_k) = \nabla f(x_k)$.

Structure of an optimization algorithm. A computer program solving an optimization problem is made up of *two distinct parts*:

1. The first is in charge of managing x and is the algorithm proper; call it (A), as Algorithm; it is generally written by a mathematician, specialized in optimization.

2. The other is the simulator which performs the required calculations for each x decided by (A); it is generally written by a practitioner (physicist, economist, etc.), the one who wishes to solve the specific optimization problem. For example, the simulator in the forthcoming §1.3 integrates the state equation (2), computes the deviation from the observations (as well as the gradient, see below), and passes the result to the Algorithm in charge of solving the problem.

Another fundamental thing to understand here is the following: for any problem considered, the only information available for f (say usually $f(x_k)$ and $\nabla f(x_k)$) is the result of a numerical calculation, generally complicated (think of the Navier-Stokes equation of the example of §1.3). Hence, (A) has to proceed by "trial and error": it assigns trial values to x, which it corrects upon observation of the answer from the simulator; and this will essentially make up one iteration of the optimization process.

Computation of the next iterate. Optimization algorithms use the available information at the current iteration to find a new iterate x_{k+1} satisfying

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

This *descent* property ensures in particular *stability*, a privilege of optimization methods, as compared for example with equation solvers. There are two fundamental strategies for moving from the current point x_k to the new iterate x_{k+1}: *line-search and trust-region*. We give here the sketch of these two strategies, and they will be develop in the sequel.

In the line-search strategy, the algorithm chooses first a descent direction d_k (in using a model f f) and then searches along this direction from the current iterate x_k for a new iterate wih lower function value: we look for a "good" t such that

$$f(x_k + td_k) < f(x_k).$$

In the trust-region strategy, the information about f is used to construct a model function $\tilde{f}_k$ whose the behaviour near the current point x_k is similar to that of the actual objective function f. The model $\tilde{f}_k$ is usually defined to be a quadratic function of the form

$$\tilde{f}_k(x_k + h) = f(x_k) + \nabla f(x_k)^\top h + \frac{1}{2} h^\top M_k \, h$$

where M_k is a symmetric matrix that can be the Hessian $\nabla^2 f(x)$ if available, or some approximation of it. Because the model $\tilde{f}_k(x)$ may not be a good approximation of $f(x)$ when x is far from x_k, we restrict the search for a minimizer of $\tilde{f}_k$ to some "trust" region around x_k: in other words

$$\min \quad \tilde{f}_k(x_k + h), \qquad \text{where } x_k + h \text{ lie inside of the trust-region.}$$

In a sense, the line-search and trust-region approaches differ in the order in which they choose the *direction* and the *distance* to move to the next iterate. Line-search starts by fixing the direction d_k and then identifies an appropriate distance, namely the step-lenght t_k. In the trust-region, we first choose a maximum distance (the trust-region radius, Δ_k in this note) and then seek a direction and step that attain the best improvement possible subject to the distance constraint.

Generalities on convergence. For the generated $(x_k)_k$, two types of convergence are relevant.

– Global convergence. In optimization, an algorithm is said to converge *globally* when

$$\liminf_k \|\nabla f(x_k)\| = 0 \qquad \text{for any initial iterate } x_0$$

This property guarantees that the stopping test "$\|\nabla f(x_k)\| \leqslant \varepsilon$?" will occur for sure, for any $\varepsilon > 0$.

– Local convergence. Now assume x_k has a limit x^*; one wants to know at what speed $\delta_k := \|x_k - x^*\|$ tends to 0. The interesting property is the so-called *superlinear convergence*, which means $\frac{\delta_{k+1}}{\delta_k} \to 0$. In particular, we say that *quadratic convergence* holds when $\delta_{k+1} = O({\delta_k}^2)$; roughly, this means that the number of exact digits doubles at each iteration (for k large enough).

Short outline of this note. For presentation purposes, this note starts with the line-search methods (§2) and then explain how the trust-region method (§4) generalize them. Before all this, we give two applications illustrating the need for unconstrainted differentiable optimization, and we explain how to compute the gradient if not readily available. (So the next three subsections can be skipped in a first reading.)

1.2 Introductive example 1: Molecular Biology

An important problem in biochemistry, for example in pharmacology, is to determine the geometry of a molecule. Various physical techniques are possible (X-ray crystallography, nuclear magnetic resonance...). We detail one of these (using optimization) which is very convenient when

– the chemical formula of the molecule is known,

– the molecule is not available, making it impossible to conduct any experiment,

– one has some knowledge of its shape and one wants to *refine* it.

The idea is to compute the positions of the atoms in the space that minimize the associated potential energy. Let N be the number of atoms and call $x_i \in \mathbb{R}^3$ the spatial position of the i^{th} atom. To the vector $X = (x_1, \ldots, x_N) \in \mathbb{R}^{3N}$ is associated a potential energy $f(X)$ (the "conformational energy"), which is the sum of several terms. For example:

– Bond length: between two atoms i and j at distance $\|x_i - x_j\|$, there is first an energy of the type

$$L_{ij}(x_i, x_j) = \lambda_{ij}(\|x_i - x_j\| - d_{ij})^2 \, .$$

– There is also a Van der Waals energy, say

$$V_{ij}(x_i, x_j) = v_{ij}\Big(\frac{\delta_{ij}}{\|x_i - x_j\|}\Big)^6 - w_{ij}\Big(\frac{\delta_{ij}}{\|x_i - x_j\|}\Big)^{12} \, .$$

Here, the $\lambda_{ij}, v_{ij}, w_{ij}, d_{ij}, \delta_{ij}$'s are known constants, depending on the pair of atoms involved (carbon-carbon, carbon-nitrogen, etc.)

– Valence angle: between three atoms i, j, k forming an angle θ_{ijk} (we can write down the value of θ_{ijk} - but it's too heavy), there is an energy

$$A_{ijk}(x_i, x_j, x_k) = \alpha_{ijk}(\theta_{ijk} - \bar{\theta}_{ijk})^2 \,,$$

where, here again, α_{ijk} and $\bar{\theta}_{ijk}$ are known constants.

Other types of energies may also be considered: electrostatic, torsion angles, etc. The total energy is then the sum of all these terms, over all pairs/triples/quadruples of atoms. The important thing to understand here, is that this energy can be computed (as well as its derivatives) for any numerical values taken by the variables x_i. And this is true even if these values do not correspond to any reasonable configuration; simply, the resulting energy will then be unreasonably large (if the model is reasonable!); the optimization process, precisely, will aim at eliminating these values.

Note that the objective function is disagreeable: First with its many terms, it is long to compute. Second, with its strong nonlinearities, it does not enjoy some nice properties useful for optimization: it is definitely not quadratic, and not even convex. Actually, in most examples there are many equilibrium points X^* (local minima); this is why the only hope is to *refine* a specific one: by assumption, some estimate X_0 is available, close to the sought "optimal" X^*. Otherwise the optimization algorithm could only find some uncontrolled equilibrium, "by chance". Note also that nowaday's "interesting" molecules have 10^3 atoms and more, so we have to fight against numerical difficulties...

1.3 Introductive example 2: Meteorology

Consider the problem of forecasting the weather, i.e. of knowing the state of the atmosphere in the future. For this, let $p(z,t)$ be the state of the atmosphere at point $z \in \mathbb{R}^3$ and time $t \in [0,7]$ (with t expressed in days, assuming a forecast over one week); p is actually a vector made up of pressure, wind speed, humidity... The evolution of p along time can be modelled: avoiding technicalities, fluid mechanics tells us that

$$\frac{\partial p}{\partial t}(z,t) = \Phi(p(z,t)) \,, \tag{2}$$

where Φ is a certain differential operator. For example, (2) could be the Navier-Stokes equation, but approximations are generally introduced.

Once our model Φ is chosen, it "suffices" to integrate (2). For this, initial conditions are needed (the question of boundary conditions is neglected here; for example, we shall say that they are periodicity conditions, (2) being integrated on the whole earth). Here comes optimization, in charge of estimating $p(\cdot, 0)$ via an *identification* process, which we explain roughly.

In fact, the available information also contains all the meteorological observations collected in the past, say during the preceding day. Let us denote by $\omega = \{\omega_i\}_{i \in I}$ these observations. To fix ideas, we could say that each ω_i represents the value of p at a certain point (z_i, t_i). To take this data into account, a natural idea is to consider the problem

$$\min_p \|p - \omega\| \,, \tag{3}$$

(2) being considered as a constraint (called in this context the *state equation*; in fact we have here an *optimal control* problem, in which the objective function depends on a *state* (here p) evolving along time).

At this point, it is a good idea not to view (2), (3) as a nonlinearly constrained optimization problem, but rather as an unconstrained one. In fact, call $x(z) = p(z, -1)$ the state of the atmosphere at z, at initial time $t = -1$. A fundamental remark is then: assuming x to be known, (2) gives $p = p_x$ unambiguously, and hence the objective value in (3) as well: the unknown p_x depends on the variable x *only*. Our problem can therefore be formulated as $\min_x \|p_x - \omega\|$, which means:

– to minimize with respect to x (unconstrained variable)
– the function defined by (3),
– where $p = p_x$ is obtained from (2)
– via the initial condition $p(\cdot, -1) = x$.

NB. Just as p is called the state, x – i.e. $p(\cdot, -1)$ here – is the *control* variable: there is a well-defined mapping

$$\text{control} \quad \mapsto \quad \text{state} \quad \mapsto \quad \text{objective function},$$

and this is characteristic of optimal control problems.

1.4 Gradient computation in optimal control, Generalisation

For an optimization algorithm to work correctly, first-order derivatives are needed. Computing them is not always easy, in particular in optimal control. Consider for example the meteorological problem of §1.3: we want to differentiate the function $x \mapsto \|p_x - \omega\|$. For this we must differentiate $x \mapsto p_x$, given implicitly through the state equation. Computing derivatives is indeed possible via the use of the adjoint state; below is an illustration.

Take the following problem: the control variables are $\{u_t\}_{t=1}^T$ where $u_t \in \mathbb{R}^n$ for each t; the state variables are likewise $\{y_t\}_t$ with $y_t \in \mathbb{R}^m$, given by the state equation

$$\begin{cases} y_t = F_t(y_{t-1}, u_t)\,, \quad \text{for } t = 1, \dots, T, \\ y_0 \text{ given.} \end{cases} \tag{4}$$

In §1.3, the notation was x, p rather than u, y; and the state equation was a differential equation (or even partial differential), instead of a discrete process. Here, for each t, F_t is a function (possibly nonlinear) from $\mathbb{R}^m \times \mathbb{R}^n$ to $\mathbb{R}^m$. Besides, a function is given, say

$$f = \sum_{t=1}^{T} f_t(y_t, u_t)$$

where, for each t, f_t sends $\mathbb{R}^m \times \mathbb{R}^n$ to $\mathbb{R}$. It is purposedly that we do not specify formally which variables f depends on.

Call $v = \mathrm{d}u \in \mathbb{R}^{nT}$ a differential of u; it induces from (4) a differential $z = \mathrm{d}y \in \mathbb{R}^{mT}$, and finally a differential $\mathrm{d}f$. We use the notation $\langle \cdot, \cdot \rangle_n$ [resp. $\langle \cdot, \cdot \rangle_m$] for the dot-product in $\mathbb{R}^n$ [resp. $\mathbb{R}^m$]. In the control space, the scalar product is therefore $\langle g, v \rangle = \sum_{j=1}^T \langle g_t, v_t \rangle_n$. Our problem is then as follows: find $\{g_t\}_{t=1}^T$ such that the differential of f is given by $\mathrm{d}f = \langle g, v \rangle$. This will yield $\{g_t\}_t \in \mathbb{R}^{nT}$ as the gradient of f, considered as a function of the control variable u alone.

To solve this problem, we have from (4) (assuming appropriate smoothness of the data)

$$\begin{cases} z_t = J_y F_t(y_{t-1}, u_t) z_{t-1} + J_u F_t(y_{t-1}, u_t) v_t \quad \text{for } t = 1, \dots, T, \\ z_0 = 0 \end{cases} \tag{5}$$

($z_0 = 0$ because y_0 is fixed!) In this writing, the Jacobian $J_y F_t(y_{t-1}, u_t)$ is an $m \times m$ matrix and $J_u F_t(y_{t-1}, u_t)$ is $m \times n$. We have also

$$\mathrm{d}f = \sum_{t=1}^{T} \langle \nabla_y f_t(y_t, u_t), z_t \rangle_m + \sum_{t=1}^{T} \langle \nabla_u f_t(y_t, u_t), v_t \rangle_n \,; \tag{6}$$

here $\nabla_y f_t(y_t, u_t) \in \mathbb{R}^m$ and $\nabla_u f_t(y_t, u_t) \in \mathbb{R}^n$. We need to eliminate z between all these relations; this is done by a series of tricks:

Trick 1 Multiply the t^{th} linearized state equation in (5) by a vector $p_t \in \mathbb{R}^m$ (unspecified for the moment) and sum up. Setting $G_t := J_y F_t(y_{t-1}, u_t)$ and $H_t := J_u F_t(y_{t-1}, u_t)$, we obtain

$$\sum_{t=1}^{T} (p_t, z_t)_m = \sum_{t=1}^{T} \langle p_t, G_t z_{t-1} \rangle_m + \sum_{t=1}^{T} \langle p_t, H_t v_t \rangle_m \,.$$

Single out $\langle p_T, z_T \rangle_m$ in the lefthand side, transpose G_t and H_t, and re-index the sum in z; remembering that $z_0 = 0$, this gives

$$0 = -\langle p_T, z_T \rangle_m - \sum_{t=1}^{T-1} \langle p_t, z_t \rangle_m + \sum_{t=1}^{T-1} \langle G_{t+1}^\top p_{t+1}, z_t \rangle_m + \sum_{t=1}^{T} \langle H_t^\top p_t, v_t \rangle_n \,.$$

Trick 2 Add to the expression (6) of $\mathrm{d}f$ and identify with respect to the z_t's. Setting $\gamma_t := \nabla_y f_t(y_t, u_t)$ and $h_t := \nabla_u f_t(y_t, u_t)$:

$$\mathrm{d}f = \langle -p_T + \gamma_T, z_T \rangle_m + \sum_{t=1}^{T-1} \langle -p_t + G_{t+1}^\top p_{t+1} + \gamma_t, z_t \rangle_m + \sum_{t=1}^{T} \langle H_t^\top p_t + h_t, v_t \rangle_n \,.$$

Trick 3 Now it suffices to choose p so as to cancel out the coefficient of each z_t: requiring

$$\begin{cases} p_T = \gamma_T \,, \\ p_t = G_{t+1}^\top p_{t+1} + \gamma_t \quad \text{for } t = T-1, \ldots, 1 \,, \end{cases} \tag{7}$$

we obtain the gradient in the desired form:

$$g_t = H_t^\top p_t + h_t \quad \text{for } t = 1, \ldots, T.$$

The (backward) recurrence relations (7) form the so-called *adjoint equation*, whose solution p is the *adjoint state*.

The adjoint state generalized: automatic differentiation. The adjoint technique opens the way to the so-called *automatic* or *computational differentiation*. Indeed, consider any computer code which, taking an entry u, computes an output f. Such a code can be viewed as a "control process" of the type (4):

- The t^{th} line of this code is the t^{th} equation in (4).
- The intermediate results of this code (the lefthand sides of the assignment statements) form altogether a "state" y, which is a function of the "control" u.
- Forming the righthand side of the adjoint equations then amounts to differentiating one by one each line of the code.
- Afterwards, solving the adjoint equations – to obtain eventually the gradient ∇f – amounts to writing these "linearized lines" bottom up.

All these operations are purely mechanical and lend themselves to automatization. Thus, one can conceive the existence of software which

- take as entry a computer code able to calculate $f(u)$ (for given u),
- and produce as output another computer code able to calculate $\nabla f(u)$ (again for given u).

It is worth mentioning that such software do not need to know anything about the problem. They do not even need mathematical formulae representing the computation of f. What they need is just the first half of a simulator; and then they write down its second half.

2 Line-Searches

As explained in §1.1, one iteration k of a descent optimization algorithm is made up of two phases:

1. Computing a direction: f is replaced by a model f_k, which is simpler; then f_k is minimized to yield a new approximation, call it $x_k + d$. In brief, the direction is computed by minimizing (usually accurately) an *approximation* of f; this is the topic of the next section.
2. Line-search: a stepsize $t > 0$ is computed so that $f(x_k + td) < f(x_k)$. The stepsize t is computed by observing the *true* f on the restriction of $x \in \mathbb{R}^n$ to the half-line $\{x_k + td\}_{t \in \mathbb{R}_+}$ (x_k and d fixed). This is studied in this section.

To study the second step, we are given: the starting point x of the line-search (x is the current iterate x_k); the direction of search d; a merit-function $t \mapsto q(t)$, defined for $t \geqslant 0$, representing $f(x + td)$. In the following, we always suppose $q'(0) = \nabla f(x)^\top d < 0$.

2.1 General scheme and Wolfe's rule

The whole business of a line-search algorithm is to define a test with three possible outputs which, given $t > 0$, answers whether

a) t is satisfactory

b) t is too large

c) t is too small.

This test is performed upon observation of $q(t)$, and possibly of $q'(t)$. We will call t_L a too small t (on the left of a desired t), t_R a too large t (on the right of a desired t). To initialize the search, 0 is obviously a t_L, while t_R can be initialized to $+\infty$, whatever it means. Schematically, the algorithm is then the following:

Schematic line-search algorithm

Step 0. Start from an initial $t > 0$. Initialize $t_L = 0$ and $t_R = +\infty$.

Step 1. Test t;

if a) terminate;

if b) declare $t_R = t$ and go to 2;

if c) declare $t_L = t$ and go to 2.

Step 2

If no real t_R has been found ($t_R = +\infty$), compute a new $t > t_L$.

Else compute a new $t \in]t_L, t_R[$.

Loop to 1.

Thus, the line-search algorithm is a sequence of interpolations, reducing the *bracket* $[t_L, t_R]$, possibly preceded by a sequence of extrapolations (as long as $t_R = +\infty$). The observations below are straightforward, but fundamental for a good understanding of the mechanism:

– Extrapolations are performed until a finite t_R is found (which may happen at the first try).

– Then the interpolation phase starts.

– In any case, t_L increases each time it is modified,

– and t_R decreases each time it is modified;

– but there always holds $t_L < t_R$.

A line-search must terminate as soon as possible, since it is a subalgorithm within the optimization process. In particular, finding an optimal $t > 0$ would be absurd: what we want is to minimize f, not q. From this point of view, the method admitted as the most efficient is the following, commonly called the *Wolfe* line-search. Two coefficients $0 < m_1 < m_2 < 1$ are chosen, and cases a) b) c) are the following:

a) $q(t) \leqslant q(0) + m_1 t q'(0)$	and	$q'(t) \geqslant m_2 q'(0)$	(then terminate);
b) $q(t) > q(0) + m_1 t q'(0)$			(then $t_R = t$);
c) $q(t) \leqslant q(0) + m_1 t q'(0)$	and	$q'(t) < m_2 q'(0)$	(then $t_L = t$).

Note: in (rare) occasions, computing the gradient in the simulator is much more expensive than the function (in terms of CPU). For such problems, a drawback of Wolfe's line-search is that q' – hence ∇f – is needed at each cycle. An alternative line-search exists, called Goldstein & Price, in which $q'(t)$ (the slope at t) is replaced by $(q(t) - q(0))/t$ (the average slope between 0 and t. It is written

a) $q(t) \leqslant q(0) + m_1 t q'(0)$	and	$q(t) \geqslant q(0) + m_2 q'(0)$	(then terminate);
b) $q(t) > q(0) + m_1 t q'(0)$			(then $t_R = t$);
c) $q(t) \leqslant q(0) + m_1 t q'(0)$	and	$q(t) < q(0) + m_2 q'(0)$	(then $t_L = t$).

2.2 Elements of convergence

First one must make sure that the above (sub)algorithm terminates. For this, the new trial stepsize in the scheme of §2.1 – call it t_+ – must satisfy some consistency property: for example, t_+ should not be unduly close to t_R, otherwise the bracket will not be properly reduced. Thus, consistency of t_+ means two things:

1. Extrapolations should "significantly" increase t; say $t_+ \geqslant 2t_L$.
2. Interpolations should "significantly" decrease $[t_L, t_R]$; say $t_+ \in [t_L + \delta, t_R - \delta]$ with $\delta = 0.1(t_R - t_L)$.

Theorem [Consistency of Wolfe line-search] Suppose $q \in C^1$, $q'(0) < 0$, and t_+ is consistently chosen. Then Wolfe's line-search

– either produces $t_L \to +\infty$ with $q(t_L) \to -\infty$,

– or produces a) after finitely many cycles.

The next question is whether the resulting sequence (x_k) converges to a minimum point. Of course, convergence cannot hold independently of the choice of the direction (the line-search will be helpless if d_k is "too orthogonal" to the gradient). The angle between the direction and the gradient thus appears as essential, and we set

$$c_k := \frac{-g_k^\top d_k}{\|g_k\| . \|d_k\|} .$$

Theorem [Convergence Wolfe line-search] With the above notation, let an algorithm generate directions satisfying $\sum_{k=1}^{+\infty} c_k^2 = +\infty$ and use Wolfe's line-search. Assume that g is Lipschitz continuous on the slice $\{x : f(x) \leqslant f(x_1)\}$. Then: either the objective-function tends to $-\infty$, or $\liminf \|g_k\| = 0$.

Note that this theorem guarantees in particular the good behaviour of an optimization method that combines gradient descent and Wolfe line-search. Nevertheless, we can't be completely content with just a convergent algorithm. We have to demand some efficiency. Here "efficiency" means minimizing the number of calls to the simulator (since in pratice we have no control in what happens inside the simulator).

In practice A last remark: an optimization program fails rather often at the begining and experience indicates that, in 90% of cases, the mistake is not in the Algorithm proper but in the simulator, more precisely in the gradient computation. Such a mistake can be detected rather reliably, upon observation of Wolfe's line-search. If the gradient is bugged, the line-seach eventually produces the following paradoxical behaviour: a sequence $\{t = t_R\}$ is produced, tending to 0 with $q(t)$ staying stubbornly larger than $q(0)$, while $q'(t)$ stays stubbornly negative.

3 (Quasi-)Newton methods

We now focus on the first phase of a descent (or line-search) method: computing the descent direction. The steepest descent $d_k = -\nabla f(x_k)$ is not a good option: being short-sighted, it leads to some zig-zaging behaviour. Efficient algorithms require to use more information - in particular some information about the second-order of the function. Newton methods are the panacea in theory; Quasi-Newton method in practice.

3.1 Newton methods

To solve the optimality condition $g(x) = 0$, the Newton principle is as follows. Starting from the current iterate x_k, replace g by its linear approximation:

$$g(x_k + d) = g(x_k) + Jg(x_k)d + o(\|d\|)$$

where $Jg(x_k)$ is the Jacobian of g at x_k. We then neglect the term $o(\|d\|)$; this gives the linearized problem $g(x_k) + Jg(x_k)d = 0$. Its solution is $d^N = -[Jg(x_k)]^{-1}g(x_k)$, producing the Newton iterate $x^N = x_k + d^N$.

In the case of an optimization problem, g is the gradient of f, $Jg = \nabla^2 f$ is its Hessian. Just as g was approximated to first order, f can be approximated to second order:

$$f(x_k + d) = f(x_k) + g(x_k)^\top d + \frac{1}{2} d^\top \nabla^2 f(x_k) d + o(\|d\|^2)\,.$$

The quadratic approximation thus obtained is minimized (if $\nabla^2 f(x_k)$ is positive definite) when its gradient vanishes: $g(x_k) + \nabla^2 f(x_k)d = 0$. We realize an evidence: Newton's method on $\min f(x)$ is Newton's method on $g(x) = 0$.

The big advantage of this method is well known: it converges very fast.

Theorem [Local convergence of Newton] Let f be C^2 near a solution x^*, where $\nabla^2 f(x^*)$ is positive definite. Then the convergence of Newton's method is superlinear. If, in addition, $f \in C^3$, this convergence is quadratic.

This theorem implies by no means global convergence: it says that, if x is close to the solution, then x^N is infinitely closer. Besides, drawbacks of Newton's method are also well-known:

- The Hessian is necessary ! (what is it in the meteo problem ?)
- in addition, it requires to compute the Hessian, and then solve a linear system; this is heavy;
- in general, it diverges violently;
- in our situation where g is the gradient of a function to be *minimized*, another drawback is that the sequence (x_k) will probably rush to the closest stationary point, possibly a local maximum.

3.2 Quasi-Newton methods

The following ideas will be used to suppress the drawbacks above.

- In optimization, stability can be enforced by the requirement $f(x_{k+1}) < f(x_k)$; and this is the duty of the line-search. Then it suffices to consider the Newton increment d^N as a *direction*, along which a line-search will be performed to decrease the function $q(t) = f(x_k + td^N)$. This will take care of the first and third drawbacks. However, this line-search will be possible only if $q'(0) = g_k^\top d^N < 0$ (see §2). Using the definition of d^N, this means that $g_k^\top \nabla^2 f(x_k)^{-1} g_k$ must be positive, i.e. in practice $\nabla^2 f(x_k)$ positive definite; if it is not, something more must be done.
- Consider now the second drawback of Newton's method. Rather than computing explicitly $\nabla^2 f$ at each iteration, and solving the corresponding linear system, we can approximate directly $\nabla^{-2} f$ by a matrix W, which we deliberately choose symmetric positive definite. Then, alongside with the descent process on f, an identification process of the Hessian (or rather its inverse) is performed.

This is the *quasi-Newton* idea; it results in a general algorithm of the following form. Hereafter we drop the index k; and a superscript "+" will mean $k+1$.

Schematic quasi-Newton algorithm

Step 0. An initial iterate x and stopping tolerance ε are given; an initial matrix W, positive definite, is also chosen. Compute the initial gradient $g = g(x)$.

Step 1. If $\|g\| \leqslant \varepsilon$ stop.

Step 2. Compute $d = -Wg$.

Step 3. Make a line-search initialized on $t = 1$, to obtain the next iterate $x_+ = x + td$ and its gradient $g_+ = g(x_+)$.

Step 4. Compute the new matrix W_+ for the next iteration and loop to 1.

The ingredients characterizing this method are therefore: the line-search (which will be of course that of Wolfe), the initial matrix W (which can be the identity), the computation of W_+ in Step 4. Let us explain how this last matrix is computed. In the sequel, we will use the notation

$$s = s_k = x_{k+1} - x_k \quad \text{and} \quad y = y_k = g_{k+1} - g_k$$

(observe that s and y are known when W_+ must be computed). Knowing that we want to approximate a symmetric matrix (an inverse Hessian) and to obtain descent directions, W_+ is of course required to be *symmetric positive definite*. Besides, to give W_+ a chance to approximate what we want, W_+ is required to satisfy $W_+ y = s$, called the *quasi-Newton*, or *secant* equation. Its explanation is as follows: the mean-value G of $\nabla^2 f$ between x and x_+ satisfies $y = Gs$. The quasi-Newton equation has therefore the effect of forcing $W_+ \simeq G^{-1}$, in the sense that these two matrices have the same action on y, a subspace of dimension 1. Of course, the two above requirements leave infinitely many possible matrices.

A quasi-Newton method is the realization of the schematic algorithm 3.2, where the matrices W_k are computed sequentially: $W_+ = W + B$, the corrections B being chosen so that

(i) W_k is symmetric positive definite for all k,

(ii) the quasi-Newton equation $W_+ y = s$ is satisfied for all k.

Among all the possible corrections, stability reasons lead us to the additional requirement:

(iii) each B is minimal in some sense.

This still leaves a large range of possibilities, depending on the sense chosen in (iii). At present, a consensus is obtained for a method found independently, and using different arguments, by C. Broyden, R. Fletcher, D. Goldfarb, D. Shanno:

$$W_+ = W_+^{BFGS} = W - \frac{sy^\top W + Wys^\top}{y^\top s} + \Big[1 + \frac{y^\top W y}{y^\top s}\Big] \frac{ss^\top}{y^\top s}\,.$$

Check that it is symmetric and satisfies the quasi-Newton equation. Also, observe from the quasi-Newton equation $s = W_+ y$ that positive definiteness requires $y^\top s > 0$. This last condition is actually sufficient:

Theorem Suppose W is positive definite. Then $y^\top s > 0$ is a necessary and sufficient condition for W_+^{BFGS} to be positive definite. An interesting point in this result is that the property $y^\top s > 0$ means $g_+^\top s > g^\top s$, and this is just the same as the second part of Wolfe's rule: $q'(t) \geqslant m_2 q'(0)$.

Remark If $n = 1$, the quasi-Newton equation defines a unique W_+ by $W_+ = s/y$. Starting from two initial iterates x_1 and x_2, the algorithm becomes

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{g_k - g_{k-1}} g_k$$

known as the secant method, or "regula falsi": the tangent to the graph of g (which is used in Newton's method) is replaced by the secant between x and x_-.

The opposite case (big n) also deserves comment. A big drawback of Newtonian methods (including the present quasi-Newton variant) is the necessity of storing an $n \times n$ matrix (what if $n = 10^6$, as in the meteo problem of §1.3?) Yet, as long as k is small, computing the direction d_k of a quasi-Newton method needs only $2k$ vectors s_i and $y_i, i = 1, \ldots, k-1$. In the case of a really large-scale problem, at least a few iterations can be performed, just by developing the product $W_k g_k$ (as a function of these k vector pairs), instead of computing explicitly the whole matrix W_k. It is indeed possible to take advantage of this remark, which gives birth to *limited memory* quasi-Newton methods.

3.3 Elements of convergence

Rather unexpectedly, it seems impossible to prove global convergence without convexity.

Theorem [Global convergence of BFGS] Suppose f is convex with a Lipschitzian gradient on the domain $\{x : f(x) \leqslant f(x_1)\}$. Then the BFGS algorithm with Wolfe's line-search and W_1 positive definite satisfies:
– either the objective function tends to $-\infty$,
– or $\liminf \|g(x_k)\| = 0$.

Remember that Newtonian methods are designed to converge fast; it is therefore important to study their speed of convergence. Recall that speed of convergence usually makes additional assumptions. Along these lines, the folllowing criterion for superlinear convergence is the fundamental tool.

Lemma [Dennis & Moré] Consider a mapping g from $\mathbb{R}^n$ to $\mathbb{R}^n$ and let a sequence (x_k) be generated by the formula $x_{k+1} = x_k - W_k g_k$. Assume that $x_k \to x^*$ such that $g(x^*) = 0$, g' is continuous in a neighborhood of x^*, and $g'(x^*)$ is nonsingular. Then

$$\frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|} \to 0 \quad \Longleftrightarrow \quad \frac{\|g_{k+1}\|}{\|x_{k+1} - x_k\|} \to 0 \quad \Longleftrightarrow \quad [W_k^{-1} - g'(x^*)] \frac{x_{k+1} - x_k}{\|x_{k+1} - x_k\|} \to 0\,.$$

This gives a criterion for quasi-Newton methods without line-search (we have assumed $t_k = 1$). This explains that superlinear convergence of a quasi-Newton algorithm must be established in two steps:

1. to show that the matrices W_k^{-1} satisfy the criterion of Dennis and Moré,
2. to show that the stepsize $t_k = 1$ eventually satisfies Wolfe's rule.

Proposition Assume that the BFGS algorithm with Wolfe's line-search converges to x^*, where $\nabla^2 f$ is positive definite and Lipschitz continuous. Then the criterion of Dennis and Moré is satisfied.

Proposition Suppose f has a minimum point x^* with a Hessian $\nabla^2 f(x^*)$ positive definite. Then there exist $\delta > 0$ and $\varepsilon > 0$ such that: if $\|x - x^*\| \leqslant \delta$, then every x_+ satisfying $\|x_+ - x^*\| \leqslant \varepsilon \|x - x^*\|$ is accepted by Wolfe's rule, providing that $0 < m_1 < \frac{1}{2}$ and $m_2 > 0$.

In other words, a superlinear algorithm eventually needs no line-search (namely when x_+ is superlinearly better than x). Piecing together, we obtain the final result:

Theorem [Superlinear convergence of BFGS + Wolfe] Let BFGS+Wolfe generate a sequence (x_k) converging to x^* where $\nabla^2 f$ is positive definite and Lipschitz continuous. Assume $0 < m_1 < \frac{1}{2}$, $m_2 > 0$, and the stepsize $t = 1$ is tried first, at each iteration. Then the convergence is superlinear: $\|x_+ - x^*\| = o(\|x - x^*\|)$.

3.4 Large-scale problems: limited-memory quasi-Newton

In the above quasi-Newton method, for a large number of variables n (exceeding 10^5, say) storing and managing the corresponding matrix W exceeds the abilities of computers. But note that a quasi-Newton matrix W_k is completely defined by (the initial matrix W_1 and) the $2(k-1)$ vectors s_1, y_1, ..., s_{k-1}, y_{k-1}. A mean therefore exists to compute the direction $d_k = -W_k g_k$ via an explicit use of only these $2(k-1)$ vectors, which makes $2(k-1)n$ numbers, instead of $n(n+1)/2$; the next formulae do the trick.

Algorithm [qN formulae]
Given: an initial matrix W^0, a vector g and m vector pairs $\{s_i, y_i\}$ for $i = 1, \ldots, m$.
Problem: compute $d = -Wg$, where W is the result of m BFGS updates of W^0.
Answer: $d = -h^m$, obtained as follows.
– Set $q^m = g$; for $i = m, \ldots, 1$ do

$$\alpha^i = \frac{(q^i)^\top s_i}{y_i{}^{top} s_i} \quad \text{and} \quad q^{i-1} = q^i - \alpha^i y_i \,.$$

– Set $h^0 = W^0 q^0$; for $i = 1, \ldots, m$ do

$$\beta^i = \frac{y_i{}^\top h^{i-1}}{y_i{}^\top s_i} \quad \text{and} \quad h^i = h^{i-1} + (\alpha^i - \beta^i) s_i \,.$$

We use superscripts (q^i, α^i, etc.) to suggest that the above iterations need not be related to the iterations minimizing f. For example, if Algorithm [qN formula] is used to compute the k^{th} direction d_k of such a minimization algorithm, then we will set $g = g_k$ and $d_k = -h^m$. Said otherwise, the m iterations of the algorithm will be systematically executed at each iteration k of the minimization algorithm.

Equipped with these formulae, take a computer allowing the storage of $2m$ vectors of dimension n; if $n = 10^5$ for example, a standard personal computer will easily accept $m = 10$. On this computer, start a quasi-Newton algorithm without explicit storage of the matrices W_k, but with a direct computation of $W_k g_k$ by qN formulae algorithm. The iterations $k = 1, \ldots, m$ can be performed normally. Afterwards, the "standard" matrix W_k can no longer be used: it needs $k > m$ pairs $\{s_i, y_i\}$; among these k pairs, m must be chosen. To respect the essence of quasi-Newton, which constructs a local model of f in a neighborhood of x_k, one chooses of course the last m computed pairs: normally, they were computed at corresponding x's close to x_k. This results in an algorithm with the first m iterations are identical to those of Chap. 3, let us describe the current iteration after the m^{th}:

Algorithm [limited memory qN]
Step 1 (initializing iteration k). We have the m pairs of vectors

$$\begin{aligned} s_1 &= x_{k-m+1} - x_{k-m}, \quad \ldots, \quad s_m = x_k - x_{k-1} \,, \\ y_1 &= g_{k-m+1} - g_{k-m}, \quad \ldots, \quad y_m = g_k - g_{k-1} \,. \end{aligned}$$

Choose a "simple" matrix W^0, for example the identity.
Step 2 (poor-man qN formulae). Compute the direction $d_k = -W_k g_k$ by an application of Algorithm to the matrix W^0, starting from $g = g_k$.
Step 3. Obtain the next iterate x_{k+1} and its gradient g_{k+1} by a Wolfe line-search along d_k, or alternatively by a curvilinear search of the type trust-region.
Step 4 (refreshing the information). For $i = 1, \ldots, m-1$, replace each pair $\{s_i, y_i\}$ by $\{s_{i+1}, y_{i+1}\}$. Store $s_m = x_{k+1} - x_k$ and $y_m = g_{k+1} - g_k$ and loop to Step 1.

In standard quasi-Newton methods, the role of the initial matrix W_1 is not too crucial: in view of the successive updates, this role fades away along the iterations. Nevertheless, it is experimentally observed that a bad initial W_1 entails bad W_k's for many iterations. In a limited memory method such as the limited memory qN algorithm, we have an initial matrix $W^0 = W_k^0$ at each iteration k. According to the above experimental observation, W^0 must be chosen with great care; to content oneself with $W^0 = I$ at each iteration results in a bad algorithm. We do not elaborate here on appropriate initializations of W^0 in qN formulae.

The limited memory quasi-Newton is excellent; it is a good choice, often the only possible one for large-scale problems. However, it raises a rather frustrating question. One could think that its convergence is faster when the number m of stored pairs is larger; yet, one empirically observes that its speed of convergence stalls beyond a modest value of m; for larger m, the total computing time worsens, since the cost of one iteration is proportional to m. Rather frequently, one sees the limited memory with $m = 20$ terminating faster than the standard BFGS algorithm. Even worse: the "critical" value of m, beyond which the above stalling phenomenon occurs, seems relatively stable, say $m \simeq 20$, independently of n. No satisfactory explanation has been suggested so far.

4 Trust-Regions

We study trust-region methods that can be seen as variant of line-searches using other strategy for the correction step. These methods are particularly well-suited in a Newtonian context; they also appeared in the context of nonlinearly constrained optimization.

4.1 The elementary problem

Starting from the current iterate $x = x_k$, one wishes to go to a supposedly excellent estimate x^N, obtained by the minimization of a model representing f in a neighborhood of x. To fix ideas, we consider $\tilde{f}$ this model

$$\tilde{f}(x+h) := f(x) + g^\top h + \frac{1}{2} h^\top M h \,; \tag{8}$$

here $M = M_k$ can be the Hessian $\nabla^2 f(x_k)$, (or a quasi-Newton approximation, or the Gauss-Newton approximation as in the forthcoming Section 4.4. We set $x^N = x + h^N$, where h^N minimizes $\tilde{f}(x + \cdot)$.

The desired iterate x^N may not be convenient, and must perhaps be corrected; preceedingly this was the task of the line-search. Most often, x^N is not convenient because $x^N - x = h^N$ is too large; in a Newtonian method, for example, this could occur when

– the model is valid only for small h, and we have $f(x^N) \geqslant f(x)$,

– and/or the Hessian $M = \nabla^2 f(x)$ is not positive definite, which destroys any meaning of h^N.

The reason x^N is not convenient is that $\tilde{f}$ itself is not convenient. Based on this idea, one can proceed as follows:

– Enhance the model $\tilde{f}$ in a certain way (see below), depending on a parameter Δ.

– Solve (8) and obtain a solution h_Δ.

– Adjust Δ so that this solution h_Δ is "convenient".

– This adjustment is made according to the principles developed in Chap. 2.

On way to enhance the model the next iterate is forced into the ball centered at x and of radius Δ (the trust-region). In other words, h_Δ solves

$$\min_h \quad \tilde{f}(x+h)\,, \quad \|h\| \leqslant \Delta\,. \tag{9}$$

The key is that (9) can be efficiently solved, even when M is not positive definite. The technique is to solve actually the "dual problem", let us give an idea of how this is done. For $\mu \geqslant 0$ such that $M + \mu I$ is positive semi-definite (i.e. $\mu \geqslant \max\{0, -\lambda\}$, if λ is the smallest eigenvalue of M), consider the Lagrangian

$$\ell(h, \mu) = \tilde{f}(x+h) + \frac{\mu}{2}(\|h\|^2 - \Delta^2)\,.$$

For fixed μ large enough, ℓ has a minimum point h, given by

$$h = h(\mu) := -(M + \mu I)^{-1} g\,. \tag{10}$$

It happens that the solutions to (9) are those $h(\mu^*)$ of (10) satisfying $\|h(\mu^*)\| \leqslant \Delta$, where $\mu^* \geqslant \max\{0, -\lambda\}$ is such that $\mu^*(\|h(\mu^*)\| - \Delta) = 0$. Therefore the actual computation of h_Δ essentially reduces to a search on $\mu \geqslant 0$. This search can be viewed either as the resolution of the equation $\|h(\mu)\| = \Delta$, or as the maximization of the concave function $\ell(h(\mu), \mu)$. In both cases, an adequate Newton in one dimension does the trick, via a differentiation of (10). Normally, one solves rather $1/\|h(\mu)\| - 1/\Delta = 0$, which seems a better conditioned problem.

4.2 The elementary mechanism: curvilinear search

Considering now the computation of h_Δ as solved, let us see how Δ can be adjusted, following the general principles of Chap. 2. The merit function called there $t \to q(t) = f(x + td)$, with d fixed, is now $\Delta \to f(x + h_\Delta)$, which implies a few differences

(i) The trajectory $\{x + td\}_{t>0}$ becomes $\{x + h_\Delta\}_{\Delta>0}$, which is no longer a half-line but rather a curve in the space $\mathbb{R}^n$; we will rather speak of a *curvilinear* search.

(ii) If $\tilde{f}$ has a minimum point at finite distance, this trajectory has in turn a finite length. In case of extrapolations, there is then a difficulty: h_Δ eventually stops at a point minimizing $\tilde{f}(x + \cdot)$, even if $\Delta \to +\infty$.

(iii) The initial derivative $q'(0)$ becomes impossible to compute; the test $q(t) \leqslant q(0) + m_1 t q'(0)$, central in §2, is now impossible to implement. But the analogy with line-search still holds by interpreting $tq'(0)$ and $\frac{1}{2}tq'(0)$ as "nominal decreases" of f. Here, a nominal decrease can be simply taken as the decrease of the model:

$$\delta(\Delta) := \tilde{f}(x) - \tilde{f}(x + h_\Delta) \; [= f(x) - \tilde{f}(x + h_\Delta)] \,. \tag{11}$$

(iv) In addition to $q'(0)$ being unknown, $q'(t)$ becomes $\frac{\mathrm{d}}{\mathrm{d}\Delta} f(x + h_\Delta)$, a number which can be computed but which has little meaning. As a result, Wolfe's rule is no longer so natural.

In view of all this, the only Armijo's rule makes life simpler: this rule excludes any extrapolation and accommodates easily a trajectory $\{x + h_\Delta\}$ of finite length. It becomes here

$$f(x + h_\Delta) \leqslant f(x) - m\delta(\Delta) \,.$$

However, a mechanism is needed to force an increase of Δ in case of necessity; the algorithm loses some purity. So when the above criteria is not satisfied, h_Δ is too large, Δ must be decreased. In summary, the following algorithm is a schematic implementation of the trust-region technique.

Algorithm [Schematic trust-region] Given: a starting point x, a model $\tilde{f}$, an Armijo coefficient $m > 0$, an initial size Δ of the trust-region, and a simulator computing f-values.
– Compute h_Δ solving (9).
– If $f(x + h_\Delta) \leqslant f(x) - m\delta(\Delta)$ terminate.
– Else decrease Δ and loop.

Note that the above algorithm considers the static aspect only: its calls must be chained within an algorithm updating $x = x_k$, to finally minimize f. Note also a systematic initialization $\Delta = 1$ is dangerous. It is rather recommended to allow an increase of Δ when the search is finished, to initialize the next search.

4.3 Elements of convergence

It remains to check that this method does have the qualities ensuring convergence (global and possibly superlinear) of the sequence $\{x_k\}$ that it generates.

For quadratic models (8), everything relies upon the behaviour of $M = M_k$. To study the method, one must of course express that $x_+ = x + h_\Delta$ is not arbitrary, but really a minimum point of $\tilde{f}$ in the trust region. We can prove that if we set $\tilde{L} := \max\{1, \Lambda\}$, where Λ is the largest eigenvalue of M in (8), then we have in (11)

$$\delta(\Delta) \geqslant \frac{1}{2}\|g\| \min\Big(\Delta, \frac{\|g\|}{\tilde{L}}\Big).$$

Armijo's rule then gives directly

$$[f(x_+) =]\ f(x + h_\Delta) \leqslant f(x) - \frac{m}{2}\|g\| \min\Big(\Delta, \frac{\|g\|}{\tilde{L}}\Big)\,,$$

which is the necessary bound of $f(x) - f(x_+)$ in a convergence proof. We therefore see that, to obtain this bound, an upper bound of the largest eigenvalue of M is needed, the situation is no longer as in line-searches.

We stop technicalities here and we just mention the main convergence results allowed by trust-regions. With an appropriate (but rudimentary) management of $\Delta = \Delta_k$, and assuming that $\|M_k\|$ stays bounded, it can be shown that:

(i) either $f(x_k) \to -\infty$ or $\liminf \|g_k\| = 0$,

(ii) and the criterion of Dennis-Moré implies superlinear convergence.

Nothing particularly amazing so far; but trust-regions enjoy two "pluses":

(iii) indeed $g_k \to 0$; and more importantly:

(iv) take $M_k = \nabla^2 f(x_k)$; if $\{x_k\}$ is bounded, it has a cluster point x^* such that ($\nabla f(x^*) = 0$ and) $\nabla^2 f(x^*)$ is positive semi-definite. Thus, trust-region methods can avoid critical points which are not local minima. This is one of their strengths.

4.4 Least-square problems: Gauss-Newton

Important optimization problems come from *identification*, where the objective function is a sum of squares: we want to solve

$$\min f(x)\,, \quad \text{where} \quad f(x) = \frac{1}{2}\sum_{j=1}^{p} {f_j}^2(x)\,.$$

We show here that Newton-like algorithms to minimize such functions naturally appeal for trust-region techniques.

Elementary calculations give the gradient $\nabla f(x) = \sum f_j(x)\nabla f_j(x)$ and the Hessian

$$\nabla^2 f(x) = \sum_{j=1}^{p} \nabla f_j(x) {\nabla f_j}^\top(x) + \sum_{j=1}^{p} f_j(x)\nabla^2 f_j(x)$$

which is thus a sum of two terms; the first one depends on the gradients, only the second one involves second derivatives. Remembering that one of the drawbacks of Newton's method is the need to compute such second derivatives, it is tempting to neglect this second term. This amounts to considering

$$G(x) := \sum_{j=1}^{p} \nabla f_j(x) {\nabla f_j}^\top(x) = J(x)J^\top(x)\,,$$

where $J(x)$ denotes the matrix whose columns are the p gradients ∇f_j; $G(x)$ is called the *Gauss-Newton* matrix. A question is then: to what extent is $G(x)$ a good approximation of $\nabla^2 f(x)$. Answer:

- When the f_j's are mildly non-affine, the ∇f_j's are quasi-constant, the $\nabla^2 f_j$'s are small. Then $G(x) \simeq \nabla^2 f(x)$. Indeed, the whole idea of Gauss-Newton consists in saying: suppose the f_j's are affine, i.e. suppose the initial problem is a mere linear least-square problem $\min \frac{1}{2}\|J^\top x - z\|^2$; then it is solved by the Newton iterate $x - (JJ^\top)^{-1}g(x)$ (note that $g(x) = J(J^\top x - z)$). This amounts to assuming that the matrix $G(x) = JJ^\top$ does not depend on x – and is invertible, of course.
- When the least-square solution gives a good fit, the optimal value of f is close to 0. Then all the $f_j(x)$'s are small within convergence.

Anyhow, the Gauss-Newton method consists in computing the direction $-G_k^{-1}g_k$ at each iteration k. A (Wolfe) line-search then allows the computation of the next iterate; the resulting algorithm is well-defined, provided that each G_k is positive definite. Actually, the difficulty precisely lies at this point.

- When a matrix G_k is "hardly" positive definite, i.e. ill-conditioned, the method is in trouble; convergence may even be impaired.
- For this same reason, the line-search itself can often be in trouble: the Gauss-Newton direction can be orthogonal to the gradient, at least within roundoff.

Indeed, the designers of the method (neither Gauss nor Newton but Levenberg in 1944 and Marquardt in 1963) were aware of these difficulties due to ill-conditioning. They imagined, independently of each other, a stabilizing process: add to G_k a "suitably chosen" multiple $\lambda > 0$ of the identity. In view of §4.1 (and setting $\lambda = \mu$), we see that this is nothing other than a trust-region device. In summary: the Gauss-Newton method is dangerous when used with a line-search, and calls for trust-region. This gives an algorithm of the following type:

Algorithm [Gauss-Newton] Given: the Armijo coefficient $m \in]0, 1[$ and the stopping tolerance $\varepsilon > 0$. Set $k = 1$.

Step 1. Compute the gradient g_k of f at x_k. If $\|g_k\| \leqslant \varepsilon$ stop.

Step 2. Compute the Gauss-Newton matrix $G_k = G(x_k)$ and define the model

$$\tilde{f}(x_k + h) = f(x_k) + g_k^\top h + \frac{1}{2} h^\top G_k h \,.$$

Step 3. Use a trust-region algorithm as in §4 to obtain x_{k+1} satisfying

$$f(x_{k+1}) \leqslant f(x_k) - m(f(x_k) - \tilde{f}(x_{k+1})) \,.$$

Step 4. Increase k by 1 and loop to Step 1.

5 Conclusion

This study of unconstrained optimization conveys two important messages. One is that optimization algorithms can be given a "global" character, thanks to stabilization devices, such as line-search or trust-region. Second, the whole business for efficient algorithms is a proper use of second-order information. Concerning the latter, we have seen a variety of possibilities:

(i) The "standard" one, in which nothing is known beyond first order. This is unambiguously the realm of quasi-Newton methods, possibly with limited memory. It can be used in conjunction with line-search or with trust-region.

(ii) The case of least-squares problems, where an attractive approximation G_k (the matrix of Gauss-Newton) is directly available. Even though G_k is certainly positive semi-definite, it may be ill-conditioned. An appropriate parameter λ (of Levenberg-Marquardt) is needed for efficiency, as well as successive solutions to the system $(G_k + \lambda I)d = -g_k$, for successive values of λ.

(iii) In some situations, second derivatives can be computed, or conveniently approximated. The resulting matrix M_k need not be positive semi-definite. As a result, line-searches are definitely inappropriate: one needs either truncated conjugate gradient or trust-region.

Even when several of these possibilities are available, the best strategy is not necessarily obvious. Indeed consider a least-squares problem, where the exact Hessian can be computed, for example via automatic differentiation. Should one choose (i), (ii) or (iii)? and in case (i), should one choose a line-search? Actually the answer depends largely on *computing times*: how long does it take

– to compute f and ∇f?

– to compute $\nabla^2 f$?

– to obtain an appropriate solution to $\nabla^2 f(x)h = -g$ via truncated conjugate gradient?

– to obtain an appropriate λ in Gauss-Newton?

– to obtain an appropriate Δ in trust-region?

Thus the question cannot be given a general answer, and the particularities of each applications should be used for efficient computing.