Who Invented the Reverse Mode of Differentiation?

Andreas Griewank

2010 Mathematics Subject Classification: 05C85, 49M99, 65D25, 68Q17
Keywords and Phrases: Adjoints, gradient evaluation, round-off estimation, program reversal

Prologue

Nick Trefethen [13] listed automatic differentiation as one of the 30 great numerical algorithms of the last century. He kindly credited the present author with facilitating the rebirth of the key idea, namely the *reverse mode*. In fact, there have been many incarnations of this reversal technique, which has been suggested by several people from various fields since the late 1960s, if not earlier.

Seppo Linnainmaa (Lin76) of Helsinki says the idea came to him on a sunny afternoon in a Copenhagen park in 1970. He used it as a tool for estimating the effects of arithmetic rounding errors on the results of complex expressions. Gerardi Ostrowski (OVB71) discovered and used it some five years earlier in the context of certain process models in chemical engineering. Here and throughout references that are not listed in the present bibliography are noted in parentheses and can be found in the book [7].

Also in the sixties Hachtel et al. [6] considered the optimization of electronic circuits using the costate equation of initial value problems and its discretizations to compute gradients in the reverse mode for explicitly time-dependent problems. Here we see, possibly for the first time, the close connection between the reverse mode of discrete evaluation procedures and continuous adjoints of differential equations. In the 1970s Iri analyzed the properties of dual and adjoint networks. In the 1980s he became one of the key researchers on the reverse mode.

From a memory and numerical stability point of view the most difficult aspect of the reverse mode is the reversal of a program. This problem was discussed in the context of Turing Machines by Benett (Ben73), who foreshadowed the use of checkpointing as a tradeoff between numerical computational effort and memory requirement.

Motivated by the special case of back-propagation in neural networks, Paul Werbos (Wer82) compared the forward and reverse propagation of derivatives for discrete time-depedent problems with independent numbers of input, state, and output variables. He even took into account the effects of parallel computations on the relative efficiency.

Many computer scientists know the reverse mode as the *Baur-Strassen* method (BS83) for computing gradients of rational functions that are evaluated by a sequence of arithmetic operations. For the particular case of matrix algorithms Miller et al. proposed the corresponding roundoff analysis [10]. Much more general, Kim, Nesterov et al. (KN+84) considered the composition of elementary functions from an arbitrary library with bounded gradient complexity.

Bernt Speelpenning (Spe80) arrived at the reverse mode via compiler optimization when Bill Gear asked him to automatically generate efficient codes for Jacobians of stiff ODEs. I myself rediscovered it once more in the summer of 1987 when, newly arrived at Argonne, I was challenged by Jorge Moré to give an example of an objective function whose gradient could not be evaluated at about the same cost as the function itself.

One of the earliest uses of the reverse mode was in data assimilation in weather forecasting and oceanography. This was really just a history match by a weighted least squares calculation on a time-dependent evolution, where the parameters to be approximated include the present state of the atmosphere. The recurrent substantial effort of writing an adjoint code for geophysical models eventually spawned activities to generate adjoint compilers such as Tapenade (HP04) and TAF (GK98).

The first implementations of the reverse mode based on the alternative software technology of operator overloading was done in PASCAL-SC, an extension of PASCAL for the purposes of interval computation. The corresponding verified computing community has later included the revers mode in their analysis and some but not all of the software [8].

Relevance to Optimization

The eminent optimizer Phil Wolfe made the following observation in a TOMS article (Wol82):

> There is a common misconception that calculating a function of n variables and its gradient is about $(n + 1)$ times as expensive as just calculating the function. This will only be true if the gradient is evaluated by differencing function values or by some other emergency procedure. If care is taken in handling quantities, which are common to the function and its derivatives, the ratio is usually 1.5, not $(n + 1)$, whether the quantities are defined explicitly or implicitly, for example, the solutions of differential equations ...

Obviously this *Cheap Gradient Principle* is of central importance for the design of nonlinear optimization algorithms and, therefore, fits very well into this volume. Even now it is generally not well understood that there is no corresponding *Cheap Jacobian Principle*, which one might have hoped to obtain by computing Jacobians row-wise. On the other hand, many of the authors mentioned above noted that Hessian times vector products and other *higher order adjoint vectors* can be obtained roughly with the same complexity as the underlying scalar and vector functions.

The salient consequence of the cheap gradient principle for nonlinear optimization is that calculus-based methods can, in principle, be applied to large-scale problems in thousands and millions of variables. While there are challenges with regards to the memory management and the software implementation, we should not yield to the wide spread engineering practice of optimizing only on reduced order models with derivative free direct search methods. On a theoretical level there has been a lot of activity concerning the use of continuous and discrete adjoints in PDE constrained optimization [1] recently .

If everything is organized correctly, the cheap gradient principle generalizes to what one might call the holy grail of large scale optimization, namely

$$\frac{\text{Cost(Optimization)}}{\text{Cost(Simulation)}} \quad \sim \quad \mathcal{O}(1)$$

By this we mean that the transition from merely simulating a complex system (by evaluating an appropriate numerical model) to optimizing a user specified objective (on the basis of the given model) does not lead to an increase in computational cost by orders of magnitude. Obviously, this is more a rule of thumb than a rigorous mathematical statement.

The selective name-dropping above shows that, especially from 1980 onwards, there have been many developments that cannot possibly be covered in this brief note. Since we do not wish to specifically address electronic circuits or chemical processes we will describe the reverse mode from Seppo Linnainmaa's point of view in the following two sections. In the subsequent sections we discuss temporal and spatial complexity of the reverse mode. In the final section we draw the connection to the adjoint dynamical systems, which go back to Pontryagin.

Round-off Analysis Á la Linnainmaa

Seppo Linnainmaa was neither by training nor in his later professional career primarily a mathematician. In 1967 he enrolled in the first computer science class ever at the University of Helsinki. However, since there were still only very few computer science courses, much of his studies were in mathematics. Optimization was one of the topics, but did not interest him particularly. His supervisor Martti Tienari had worked for Nokia until he became an associate professor of computer science in 1967. The local system was an IBM 1602 and for heavy jobs one had to visit the Northern European Universities Computing

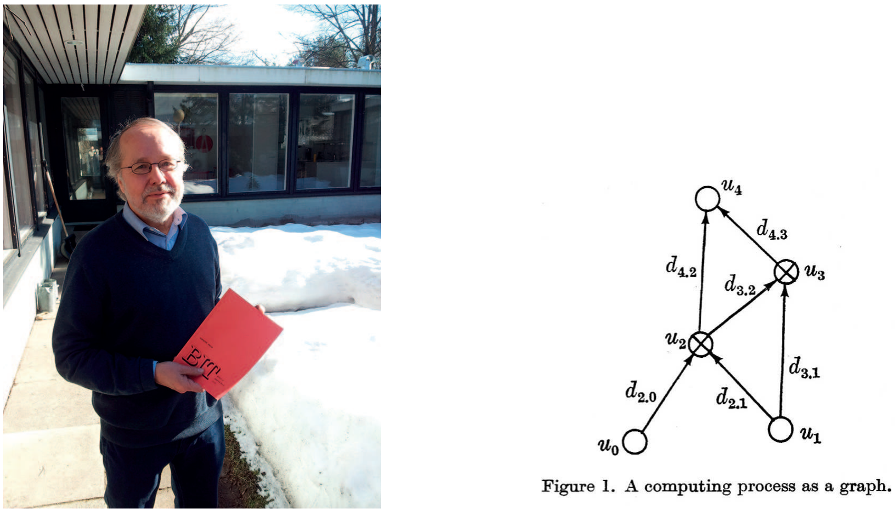

Figure 1

Center at Copenhagen, which had an IBM 7094. All computer manufacture had their own floating point system.

After finishing his Master Thesis concerning the Estimation of Rounding Errors in 1970 he obtained, four years later, the first doctorate ever awarded in computer science at Helsinki University. In 1977 he got a Finnish grant as a visiting scholar with William Kahan at Berkeley, whose group was instrumental in developing the later IEEE Standard 754. Linnainmaa does not think that the results of his thesis had any specific impact on the development of the standard.

Moreover, he did not *market* his approach as a method for cheaply evaluating gradients either, so there was little resonance until I called him up from Argonne in the late eighties. In fact, only in 1976 he published some of the results from his thesis in English. In Figure 1 one sees him holding up a reprint of this BIT paper inside his house in Helsinki in March this year. After continuing his work in numerical analysis he became, a few years later, primarily interested in *artificial intelligence*. Curiously, as he describes it, this meant at that time the simulation and optimization of complex transport systems, so he might have felt at home in todays Matheon application area B. Later on he worked in other areas of artificial intelligence and was a long time employee of the Technical Research Centre of Finland.

His motivation was classical numerical analysis in the sense of floating point arithmetic. On the right-hand side of Figure 1, we took from his BIT paper the interpretation of a simple evaluation process

$$u_2 = \varphi_2(u_0, u_1); \quad u_3 = \varphi_3(u_1, u_2); \quad u_4 = \varphi_4(u_2, u_3);$$

as a computational graph, drawn bottom up. Here the binary functions $\varphi_i()$

for $i = 2, 3, 4$ might be arithmetic operations and the arcs are annotated by the partial derivatives d_{ij}.

More generally, Linnainmaa assumed that the vector function $\tilde{\mathbf{F}} : \mathcal{D} \subset \mathbb{R}^n \to \mathbb{R}^m$ in question is evaluated by a sequence of assignments

$$u_i \;=\; \varphi_i(\mathbf{v}_i) \quad \text{with} \quad \mathbf{v}_i \equiv (u_j)_{j \prec i} \quad \text{for} \quad i = n \ldots l$$

Here the elemental functions φ_i are either binary arithmetic operations or unary intrinsic functions like

$$\varphi_i \in \Phi \;\equiv\; \{\text{rec}, \text{sqrt}, \sin, \cos, \exp, \log, \ldots\}$$

The precedence relation $\prec$ represents direct data dependence and we combine the arguments of φ_i to a vector $\mathbf{v}_i$. Assuming that there are no cyclic dependencies, we may order the variables such that $j \prec i \Rightarrow j < i$. Then we can partition the sequence of scalar variables u_i into the vector triple

$$(\mathbf{x}, \mathbf{z}, \mathbf{y}) \;=\; (u_0, \ldots, u_{n-1}, u_n, \ldots, u_{l-m}, u_{l-m+1}, \ldots, u_l) \;\in\; \mathbb{R}^{n+l}$$

such that $\mathbf{x} \in \mathbb{R}^n$ is the vector of independent variables, $\mathbf{y} \in \mathbb{R}^m$ the vector of dependent variables, and $\mathbf{z} \in \mathbb{R}^{l+1-m-n}$ the (internal) vector of intermediates. In a nonlinear optimization context the components of the vector function F may represent one or several objectives and also the constraints that are more or less active at the current point. In this way one may make maximal use of common subexpressions, which can then also be exploited in derivative evaluations.

In finite precision floating point arithmetic, or due to other inaccuracies, the actual computed values $\tilde{u}_i$ will satisfy a recurrence

$$\tilde{u}_i = \tilde{u}_j \circ \tilde{u}_k + \delta_i \quad \text{or} \quad \tilde{u}_i = \varphi_i(\tilde{u}_j) + \delta_i \quad \text{for} \quad i = n \ldots l$$

Here $\delta \equiv (\delta_i)_{i=0\ldots l} \in \mathbb{R}^{l+1}$ is a vector of hopefully small perturbations. The first n perturbations δ_i are supposed to modify the independents so that $\tilde{u}_{i-1} = x_i + \delta_{i-1}$ for $i = 1 \ldots n$. Now the key question is how the perturbations will effect the final result

$$\tilde{\mathbf{y}} \;\equiv\; (\tilde{u}_i)_{i=l-m+1\ldots l} \;\equiv\; \tilde{\mathbf{F}}(\mathbf{x}, \delta)$$

When the perturbations δ_i vanish we have obviously $\tilde{\mathbf{F}}(\mathbf{x}, 0) = \mathbf{F}(\mathbf{x})$ and, assuming all elemental functions to be differentiable at their respective (exact) arguments, there must be a Taylor expansion

$$\tilde{\mathbf{F}}(\mathbf{x}, \delta) = \mathbf{F}(\mathbf{x}) + \sum_{i=0}^{l} \bar{\mathbf{u}}_i \, \delta_i + o(\|\delta\|)$$

Here the coefficients

$$\bar{\mathbf{u}}_i \;\equiv\; \bar{\mathbf{u}}_i(\mathbf{x}) \in \mathbb{R}^m \;\equiv\; \left.\frac{\partial \mathbf{F}(\mathbf{x}, \delta)}{\partial \delta_i}\right|_{\delta=0}$$

are variously known as *adjoints* or *impacts factors.* They may be thought of as partial derivatives of the end result $\tilde{\mathbf{y}}$ with respect to the intermediates u_i for $i = n \dots l$ and the independents $u_{j-1} = \mathbf{x}_j$ for $j = 1 \dots n$. The latter form clearly the Jacobian

$$\mathbf{F}'(\mathbf{x}) \;\equiv\; \frac{\partial \mathbf{F}(\mathbf{x})}{\partial \mathbf{x}} \;\equiv\; \left(\bar{\mathbf{u}}_{j-1}^\top\right)_{j=1\dots n} \;\in\; \mathbb{R}^{m\times n}$$

Moreover, provided the m dependent variables do not directly depend on each other so that $j \prec i \Rightarrow j \le l-m$, we have $\left(\bar{\mathbf{u}}_{l-m+i}^\top\right)_{i=1\dots m} = I = (\mathbf{e}_i^\top)_{i=1\dots m}$, which is used as initialization in the recursive procedures below.

For discretizations of ODEs or PDEs the perturbations δ_i may also be interpreted as discretization errors. Controlling them in view of the adjoints $\bar{\mathbf{u}}_i$ by mesh adaptions is called the dual weighted residual approach [4]. In that context the $\bar{\mathbf{u}}_i$ are usually computed by solving discretizations of the corresponding adjoint ODE or PDE, which are always linear. Questions of the commutativity of discretization and adjoining or at least consistency to a certain order have been considered by Hager and Walther, for recent developments see [2].

When the perturbations are exclusively produced by rounding and there is no exponent overflow, we may estimate the perturbations by $|\delta_i| \le |\tilde{v}_i|\texttt{eps}$, with `eps` denoting the relative machine precision. Following Linnainmaa we obtain from the triangle inequality the estimates

$$\|\tilde{\mathbf{F}}(\mathbf{x},\delta) - \mathbf{F}(\mathbf{x})\| \;\lesssim\; \sum_{i=0}^{l} \|\bar{\mathbf{u}}_i\||\delta_i| \;\lesssim\; \texttt{eps} \sum_{i=0}^{l} \|\bar{\mathbf{u}}_i\||u_i|$$

where we have replaced $\tilde{u}_i$ by u_i in the last approximate relation. This estimate of the conditioning of the evaluation process was applied to matrix algorithms in (Stu80) and [10]. It was also studied by Iri, whose results can be traced backward from (ITH88). Koichi Kubota [9] developed and implemented a strategy for adaptive multi-precision calculations based on the impact factors $\bar{\mathbf{u}}_i$.

Jacobian accumulation

Now we turn to the aspect of Seppo Linnainmaa's thesis that is most interesting to us, namely the fact that he proposed what is now known as the reverse mode for calculating the adjoint coefficients $\bar{\mathbf{u}}_i$.

Assuming that all elementary functions φ_i are continuously differentiable at the current argument, we denote their partial derivatives by $d_{i\,j} = \partial\varphi_i/\,\partial u_j \in \mathbb{R}$. These scalars $d_{i\,j}$ are directly functions of $\mathbf{u_i}$ and indirectly functions of the vector of independents $\mathbf{x}$.

The partial ordering $\prec$ allows us to interpret the variables u_i as nodes of a directed acyclical graph whose edges can be annotated by the elementary partials $d_{i\,j}$. For the tiny example considered above this so-called Kantorovich graph (see [3]) is depicted on the right-hand side of Figure 1. It is rather important to understand that DAGs are not simply expression trees, but that there may be diamonds and other semi-cycles connecting certain pairs of nodes u_j and u_i. It is intuitively clear that the partial derivative of any dependent variable $\mathbf{y}_i \equiv v_{l-m+i}$ with respect to any independent variable $\mathbf{x}_j \equiv u_{j-1}$ is equal to the sum over all products of partials $d_{i\,j}$ belonging to edge disjoint paths that connect the pair $(\mathbf{x}_j, \mathbf{y}_i)$ in the computational graph. The resulting determinant-like expression is usually called Bauer's formula ([3]). In the tiny example above we obtain the two gradient components

$$\partial u_4/\,\partial u_0 = d_{4\,2}\,d_{2\,0} + d_{4\,3}\,d_{3\,2}\,d_{2\,0}; \quad \partial u_4/\,\partial u_1 = d_{4\,2}\,d_{2\,1} + d_{4\,3}\,d_{3\,2}\,d_{2\,1} + d_{4\,3}\,d_{3\,1}$$

In general, the direct application of Bauer's formula to *accumulate* complete Jacobians involves an effort that is proportional to the length of an explicit algebraic representation of the dependents $\mathbf{y}$ in terms of the independents $\mathbf{x}$. As this effort typically grows exponentially with respect to the depth of the computational graph, one can try to reduce it by identifying common subexpressions, which occur even for our tiny example. Not surprisingly, absolutely minimizing the operations count for Jacobian accumulation is NP hard (Nau06).

However, if the number m of dependents is much smaller than the number n of independents, Jacobians should be accumulated in the reverse mode as already suggested by Linnainmaa. Namely, one can traverse the computational graph backward to compute the adjoint vectors $\bar{\mathbf{u}}_i$ defined above by the recurrence

$$\bar{\mathbf{u}}_j \;=\; \sum_{i \succ j} \bar{\mathbf{u}}_i\, d_{i\,j} \;\in\; \mathbb{R}^m \quad \text{for} \quad j = l - m \dots 0$$

This relation says that the (linearized) impact of the intermediate or independent variable u_j on the end result $\mathbf{y}$ is given by the sum of the impact factors over all successors $\{u_i\}_{i \succ j}$ weighted by the partials $d_{i\,j}$. Note that the $\bar{\mathbf{u}}_j$ are computed backward, starting from the terminal values $\bar{\mathbf{u}}_{l-m+i} = \mathbf{e}_i$ for $i = 1 \dots m$. For the tiny example depicted above, one would compute from $\bar{\mathbf{u}}_4 = 1$ the adjoint intermediates

$$\bar{\mathbf{u}}_3 = 1 \cdot d_{4\,3}; \quad \bar{\mathbf{u}}_2 = 1 \cdot d_{4\,2} + \bar{\mathbf{u}}_3\, d_{3\,2}; \quad \bar{\mathbf{u}}_1 = \bar{\mathbf{u}}_2\, d_{2\,1} + \bar{\mathbf{u}}_3\, d_{3\,1}; \quad \bar{\mathbf{u}}_0 = \bar{\mathbf{u}}_2\, d_{2\,0}$$

Note that there is a substantial reduction in the number of multiplications compared to Bauer's formula above and that the process proceeds backward, i.e., here downward through the computational graph, which was drawn buttom up for the evaluation itself. Since function evaluations are usually defined in terms of predecessor sets $\{j : j \prec i\}$ rather than successor sets $\{i : i \succ j\}$, the accumulation of adjoints is usually performed in the incremental form

$$\bar{\mathbf{v}}_i \mathrel{+}= \; \bar{\mathbf{u}}_i\, \nabla\varphi_i(\mathbf{v}_i) \;\in\; \mathbb{R}^{m \times n_i} \quad \text{for} \quad i = l \dots n$$

where $\nabla\varphi_i(\mathbf{v}_i) \equiv (d_{i\,j})_{j \prec i}$ is a row vector and the matrices of adjoints $\bar{\mathbf{v}}_i \equiv (\bar{\mathbf{u}}_j)_{j \prec i} \in \mathbb{R}^{m \times n_i}$ are assumed to be initialized to zero for $i \leq l - m$. For the tiny example above we obtain the statements

$$\bar{\mathbf{v}}_4 \mathrel{+}= 1 \cdot (d_{4\,2}, d_{4\,3}); \quad \bar{\mathbf{v}}_3 \mathrel{+}= \bar{\mathbf{u}}_3\, (d_{3\,1}, d_{3\,2}); \quad \bar{\mathbf{v}}_2 \mathrel{+}= \bar{\mathbf{u}}_2\, (d_{2\,0}, d_{2\,1})$$

where $\bar{\mathbf{v}}_4 \equiv (\bar{\mathbf{u}}_2, \bar{\mathbf{u}}_3), \bar{\mathbf{v}}_3 \equiv (\bar{\mathbf{u}}_1, \bar{\mathbf{u}}_2)$ and $\bar{\mathbf{v}}_2 \equiv (\bar{\mathbf{u}}_0, \bar{\mathbf{u}}_1)$.

Temporal complexity

The mathematically equivalent incremental form shows very clearly that each elemental function $u_i = \varphi_i(\mathbf{v}_i)$ spawns a corresponding adjoint operation $\bar{\mathbf{v}}_i \mathrel{+}= \bar{\mathbf{u}}_i \nabla\varphi_i(\mathbf{v}_i)$. The cost of this operation scales linearly with respect to m, the number of dependent variables. Hence, for a fixed library Φ there is a common constant ω such that for all i

$$\mathtt{OPS}\{\mathrel{+}= \bar{\mathbf{u}}_i \nabla\varphi_i(\mathbf{v}_i)\} \;\leq\; m\,\omega\, \mathtt{OPS}\{u_i = \varphi_i(\mathbf{v}_i)\}.$$

Here OPS is some temporal measure of computational complexity, for example the classical count of arithmetic operations. This implies for the composite function F and its Jacobian that

$$\mathtt{OPS}\{\mathbf{F}'(\mathbf{x})\} \;\leq\; m\,\omega\, \mathtt{OPS}\{\mathbf{F}(\mathbf{x})\}$$

The constant ω depends on the complexity measure OPS and the computing platform. If one considers only polynomial operations and counts the number of multiplications, the complexity ratio is exactly $\omega = 3$. This is exemplified by the computation of the determinant of a dense symmetric positive matrix via a Cholesky factorization. Then the gradient is the adjugate, a multiple of the transposed inverse, which can be calculated using exactly three times as many multiplications as needed for computing the determinant itself.

The linear dependence on m cannot be avoided in general. To see this, one only has to look at the trivial example $\mathbf{F}(\mathbf{x}) = \mathbf{b}\sin(\mathbf{a}^\top\mathbf{x})$ with constant vectors $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{a} \in \mathbb{R}^n$. Here the operations count for $\mathbf{F}$ is essentially $n + m$ multiplications and for $\mathbf{F}'(\mathbf{x})$ it is clearly $n\,m$ multiplications so that for the multiplicative complexity measure $\mathtt{OPS}\{\mathbf{F}'(\mathbf{x})\} \gtrsim 0.5\, m\, \mathtt{OPS}\{\mathbf{F}(\mathbf{x})\}$ provided $m \leq n$. Hence, the cheap gradient principle does not extend to a cheap Jacobian principle. Note that this observation applies to any conceivable method of computing $\mathbf{F}'(\mathbf{x})$ as an array of $n \times m$ usually distinct numbers.

The memory issue

For general $\mathbf{F}$ the actual runtime ratio between Jacobians and functions may be significantly larger due to various overheads. In particular, it has been well known since Benett [5] that executing the reverse loop in either incremental or nonincremental form requires the recuperation of the intermediate values u_i in the opposite order to that in which they were generated initially by the forward evaluation loop. The simplest way is to simply store all the intermediate values onto a large stack, which is accessed strictly in a first-in last-out fashion. Speelpenning [12] depicted the sequential storage of all intermediate operations as shown in Figure 2. This picture quite closely reflects the storage in other AD-tools such as ADOL-C.

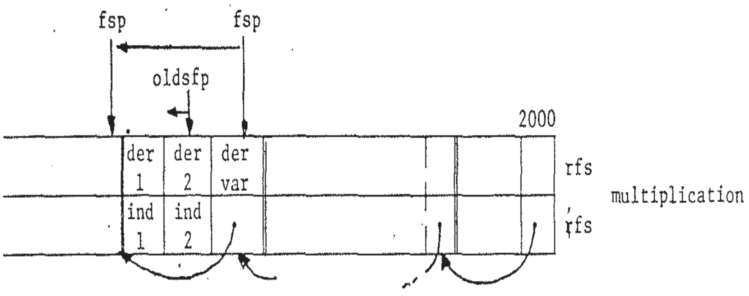

Figure 2

Since we have to store some information for every single operation performed, we obtain the spatial complexity

$$\mathtt{MEM}\{\mathbf{F}'(\mathbf{x})\} \sim \mathtt{OPS}\{\mathbf{F}(\mathbf{x})\} \gtrsim \mathtt{MEM}\{\mathbf{F}(\mathbf{x})\}$$

Note that this memory estimate applies to the vector and scalar cases $m > 1$ and $m = 1$ alike. Hence, from a memory point of view it is advantageous to propagate several adjoints simultaneously backward, for example in an optimization calculation with a handful of active constraints.

Originally, the memory usage was a big concern because memory size was severely limited. Today the issue is more the delay caused by large data movements from and to external storage devices, whose size seems almost unlimited. As already suggested by Benett and Ostrowski et al. the memory can be reduced by orders of magnitude through an appropriate compromise between storage and recomputation of intermediates, described as checkpointing in [7]. One possibility in a range of trade-offs is to realize a logarithmic increase for both spatial and temporal complexity

$$\frac{\mathtt{MEM}\{\mathbf{F}'(\mathbf{x})\}}{\mathtt{MEM}\{\mathbf{F}(\mathbf{x})\}} \sim \log(\mathtt{OPS}\{\mathbf{F}(\mathbf{x})\}) \sim \frac{\mathtt{OPS}\{\mathbf{F}'(\mathbf{x})\}}{\mathtt{OPS}\{\mathbf{F}(\mathbf{x})\}m}$$

Gradients and adjoint dynamics

Disregarding the storage issue we obtain, for the basic reverse mode for the scalar case $m = 1$ with $f(\mathbf{x}) = \mathbf{F}(\mathbf{x})$, the striking result that

$$\mathtt{OPS}\{\nabla f(\mathbf{x})\} \leq \omega\, \mathtt{OPS}\{f(\mathbf{x})\}$$

In other words, as Wolfe observed, gradients can 'always' be computed at a small multiple of the cost of computing the underlying function, irrespective of n the number of independent variables, which may be huge. Since $m = 1$, we may also interpret the scalars $\bar{\mathbf{u}}_i$ as Lagrange multipliers of the defining relations $u_i - \varphi_i(\mathbf{v}_i) = 0$ with respect to the single dependent $\mathbf{y} = u_l$ viewed as objective function. This interpretation was used amongst others by the oceanographer Thacker in (Tha91). It might be used to identify critical and calm parts of an evaluation process, possibly suggesting certain simplifications, e.g., the local coarsening of meshes.

As discussed in the prologue, the cheapness of gradients is of great importance for nonlinear optimization, but still not widely understood, except in the time dependent context. There we may have, on the unit time interval $0 \leq t \leq 1$, the primal dual pair of evolutions

$$\begin{aligned} \dot{\mathbf{u}}(t) \equiv \partial \mathbf{u}(t)/\partial t &= \mathbf{F}(\mathbf{u}(t)) && \text{with} \quad \mathbf{u}(0) = \mathbf{x}, \\ \dot{\bar{\mathbf{u}}}(t) \equiv \partial \bar{\mathbf{u}}(t)/\partial t &= \mathbf{F}'(\mathbf{u}(t))^\top \bar{\mathbf{u}}(t) && \text{with} \quad \bar{\mathbf{u}}(1) = \nabla f(\mathbf{u}(1)) \end{aligned}$$

Here the state $\mathbf{u}$ belongs to some Euclidean or Banach space and $\bar{\mathbf{u}}$ to its topological dual. Correspondingly, the right-hand side $\mathbf{F}(\mathbf{u})$ and its dual $\mathbf{F}'(\mathbf{u})^\top \bar{\mathbf{u}}$ may be strictly algebraic or involve differential operators.

Then it has been well understood since Pontryagin that the gradient of a function $y = f(\mathbf{u}(1))$ with respect to the initial point $\mathbf{x}$ is given by $\bar{\mathbf{u}}(0)$. It can be computed at maximally $\omega = 2$ times the computational effort of the forward calculation of $\mathbf{u}(t)$ by additionally integrating the second, linear evolution equation backward. In the simplest mode without checkpointing this requires the storage of the full trajectory $\mathbf{u}(t)$, unless the right-hand side $\mathbf{F}$ is largely linear. Also for each t the adjoint states $\bar{\mathbf{u}}(t)$ represent the sensitivity of the final value $y = f$ with respect to perturbations of the primal state $\mathbf{u}(t)$. Of course, the same observations apply to appropriate discretizations, which implies again the proportionality between the operations count of the forward sweep and memory need of the reverse sweep for the gradient calculation. To avoid the full trajectory storage one may keep only selected checkpoints during the forward sweep as mentioned above and then recuperate the primal trajectory in pieces on the way back, when the primal states are actually needed.

In some sense the reverse mode is just a discrete analogue of the extremum principle going back to Pontryagin. Naturally, the discretizations of dynamical systems have more structure than our general evaluation loop described on page 4, but the key characteristics of the reverse mode are the same.

Summary and outlook

The author would have hoped that the cheap gradient principle and other implications of the reverse mode regarding the complexity of derivative calculations were more widely understood and appreciated. However, as far as smooth optimization is concerned most algorithm designers have always assumed that gradients are available, notwithstanding a very substantial effort in derivative-free optimization over the last couple of decades.

Now, within modeling environments such as AMPL and GAMS, even second derivatives are conveniently available, though one hears occasionally complaints about rather significant runtime costs. That is no surprise since we have seen that without sparsity, complete Jacobians and Hessians may be an order of magnitude more expensive than functions and gradients, and otherwise, one finds that the evaluation of sparse derivatives may entail a significant interpretative overhead.

Further progress on the reverse mode can be expected mainly from the development of an adjoint calculus in suitable functional analytical settings. So far there seems to be little prospect of a generalization to nonsmooth problems in a finite dimensional setting. The capability to quantify the rounding error propagation and thus measure the conditioning of numerical algorithms, which played a central role in the evolution of the reverse mode, awaits further application. In contrast, checkpointing or windowing as it is sometimes called in the PDE community, is being used more and more to make the reverse mode applicable to really large problems.

References

[1] Constrained optimization and optimal control for partial differential equations. In G. Leugering, S. Engell, A. Griewank, M. Hinze, R. Rannacher, V. Schulz, M. Ulbrich, and St. Ulbrich, editors, *International Series of Numerical Mathematics*, pages 99–122. Springer, Basel, Dordrecht Heidelberg London New York, 2012.

[2] Mihai Alexe and Adrian Sandu. On the discrete adjoints of adaptive time stepping algorithms. *Journal of Computational and Applied Mathematics*, 233(4):1005–1020, 2009.

[3] Friedrich L. Bauer. Computational graphs and rounding errors. *SIAM J. Numer. Anal.*, 11(1):87–96, 1974.

[4] R. Becker and R. Rannacher. An optimal control approach to error control and mesh adaptation in finite element methods. *Acta Numerica 2001*, pages 1–102, 2001.

[5] C. H. Bennett. Logical Reversability of Computation. *IBM Journal of Research and Development*, 17:525–532, 1973.

[6] F.G. Gustavson G.D. Hachtel, R.K. Brayton. The sparse tableau approach to network design and analysis. *IEEE Transactions of Circuit Theory*, 18(1):102 – 113, 1971.

[7] A. Griewank and A. Walther. *Principles and Techniques of Algorithmic Differentiation, Second Edition.* SIAM, 2008.

[8] Ralph Baker Kearfott. GlobSol user guide. *Optimization Methods and Software*, 24(4–5):687–708, August 2009.

[9] Koichi Kubota. PADRE2 – Fortran precompiler for automatic differentiation and estimates of rounding error. In Martin Berz, Christian Bischof,

George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 367–374. SIAM, Philadelphia, Penn., 1996.

[10] Webb Miller and Cella Wrathall. *Software for Roundoff Analysis of Matrix Algorithms.* Academic Press, 1980.

[11] U. Naumann. Optimal Jacobian accumulation is NP-complete. *Math. Prog.*, 112:427–441, 2006.

[12] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, Ill., January 1980.

[13] Nick Trefethen. *Who invented the greatest numerical algorithms*, 2005. www.comlab.ox.ac.uk/nick.trefethen.

Andreas Griewank
Institut für Mathematik
Humboldt Universität zu Berlin
Unter den Linden 6
10099 Berlin
Germany
griewank@mathematik.hu-berlin.de