

COMPLEXITY OF GRADIENTS, JACOBIANS, AND HESSIANS, SA

The evaluation or approximation of derivatives is a central part of most nonlinear optimization calculations. The gradients of objectives and active constraints enter directly into the Kuhn–Tucker–Karush conditions so that inaccuracies in their evaluation limit the achievable solution accuracy. The latter depends also crucially on the conditioning of the projected Hessian of the Lagrangian. Hence accurate values of this symmetric matrix allow the design of appropriate stopping criteria including the verification of second order conditions. Second derivatives also facilitate a rapid final rate of convergence, provided the step-defining linear systems can be solved by factorization or iteration at a reasonable cost. The same observation applies to more general optimization calculations like the solution of nonlinear complementarity problems.

Whether or not the obvious benefits of evaluating first and higher derivatives accurately justify the costs incurred, does strongly depend on the suitability of the differentiation method employed for the particular problem at hand. We may distinguish five principal options for evaluating or approximating derivatives

- *Symbolic Differentiation*
- *Handcoded Derivatives*
- *Automatic Differentiation*
- *Difference Quotients*
- *Secant Updating*

‘Numerical’ Differentiation Methods. The last two options are widely used in practical optimization, primarily because they require no extra effort whatsoever on the part of the user. Difference quotients are often called *divided differences* or *finite differences*, though the last term invites confusion with a related method for

Symbolic Differentiation
Handcoded Derivatives
Automatic Differentiation
Difference Quotients
Secant Updating
divided differences
finite differences
differencing
numerical differentiation
symbolic

discretizing differential equations. Other popular labels are *differencing* or *numerical differentiation*, because the results are floating point numbers rather than algebraic expressions. The latter are often presumed to be the output of more *symbolic* methods. Even though we shall see that the distinction is not quite that easy, there is no doubting the importance of the fundamental relation

$$\begin{aligned} F'(x) \dot{x} &= \left. \frac{d}{d\alpha} F(x + \alpha \dot{x}) \right|_{\alpha=0} \\ &= \frac{1}{\varepsilon} [F(x + \varepsilon \dot{x}) - F(x)] + O(\varepsilon). \end{aligned}$$

Here the vector function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$ is assumed Lipschitz-continuously differentiable on some neighborhood of the base point $x \in \mathbb{R}^n$. In other words, the directional derivative of F along some vector $\dot{x} \in \mathbb{R}^n$ is the product of the Jacobian matrix $F'(x) \in \mathbb{R}^{m \times n}$ with the direction \dot{x} and it can be approximated by a difference quotient. The quality of this approximation depends strongly on the choice of ε and one must expect a halving in the number of significant digits under the best of circumstances. Quasi-Newton, or Secant, Methods may be viewed as an ingenious way of sequentially incorporating difference quotients into a Jacobian approximation while iterating towards the solution vector of a nonlinear system of equations. The corresponding theory of superlinear convergence is quite beautiful from a mathematical point of view, though perhaps not terribly relevant in practice for large, structured problems.

It is important to note that the quality of the approximate derivative matrices g generated by Quasi-Newton methods influences only the rate of convergence but not so much the solution accuracy itself. The latter depends on the accurate evaluation of residual vectors, which may be composed of gradients as in the case

for the KKT conditions. The importance of accurate residual values is particularly well understood in numerical linear algebra, and replacing them with approximations of uncertain reliability is generally a dicy proposition. Fortunately, it just so happens that gradients can usually be evaluated with working precision at a moderate cost relative to that of the underlying functions. This is far from true for Jacobians and Hessians, whose cost is very hard to predict (and even define) as we shall demonstrate further below on various examples.

The relative cost of evaluating one sided-difference quotients in p directions \hat{x} from the same base point x is clearly $(p + 1)$. Theoretically one might sometimes reduce the evaluation costs by exploiting the fact that the p points $x + \varepsilon\hat{x}$ are close to x . This proximity may arise in the topological sense that the step size $\varepsilon\|\hat{x}\|$ is small as well as in the structural sense that \hat{x} is sparse and thus leaves many components of x unchanged. In practice such savings are rarely realized and they would certainly destroy the main advantage of differencing, namely its black box quality, which does not require any insight or access to the process by which function values are generated. Of course, there is the optimistic assumption that they vary smoothly as a function of the argument x , and usually the selection of a suitable increment ε causes enough trouble for the user and possibly even quite a few extra trial evaluations.

Hence it is indeed fair to assume that one-sided or centred differences in p directions \hat{x} at a common x require $1 + p$ or $1 + 2p$ separate function evaluations but little extra storage. By letting \hat{x} range over all n Cartesian basis vectors one obtains an approximate Jacobian with first or second order accuracy at the cost of $1 + n$ or $1 + 2n$ function evaluations. The number of dependent variables does not matter for differencing so that the cost of a gradient, where $m = 1$, is also $1 + n$ or $1 + 2n$ times that of the underlying scalar function. To compute Hessian or more generally full second derivative tensor one needs $n(n+1)/2$ function evaluations for one-sided and

column incidence graph
analytical differentiation

twice that many for the more accurate centred differences.

Since multiple function evaluations are an ‘embarrassingly’ parallel task the availability of several processors can be used to achieve a nearly perfect speed up for derivative approximations by differencing. In the sparse case, the number of independent variables n can be replaced in the cost ratios above by a number $p \leq n$ that represents either the maximal number of nonzeros in any row of $F'(x)$ or the usually slightly larger chromatic number of the *column incidence graph*. The latter reduction can be achieved by the by now classical grouping or coloring technique originally due to Curtis–Powell–Reid [5] and further developed by Coleman–Moré [3]. An alternative way to compress the rows of the Jacobian even further at the expense of some linear equation solving is due to Newsam–Ramsdell [12] and has recently been adopted to automatic differentiation.

‘Analytical’ Differentiation Methods. The first three options listed at the beginning are based on the chain rule and may therefore be combined under the label *analytical differentiation*. They all would yield exact derivative values if real arithmetic could be performed in infinite precision. Moreover, even the actual sequence of operations performed to evaluate a particular partial derivative would quite likely be the same and thus yield identical results if the same floating point arithmetic was used. Only the way in which the instruction for this floating point calculation are generated and stored differ significantly between the three approaches. Also, there may be more or less recalculation of intermediates that are common to several partial derivatives, which can have drastic effects on the computational efficiency.

The result of the second option *handcoding* may in principle be always similarly obtained by symbolic or automatic differentiation, provided the computer algebra package or the differentiation software is sufficiently smart. Hence we will discuss only the pure options one and three, which might of course also be combined

by a highly sophisticated programmer or software tool.

Symbolic differentiation is usually performed in *computer algebra* packages like Maple, Mathematica and Reduce. Most users have the notion that the differentiation command in these sophisticated systems turn formulas for functions into formulas for derivatives. Moreover there is a tendency to assume that having a ‘formula’ means directly expressing dependent variables as algebraic expressions of independents without allowing any named intermediates. The natural data structure for such formulas would be expression trees. There are every node has only one parent, so that the whole thing can be easily linearized and printed by enumeration in a depth first order. In reality computer algebra packages do not restrict themselves to expression trees, because for any nontrivial function the corresponding tree structure is very likely to represent an incredible amount of *redundancy*, even before any differentiation takes place.

The Two-Stranded Chain Scenario. Consider for example a sequence of complex function evaluations

$$x_{k+1} + i y_{k+1} = \phi_k(x_k + i y_k) + i \psi(x_k + i y_k)$$

for $k = 0 \dots l - 1$ starting from some initial $x_0 + i y_0 \in \mathbb{C}$. Suppose all function pairs $\phi_k + i \psi_k$ are nonlinear and don’t allow any algebraic simplifications. Then eliminating the intermediates x_1 and y_1 yields the formula

$$x_2 + i y_2 = \phi_1(\phi_0(x_0, y_0) + i \psi_0(x_0, y_0)) + \psi_1(\phi_0(x_0, y_0) + i \psi_0(x_0, y_0)),$$

which involves already twice as many terms as the one-level original formula. The same doubling occurs at each subsequent level so that expressing x_l and y_l directly in terms of the initial components x_0 and y_0 yields an exponentially long formula with the symbols x_0 and y_0 each occurring exactly 2^l times. In this case one could avoid the highly undesirable expression swell by merely substituting $z_k \equiv x_k + i y_k$, which turns

computer algebra
redundancy

the binary expression tree into a simple chain of the same height l .

While this example may appear rather algebraic and somewhat contrived, exactly the same effect occurs if the real pairs (x_k, y_k) specify straight lines in the plane. Specifically, one might think of light-beams being reflected in a maze of mirrors or some other optical arrangement in the plane. Each ray (x_k, y_k) that is incoming to a mirror or lense uniquely determines an outgoing ray (x_{k+1}, y_{k+1}) via some simple algebraic relationship. Then expressing the final ray parameters (x_l, y_l) directly as functions of the initial parameters (x_0, y_0) will again yield an expression of size 2^l .

Rather than dealing with this algebraic monster one should of course keep all the intermediate pairs (x_k, y_k) with $0 \leq k \leq l$ as named variables. Along this chain one can easily propagate all information of interest, including the 2×2 Jacobian of (x_k, y_k) with respect to (x_0, y_0) , at a temporal and spatial complexity of order l . To achieve this result one may employ suitable variants of computer algebra, automatic differentiation or, of course, hand-coding. Before discussing them in more detail let us discuss a general model of function and derivative evaluations.

The Computational Model. All analytical differentiation methods are based on the observation that most vector functions F of practical interest are being evaluated by a sequence of assignments

$$v_i = \varphi_i(v_j)_{j < i} \quad \text{for } i = 1 \dots l + m \quad . \quad (1)$$

Here the the variables v_i are real scalars and the elemental functions φ_i are either binary arithmetic operations or univariate intrinsics. Consequently, only one or two of the partial derivatives

$$c_{ij} \equiv \frac{\partial}{\partial v_j} \varphi_i(v_k)_{k < i}$$

do not vanish identically and can be evaluated at a cost comparable to that of the underlying φ_i itself.

Without loss of generality we may require that the first n variables $v_{j-n} = x_j$

with $j = 1 \dots n$ represent the *independent variables* and the last m variables $y_i = v_{i+m}$ with $i = 1 \dots m$ represent the *dependent variables*. Then the function $y = F(x)$ is defined by the program(1). Here the nonnegative integer l represents the number of intermediate variables, which we expect to be much larger than both n and m for seriously nonlinear problems. We will also assume that within a small constant all elemental functions have the same complexity so that we have the approximate operations count

$$\boxed{OPS(x \xrightarrow{\text{prog}} y) \sim l \equiv \#\text{intermediates} .}$$

Throughout this article \sim means proportional with small constants that are independent of the particular problem at hand. Each intermediate variable may be viewed and thus later differentiated as a function $v_i \equiv v_i(x)$ of the independent variable vector x . As long as all intermediates v_i are stored in separate locations the memory requirement for evaluating F will also be proportional to l . This is a very unrealistic assumption as most evaluation programs involve shared allocation of intermediates. Due to space constraints we will not be able to discuss any aspects of spatial complexity in this article. For a detailed treatment of various trade-offs between space and time see [6].

The way in which the elemental partials c_{ij} are handled differs amongst various analytical differentiation methods. They are always evaluated as floating point numbers at the current argument in what is variously known as *automatic* or *algorithmic* or *computational differentiation*. The same can be assumed for hand written derivatives codes unless they are programmed within a computer algebra system, where the c_{ij} can be defined and manipulated as algebraic expressions. In some cases applying the chain rule to these expressions may theoretically lead to significant simplifications and thus potentially provide the user with analytical insight. In the following section we reverse engineer one such

independent variables

dependent variables

automatic

algorithmic

computational differentiation

class of examples and arrive at the tentative conclusion that the practical potential for symbolic simplifications during the differentiation process appears to be very slim indeed.

Indefinite Integral Scenario. Suppose

$$F(x) = \int_a^x P(\tilde{x})/Q(\tilde{x}) d\tilde{x}$$

for two polynomials $P(\tilde{x})$ and $Q(\tilde{x})$ with $\text{deg}(Q) > \text{deg}(P)$. Besides a rational term the symbolic expression for $F(x)$ is then likely to contain a welter of logarithms and arcus tangents, whose complexity may easily exceed that of the integrand $f(x) \equiv P(x)/Q(x)$ by orders of magnitude. Then fully symbolic differentiation will of course lead back to an algebraic expression for $f(x)$, while automatic differentiation will combine the c_{ij} in floating point arithmetic according to some variant of the chain rule and obtain ‘just’ a numerical value of $f(x)$ at the given point $x \in \mathbb{R}$. Moreover, due to cancellations that value may well be less accurate than that obtained by plugging the particular argument x into the formula for $f(x)$.

However, similar numerical instabilities are likely to already affect the evaluation of $F(x)$ itself. They may also show up in the form of an imaginary component when the coefficients of $P(x)$ and $Q(x)$ are real but given in floating point format. Then the roots of the denominator polynomial are already perturbed by unavoidable round off and symbolic differentiation of the resulting expression for $F(x)$ will usually not lead back to $f(x)$ but some other rational function with a higher polynomial degree in the numerator or denominator. To avoid this effect all coefficients of $f(x)$ must be specified as algebraic numbers so that the symbolic integration can be performed exactly. This process which typically involves rational numbers with enormous coefficients and thus requires a large computational effort.

Hence on practical models one may well be better advised to evaluate $F(x)$ by a numerical quadrature yielding highly accurate results at a fraction of the computing time. Analytically differentiating a nonadaptive quadrature procedure yields the same quadrature applied to the derivatives of the integrand, namely $f'(x) = F''(x)$. Hence the resulting values are quite likely to be good approximations to the original integrand $f(x)$ and they are the exact derivatives of the approximate values computed for $F(x)$ by the quadrature.

Lack of Smoothness. Adaptive quadratures on the other hand may vary grid points and coefficient values in a nondifferentiable or even discontinuous fashion. Then derivatives of the quadrature value may well not exist in the classical sense at some critical arguments x . This difficulty is likely to arise in the form of program branches in all substantial scientific codes and there is no agreement yet on how to deal with it. In most situations one can still compute one-sided directional derivatives as well as generalized gradients and Jacobians [6]. Naturally, computing difference quotients of nonsmooth functions is also a risky proposition. Generally, optimal results in terms of accuracy and efficiency can only be expected from a derivatives code developed by a knowledgeable user, possibly with the help of program analysis and transformation tools.

Predictability of Complexities. With regards to spatial and temporal complexity the following basic distinction applies between the analytical differentiation methods sketched above. The cost of fully symbolic differentiation seems impossible to predict. It can sometimes be very low due to fortuitous cancellations but it is more likely to grow drastically with the complexity of the underlying function. In contrast the relative cost incurred by the various modes of automatic differentiation can always be a priori bounded in terms of the number of independent and dependent variables. Moreover, as we will see below these bounds can sometimes be substantially undercut for certain structured problems.

Another advantage of automatic differentiation compared to a fully symbolic approach is

that restrictions and projections of Jacobians and Hessians to certain subspaces of the functions domain and range can be built into the differentiation process with a corresponding savings in computational complexity. In the remainder of this article we will therefore focus on the complexity of various automatic differentiation techniques; always making sure that no other known approach is superior in terms of accuracy and complexity on general vector functions defined by a sequence of elemental assignments.

Goal Oriented Differentiation. The two-stranded chain scenario above illustrates the crucial importance of suitable representations of the mathematical objects, whose complexity we try to quantify here. So one really has to be more specific about what one means by *computing* a function, gradient, Jacobian, Hessian, or their restriction and projection to certain subspaces. At the very least we have to distinguish the (repeated) *evaluation* in floating point arithmetic at various arguments from the *preparation* of a suitable procedure for doing so. This preparation stage comes actually first and might be considered the symbolic part of the differentiation process. It usually involves no floating point operations, except possibly the propagation and simplification of some constants. This happens for example when a source code for evaluating F is precompiled into a source code for jointly evaluating $F(x)$ and its Jacobian $F'(x)$ at a given argument x . In the remainder we will neglect the preparation effort presuming that it can be amortized over many numerical evaluations as is typically the case in iterative or time-dependent computations.

In general, it is not a priori understood that $F'(x)$ should be returned as a rectangular array of floating point numbers, especially if it is sparse or otherwise structured. Its cheapest representation is the sparse triangular matrix $C = C(x) \equiv (c_{ij})_{i=1-n \dots l+m}^{j=1-n \dots l+m}$. The nonzero entries in C can be obtained during the evaluation of F at a given x for little extra cost in terms of arithmetic operations so that

$$\boxed{OPS\{x \mapsto C\} \sim OPS\{x \xrightarrow{\text{prog}} F\} \quad .}$$

As we will see below the nonzeros in C allow directly the calculation of the products

$$F'(x) \dot{x} \in \mathbb{R}^m \quad \text{for} \quad \dot{x} \in \mathbb{R}^n$$

and

$$F'(x)^T \bar{y}^T \in \mathbb{R}^n \quad \text{for} \quad \bar{y}^T \in \mathbb{R}^m$$

using just one multiplication and addition per $c_{ij} \neq 0$. So if our goal is the iterative calculation of an approximate Newton-step using just a few matrix-vector products, we are well advised to just work with the collection of nonzero entries of C provided it can be kept in memory. If on the other hand we expect to take a large number of iterations or wish to compute a matrix factorization of the Jacobian we have to first *accumulate* all mn partial derivatives $\partial y_i / \partial x_j$ from the elemental partials c_{ij} . It is well understood that a subsequent in-place triangular factorization of the Jacobian $F'(x)$ yields an ideal representation if one needs to multiply itself as well as its inverse by several vectors and matrices from the left or right. Hence we have at least three possible ways in which a Jacobian can be represented and kept in storage

- unaccumulated** computational graph,
- accumulated** rectangular array,
- factorized** two triangular arrays.

Here the arrays may be replaced by sparse matrix structures. For the time being we note that Jacobians and Hessians can be provided in various representation at various costs for various purposes. Which one is most appropriate depends strongly on the structure of the problem function $F(x)$ at hand and the final numerical purpose of evaluating derivatives in the first place. The interpretation of C as computational graph goes back to Kantorovich and requires a little more explanation.

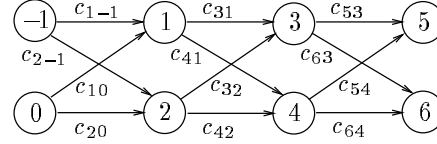
The Computational Graph. With respect to the precedence relation

$$j \prec i \iff c_{ij} \neq 0 \iff (j, i) \in E,$$

the indices $i, j \in V \equiv [1 - n \dots l + m]$ form a directed graph with the edge set E . Since by assumption $j \prec i$ implies $j < i$ the graph is acyclic and the transitive closure of \prec defines a partial order between the corresponding variables v_i

accumulate

and v_j . The minimal and maximal elements with respect to that order are exactly the independent and dependent variables $v_{j-n} \equiv x_j$ with $j = 1 \dots n$ and the $v_{m+i} \equiv y_i$ with $i = 1 \dots m$, respectively. For the two stranded chain scenario with $l = 3$ one obtains a computational graph of the following form.



Assuming that all elemental φ_i are unary functions or binary operations we find $|E| \leq 2(l + m) \approx l$. One may always annotate the graph vertices with the elemental functions φ_i and the edges with the nonvanishing elemental partials c_{ij} . For most purposes the φ_i do not really matter and we may represent the graph (V, E) simply by the sparse matrix C .

Forward Mode. Given some vector $\dot{x} \equiv (\dot{v}_{j-n})_{j=1 \dots n} \in \mathbb{R}^n$ there exist derivatives

$$\dot{v}_i \equiv \left. \frac{d}{d\alpha} v_i(x + \alpha \dot{x}) \right|_{\alpha=0} \quad \text{for} \quad 1 \leq i \leq l + m.$$

By the chain-rule these \dot{v}_i satisfy the recurrence

$$\dot{v}_i \equiv \sum_{j \prec i} c_{ij} \dot{v}_j \quad \text{for} \quad i = 1 \dots l + m. \quad (2)$$

The resulting tangent vector $\dot{y} \equiv (\dot{v}_{l+i})_{i=1 \dots m}$ satisfies by $\dot{y} = F'(x) \dot{x}$ and it is obtained at a cost proportional to l . Instead of propagating derivatives with respect to just one direction vector \dot{x} one may amortize certain overheads by bundling p of them into a matrix $\dot{X} \in \mathbb{R}^{n \times p}$ and then computing simultaneously $\dot{Y} = F'(x) \dot{X}$. The cost of this *vector forward* mode of automatic differentiation is given by

$$\boxed{OPS\{C \xrightarrow{\text{forw}} \dot{Y}\} \sim p l \sim p OPS\{x \xrightarrow{\text{prog}} y\}}.$$

If the columns of \dot{X} are Cartesian basis vectors $e_j \in \mathbb{R}^n$ the corresponding columns of the resulting \dot{Y} are the j th columns of the Jacobian. Hence by setting $\dot{X} = I$ with $p = n$ we may compute the whole Jacobian at a temporal complexity proportional to nl . Fortunately,

in many applications the whole Jacobian is either not needed at all or due to its sparsity pattern it may be reconstructed from its *compression* $\dot{Y} = F'(x)\dot{X}$ for a suitable *seed matrix* \dot{X} . As in the case of difference quotients this matrix may be chosen according to the CPR [5] or the NR[12] approach with p usually close to the maximal number of nonzeros in any row of the Jacobian.

Bauer's Formula. Using the recurrence for the \dot{v}_i given above one may also obtain an explicit expression for each individual partial $\partial y_i / \partial x_j$. Namely, it is given by the sum over the products of all arc values c_{ij} along all paths connecting the minimal node v_{j-n} with the maximal node v_{l+i} . This formula due to Bauer [1] implies in particular that the ij th Jacobian entry vanishes identically exactly when there is no path connecting nodes $j-n$ and $l+i$ in the computational graph. In general the number of distinct paths in the graph is very large and it represents exactly the lengths of the formulas obtained if one expresses each y_i directly in terms of all x_j that it depends on. Hence we may conclude

$$\boxed{OPS\{C \xrightarrow{\text{bauer}} F'\} \sim OPS\{x \xrightarrow{\text{formul}} y\}} \quad (3)$$

In the two-stranded chain scenario considered above, both operations count would be of order 2^l , which is obviously an unacceptable effort. Fortunately, vector forward and Bauer's formula are just two special choices amongst many ways for accumulating the Jacobian $F'(x)$ from the computational graph C . The most celebrated alternative is the reverse or backward mode of automatic differentiation.

Reverse Mode. Rather than propagating directional derivatives \dot{v}_i forward through the computational graph one may also propagate adjoint quantities \bar{v}_i backward. To define them properly one must perturb the original evaluation loop by rounding errors δ_i so that now

$$v_i = \delta_i + \varphi_i(v_j)_{j < i} \quad \text{for } i = 1 - n \dots l.$$

Then the resulting vector y is a function not only of x but also the vector of small perturbations $(\delta_i)_{i=1-n \dots l+m}$. Given any row vector of weights

$\bar{y} = (\bar{v}_{l+i})_{i=1 \dots n}$ we obtain the sensitivities

$$\bar{v}_i \equiv \left. \frac{\partial}{\partial \delta_i} \bar{y} y \right|_{\delta_i=0} \quad \text{for } 1 - n \leq i \leq l$$

where all other perturbations δ_j with $j \neq i$ are set to zero during the differentiation. The adjoint components $\bar{v}_{j-n} = \bar{x}_j$ form the row vector $\bar{x} = \bar{y}F'(x) \in \mathbb{R}^n$, which is simply the gradient of the linear combination $\bar{y}F(x)$. In the optimization context this scalar valued function is usually a Lagrangian, whose gradient and Hessian figures prominently in the first and second order optimality conditions. The amazing thing is that as a consequence of the chain rule such gradients can be computed at the same cost as tangents by using the backward recurrence

$$\bar{v}_i = \sum_{i > j} \bar{v}_i c_{ij} \quad \text{for } j = l \dots 1 - n. \quad (4)$$

Just like in the forward scalar recurrence (2) each elemental partial $c_{ij} \neq 0$ occurs exactly once and we may amortize costs by bundling several \bar{y} into an adjoint seed matrix $\bar{Y} \in \mathbb{R}^{q \times m}$. This vector reverse mode yields the matrix $\bar{X} = \bar{Y}F'(x) \in \mathbb{R}^{m \times n}$ at the cost

$$\boxed{OPS\{C \xrightarrow{\text{rev}} \bar{X}\} \sim q l \sim q OPS\{x \xrightarrow{\text{prog}} y\}}$$

Again the whole Jacobian is obtained directly if we seed $\bar{X} = I$ with $q = m$ and in the sparse case we may now employ column rather than row compression with q roughly equal to the maximal number of nonzeros in any column of $F'(x)$. Hence we find by comparison with (3) as a rule of thumb that the reverse mode is preferable if $m \ll n$, i.e., if there are not nearly as many dependents as independents. In classical NLP's we may think of m as the number of active constraints plus one, which is often much smaller than n , the number of variables. In unconstrained optimization we have $m = 1$ so that the gradient of the objective F can be computed with essentially the same effort as F itself.

For suitable seeds \bar{Y} the column compression $\bar{X} = \bar{Y}F'(x)$ allows the reconstruction of the complete Jacobian $F'(x)$. Furthermore, row and column compression can be combined yielding for example Jacobians with arrow head structure at the cost of roughly $p + q = 3$ function

evaluations. In that case one may use

$$\dot{X} \equiv \begin{bmatrix} 1 & 1 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix}^T \quad \text{and} \quad \bar{Y} = e_m^T .$$

Then $\bar{X} = \bar{Y}F'(x)$ is the last row of the arrowhead matrix $F'(x)$ and the two columns of $\dot{Y} = F'(x)\dot{X}$ contain all other nonzero entries. For pure row or reverse compression dense rows or columns always force $p = n$ or $q = m$, respectively. Hence the *combination* of forward and reverse differentiation offers the potential for great savings. In either case projections and restrictions of the Jacobian to subspaces of the vector functions domain and range can be built into the differentiation process, which is part of the goal-orientation we alluded to before.

Second Order Adjoints. Rather than separately propagating some first derivatives forward, others reverse, and then combining the results to compute Jacobian matrices efficiently, one may compose these two fundamental modes to compute second derivatives like Hessians of Lagrangians. More specifically, we obtain by directional differentiation of the adjoint relation $\bar{x} = \bar{y}F'(x)$ the second order adjoint

$$\dot{\bar{x}} = \bar{y}F''(x)\dot{x} \in \mathbb{R}^n .$$

Here we have assumed that the adjoint vector \bar{y} is constant. We also have taken liberties with matrix vector notation by suggesting that the $m \times n \times n$ derivative tensor $F''(x)$ can be multiplied by the row vector $\bar{y} \in \mathbb{R}^m$ from the left and the column vector $\dot{x} \in \mathbb{R}^n$ from the right yielding a row vector $\dot{\bar{x}}$ of dimension n . In an optimization context \bar{y} should be thought of as a vector of Lagrange multipliers and \dot{x} as a feasible direction. By composing the complexity bounds for the reverse and the forward mode one obtains the estimates

$$\begin{aligned} OPS\{x \xrightarrow{\text{prog}} y\} &\sim OPS\{x \xrightarrow{\text{forw}} \dot{x}\} \\ &\sim OPS\{x \xrightarrow{\text{rev}} \bar{x}\} \sim OPS\{x \xrightarrow{\text{ad}} \dot{\bar{x}}\} \end{aligned}$$

Here *ad* represents reverse differentiation followed by forward differentiation or vice versa. The former interpretation is a little easier to implement and involves only one forward and one backward swept through the computational graph.

Operations Counts and Overheads. From a practical point of view one would of course like to know the proportionality factors in the relations above. If one counts just multiplication operations then \dot{y} and \bar{x} are at worst 3 times as expensive as y and $\dot{\bar{x}}$ is at most 9 times as expensive. A nice intuitive example is the calculation of the determinate y of a $\sqrt{n} \times \sqrt{n}$ matrix whose entries from the variable vector x . Then we have $m = 1$ and $OPS\{x \mapsto y\} = \frac{1}{3}\sqrt{n}^3 + O(n)$ multiplications if one uses an LU factorization. Then it can be seen that $\bar{y} = 1/y$ makes \bar{x} the transpose of the inverse matrix and the resulting cost estimate of $\sqrt{n}^3 + O(n)$ multiplications conforms exactly with the usual substitution procedure.

However, these operations count ratios are no reliable indications of actual runtimes, which depend very strongly on the computing platform, the particular problem an hand, and the characteristics of the AD tool. Implementations of the vector forward mode like ADIFOR [2] that generate compilable source codes can easily compete with divided differences, i.e. compute p directional derivatives in the form $\dot{Y} = F'(x)\dot{X}$ at the cost of about p function evaluations. For sizeable $p \approx 10$ they are usually faster than divided differences, unless the roughly p -fold increase in storage results in too much paging onto disk. The reverse mode is an entirely different ball-game since most intermediate values v_i and some control flow hints need to be first saved and later retrieved, which can easily make the calculation of adjoints memory bound. This memory access overhead can be partially amortized in the vector reverse mode, which yields a bundle $\bar{X} = \bar{Y}F'(x)$ of q gradient vectors. For example in multi-criteria optimization one may well have $q \approx 10$ objectives or soft constraints, whose gradients are needed simultaneously.

Worst Case Optimality. Counting only multiplications we obtain for Jacobians $F' \in \mathbb{R}^{m \times n}$ the complexity bound

$$OPS\{x \xrightarrow{\text{ad}} F'\} \leq 3 \min(n, m) OPS\{x \xrightarrow{\text{prog}} y\} .$$

Here n and m can be reduced to the maximal number of nonzero entries in the rows and columns of the Jacobian, respectively. Similarly,

we have for the one-sided projection of the Lagrangian Hessian

$$H(x, \bar{y}) \equiv \bar{y} F'' \equiv \sum_{i=1}^m \bar{y}_i \nabla^2 F_i \in \mathbb{R}^{n \times n}$$

onto the space spanned by the columns of \dot{X}

$$\boxed{OPS\{x \xrightarrow{\text{ad}} H(x, \bar{y}) \dot{X}\} \leq 9p OPS\{x \xrightarrow{\text{prog}} y\} .}$$

As we already discussed for indefinite integrals there are certainly functions whose derivatives can be evaluated much cheaper than they themselves for example using a computer algebra package. Note that here again we have neglected the preparation effort, which may be very substantial for symbolic differentiation. Nevertheless, the estimates given above for AD are optimal in the sense that there are vector functions $F(x)$ defined by evaluation procedures of the form (1), for which no differentiation process imaginable can produce the Jacobian and projected Hessian significantly cheaper than the given cost bound divided by a small constant. Here, producing these matrices is understood to mean calculating all its elements explicitly, which may or may not be actually required by the overall computation.

Consider for example the cubic vector function

$$F(x) = x + b(a^T x)^3/2 \quad \text{with } a, b \in \mathbb{R}^n .$$

Its Jacobian and projected Hessian are given by

$$F'(x) = I + b(a^T x)^2 a^T \in \mathbb{R}^{m \times n}$$

and

$$H(x, \bar{y}) \dot{X} = 2a(\bar{y}b)(a^T x)a^T \dot{X} \in \mathbb{R}^{m \times p} .$$

For general a, b and \dot{X} all entries of the matrices $F'(x)$ and $H(x, \bar{y}) \dot{X}$ are distinct and depend nontrivially on x . Hence their explicit calculation by any method requires at least n^2 or np arithmetic operations, respectively. Since the evaluation of F itself can be performed using just $3n$ multiplications and a few additions, the operations count ratios given above cannot be improved by more than a constant. There are other more meaningful examples [6] with the same property, namely that their Jacobians and projected Hessians are orders of magnitude redundancy

more expensive than the vector function itself. At least this is true if we insist on representing them as rectangular arrays of reals. This does not contradict our earlier observation that gradients are cheap, because the components of $F(x)$ cannot be considered as independent scalar functions. Rather their simultaneous evaluation may involve many common subexpressions, as is the case for our rank-one example. These appear to be less beneficial for the corresponding derivative evaluation, thus widening the gap between function and derivative complexities.

Expensive \equiv Redundant?? The rank-one problem and similar examples for which explicit Jacobians or Hessians appear to be expensive have a property that one might call *redundancy*. Namely, as x varies over some open neighbourhood in its domain, the Jacobian $F'(x)$ stays in a lower-dimensional manifold of the linear space of all matrices with its format and sparsity pattern. In other words, the nonzero entries of the Jacobian are not truly independent of each other so that computing them all and storing them separately may be wasteful. In the rank-one example the Jacobian $F'(x)$ is dense but belongs at all x to the one-dimensional affine variety $\{I + b\alpha a^T : \alpha \in \mathbb{R}\}$. Note that the vectors $a, b \in \mathbb{R}$ are assumed to be dense and constant parameter vectors of the problem at hand. Their elements all play the role of elemental partials c_{ij} with the corresponding operation φ_i being multiplications. Hence accumulating the extremely sparse triangular matrix C , which involves only $O(n)$ nonzero entries, to the dense $n \times n$ array $F'(x)$ is almost certainly a bad idea, no matter what the ultimate purpose of the calculation. In particular, if one wishes to solve linear systems in the Jacobian, the inverse formula of Sherman–Morris–Woodburry provides a way of computing the solution of rank-one perturbations to diagonal matrices with $O(n)$ effort. This formula may be seen as a very special case of embedding linear systems in F' into a much larger and sparse linear system involving C as demonstrated in [8] and [4].

As of now, all our examples for which the array representation of Jacobians and Hessians are

orders of magnitude more expensive to evaluate than the underlying vector function exhibit this redundancy property. In other words, we know of no convincing example where vectors that one may actually wish to calculate as end products are necessarily orders of magnitude more expensive than the functions themselves. Especially for large problems it seems hard to imagine that array representations of the Jacobians and Hessians themselves are really something anybody would wish to look at rather than just use as auxiliary quantities within the overall calculation.

So evaluating complete derivative arrays is a bit like fitting a handle to a wooden crate that needs to be moved about frequently. If the crate is of small weight and size this job is easily performed using a few wood screws. If, on the other hand, the crate is large and heavy, fitting a handle is likely to require additional bracing and other reinforcements. Moreover, this effort is completely pointless since nobody can just pick up the crate by the handle anyhow and one might as well use a fork left in the first place.

Preaccumulation and Combinatorics. The temporal complexity for both the forward and the reverse (vector) mode are proportional to the number of edges in the linearized computational graph. Hence one may try to reduce the number of edges by certain algebraic manipulations that leave the corresponding Jacobian, i.e., the linear mapping between \dot{x} and $\dot{y} = F'(x)\dot{x}$ and equivalently also that between \bar{y} and $\bar{x} = \bar{y}F'(x)$ unchanged. It can be easily checked that this is the case if given an index j one updates first

$$c_{ik} \dagger = c_{ij} c_{jk}$$

either for fixed $i \succ j$ and all $k \prec j$, or for fixed $k \prec j$ and all $i \succ j$, and then sets $c_{ij} = 0$ or $c_{jk} = 0$, respectively. In other words, either the edge (j, i) or the edge (k, j) is eliminated from the graph. This leads to fill-in by the creation of new arcs, unless all updated c_{ik} were already nonzero beforehand. Eliminating all edges (k, j) with $k \prec j$ or all edges (j, i) with $i \succ j$ is equivalent and amounts to eliminating the vertex j

completely from the graph. After all intermediate vertices $1 \leq j \leq l$ are eliminated in some arbitrary order, the remaining edges c_{ij} directly connect independent variables with dependent variables and are therefore entries of the Jacobian $F'(x)$. Hence, one refers to the *accumulation* of the Jacobian F' if all intermediate nodes are eliminated and to *preaccumulation* if some of them remain so that the Jacobian is represented by a simplified graph.

As we have indicated in the section on goal oriented differentiation one would have to carefully look at the problem function and the overall computational task to decide how much preaccumulation should be performed. Moreover, there are $\tilde{l}!$ different orders in which a particular set of $\tilde{l} \leq l$ intermediate nodes can be eliminated and even many more different ways of eliminating the corresponding set of edges. So far there have only been few studies of heuristic criteria for finding efficient elimination orderings down to an appropriate preaccumulation level [6].

Summary. First and second derivative vectors of the form $\dot{y} = F'(x)\dot{x}$, $\bar{x} = \bar{y}F'(x)$ and $\hat{x} = \bar{y}F''(x)\dot{x}$ can be evaluated for a fixed small multiple of the temporal complexity of the underlying relation $y = F(x)$. The calculation of the gradient \bar{x} and the second order adjoint \hat{x} by the basic reverse method may require storage of order $l \equiv \#\text{intermediates}$. This possibly unacceptable amount can be reduced to order $\log(l)$ at a slight increase in the operations count (see [7]).

Jacobians and one-sided projected Hessians can be composed column by column or row by row from vectors of the kind \dot{y} , \bar{x} and \hat{x} . For sparse derivative matrices row and/or column compression using suitable seed matrices of type CPR or NR allow a substantial reduction of the computational effort. In some cases the nonzero entries of derivative matrices may be redundant, so that their calculation should be avoided, if the overall computational goal can be reached in some other way. The attempt to evaluate derivative array with absolutely minimal effort leads to hard combinatorial problems.

References

- [1] F.L. Bauer, *Computational graphs and rounding error*, SIAM J. Numer. Anal. **11** (1974), 87–96.
- [2] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *ADIFOR: Generating derivative codes from Fortran programs*, Scientific Programming, 1 (1992), pp. 1–29.
- [3] T.F. Coleman and J.J. Moré, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal. **20** (1984), 187–209.
- [4] T.F. Coleman and A. Verma, *Structure and efficient Jacobian calculation*, in [10], pp. 149–159.
- [5] A.R. Curtis, M.J.D. Powell, and J.K. Reid, *On the estimation of sparse Jacobian matrices*, J. Inst. Math. Appl. **13** (1974), 117–119.
- [6] A. Griewank, *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*, Frontiers in Appl. Math., no. 19, SIAM, Philadelphia, (2000).
- [7] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software, 1 (1992), pp. 35–54.
- [8] A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in [11], pp. 126–135.
- [9] A. Griewank, *The Chain Rule Revisited in Scientific Computing*, I-II, *SIAM News*, May/July 1991.
- [10] M. Berz, C.H. Bischof, G. Corliss, and A. Griewank (eds.), *Computational differentiation—techniques, applications, and tools*, SIAM, Philadelphia, 1996.
- [11] A. Griewank and G.F. Corliss (eds.), *Automatic differentiation of algorithms: Theory, implementation, and application*, SIAM, Philadelphia, 1991.
- [12] G. N. Newsam and J. D. Ramsdell, *Estimation of sparse Jacobian matrices*, SIAM J. Algebraic Discrete Methods **4** (1983), 404–417.

Andreas Griewank

Institute of Scientific Computing,
Department of Mathematics
Technical University Dresden
Germany

E-mail address: `griewank@math.tu-dresden.de`

AMS1991 Subject Classification: 65D25,68W30.

Key words and phrases: automatic differentiation, divided differences, computer algebra, symbolic manipulation.