ACTA MATHEMATICA VIETNAMICA
Volume 34, Number 1, 2009, pp. 91–103

IMPLEMENTATION ISSUES FOR HIGH-ORDER ALGORITHMS

JEAN-PIERRE DUSSAULT, BENOIT HAMELIN AND BILEL KCHOUK

Dedicated to Nguyen Van Hien on the occasion of his sixty-fifth birthday

Abstract. Newton-Raphson method, which dates back to 1669–1670, is widely used to solve systems of equations and unconstrained optimization problems. Newton-Raphson consists in linearizing the system of equations and provides quadratic local convergence order. Quite soon after Newton and Raphson introduced their iterative process, Halley in 1694 proposed a higher-order method providing cubic asymptotic convergence order. Chebyshev in 1838 proposed another high-order variant. In *High-order Newton-penalty Algorithms* [2], by interpreting Newton's iteration as a linear extrapolation, formulæ were proposed to compute higher-order extrapolations generalizing Newton-Raphson's and Chebyshev's methods.

In this paper, we provide details using an automatic differentiation (AD) tool to implement those high-order extrapolations. We present a complexity analysis allowing to predict the efficiency of those high-order strategies.

Introduction

Over the years, the so called Newton-Raphson [11, 12] iterative method has proved a very efficient solution for solving nonlinear equations and optimization problems. When applied to an unconstrained optimization problem $\min f(x)$, Newton's method reduces to iteratively solve a second-order approximation of f. A local minimum of f being a stationary point, one observes that Newton's method consists in iteratively solving a linear approximation of $g(x) = \nabla f(x)$. The linear approximation of g is defined using its Jacobian $H(x) = \nabla g(x) = \nabla^2 f(x)$. When f is more than twice differentiable, higher-order variants may be developed. Chebyshev's and Halley's methods both use third-order derivatives of f within an iterative scheme. Both those methods avoid solving the quadratic equations underlying a second-order Taylor expansion of g. This contrasts with the so-called tensor methods proposed in the 80's [13], which use clever approximations to the second-order model of g.

Received April 10, 2008; in revised form September 22, 2008.
2000 Mathematics Subject Classification. 65Y20.
Key words and phrases. Automatic differentiation, Newton-Raphson's method, Halley's method, Chebyshev's method.
This research was partially supported by NSERC grant OGP0005491.

In *High-order Newton-penalty Algorithms* [2], we proposed a different interpretation of the Newton-Raphson method, viewing it as a linear extrapolation of the implicit function $x(r)$, r being the residual of the nonlinear function at the current iterate. If the function is p times differentiable, we are then able to define extrapolations of orders as high as p. As it happens, a first order extrapolation is nothing else than Newton's method while a second-order extrapolation is Chebyshev's method.

These high-order approaches can be made practical using automatic differentiation techniques. The purpose of this paper is to present details of the computations of those higher-order variants within a specific automatic differentiation tool, SciAD [9], developed in the Scilab environment [14]. We will compare the per-iteration cost of the competing variants, and give some hints on their convergence properties.

1. High-order methods

In this section, we consider solving a system of equations $F(x) = 0$, where $F : \mathbb{R}^n \to \mathbb{R}^n$ is p times continuously differentiable. For an iterative method, at iteration k, we express

$$F(x_k) - r_k = 0$$

when solving optimization problems $\min f(x)$, $F(x) = g(x) = \nabla f(x)$.

1.1. **Newton-Raphson's method.**

This famous method consists in solving a linearization of F at x_k:

$$l_k(d) = F(x_k) + \nabla F(x_k)d = 0$$

and iterating $x_{k+1} = x_k + d_k$ where d_k solves $l_k(d_k) = 0$.

1.2. **Halley class of iterations.**

Chebyshev's, Halley's and a so-called super Halley's method may all be derived from the general iterative scheme [5, 6], where $k = 0, 1, 2...$:

$$x_{k+1} = x_k - \left[I + \frac{1}{2}L(x_k)[I - \alpha L(x_k)]^{-1}\right] \nabla F(x_k)^{-1} F(x_k),$$

and

$$L(x) = \nabla F(x)^{-1} \nabla^2 F(x) \nabla F(x)^{-1} F(x).$$

The three most usual cases are

- $\alpha = 0$: Chebyshev's method [1];
- $\alpha = \frac{1}{2}$: Halley's method [7];
- $\alpha = 1$: super Halley's method [6].

The above formulation is better rewritten as follows for implementation (omitting the iteration index k):

$$\begin{aligned}\nabla F(x)d_1 &= F(x)\\ (\nabla F(x)+\alpha\nabla^2 F(x)d_1)d_2 &= -\frac{1}{2}\nabla^2 F(x)d_1d_1.\end{aligned}$$

Therefore, each iteration requires the solution of two linear systems, and the evaluation of $\nabla^2 F(x)d_1$ and $\nabla^2 F(x)d_1d_1$ once d_1 is computed. When $\alpha = 0$, for Chebyshev's method, the two linear system are both defined by the same matrix $\nabla F(x)$, and only $\nabla^2 F(x)d_1d_1$ is needed. As we will see, this has a tremendous impact on the computational cost per iteration.

All three third-order variants (using third-order derivatives of the objective function f, $\nabla^2 F(x) = \nabla^3 f(x)$) share an asymptotic convergence of order three. It has been reported that super Halley's method usually requires less iterations than the other two variants, even if it does not benefit from a higher convergence order.

1.3. **Extrapolations.** An unusual way of obtaining Newton's method consists in writing $\tilde{r}_k = \frac{r_k}{\|r_k\|}$,

$$F(x_k) - \tau\tilde{r}_k = 0,$$

and devising extrapolation formulæ from $\tau_k = \|r_k\|$ to 0. Let us apply the implicit function theorem to obtain x as a function of τ, with $x_k = x(\tau_k)$:

$$\nabla F(x_k)\dot{x}(\tau_k) - \tilde{r}_k = 0.$$

It is now clear that $\dot{x}(\tau_k) = \nabla F(x_k)^{-1}\tilde{r}_k$ and that the first order extrapolation is nothing else than Newton's direction, $d_N = \dot{x}(\tau_k)(0-\tau_k) = -\nabla F(x_k)^{-1}F(x_k)$. Whenever F is many times continuously differentiable at x^*, we may generalize the extrapolation to a higher-order Taylor expansion of x,

$$\hat{x}^p(0) = x(\tau) + \dot{x}(\tau)(0-\tau)\ldots + \frac{1}{p!}x^{(p)}(\tau)(0-\tau)^p,$$

thus providing a superlinear convergence of order $p+1$.

Even if high-order derivatives of F are required, this high-order Taylor expansion only requires solving linear systems, all involving the same matrix $\nabla F(x_k)$. For example,

$$\nabla^2 F(x)\dot{x}(\tau)\dot{x}(\tau) + \nabla F(x)\ddot{x}(\tau) = 0 \tag{1}$$

defines $\ddot{x}(\tau)$ as

$$\ddot{x}(r) = -\nabla F(x)^{-1}\left(\nabla^2 F(x)\dot{x}(r)\dot{x}(r)\right).$$

The extrapolation itself is $\hat{x}^2 = x(\tau)+\dot{x}(\tau)(0-\tau)+\frac{1}{2}\ddot{x}(\tau)(0-\tau)^2$. One recognizes Chebyshev's iteration in this $\hat{x}^2$.

1.4. **Convergence.** The local convergence properties of the above methods are quite well documented. Newton's method converges quadratically while all the variants in the Halley family converge with an asymptotic order three. It is generally considered that super Halley's method requires marginally less iterations than Chebyshev's method, both third-order variants reputedly outperform Newton's method in terms of the number of iterations. We are concerned with more global efficiency considerations, taking into account the per-iteration computational costs as well.

For optimization problems, Newton's method may be adapted (modified Cholesky factorization of the Hessian matrix, line search, trust regions) to be globally convergent, thanks to the descent property of those modifications. The high-order variants will benefit from such modifications, but the details are left to future works.

2. Computational cost per iteration

Without presuming on the cost of involving high-order tensors, we now compare the main operations required by the three main variants, Newton's, Chebyshev's and super Halley's methods. Halley's method uses the same per iteration operations as super Halley (see (1)), but is generally reported less efficient.

The main computational items involve computing f, ∇f, $\nabla^2 f$, $\nabla^3 f d_1$, $\nabla^3 f d_1 d_1$ and solving a linear system $Mz = b$ for some right hand side vector b. For our unconstrained optimization application, the matrix M is symmetric, and in the neighborhood of a non-degenerate minimum, positive definite. Therefore, the solution of the linear system may be divided into two steps: factorization of $M = LL^t$, and two back substitutions involving the triangular matrix L.

In this investigation, we consider unstructured problems, either with no sparsity or unexploited sparsity. Similar observations on variants of Rosenbrock's function using highly efficient sparse linear algebra are reported in [4]. For such unstructured problems, the algebraic costs are dominated by the factorization of the second derivative matrix, an operation that has $O(n^3)$ complexity. As we will see, the tensor operations may be performed, thanks to the reverse mode of automatic differentiation, at a lower complexity.

As all three methods will evaluate f, ∇f and $\nabla^2 f$ at each iteration, and thus the associated cost will be the same. By using the reverse mode of automatic differentiation, the cost of evaluating ∇f is bounded by 5 times the cost of evaluating f [3], but the cost to obtain $\nabla^2 f$ is proportional to n times the cost of f.

For unstructured problems, one may expect that the cost of evaluating f will grow as $O(n^2)$. This cost will vary from one problem to another, very sparse functions being closer to $O(n)$, and parameter estimation problems bearing a very large constant cost multiplied by the problem's dimension, but for the moment, let us assume the relation that $C(f) \sim O(n^2)$. We will see below that in this case, the only $O(n^3)$ computations are the evaluation of $\nabla^2 f$, the factorization of $\nabla^2 f$ and the tensor operation $\nabla^3 f d_1$. All other operations may be performed at a cost proportional to n^2.

Method	$M = LL^t$	back-substitution	$\nabla^3 f d_1$	$\nabla^3 f d_1 d_1$
Newton	1	1	—	—
Chebyshev	1	2	—	1
superHalley	2	2	1	1

TABLE 1. Computationally important operations

3. Computations with higher-order derivatives

We now discuss the computation of the quantities involving the third-order tensor, $\nabla^3 f(x) d_1$ and $\nabla^3 f(x) d_1 d_1$. The formulation using the extrapolation introduced in Section 1.3 allows to obtain a simple presentation. Therefore, we assume that the quantity $d_1 = \dot{x}$. Up to a constant, this same quantity is used within the Halley family of iterations.

Therefore, we now obtain formulations allowing efficient computations of $\hat{x}^2$, and higher-orders $\hat{x}^p$ using automatic differentiation techniques. Those techniques rely on the representation of the computational graph of a given function, and produce a representation of the computational graph of its derivatives.

3.1. **A first attempt.**

Now, to rework the expression (1), we fix $\bar{x}^1 = \dot{x}(\tau)$, and write

$$\bar{B}_2 = \nabla^2 F(x)\dot{x}(\tau)\dot{x}(\tau) = \nabla_\tau(\nabla F(x(\tau))\bar{x}^1),$$

so that

$$\ddot{x}(\tau) = -\nabla F(x(\tau))^{-1}\bar{B}_2.$$

Thus expressed, $\bar{B}_2$ is obtained by differentiating the vector valued function $\nabla F(x(\tau))\bar{x}^1$, requiring n times its cost.

This may be improved as follows, but let us mention that the computation of $\nabla^2 F(x) d_1$ may be reduced to $\nabla_x(\nabla F(x) d_1)$ and $\nabla F(x) d_1$ is a vector valued function whose derivative costs n times the cost of its evaluation.

3.2. **An improved formulation for symmetric systems.**

Still using the notation above, we define the scalar function $\phi(x) = F(x)^t \bar{x}^1$, and further $\psi(x) = \nabla\phi(x)\bar{x}^1$. We claim that if $\nabla F(x)$ is symmetric, i.e. $F(x) = \nabla f(x)$ for some scalar valued function $f(x)$, $\bar{B}_2 = \nabla\psi(x)$. Observe that the above computation yields the *value* of $\bar{B}_2$, as opposed to its computational graph. It would then be more convenient to denote $B_2(u)$ the general quantity, and the value of $\bar{B}_2$ above is simply $\bar{B}_2 = B_2(\bar{x}^1)$.

Therefore, using the backward mode of automatic differentiation, we obtain B_2 with no more than 5 times the number of operations required to compute ψ [3], and ψ is computed in no more than 5 times the number of operations needed for ϕ, itself requiring one operation more than the computation of F itself, for an overall cost of no more than 25 times the cost of F. On the other hand, to obtain ∇F costs n times the cost of F.

3.3. **High-orders.** Successive orders have to be computed sequentially, since the right hand side of the relevant equation involves lower order derivatives. We then obtain a recursion

$$\nabla F(x(\tau))x^{(p)} + B_p(\dot{x}, \ddot{x}, \ldots, x^{(p-1)}) = 0$$

where

$$B_p(\dot{x}, \ddot{x}, \ldots, x^{(p-1)}) = \nabla_\tau \left(\nabla F(x)\bar{x}^{p-1} + B_{p-1}(\dot{x}, \ddot{x}, \ldots, x^{(p-2)})\right),$$

with $B_1 = 0$ and $\bar{x}^p$ a constant vector of value $x^{(p)}$. We were able to obtain an efficient formulation to compute $\bar{B}_2$. Let us examine the computation of $\bar{B}_3$.

$$\begin{aligned} B_3(\dot{x}, \ddot{x}) &= \nabla_r \left(\nabla F(x)\bar{x}^2 + B_2(\dot{x})\right) \\ &= \nabla_r \left(\nabla F(x)\bar{x}^2 + \nabla^2 F(x)\dot{x}\dot{x}\right) \\ &= \nabla^2 F(x)\dot{x}\bar{x}^2 + \nabla^3 F(x)\dot{x}\dot{x}\dot{x} + \nabla^2 F(x)\ddot{x}\dot{x} + \nabla^2 F(x)\dot{x}\ddot{x}. \end{aligned}$$

For symmetric systems, we then have

$$\bar{B}_3 = \nabla^3 F(x)\bar{x}^1\bar{x}^1\bar{x}^1 + 3\nabla^2 F(x)\bar{x}^1\bar{x}^2,$$

and $\nabla^3 F(x)\bar{x}^1\bar{x}^1\bar{x}^1 = \nabla(\nabla\psi(x)\bar{x}^1)$ while $\nabla^2 F(x)\bar{x}^1\bar{x}^2$ may be obtained similarly to $\bar{B}_2$ by using $\bar{x}^1$ in the definition of ϕ and $\bar{x}^2$ in the definition of ψ. The most costly term is thus the term involving third derivatives, whose cost is bounded above by 5 times the cost of computing $\nabla\psi$, i.e. less than 125 times the cost of evaluating F itself.

It seems reasonable to estimate the cost of evaluating F as $\mathcal{O}(n^2)$; in that case, automatic differentiation techniques allow to compute each of the required quantities in $\mathcal{O}(n^2)$ arithmetic operations, so that the cost of evaluating those high-order Taylor coefficients is still dominated by the factorization of the matrix $\nabla F(x)$, $\mathcal{O}(n^3)$ operations, which has to be performed only once. However, the constant in the $\mathcal{O}(n^2)$ grows exponentially with respect to the derivative order p, but (of course) is independent of n.

4. Experimental efficiency

The computations sketched above require to interleave derivative operations and numeric computations. To our knowledge, no existing AD tool except our implementation SciAD [8] allows such combination.

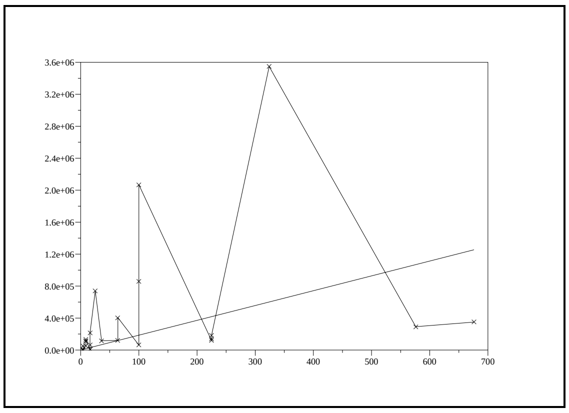
SciAD has been instrumented to collect statistics on operations performed while evaluating a given computation tree. Since every arithmetic operation is overloaded, it is easy to obtain the total number of each operation, whose grand total is accumulated using the weights given in Table 2, where complex operation may be logarithms, exponentials, trigonometric functions and so on. Those weights correspond to the Sun SPARC architecture [15]. Intel with floating point accelerators behave similarly, but of course, the exact weights are slightly different. Our purpose is not to provide accurate predictions, but exhibit the general trend. We do not rely on CPU clock measurements because our experiment is performed under the Scilab environment, which is an interpreted environment plagued by severe non-arithmetic overhead.

operation	Memory		Basic ops			complex ops
	fetch	write	$+-$	$*$	$/$	sin, cos, etc.
cost	2	1	4	8	24	247

Table 2. Operation costs for the Sun SPARC architecture

Hereafter, we present graphically several statistics. The function F is actually the gradient of an objective function. We present statistics for the MGH [10] collection translated to Scilab from the MATLAB version.

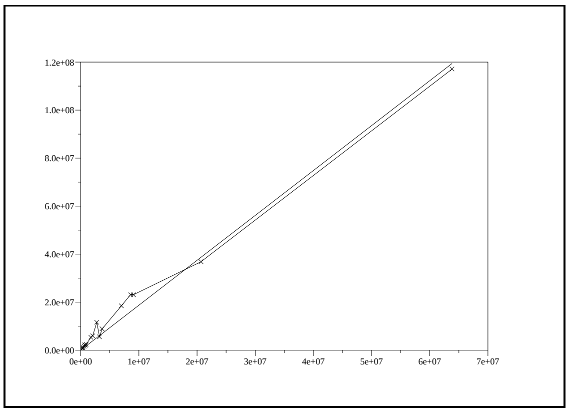
4.1. **Computational cost of ∇f.** Our first experiment plots the cost of evaluating $\nabla f(x)$ given the evaluation graph for the function $f(x)$. We plot the cost with respect to n^2. A regression gives $C(\nabla f) = 1855.44\ n^2$. This cost accumulates the cost of obtaining the evaluation graph for ∇f and of evaluating it for a given x.



The gradient of the function f involves n variables. It is reasonable to guess that its evaluation cost is proportional to n^2, a cost that would be realistic for a quadratic function. For the problems we tested, we observe that the variance is large, but that the trend proportional to n^2 is not unrealistic. The blue line illustrates the regression $C(\nabla f) = 1855.44\ n^2$

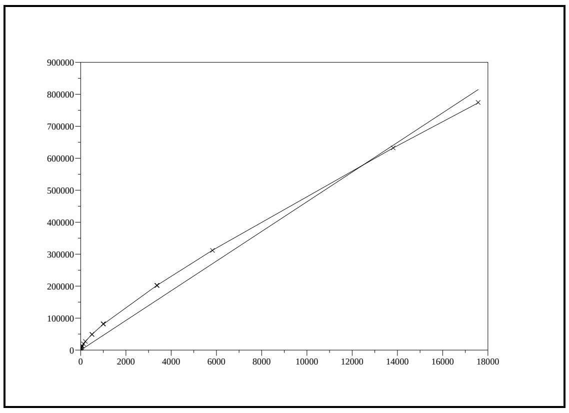
Figure 1. Cost of the gradient

4.2. **Computational cost of $\nabla^2 f$.** Next, we plot the cost of computing $\nabla^2 f$ once the evaluation graph for ∇f is available. A regression gives $C(\nabla^2 f) = 1.87\ n\ C(\nabla f)$.



In order to obtain $\nabla^2 f$, we must evaluate n times the derivatives of the gradient's components, so that we expect the cost to be of the order of n times the cost of ∇f. This time, the blue line representing the regression $C(\nabla^2 f) = 1.87\ n\ C(\nabla f)$ predicts much more faithfully the actual cost .

FIGURE 2. Cost of the Hessian



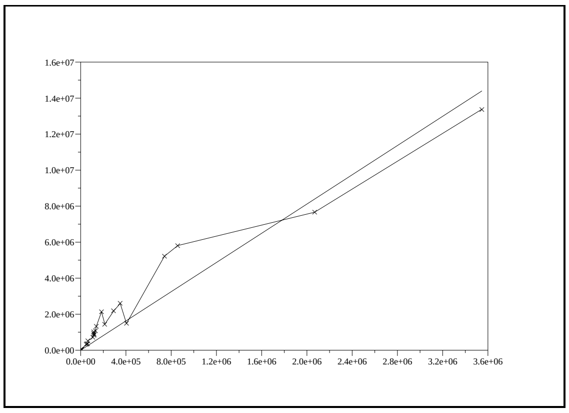
When computing $\dot{x}$, we have to solve a linear system. Actually, computing $\dot{x}$ is equivalent to performing a Newton-Raphson iteration, $\mathcal{O}(n^3)$. The regression $C(\dot{x}) = 46.38\ n^3$ is quite faithful to the observed data. .

FIGURE 3. Cost of the first order derivative $\dot{x}$

4.3. **Computational cost of $\dot{x}$.** Now, we give the cost of obtaining $\dot{x}$, i.e. obtaining the Cholesky factors of $\nabla^2 f$, and solve the linear system defining $\dot{x}$, the Newton direction. The regression yields $C(\dot{x}) = 46.38\ n^3$.

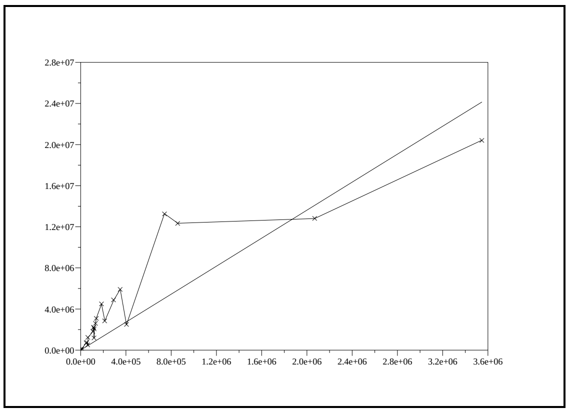
4.4. **Computational cost of $\ddot{x}$.** The additional work to obtain $\ddot{x}$ requires to evaluate B_2, as explained in Section 3.2, and solve a linear system reusing the Cholesky factors already computed to obtain $\dot{x}$. The regression yields $C(\ddot{x}) = 4.06\ C(\nabla f)$.

4.5. **Computational cost of $x^{(3)}$.** Our last experiment compares the additional work for obtaining the third derivative of x with respect to the cost of obtaining the gradient. The regression yields $C(x^{(3)}) = 6.81\ C(\nabla f)$.



When computing $\ddot{x}$, we have to solve a linear system, but the Cholesky factors of the matrix defining the linear system are known, so that we expect the cost to be $\mathcal{O}(n^2)$. The regression $C(\dot{x}) = 4.06\ C(\nabla f)$ is quite realistic; we already saw that $C(\nabla f) \sim \mathcal{O}(n^2)$.

FIGURE 4. Cost of the second-order derivative $\ddot{x}$



To compute the third-order derivative of $x(\tau)$, we need obtain high-order derivatives of functions constructed by interleaving arithmetic operations and differentiation operations. The order should be quadratic, but the underlying constant could, in worst case analysis, explode as 5^p. The good news is, for our test set, that this exponential explosion seems not to occur, as the regression (the blue line) $C(x^{(3)}) = 6.81\ C(\nabla f)$ is quite close to the observed data.

FIGURE 5. Cost of the third-order derivative $x^{(3)}$

4.6. **Discussion.** We confirm the complexity estimations experimentally. Actually, we observe that the exponential dependence of the cost of higher derivative is much less in practice than the predicted 5^p.

5. Local convergence

We present a few examples illustrating the gains provided by the use of high-order variants close to a solution. We picked an x_0 such that $\|x_0 - x^*\| \sim 0.1$, except for the Box function, for which no variant converged from that far, and we used $\|x_0 - x^*\| \sim 0.01$. As can be observed, on Rosenbrock, the initial point

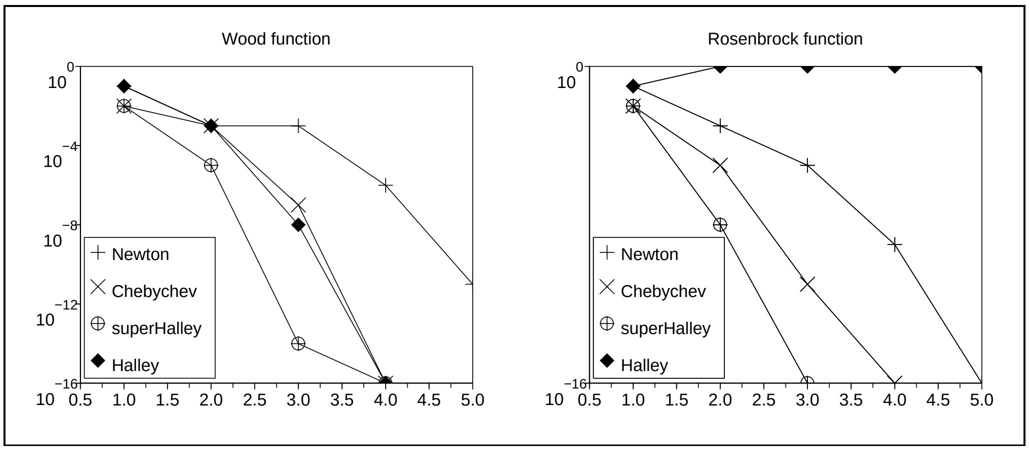


Figure 6. Wood and Rosenbrock functions

was outside the convergence radius of Halley's method. One can appreciate that super Halley's method is indeed the most efficient in terms of local convergence speed, but the gain does not appear to be enough to compensate its much higher per iteration cost when compared to Chebyshev's method.

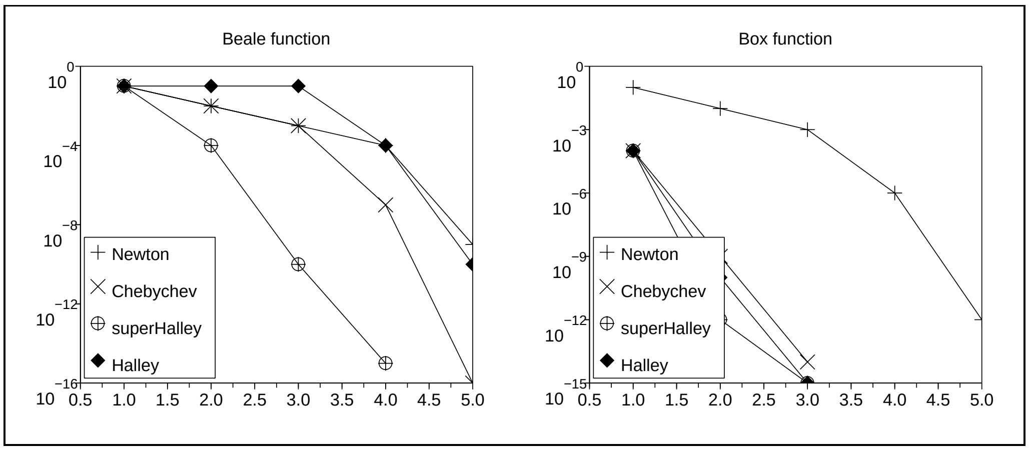


Figure 7. Beale and Box functions

6. High-order Newton-Raphson iteration

In the unconstrained optimization context, one expects algorithms to be modified to ensure global convergence while retaining fast asymptotic convergence. The use of higher-order variants will hopefully reduce the number of iterations, but at some per-iteration cost. We here compare Chebyshev's and Newton-Raphson's iterations, both suitably modified using the Cholesky modified factorization and a line search (simple Armijo backtracking method) to ensure global convergence. We use a very simple heuristic to trigger the use of Chebyshev's method. As discussed in Section 1.2, Chebyshev's is obtained as a correction to the Newton direction. Therefore, we use the correction only when the Cholesky factorization returns an unmodified result, i.e. when $\nabla^2 f(x)$ is sufficiently positive definite.

Problem number	Dimension	Newton	Chebychev	Best
		# weighted ops	# weighted ops	
1	2	91916	155773.	Newton
2	2	51076	63422	Newton
3	2	1006441	1165651	Newton
4	2	230426	248213	$\sim=$
5	2	568310	608360	$\sim=$
6	2	2502457	2986719	Newton
8	3	7885767	9088827	Newton
9	3	3719897	5279117	Newton
10	3	2.392×10^8	2.032×10^8	Chebychev
12	3	3493107	3659552	$\sim=$
13	4	1281523	872617	Chebychev
14	4	3026279	3197496	$\sim=$
15	4	6998007	8818893	Newton
16	4	26621673	32473938	Newton
21	10	19638946	17697387	Chebychev
22	8	18706802	11516356	Chebychev
23	10	55189602	36882347	Chebychev
24	10	8.441×10^8	7.487×10^8	Chebychev
25	10	29723031	20044968	Chebychev
26	10	1.145×10^8	1.105×10^8	$\sim=$
28	12	30712372	34906996	Newton
29	12	1.956×10^8	2.192×10^8	Newton
30	10	27017447	22318732	Chebychev
31	10	55331379	47741981	Chebychev
32	10	909059	1038633	Newton
33	10	3.545×10^8	3.546×10^8	$\sim=$
34	10	3.453×10^8	3.454×10^8	$\sim=$

Table 3. Newton vs Chebyshev on the MGH collection

In our experiment, we require precisions close to the machine precision, i.e. 1×10^{-12}. However, the value of the improved iteration lies in the actual computational cost. With the instrumented overloaded arithmetic operations and standard functions, we may quantify the cost for a standard Newton method (modified for optimization problems in order to ensure global convergence) and a hopefully improved version using higher-order derivatives.

Our first attempt used the formulation of Section 3.1, and the results were disappointing, the plain Newton iteration exhibiting lower costs than the second-order version.

In Table 3, we observe that the version of a second order iteration using the strategy developed above actually slightly improves upon the usual Newton iteration; note that for very small instances ($n = 2, 3$ or 4) Newton's method is more efficient. This is predictable, since 2 Newton iterations have a cost proportional to $2n^3$ and provide a quadratic convergence order while a Chebyshev iteration have a cost proportional to n^3+n^2 and provides a cubic convergence order. When $n = 2$ or 3, the gain from $n^3 + n^2$ over $2n^3$ is not enough to warrant the use of the higher-order method. In contrast, for $n = 8, 10, 12$, the Chebyshev variant is more efficient in 7 out of 13 instances, and equivalent in 3 other.

Conclusion

We presented a clever analysis and implementation proposal for high-order methods of the Halley family, and high-order methods introduced in [2]. The use of automatic differentiation tools allows to obtain the required higher derivatives efficiently. We reported on a preliminary experimental implementation which confirms that high-order variants may well be competitive, and even better than the usual Newton's method.

We limited ourselves to unstructured dense problems. For structured sparse problems, Gundersen and Steihaug [4] obtained similar conclusions with super Halley's method on high dimensional chained and extended Rosenbrock functions. They did not address the subtleties in evaluating directional tensors as we presented in Section 3.2, and their leading complexity is much lower than n^3 since they consider very sparse problems.

In all cases, high-order methods have been shown viable alternatives to the usual Newton method, and those promising aspects certainly warrant further research.

References

[1] P. L. Chebyshev, *Collected works*, Number 5, Moscow-Leningrad, 1951 (in Russian), an early work while he was a student. In 1841 Chebyshev was awarded the silver medal for his work "calculation of the roots of equations" which he had finished in 1838.

[2] J.-P. Dussault, High order Newton-penalty algorithms, *J. Comput. Appl. Math.* **182** (1) (2005), 117–133.

[3] A. Griewank, On automatic differentiation, In M. Iri and K. Tanabe, editors, *Math. Program.: Recent Developments and Applications*, pp. 83 – 108. Kluwer Academic Publishers, 1989.

[4] G. Gundersen and T. Steihaug, On large scale unconstrained optimization problems and higher order methods, *Optimization online*, 2007. A revised version will appear in *Optimisation Methods and Software.*

[5] J. M. Gutiérrez and M. A. Hernández, A family of Chebyshev-Halley type methods in Banach spaces, *Bull. Austral. Math. Soc.* **55** (1997), 113–130.
[6] J. M. Gutiérrez and Miguel A. Hernández, An acceleration of Newton's method: Super-Halley method, *Appl. Math. Comput.* **117** (2001) (2–3), 223–239.
[7] E. Halley, A new, exact, and easy method of finding roots of any equations generally, and that without any previous reduction, *Philos. Trans. Roy. Soc. London* **18** (1694), 136–145.
[8] B. Hamelin and J.-P. Dussault, Object-oriented implementation of an automatic differentiation toolkit in a high-level numerical processing functional language, in *Scilab 2004 INRIA Rocquencourt*, 2004.
[9] B. Hamelin and J.-P. Dussault, SCIAD Toolbox of automatic differentiation software and related algorithmic computation tools.
http://www.dmi.usherb.ca/ hamelin/autodiff/html/sciad.html, 2004.
[10] J. J. Moré, B. S. Garbow, and K. E. Hillstrom, Testing unconstrained optimization software, *ACM Trans. Math. Softw.* **7** (1) (1981), 17–41.
MATLAB version: ftp://ftp.mathworks.com/pub/contrib/v4/optim/uncprobs/.
[11] Isaac Newton, *Methodus fluxionium et serierum infinitarum*, 1664–1671.
[12] J. Raphson, *Analysis Aequationum Universalis*, London, 1690.
[13] R. B. Schnabel and T. Chow, Tensor methods for unconstrained optimization using second derivatives, *SIAM J. Optim.* **1** (1991), 293–315.
[14] Scilab group, ψlab 4.1.2. http://www.scilab.org, 2008.
[15] R. St-Denis, *La programmation en langage d'assemblage SPARC*, Les Éditions G.G.C., Sherbrooke, 1998.

Professeur titulaire, département d'Informatique, Université de Sherbrooke, Sherbrooke (Québec), Canada J1K 2R1

E-mail address: `Jean-Pierre.Dussault@USherbrooke.CA`

École Polytechnique, Montréal

E-mail address: `benoit@benoithamelin.com`

Département d'Informatique, Université de Sherbrooke,
Sherbrooke (Québec), Canada J1K 2R1

E-mail address: `Bilel.Kchouk@USherbrooke.CA`