Acta Numerica (2003), pp. 321–398
DOI: 10.1017/S0962492902000132

A mathematical view of automatic differentiation

Andreas Griewank
Institute of Scientific Computing,
Department of Mathematics,
Technische Universität Dresden,
01052 Dresden, Germany
E-mail: griewank@math.tu-dresden.de

Automatic, or algorithmic, differentiation addresses the need for the accurate and efficient calculation of derivative values in scientific computing. To this end procedural programs for the evaluation of problem-specific functions are transformed into programs that also compute the required derivative values at the same numerical arguments in floating point arithmetic. Disregarding many important implementation issues, we examine in this article complexity bounds and other more mathematical aspects of the program transformation task sketched above.

CONTENTS

1. Introduction

Practically all calculus-based numerical methods for nonlinear computations are based on truncated Taylor expansions of problem-specific functions. Naturally we have to exempt from this blanket assertion those methods that are targeted for models defined by functions that are very rough, or even nondeterministic, as is the case for functions that can only be measured experimentally, or evaluated by Monte Carlo simulations. However, we should bear in mind that, in many other cases, the roughness and nondeterminacy may only be an artifact of the particular way in which the function is evaluated. Then a reasonably accurate evaluation of derivatives may well be possible using variants of the methodology described here. Consequently, we may view the subject of this article as an effort to extend the applicability of classical numerical methodology, dear to our heart, to the complex, large-scale models arising today in many fields of science and engineering. Sometimes, numerical analysts are content to verify the efficacy of their sophisticated methods on suites of academic test problems. These may have a very large number of variables and exhibit other complications, but they may still be comparatively easy to manipulate, especially as far as the calculation of derivatives is concerned. Potential users with more complex models may then give up on calculus-based models and resort to rather crude methods, possibly after drastically reducing the number of free variables in order to avoid the prohibitive runtimes resulting from excessive numbers of model reruns.

Not surprisingly, on the kind of real-life model indicated above, nothing can be achieved in an entirely automatic fashion. Therefore, the author much prefers the term 'algorithmic differentiation' and will refer to the subject from here on using the common acronym AD. Beyond this minor labelling issue lies the much more serious task of deciding which research and development activities ought to be described in a fair but focused survey on AD. To some, AD is an exercise in software development, about which they want to read (or write) nothing but a crisp online documentation. For others, AD has become the mainstay of their research activity with plenty of intriguing questions to resolve. As in all interdisciplinary endeavours, there are many close connections to other fields, especially numerical linear algebra, computer algebra, and compiler writing. It would be preposterous as well as imprudent to stake out an exclusive claim for any particular subject area or theoretical result. There has been a series of workshops and conferences focused on AD and its applications (Griewank and Corliss 1991, Berz, Bischof, Corliss and Griewank 1996, Corliss, Faure, Griewank, Hascoët and Naumann 2001), but nobody has seen the need for a dedicated journal. Results appear mostly in journals on optimization, or more generally, numerical analysis and scientific computing.

1.1. How numeric and how symbolic is AD?

There is a certain dichotomy in that the final results of AD are numerical derivative values, which are usually thought to belong to the domain of nonlinear analysis and optimization. On the other hand, the development of AD techniques and tools can for the most part stay blissfully oblivious to issues of floating point arithmetic, much like sparse matrix factorization in the positive definite case. Instead, we need to analyse and manipulate discrete objects like computer codes and related graphs, sometimes employing tools of combinatorial optimization. There are no division operations at all, and questions of numerical stability or convergence orders and rates arise mostly for dynamic codes that represent iterative solvers or adaptive discretizations.

The last assertion flies in the face of the notion that differentiation is an ill-conditioned process, which is firmly ingrained in the minds of numerical mathematicians. Whether or not this conviction is appropriate depends on the way in which the functions to be differentiated are provided by the *user*. If we merely have an oracle generating function values with prescribed accuracy, derivatives can indeed only be estimated quite inaccurately as divided differences; possibly averaged over a number of trial evaluations in situations where errors can be assumed to have a zero mean statistically. If, on the other hand, the *oracle* takes the form of a computer code for evaluating the function, then this code can often be analysed and transformed to yield an extended code that also evaluates desired derivatives. In both scenarios we have backward stability *à la* Wilkinson, in that the floating point values obtained can be interpreted as the exact derivatives of a 'slightly' perturbed problem with the same structure. The crucial difference is that in the first scenario there is not much structure to be preserved, as the functions may be subjected to discontinuous perturbations, generating slope changes of arbitrary size. In the second scenario the function is effectively prescribed as the composite of elementary functions, whose values and derivative are only subject to variations on the order of the machine accuracy.

More abstractly, we may simply state that, as a consequence of the chain rule, the composition operation

$$F \equiv F_2 \circ F_1 \quad \text{for} \quad F_i : \Omega_{i-1} \mapsto \Omega_i$$

is a jointly continuous mapping between Banach spaces of differentiable functions, *i.e.*, it belongs to

$$\mathcal{C}^1(\Omega_0, \Omega_1) \times \mathcal{C}^1(\Omega_1, \Omega_2) \mapsto \mathcal{C}^1(\Omega_0, \Omega_2).$$

In practical terms this means that we differentiate the composite function F by appropriately combining the derivatives of the two constituents F_1 and F_2, assuming that procedures for their evaluation are already in place. By doing this recursively, we effectively extend an original procedure for evaluating function values by themselves into one that also evaluates derivatives. This is essentially an algebraic manipulation, which again reflects the two-sided nature of AD, partly symbolic and partly numeric.

The term *numeric differentiation* is widely used to describe the common practice of approximating derivatives by divided differences (Anderssen and Bloomfield 1974). This approximation of differential quotients by difference quotients means geometrically the replacement of tangent slopes by secant slopes over a certain increment in each independent variable. It is well known that, even for the increment size that optimally balances truncation and rounding error, half of the significant digits are lost. Of course, the optimal increment typically differs for each variable/function pair and the situation is still more serious when it comes to approximating higher derivatives. No such difficulty occurs in AD, as no parameter needs to be selected and there is no significant loss of accuracy at all. Of course, the application of the chain rule does entail multiplications and additions of floating point numbers, which can only be performed with platform-dependent finite precision.

Since AD has really very little in common with divided differences, it is rather inappropriate to describe it as a 'halfway house' between numeric and symbolic differentiation, as is occasionally done in the literature. What then is the key feature that distinguishes AD from *fully symbolic differentiation*, as performed by most computer algebra (CA) systems? A short answer would be to say that AD applies the chain rule to floating point numbers rather than algebraic expressions. Indeed, no AD package yields tidy mathematical formulas for the derivatives of functions, no matter how algebraically simple they may be. All we get is more code, typically source or binary. It is usually not nice to look at and, as such, it never provides any analytical insight into the nature of the function and its derivatives. Of course, for multi-layered models, such insight by inspection is generally impossible anyway, as the direct expression of function values in terms of the independent variables leads to formulas that are already impenetrably complex. Instead, the aim in AD is the accurate evaluation of derivative values at a sequence of arguments, with an *a priori* bounded complexity in terms of the operation count, memory accesses, and memory size.

1.2. Various phases and costs of AD

Relative to the complexity of the function itself, the complexity growth is at worst linear in the number of independent or dependent variables and quadratic in the degree of the derivatives required. This moderate and predictable extra cost, combined with the achievement of good derivative accuracy and the absence of any free method parameter, has led to the wide-spread use of AD as built-in functionality in AMPL, GAMS, NEOS, and other modelling or optimization systems. In contrast, the complexity of symbolic manipulations typically grows exponentially with the depth of the expression tree or the computational graph that represent a function evaluation procedure. By making such blanket statements we are in danger of comparing apples to oranges.

In computing derivatives using either CA or AD we have to distinguish at least two distinct phases and their respective costs: first, a *symbolic phase* in which the given function specification is analysed and a procedure for evaluating the desired derivatives is prepared in a suitable fashion; second, a *numeric phase* where this evaluation is actually carried out at one or several numerical arguments. The more effort we invest in the symbolic phase the more runtime efficiency we can expect in the numeric phase, which will hopefully be applied sufficiently often to amortize the initial symbolic investment. For example, extensive symbolic preprocessing of a routine specifying a certain kind of finite element or the right-hand side of a stiff ODE is likely to pay off if the same routine and its derivative procedure are later called in the numeric phase at thousands of grid points or time-steps, respectively. On the other hand, investing much effort in the symbolic phase may not pay off, or simply be impossible, if the control flow of the original routine is very dynamic. For example, one may think of a recursive quadrature routine that adaptively subdivides the domains depending on certain error estimates that can vary dramatically from call to call. Then very little control flow information can be gleaned at compile-time and code optimization is nearly impossible.

Of course, we can easily conceive of a multi-phase scenario where the function specification is specialized at various levels and the corresponding derivative procedures are successively refined accordingly. For example, we may have an intermediate *qualitative phase* where, given the specification of certain parameters like mesh sizes and vector dimensions, the AD tool determines the sparsity pattern of a Jacobian matrix and correspondingly allocates suitable data structures in preparation for the subsequent numeric phase. Naturally, it may also output the sparsity pattern or provide other qualitative dependence information, such as the maximal rank of the Jacobian or even the algebraic multiplicity of certain eigenvalues.

Traditionally in AD the symbolic effort has been kept to the equivalent of a few compiler-type passes through the function specification, possibly including a final compilation by a standard compiler. Hence the symbolic effort is at most of the order of a single function evaluation, albeit probably with a rather large constant. However, this characteristic of all current AD tools may change when more extensive dependence analyses or combinatorial optimizations of the way in which the chain rule is applied are incorporated. For the most part these considerable extra efforts in the symbolic phase will reduce the resulting costs in the numerical phase only by constants but not by orders of magnitude. Some of them are rather mundane rearrangements or software modifications with no allure for the mathematician. Of course, such improvements can make all the difference for the practical feasibility of certain calculations.

This applies in particular to the calculation of gradients in the reverse mode of AD. Being a discrete analogue of adjoints in ODEs, this backward application of the chain rule yields all partials of a scalar function with respect to an arbitrary number of variables at the cost of a small multiple of the operation count for evaluating the function itself. If only multiplications are counted, this growth factor is at most 3, and for a typical evaluation code probably more like 2 on average. That kind of value seems reasonable to people who skilfully write adjoint code by hand (Zou, Vandenberghe, Pondeca and Kuo 1997), and it opens up the possibility of turning simulation codes into optimization codes with a growth of the total runtime by a factor in the tens rather than the hundreds. Largely because of memory effects, it is not uncommon that the runtime ratio for a single gradient achieved by current AD tools is of the order 10 rather than 2. Fortunately, the memory-induced cost penalty is the same if a handful of gradients, forming the Jacobian of a vector function, is evaluated simultaneously in what is called the *vector-reverse mode*.

In general, it is very important that the user or algorithm designer correctly identifies which derivative information is actually needed at any particular point in time, and then gets the AD tool to generate all of it jointly. In this way optimal use can be made of common subcalculations or memory accesses. The latter may well determine the wall clock time on a modern computer. Moreover, the term *derivative information* needs to be interpreted in a wider sense, not just denoting vectors, matrices or even tensors of partial derivatives, as is usually understood in hand-written formulas or the input to computer algebra systems. Such rectangular derivative arrays can often be contracted by certain weighting and direction vectors to yield derivative objects with fewer components. By building this contraction into the AD process all aspects of the computational costs can usually be significantly reduced. A typical scenario is the iterative calculation of approximate Newton steps, where only a sequence of Jacobian–vector, and possibly vector–Jacobian, products are needed, but never the Jacobian as such. In fact, as we will discuss in Section 6, it is not really clear what the 'Jacobian as such' really is. Instead we may have to consider various representations depending on the ultimate purpose of our numerical calculation. Even if this ultimate purpose is to compute exact Newton steps by a finite procedure, first calculating all Jacobian entries may not be a good idea, irrespective of whether it is sparse or not. In some sense Newton steps themselves become *derivative objects*. While that may seem a conceptual stretch, there are some other mathematical objects, such as Lipschitz constants, error estimates, and interval enclosures, which are naturally related to derivatives and can be evaluated by techniques familiar from AD.

1.3. Some historical remarks

Historically, there has been a particularly close connection between the reverse mode of AD and efforts to estimate the effects of rounding errors in evaluating functions or performing certain numerical algorithms. The adjoint quantities generated in the reverse mode for each intermediate variable of an evaluation process are simply the sensitivities of the weighted output with respect to perturbations of this particular variable. Hence we obtain the first-order Taylor expansion of the output error with respect to all intermediate errors, a linearized estimate apparently first published by Linnainmaa in 1972 (see Linnainmaa (1976)). Even earlier, in 1966, Volin and Ostrovski (see G. M. Ostrovskii and Borisov (1971)) suggested the reverse mode for the optimization of certain process models in chemical engineering. Interestingly, the first major publication dedicated to AD, namely the seminal book by Louis Rall (1983) did not recognize the reverse, or adjoint, mode as a variant of AD at all, but instead covered exclusively the forward or direct mode. This straightforward application of the chain rule goes back much further, at least to the early 1950s, when it was realized that computers could perform symbolic as well as numerical manipulations. In the past decade there has been growing interest in combinations of the forward and reverse mode, a wide range of possibilities, which had in principle already been recognized by Ostrovski *et al.* (see Volin and Ostrovskii (1985)).

Based on theoretical work by Cacuci, Weber, Oblow and Marable (1980), the first general-purpose tool implementing the forward and reverse mode was developed in the 1980s by Ed Oblow, Brian Worley and Jim Horwedel at Oak Ridge National Laboratory. Their system GRESS/ADGEN was successfully applied to many large scientific and industrial codes, especially from nuclear engineering (Griewank and Corliss 1991). Hence there was an early proof of concept, which did, however, have next to no impact in the numerical analysis community, where the notion that derivatives are always hard if not impossible to come by persisted for a long time, possibly to this day. Around the time of the first workshop on AD at Breckenridge in 1991, several system projects were started (*e.g.*, ADIFOR, Odyssée, TAMC, ADOL-C), though regrettably none with sufficient resources for the development of a truly professional tool. Currently several projects are under way, some to cover MATLAB, and others that aim at an even closer integration into a compilation platform for several source languages (Hague and Naumann 2001).

As suggested by the title, in this article we will concentrate on the mathematical questions of AD and leave the implementation issues largely aside. Following current practice in numerical linear algebra, we will measure computational complexity by counting fused multiply-adds and denote them simply by OPS. They can be performed in one or two cycles on modern super-scalar processors, and are essentially the only extra arithmetic operations introduced by AD. Furthermore, in Section 4 on reversal schedules, we will also emphasize the maximal memory requirement and the total number of memory accesses. They may dominate the runtime even though most of them occur in a strictly sequential fashion, thus causing only a minimal number of cache misses.

1.4. Structure of the article

The paper is organized as follows. In Section 2 we set up a framework of function evaluation procedures that is invariant with respect to adjoining, *i.e.*, application of reverse mode AD. To achieve this we consider from the beginning vector-valued and incremental elemental functions, whereas in Griewank (2000) and most other presentations of AD only scalar-valued assignments are discussed. These generalizations make the notation in some respects a little more complicated, and will therefore be suspended in the second part of Section 6. That final section discusses the rather intricate relationship between Jacobian matrices and computational graphs. It is in part speculative and meant to suggest directions of future research in AD. Section 3 reviews the basic and well-established techniques of AD, with methods for the evaluation of sparse Jacobians being merely sketched verbally. For further details on this and other aspects one may consult the author's book (Griewank 2000).

The following three sections treat rather different topics and can be read separately from each other. In many people's minds the main objection to the reverse mode in its basic form is its potentially very large demand for temporary memory. This may either exceed available storage or make the execution painfully slow owing to extensive data transfers to remote regions of the memory hierarchy. Therefore, in Section 4 we have elaborated checkpointing techniques for program reversals on serial and parallel machines in considerable detail. The emphasis is on the discrete optimization problem of checkpoint placement rather than their implementation from a computer science perspective. In Section 5 we will consider adjoints of iterative processes for the solution of linear or nonlinear state equations. In that case loop reversals can be avoided altogether using techniques that were also considered in Hascoët, Fidanova and Held (2001), Giles and Süli (2002) and Becker and Rannacher (2001).

Apart from second-order adjoints, which are covered in Sections 3 and 5, we will not discuss the evaluation of higher derivatives even though there are some interesting recent applications to DAEs and ODEs (see Pantelides (1988), Pryce (1998) and Röbenack and Reinschke (2000)). Explicit formulas for the higher derivatives of composite functions were published by Faa di Bruno (1856). Recursive formulas for the forward propagation of Taylor series with respect to a single variable were published by Moore (1979) and have been reproduced many times since: see, for instance, Kedem (1980), Rall (1983), Lohner (1992) and Griewank (2000). The key observation is that, because all elemental functions of interest, *e.g.*, $\exp(x)$, $\sin(x)$, *etc.*, are solutions of linear ODEs, univariate Taylor series can be propagated with a complexity that grows only quadratically with the degree d of the highest coefficient. Using FFT or other fast convolution algorithms we can reduce this complexity to order $d\ln(1+d)$, a possibility that has not been exploited in practice. Following a suggestion by Rall, it was shown in Bischof, Corliss and Griewank (1993), Griewank, Utke and Walther (2000) and Neidinger (200x) that multivariate Taylor expansions can be computed rather efficiently using families of univariate polynomials. Similarly, the reverse propagation of Taylor polynomials does not introduce any really new aspects, and can be interpreted as performing the usual reverse mode in Taylor series arithmetic (Christianson 1992, Griewank 2000).

Like the kind of material presented, the style and depth of treatment in this article is rather heterogeneous. Many observations are just asserted verbally with references to the literature but others are formalized as lemmas or propositions. Proofs have been omitted, except for a new, shorter demonstration that binomial reversal schedules are optimal in Proposition 4.2, the proof of the folklore result, Proposition 6.3, concerning the generic rank of a Jacobian, and some simple observations regarding the new concept of Jacobian scarcity introduced in Section 6. For a more detailed exposition of the basic material the reader should consult the author's book (Griewank 2000), and for more recent results and application studies the proceedings volume edited by Corliss *et al.* (2001). A rather comprehensive bibliography on AD is maintained by Corliss (1991).

2. Evaluation procedures in incremental form

Rather than considering functions 'merely' as mathematical mappings we have to specify them by evaluation procedures and to a large extent identify the two. These conceptual codes are abstractions of actual computer codes, as they may be written in Fortran or C and their various extensions. Generally, the quality and complexity of the derived procedures for evaluating derivatives will reflect the properties of the underlying function evaluation procedure quite closely. The following formalism does not make any really substantial assumptions other than that the procedure represents a finite algorithm without branching of the control flow. We just have to be a little careful concerning the handling of intermediate quantities.

The basic assumption of AD is that the function to be differentiated is, at least conceptually, evaluated by a sequence of *elemental* statements, such as

$$v_i = \varphi_i(u_i) \quad \text{or} \quad v_i \mathrel{+}= \varphi_i(u_i) \quad \text{for all} \quad i \in \mathcal{I}. \tag{2.1}$$

In other words we have the evaluation of an elemental function φ_i followed by a standard assignment or an additive incrementation. Here the C-style notation $a \mathrel{+}= b$ abbreviates $a = a+b$ for any two compatible vectors a and b. We include incremental assignments and, later, allow overlap between the left-hand sides v_i, because both aspects occur frequently in adjoint programs. To obtain a unified notation for both kinds of assignment we introduce for each index $i \in \mathcal{I}$ a *flag* $\sigma_i \in \{0, 1\}$, and write

$$v_i = \sigma_i v_i + \varphi_i(u_i) \quad \text{for all} \quad i \in \mathcal{I}. \tag{2.2}$$

To make the adjoint procedure even more symmetric we may prefer the equivalent statement pair

$$\big[v_i \mathrel{*}= \sigma_i;\ v_i \mathrel{+}= \varphi_i(u_i)\big] \quad \text{for all} \quad i \in \mathcal{I}, \tag{2.3}$$

where $a \mathrel{*}= \alpha$ abbreviates $a = \alpha * a$ for any vector a and scalar α. In this way we have a fully incremental form, which turns out to be convenient for adjoining later on. Throughout we will neglect the cost of the additive and multiplicative incrementations. Normally the φ_i will be taken from a library of arithmetic operations and intrinsic functions. However, we may also include basic linear algebra subroutines (BLAS) and other library or user-defined functions, provided corresponding derivative procedures $\dot{\varphi}_i$ and $\bar{\varphi}_i$, as defined in Section 3, can be supplied.

The index set $\mathcal{I}$ is assumed to be partially ordered by an acyclic, nonreflexive precedence relation $j \prec i$, which indicates that φ_j must be applied before φ_i. Deviating from the more mathematical AD literature, including Griewank (2000), we will identify the indices $i \in \mathcal{I}$ with the whole statement (2.2) rather than just the output variable v_i. This approach is more in line with compiler literature (Hascoët 2001). The difference disappears whenever we exclude incremental statements. We may interpret $\mathcal{I}$ as the vertex set of the directed acyclic graph $\mathcal{G} = (\mathcal{I}, \mathcal{E})$ with edge set $\mathcal{E} \equiv \{(j, i) \in \mathcal{I} \times \mathcal{I} \big| \ j \prec i\}$. When the vertices of $\mathcal{G}$ are annotated with the elemental functions φ_i, it is often called the *computational graph*, a concept apparently due to Kantorovich (1957). As a small example we consider the scalar function

$$y = \exp(x_1) * \sin(x_1 + x_2). \tag{2.4}$$

It can be evaluated using 3 intermediates by the program listed in Figure 2.1; on the right is displayed the corresponding computational graph.

Program:

$v_3 = \exp(v_1)$

$v_4 = \sin(v_1 + v_2)$

$v_5 = v_3 * v_4$

Graph:

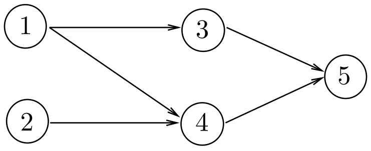

Figure 2.1. Program and graph associated with (2.4).

Without loss of generality, we may assume that $\mathcal{I}$ equals the range of integers $\mathcal{I} = [1 \ldots l]$, and consider (2.2) as a loop representing the succession of statements executed by a computer program for a given set of data.

In particular we consider the control flow to be fixed. The partial ordering $\prec$ of $\mathcal{I}$ allows us to discuss the runtime ratios between the derived and the original evaluation procedures on parallel as well as serial machines. We will relax the usual convention that the variables v_i are real scalars, and instead allow them to be elements of an arbitrary real Hilbert space $\mathcal{H}_i$ of dimension $m_i \equiv \dim(\mathcal{H}_i)$. We will refer to the *scalar case* whenever $m_i = 1$ for all $i = 1 \ldots l$.

2.1. The state space and its transformations

In real computer programs several variables are often stored in the same location. Hence we do not require the $\mathcal{H}_i$ to be completely separate but to have a natural embedding into a common *state space* $\mathcal{H}$, such as

$$P_i \, \mathcal{H}_i \subset \mathcal{H} \quad \text{and} \quad v_i = P_i^T \, \mathbf{v} \quad \text{for} \quad \mathbf{v} \in \mathcal{H},$$

where P_i is orthogonal, in that $P_i^T P_i = I$ is the identity on $\mathcal{H}_i$. Here and throughout we will identify linear mappings between finite-dimensional spaces with their matrix description in terms of suitable orthonormal bases. Correspondingly we will denote the adjoint mapping of P as the transpose P^T. We will choose $\mathcal{H}$ minimal such that

$$\mathcal{H} \equiv P_1 \, \mathcal{H}_1 + P_2 \mathcal{H}_2 + \cdots + P_{l-1} \, \mathcal{H}_{l-1} + P_l \, \mathcal{H}_l.$$

In the simplest scenario $\mathcal{H}$ is the Cartesian product of all $P_i \, \mathcal{H}_i$, in which case we have a so-called *single assignment procedure*. Otherwise the variables v_i are said to *overlap*, and we have to specify more carefully what (2.2) and thus (2.3) actually mean. Namely, we assume that there is an underlying state vector $\mathbf{v} \in \mathcal{H}$, which is transformed according to

$$\Phi_i(\mathbf{v}) \equiv \big[I - (1 - \sigma_i) \, P_i \, P_i^T\big] \mathbf{v} + P_i \, \varphi_i(Q_i \, \mathbf{v}). \tag{2.5}$$

Here the *argument selections* Q_i denote orthogonal projections from $\mathcal{H}$ into the domains $\mathcal{D}_i$ of the elemental functions φ_i so that

$$u_i = Q_i \, \mathbf{v} \in \mathcal{D}_i = \mathrm{dom}(\varphi_i) \quad \text{with} \quad n_i = \dim(\mathcal{D}_i). \tag{2.6}$$

Rather than restricting ourselves to Cartesian projections that pick out coordinate components of $\mathbf{v}$, we allow general linear Q_i, and thus include the important concept of group partial separability (Conn, Gould and Toint 1992) in a natural way. Of course, the φ_i need not be well defined on the whole Euclidean space $\mathcal{D}_i$ in practice, but we will assume here that this difficulty does not occur at arguments of interest. For basic observations regarding exceptional arguments consider Sections 11.2 and 11.3 in Griewank (2000).

To distinguish them verbally from the underlying *elemental functions* φ_i we will refer to the Φ_i as *elemental transitions.* Given any initial state $\mathbf{u} \in \mathcal{H}$ we can now apply the Φ_i in any specified order and obtain a corresponding final state $\mathbf{v} \equiv \Phi(\mathbf{u}) \in \mathcal{H}$ where $\Phi \equiv \Phi_l \circ \Phi_{l-1} \circ \cdots \circ \Phi_1$ denotes the composition of the elemental transitions.

For example (2.4) we may set

$$\mathcal{H} = \mathbb{R}^5, \quad \mathcal{H}_i = \mathbb{R} \quad \text{and} \quad P_i = \mathbf{e}_i \quad \text{for} \quad i = 1 \ldots 5.$$

Here $\mathbf{e}_i \in \mathbb{R}^5$ denotes the ith Cartesian basis vector. Furthermore we select

$$Q_3 = (1\,0\,0\,0\,0), \quad Q_4 = (1\,1\,0\,0\,0), \quad Q_5 = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

and may thus evaluate (2.4) by the transformations

$$\begin{aligned}
\mathbf{v}^{(0)} &= 0,\\
\mathbf{v}^{(1)} &= \big(I - \mathbf{e}_1\,\mathbf{e}_1^T\big)\mathbf{v}^{(0)} + \mathbf{e}_1\,x_1,\\
\mathbf{v}^{(2)} &= \big(I - \mathbf{e}_2\,\mathbf{e}_2^T\big)\mathbf{v}^{(1)} + \mathbf{e}_2\,x_2,\\
\mathbf{v}^{(3)} &= \big(I - \mathbf{e}_3\,\mathbf{e}_3^T\big)\mathbf{v}^{(2)} + \mathbf{e}_3\,\exp\big(Q_3\,\mathbf{v}^{(2)}\big),\\
\mathbf{v}^{(4)} &= \big(I - \mathbf{e}_4\,\mathbf{e}_4^T\big)\mathbf{v}^{(3)} + \mathbf{e}_4\,\sin\big(Q_4\,\mathbf{v}^{(3)}\big),\\
\mathbf{v}^{(5)} &= \big(I - \mathbf{e}_5\,\mathbf{e}_5^T\big)\mathbf{v}^{(4)} + \mathbf{e}_5\,\mathrm{prod}\big(Q_5\,\mathbf{v}^{(4)}\big),
\end{aligned}$$

where $I = I_5$ and $\mathrm{prod}(a, b) \equiv a * b$ for $(a, b) \in \mathbb{R}^2$.

2.2. No-overwrite conditions

Example (2.4) already shows some properties that we require in general. Namely, we normally impose the natural condition that the function φ_i may only be applied when its argument u_i has reached its final value. In other words we must have evaluated all φ_j that precede φ_i so that

$$Q_i\,P_j \neq 0 \quad \Rightarrow \quad j \prec i. \tag{2.7}$$

In particular we always have $Q_i\,P_i = 0$, as $\prec$ is assumed nonreflexive. In addition to these write–read dependences between the two statements φ_j and φ_i, we also have to worry about write–write dependences where several elemental functions 'overlap', in that equivalently

$$P_i^T P_j \neq 0 \quad \Leftrightarrow \quad P_j^T P_i \neq 0.$$

Overlapping really only makes sense if all but possibly one of the φ_i are incremental, *i.e.*, $\sigma_i = 1$, since otherwise values are overwritten before they are ever used. Moreover, since by their very nature the incremental φ_i with overlapping P_i do commute, their relative order does not matter, but they all must succeed a possible, nonincremental one. Formally we impose the condition

$$P_j^T P_i \neq 0 \quad \Rightarrow \quad \big(j \prec i \quad \text{and} \quad \sigma_i = 1\big) \quad \text{or} \quad \big(i \prec j \quad \text{and} \quad \sigma_j = 1\big). \tag{2.8}$$

From now on we will assume that we have an acyclic ordering satisfying conditions (2.7), (2.8) and furthermore make the monotonicity assumption $j \prec i \Rightarrow j < i$, where $<$ denotes the usual ordering of integers by size.

2.3. Independent and dependent variables

Even though they are patently obvious in many practical codes, we may characterize the independent and dependent variables in terms of the precedence relation $\prec$ as follows. All indices j that are minimal with respect to $\prec$ must have a trivial argument mapping $Q_j = 0$. Otherwise, by the assumed minimality of the state space we would have $Q_j P_k \neq 0$ for some k. All such minimal φ_j initialize v_j to some constant vector x_j unless j is incremental, which we will preclude by assumption. Furthermore, we assume without loss of generality that the minimal indices are given by $j = 1 \dots n$ and may thus write

$$v_j = x_j \quad \text{for} \quad j = 1 \dots n.$$

We consider

$$\mathbf{x} = (x_j)_{j=1\dots n} \in X \equiv P_1 \mathcal{H}_1 \times P_2 \mathcal{H}_2 \times \cdots \times P_{n-1} \mathcal{H}_{n-1} \times P_n \mathcal{H}_n.$$

as the vector of *independent* variables. Similarly, we assume that the maximal indices $i \in \mathcal{I}$ with respect to $\prec$ are given by $i = l - m + 1 \dots l$. The values v_i with $i > l - m$ do not impact any other elemental function and are therefore considered as *dependent* variables

$$y_i = v_{\hat{\imath}} \quad \text{with} \quad \hat{\imath} \equiv l - m + i \quad \text{for} \quad i = 1 \dots m.$$

We consider

$$\mathbf{y} = (y_i)_{i=1\dots m} \in Y \equiv P_{l-m+1} \mathcal{H}_{l-m+1} \times \cdots \times P_{l-1} \mathcal{H}_{l-1} \times P_l \mathcal{H}_l$$

as the vector of dependent variables. Combining our assumption on the independents and dependents in the following additional condition, we obtain

$$j \prec i \quad \Rightarrow \quad i > n \quad \text{and} \quad j \leq l - m. \tag{2.9}$$

Also, we exclude independent or dependent elementals from being incremental, so that

$$i \leq n \quad \text{or} \quad i > l - m \quad \Rightarrow \quad \sigma_i = 0.$$

Consequently, by (2.8) the $P_i \mathcal{H}_i$ for $i = 1 \dots n$ and $i = l - m + 1 \dots l$ must be mutually orthogonal, so that we have in fact $X \subset \mathcal{H}$ and $Y \subset \mathcal{H}$.

2.4. Four-part form and complexity

To ensure that the result of our evaluation procedure is uniquely defined even when there are incremental assignments, we will assume that the state

Table 2.1. Original function evaluation procedure.

$\mathbf{v}$	$=$	0		
v_i	$=$	x_i	for	$i = 1 \dots n$
v_i v_i	$*=$ $+=$	σ_i $\varphi_i(u_i)$	for	$i = n+1 \dots l$
y_{m-i}	$=$	v_{l-i}	for	$i = m-1 \dots 0$

vector $\mathbf{v}$ is initialized to zero. Hence we obtain the program structure displayed in Table 2.1.

With P_X and P_Y the orthogonal projections from $\mathcal{H}$ onto its subspaces X and Y, we obtain the mapping from $\mathbf{x} = \big(x_j\big)_{j=1}^{n}$ to $\mathbf{y} = \big(y_i\big)_{i=1}^{m}$ as the composite function

$$F \quad \equiv P_Y \circ \Phi_l \circ \Phi_{l-1} \circ \cdots \circ \Phi_2 \circ \Phi_1 \circ P_X^T, \tag{2.10}$$

where the transitions Φ_i are as defined in (2.5). The application of (2.10) to a particular vector $\mathbf{x} \in X$ can be viewed as the loop

$$\mathbf{v}^{(0)} = P_X^T \mathbf{x}, \quad \mathbf{v}^{(i)} = \Phi_i\big(\mathbf{v}^{(i-1)}\big) \quad \text{for} \quad i = 1 \dots l, \quad \mathbf{y} = P_Y \, \mathbf{v}^{(l)}. \tag{2.11}$$

We may summarize the mathematical development in this section as follows.

General Assumption: Elemental Decomposition.

(i) The elemental functions $\varphi_i : \mathcal{D}_i \mapsto \mathcal{H}_i = P_i^T \mathcal{H}$ are $d \geq 1$ times continuously differentiable on their open domains $\mathcal{D}_i = Q_i \, \mathcal{H}$.

(ii) The pairs of linear mappings $\{P_i, Q_i\}$ and the partial ordering $\prec$ are consistent in that (2.7), (2.8), and (2.9) hold.

As an immediate consequence we state the following result without proof.

Proposition 2.1. Given the General Assumption, Table 2.1 yields, for any monotonic total ordering of $\mathcal{I}$, the same unique vector function defined in (2.10), that is,

$$X \ni \mathbf{x} \mapsto \mathbf{y} = F(\mathbf{x}) \in Y,$$

which is d times continuously differentiable, by the chain rule.

Only in Sections 3.6 and 5.4 will we need $d > 1$. The partial ordering in $\mathcal{I}$ is only important for parallel evaluation and reversal without a value stack, as described in Section 3.2.

As to the computational cost, we will assume that the operation count is additive in that

$$\texttt{OPS}\,(F(\mathbf{x})) = \sum_{i=1}^{l} \texttt{OPS}\,(\varphi_i(u_i)), \tag{2.12}$$

where we have neglected the cost of performing the incrementations and projections.

On a parallel machine elemental functions φ_i and φ_j that do not depend on each others' results can be executed simultaneously. It is customary to assume that a read conflict, *i.e.*, $Q_i\, Q_j^T \neq 0$, is no inhibition for concurrent execution, and we will assume here that the same is true for the incrementation of results, *i.e.*, $P_i^T P_j \neq 0$. This is a crucial assumption for showing that adjoints have the same degree of parallelism as the original evaluation procedure. Whether or not it is realistic depends on the computing platform and the nature of the elemental functions φ_i. If they are rather chunky, involving many internal calculations, the delays through incremental write conflict are indeed likely to be negligible. In any case, it makes no sense to schedule individual operations separately on a single processor machine, so that we may consider the φ_i to be more substantial subtasks in a parallel context. Then we can estimate the wall clock time by the longest, and thus critical path, *i.e.*,

$$\texttt{CLOCK}\,(F(\mathbf{x})) = \max_{\mathcal{P}\subset\mathcal{G}} \sum_{i\in\mathcal{P}} \texttt{OPS}\,(\varphi_i). \tag{2.13}$$

Here $\mathcal{P}$ denotes a directed path in $\mathcal{G}$ with $i \in \mathcal{P}$ ranging over its vertices. Throughout we will use CLOCK to denote the idealized wall clock time on a parallel machine with an unlimited supply of processors and zero communication costs. For practical implementations of AD with parallelism, see Carle and Fagan (1996), Christianson, Dixon and Brown (1997) and Mancini (2001).

3. Basic forward and reverse mode of AD

In this section we introduce the basic forward and reverse mode of AD and develop estimates for its computational complexity. They may be geometrically interpreted as the forward propagation of tangents or the backward propagation of normals or cotangents. Therefore they are often referred to as the *tangent* and the *cotangent mode*, respectively. Rather than propagating a single tangent or normal, we may also carry forward or back bundles of them in the so-called *vector mode.* It amortizes certain overhead costs and allows the efficient evaluation of sparse Jacobians by matrix compression. These aspects as well as techniques for propagating bit pattern will be sketched in Section 3.5. The section concludes with some remarks on higher-order adjoints and a brief section summary.

3.1. Forward differentiation

Differentiating (2.10), we obtain by the chain rule the Jacobian

$$F'(\mathbf{x}) \quad \equiv P_Y \;\Phi'_l \;\Phi'_{l-1} \;\cdots\; \Phi'_2 \;\Phi'_1 \;P_X^T, \tag{3.1}$$

where

$$\Phi'_i = I - (1-\sigma_i)P_i\,P_i^T + P_i\,\varphi'_i(u_i)\,Q_i. \tag{3.2}$$

Rather than computing the whole Jacobian explicitly, we usually prefer in AD to propagate directional derivatives along a smooth curve $\mathbf{x}(t) \subset X$ with $t \in (-\varepsilon, \varepsilon)$ for some $\varepsilon > 0$. Then $\mathbf{y}(t) \equiv F(\mathbf{x}(t)) \subset Y$ is also a smooth curve with the tangent

$$\dot{\mathbf{y}}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\,F(\mathbf{x}(t)) = F'(\mathbf{x}(t))\,\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}(t) \;\subset Y. \tag{3.3}$$

Similarly all states $\mathbf{v}^{(i)} = \mathbf{v}^{(i)}(t)$ and intermediates $v_i = v_i(t) = P_i^T\mathbf{v}^{(i)}$ are differentiable functions of $t \in (-\varepsilon, \varepsilon)$ with the tangent values

$$\dot{v}_i \equiv \frac{\mathrm{d}}{\mathrm{d}t}v_i(t)\bigg|_{t=0}.$$

Multiplying (3.1) by the input vector $\dot{\mathbf{x}} = \mathrm{d}\mathbf{x}(t)/\mathrm{d}t|_0$, we obtain the vector equation

$$\dot{\mathbf{y}} = P_Y\,\Phi'_l\left(\Phi'_{l-1}\left(\cdots\left(\Phi'_2\left(\Phi'_1\,P_X^T\,\dot{\mathbf{x}}\right)\right)\cdots\right)\right).$$

The bracketing is equivalent to the loop of matrix-vector products

$$\dot{\mathbf{v}}^{(0)} = P_X^T\,\dot{\mathbf{x}},\; \dot{\mathbf{v}}^{(i)} = \Phi'_i(\mathbf{v}^{(i-1)})\dot{\mathbf{v}}^{(i-1)} \;\text{ for }\; i = 1\ldots l,\; \dot{\mathbf{y}} = P_Y\dot{\mathbf{v}}^{(l)}. \tag{3.4}$$

With $u_i = Q_i\mathbf{v}^{(i-1)}$, $\dot{u}_i = Q_i\,\dot{\mathbf{v}}^{(i-1)}$ and $\dot{\varphi}_i(u_i, \dot{u}_i) \equiv \varphi'_i(u_i)\dot{u}_i$ this may be rewritten as the so-called *tangent procedure* listed in Table 2.1, and it can now be stated formally.

Proposition 3.1. Given the General Assumption, the procedure listed in Table 3.1 yields the value $\dot{\mathbf{y}} = F'(\mathbf{x})\dot{\mathbf{x}}$ with F, as defined in Proposition 2.1.

Obviously Table 3.1 has the same form as Table 2.1 with $\mathcal{H}$ replaced by $\mathcal{H} \times \mathcal{H}$, Q_i by $Q_i \times Q_i$, and P_i by $P_i \times P_i$. In other words, all argument and value spaces have been doubled up by a derivative companion, but the flags σ_i, the precedence relation $\prec$ and thus the structure of the computational graph remain unchanged.

As we can see, the evaluation of the combined function

$$[F(\mathbf{x}), \dot{F}(\mathbf{x}, \dot{\mathbf{x}})] \equiv [F(\mathbf{x}), F'(\mathbf{x})\dot{\mathbf{x}}] \tag{3.5}$$

requires exactly one evaluation of the corresponding elemental combinations

$$[\varphi_i(u_i), \dot{\varphi}_i(u_i, \dot{u}_i)] \equiv [\varphi_i(u_i), \varphi'_i(u_i)\dot{u}_i].$$

Table 3.1. Tangent procedure derived from original procedure in Table 2.1.

$[\mathbf{v}, \dot{\mathbf{v}}]$	$=$	0		
$[v_i, \dot{v}_i]$	$=$	$[x_i, \dot{x}_i]$	for	$i = 1 \ldots n$
$[v_i, \dot{v}_i]$	$*=$	σ_i	for	$i = n+1 \ldots l$
$[v_i, \dot{v}_i]$	$+=$	$[\varphi_i(u_i), \dot{\varphi}_i(u_i, \dot{u}_i)]$		
$[y_{m-i}, \dot{y}_{m-i}]$	$=$	$[v_{l-i}, \dot{v}_{l-i}]$	for	$i = m-1 \ldots 0$

We have bracketed $[\varphi_i, \dot{\varphi}_i]$ side-by-side to indicate that good use can often be made of common subexpressions in evaluating the function and its derivative. In any case we can show that

$$\frac{\texttt{OPS}\big([F(\mathbf{x}), \dot{F}(\mathbf{x}, \dot{\mathbf{x}})]\big)}{\texttt{OPS}\,\big(F(\mathbf{x})\big)} \le \max_{1 \le i \le l} \frac{\texttt{OPS}\,\big([\varphi_i(u_i), \dot{\varphi}_i(u_i, \dot{u}_i)]\big)}{\texttt{OPS}\,\big(\varphi_i(u_i)\big)} \le 3. \tag{3.6}$$

The upper bound 3 on the relative cost of performing a single tangent calculation is arrived at by taking the maximum over all elemental functions. It is actually attained for a single multiplication, which spawns two extra multiplications by the chain rule

$$v = \varphi(u, w) \equiv u * w \quad \rightarrow \quad \dot{v} = \dot{\varphi}(u, w, \dot{u}, \dot{w}) = u * \dot{w} + w * \dot{u}.$$

Since the data dependence relation $\prec$ applies to the $\dot{\varphi}_i$ in exactly the same way as to the φ_i, we obtain on a parallel machine

$$\texttt{CLOCK}\big([F(\mathbf{x}), \dot{F}(\mathbf{x}, \dot{\mathbf{x}})]\big) \le 3\,\texttt{CLOCK}\big(F(\mathbf{x})\big),$$

where CLOCK is the idealized wall clock time introduced in (2.13).

The bound 3 is pessimistic as the cost ratio between φ and $\dot{\varphi}$ is more advantageous for most other elemental functions. On the other hand actual runtime ratios between codes representing Table 3.1 and Table 2.1 may well be worse on account of various effects including loss of vectorization. This gap between operation count and actual runtime is likely to be even more marked for the following adjoint calculation.

3.2. Reverse differentiation and adjoint vectors

Rather than propagating tangents forward we may propagate normals backward, that is, instead of computing $\dot{\mathbf{y}} = \dot{F}(\mathbf{x}, \dot{\mathbf{x}}) = F'(\mathbf{x})\dot{\mathbf{x}}$ we may evaluate $\bar{\mathbf{x}} = \bar{F}(\mathbf{x}, \bar{\mathbf{y}}) \equiv \bar{\mathbf{y}} F'(\mathbf{x})$. Here the dual vectors $\bar{\mathbf{y}} \in Y^* = Y^T$ and $\bar{\mathbf{x}} \in X^* = X^T$ are thought of as row vectors. Using the transpose of the

Jacobian product representation (3.1) we find that

$$\bar{\mathbf{x}}^T = P_X\ [\Phi_1']^T\big([\Phi_2']^T\big(\cdots\big([\Phi_{l-1}']^T\big([\Phi_l']^T\ P_Y^T\bar{\mathbf{y}}^T\big)\big)\cdots\big)\big).$$

The bracketing is equivalent to the loop of vector-matrix products

$$\bar{\mathbf{v}}^{(l)} = \bar{\mathbf{y}}\, P_Y, \quad \bar{\mathbf{v}}^{(i-1)} = \bar{\mathbf{v}}^{(i)}\, \Phi_i'(\mathbf{v}^{(i-1)}) \quad \text{for} \quad i = l \ldots 1, \quad \bar{\mathbf{x}} = \bar{\mathbf{v}}^{(0)}\, P_X^T.$$

Formally we will write, in agreement with (3.2),

$$\bar{\mathbf{v}}^{(i-1)} = \bar{\Phi}_i\big(\mathbf{v}^{(i-1)}, \bar{\mathbf{v}}^{(i)}\big) \equiv \bar{\mathbf{v}}^{(i)}\Phi_i'\big(\mathbf{v}^{(i-1)}\big),$$

where, as ever, adjoints are interpreted as row vectors. With $\bar{v}_i \equiv \bar{\mathbf{v}}^{(i)}P_i$ and $\bar{u}_i = \bar{\mathbf{v}}^{(i)}Q_i^T$, we obtain from (3.2), using $Q_iP_i = 0$, the adjoint elemental function

$$\big[\bar{u}_i \mathrel{+}= \bar{v}_i\, \varphi_i'(u_i); \quad \bar{v}_i \mathrel{*}= \sigma_i\big]. \tag{3.7}$$

In other words the multiplicative statement $v_i \mathrel{*}= \sigma_i$ is self-adjoint in that, correspondingly, $\bar{v}_i \mathrel{*}= \sigma_i$. The incremental part $v_i \mathrel{+}= \varphi(u_i)$ generates the equally incremental statement

$$\bar{u}_i \mathrel{+}= \bar{\varphi}(u_i, \bar{v}_i) \equiv \bar{v}_i\, \varphi_i'(u_i). \tag{3.8}$$

Following the forward sweep of Table 2.1, we may then execute the so-called reverse sweep listed in Table 3.2. Now, using the assumptions on the partial ordering in $\mathcal{I}$, we obtain the following result.

Proposition 3.2. Given the General Assumption, the procedure listed in Table 3.1 yields the value $\bar{\mathbf{x}} = \bar{\mathbf{y}}\, F'(\mathbf{x})$, with F as defined in Proposition 2.1.

Proof. The only fact to ascertain is that the argument u_i of the $\bar{\varphi}_i$ defined in (3.8) still has the correct value when it is called up in the third line of Table 3.2. However, this follows from the definition of u_i in (2.6) and our assumption (2.7), so that none of the statements φ_j with $j > i$, nor of course the corresponding $\bar{\varphi}_j$, can alter u_i before it is used again by $\bar{\varphi}_i$. □

So far, we have treated the $\bar{v}_i$ merely as auxiliary quantities during reverse matrix multiplications. In fact, their final values can be interpreted as adjoint vectors in the sense that

$$\bar{v}_i \equiv \frac{\partial}{\partial v_i}\bar{\mathbf{y}}\,\mathbf{y}. \tag{3.9}$$

Here $\bar{\mathbf{y}}$ is considered constant and the notation $\frac{\partial}{\partial v_i}$ requires further explanation. Partial differentiation is normally defined with respect to one of several independent variables whose values uniquely determine the function being differentiated. However, here $\mathbf{y} = F(\mathbf{x})$ is fully determined, via the evaluation procedure of Table 2.1, as a function of $x_1 \ldots x_n$, with each v_i, for $i > n$, occurring merely as an intermediate variable. Its value, $v_i = v_i(\mathbf{x})$,

Table 3.2. Reverse sweep following forward sweep of Table 2.1.

$\bar{\mathbf{v}}$	$=$	0		
$\bar{v}_{l-m+i}$	$=$	$\bar{y}_i$	for	$i = m \dots 1$
$\bar{u}_i$	$+=$	$\bar{\varphi}_i(u_i, \bar{v}_i)$	for	$i = l \dots n+1$
$\bar{v}_i$	$*=$	σ_i		
$\bar{x}_j$	$=$	$\bar{v}_j$	for	$j = n \dots 1$

being also uniquely determined by $\mathbf{x}$, the intermediate v_i is certainly not another independent variable. To define $\bar{v}_i$ unambiguously, we should perturb the assignment (2.2) to

$$v_i = \sigma_i v_i + \varphi_i(u_i) + \delta_i \quad \text{for} \quad \delta_i \in \mathcal{H}_i,$$

and then set

$$\bar{v}_i = \nabla_{\delta_i} \bar{\mathbf{y}}\, \mathbf{y}\big|_{\delta_i = 0} \in \mathcal{H}_i^* = \mathcal{H}_i. \tag{3.10}$$

Thus, $\bar{v}_i$ quantifies the sensitivity of $\bar{\mathbf{y}}\, \mathbf{y} = \bar{\mathbf{y}}\, F(\mathbf{x})$ to a small perturbation δ_i in the ith assignment. Clearly, δ_i can be viewed as a variable independent of $\mathbf{x}$, and we may allow one such perturbation for each v_i.

The resulting perturbation δ of $\bar{\mathbf{y}}\, \mathbf{y}$ can then be estimated by the first-order Taylor expansion

$$\delta \approx \sum_{i=1}^{l} \bar{v}_i \cdot \delta_i. \tag{3.11}$$

When v_i is scalar the δ_i may be interpreted as rounding errors incurred in evaluating $v_i = \varphi_i(u_i)$. Then we may assume $\delta_i = v_i\, \eta_i$ with all $|\, \eta_i| \leq \eta$, the relative machine precision. Consequently, we obtain the approximate bound

$$|\, \delta\, | \lesssim \left| \sum_{i=1}^{l} \bar{v}_i\, v_i\, \eta_i \right| \leq \eta \sum_{i=1}^{l} |\, \bar{v}_i\, v_i\, |. \tag{3.12}$$

This relation has been used by many authors for the estimation of round-off errors (Iri, Tsuchiya and Hoshi 1988). In particular, it led Seppo Linnainmaa to the first publication of the reverse mode in English (Linnainmaa 1983). The factor $\sum_{i=1}^{l} |\bar{v}_i v_i|$ has been extensively used by Stummel (1981) and others as a condition estimate for the function evaluation procedure. Braconnier and Langlois (2001) used the adjoints $\bar{v}_i$ to compensate evaluation errors.

In the context of AD we mostly regard adjoints as vectors rather than adjoint equations. They represent gradients, or more generally Fréchet derivatives, of the weighted final result $\bar{\mathbf{y}}\, \mathbf{y}$ with respect to all kinds of intermediate vectors that are generated in a well-defined hierarchical fashion. This procedural view differs strongly from the more global, equation-based concept prevalent in the literature (see, *e.g.*, Marcuk (1996)).

Viewing all u_i as given constants in Table 3.2, this procedure may by itself be identified with the form of Table 2.1. However, the roles of P_i and Q_i have been interchanged and the precedence relation $\prec$ must be reversed. To interpret the combination of Tables 2.1 and 3.2 as a single evaluation procedure, we introduce an isomorphic copy $\bar{\mathcal{I}}$ of $\mathcal{I}$, denoting the bijection between elements of $\mathcal{I}$ and $\bar{\mathcal{I}}$ by an overline:

$$i \in \mathcal{I} \quad \Leftrightarrow \quad \bar{\imath} \in \bar{\mathcal{I}} \quad \Rightarrow \quad \varphi_{\bar{\imath}}(u_i, \bar{v}_i) \equiv \bar{\varphi}_i(u_i, \bar{v}_i). \tag{3.13}$$

Identifying the Hilbert space $\mathcal{H}$ and its subspaces with their duals, we obtain the state space $\mathcal{H} \times \mathcal{H}$ for the combined procedure. Correspondingly the argument selections and value projections can be defined such that the computational graph for $\bar{F}(\mathbf{x}, \bar{\mathbf{y}}) = \bar{\mathbf{y}} F'(\mathbf{x})$ has the following ordering.

Lemma 3.3. With $\varphi_{\bar{\imath}}$ as defined in (3.13), the extension of the precedence relation $\prec$ to $\mathcal{I} \cup \bar{\mathcal{I}}$ is given by

$$\bar{\imath} \prec \bar{\jmath} \quad \Leftrightarrow \quad j \prec i \quad \Leftrightarrow \quad j \prec \bar{\imath}.$$

Proof. The extension can be written as

$$(Q_i, 0) \quad \text{for} \quad i \in \mathcal{I} \quad \text{and} \quad Q_{\bar{\imath}} \equiv \begin{pmatrix} Q_i & 0 \\ 0 & P_i^T \end{pmatrix} \quad \text{for} \quad \bar{\imath} \in \bar{\mathcal{I}}$$

and

$$\begin{pmatrix} P_i \\ 0 \end{pmatrix} \quad \text{for} \quad i \in \mathcal{I} \quad \text{and} \quad P_{\bar{\imath}} = \begin{pmatrix} 0 \\ Q_i^T \end{pmatrix} \quad \text{for} \quad \bar{\imath} \in \bar{\mathcal{I}}.$$

Hence we find, for the precedence between adjoint indices,

$$Q_{\bar{\imath}} \, P_{\bar{\jmath}} = P_i^T \, Q_j^T = (Q_j \, P_i)^T \quad \Leftrightarrow \quad \bar{\jmath} \prec \bar{\imath} \quad \Leftrightarrow \quad j \succ i.$$

For the precedence between direct and adjoint indices, we obtain

$$Q_{\bar{\imath}} \, P_j = Q_i \, P_j \quad \text{so that} \quad j \prec \bar{\imath} \quad \Leftrightarrow \quad j \prec i. \qquad \square$$

Hence we see that the graph with the vertex set $\mathcal{I} \cup \bar{\mathcal{I}}$ and the extended precedence relation consists of two halves whose internal dependencies are mirror images of each other. However, the connections between the two halves are not symmetric, in that $j \prec \bar{\imath}$ normally precludes rather than implies the relation $i \succ \bar{\jmath}$, as shown in Figure 3.1.

It is very important to notice that the adjoint operations are executed in reverse order. As we can see from Tables 2.1 and 3.2, the evaluation of the combined function $[F(\mathbf{x}), \bar{F}(\mathbf{x}, \bar{\mathbf{y}})]$ requires exactly one evaluation of the corresponding elemental combinations $\varphi_i(u_i)$ and $\bar{\varphi}_i(u_i, \bar{v}_i)$, though this time the two must be evaluated separately, *i.e.*, at different times. In any case, we find that

$$\frac{\mathtt{OPS}([F(\mathbf{x}), \bar{F}(\mathbf{x}, \bar{\mathbf{y}})])}{\mathtt{OPS}(F(\mathbf{x}))} \leq \max_{1 \leq i \leq l} \frac{\mathtt{OPS}(\varphi_i(u_i)) + \mathtt{OPS}(\bar{\varphi}_i(u_i, \bar{v}_i))}{\mathtt{OPS}(\varphi_i(u_i))} \leq 3. \tag{3.14}$$

The upper bound 2 on the relative cost of performing a single reverse operation can again be found by taking the maximum over all elemental functions, which is actually obtained for a single multiplication,

$$v \mathrel{+}= \varphi(u, w) \equiv u * w \quad \rightarrow \quad [\bar{u}, \bar{w}] \mathrel{+}= [\bar{v} * w, \bar{v} * u],$$

only counting multiplications as elsewhere. For the estimate (2.13) of parallel execution time, our assumptions now allow us to conclude that

$$\mathtt{CLOCK}\left([F(\mathbf{x}), \bar{F}(\mathbf{x}, \bar{\mathbf{y}})]\right) \leq 3\,\mathtt{CLOCK}(F(\mathbf{x})). \tag{3.15}$$

This estimate holds because every critical path in $\mathcal{I} \cup \bar{\mathcal{I}}$ must be the union of two paths $\mathcal{P} \subset \mathcal{I}$ and $\bar{\mathcal{P}} \subset \bar{\mathcal{I}}$, whose complexity is bounded by one and two times $\mathtt{CLOCK}(F(\mathbf{x}))$, respectively.

3.3. Symmetry of Hessian graphs

To make the extended graph fully symmetric we must be able to rewrite $\varphi_{\bar{\imath}}(u_i, \bar{v}_i)$ as $\varphi_{\bar{\imath}}(v_i, \bar{v}_i)$ so that

$$Q_{\bar{\imath}} \equiv \begin{pmatrix} P_i^T & 0 \\ 0 & P_i^T \end{pmatrix} \quad \text{and hence} \quad j \prec \bar{\imath} \quad \Leftrightarrow \quad P_i^T P_j \neq 0 \quad \Leftrightarrow \quad i \prec \bar{\jmath}.$$

In other words, we must pack all information needed for the adjoint operation into the value of the original one. For certain elementals such as $v = \varphi(u) \equiv \exp(u)$ this is very natural, for in that particular case $\varphi'(u) = v$. For others like $\varphi(u, w) = u * w$ we would have to append the partials $\partial\varphi/\partial u = w$ and $\partial\varphi/\partial w = u$ to the result, and thus return the vector value $v = (u * w, w, u)$. This is more or less what is done when a tangent linear code is first developed and then the partials are used in the cotangent linear, or adjoint code.

As an example we may again consider the function defined in (2.4). In Table 3.3 we have listed on the left the combination of the forward and reverse sweep without the inclusion of partials in the elemental values. On the right we have listed the extended symmetrized version.

The corresponding computational graphs are depicted in Figure 3.1. With the curved, dotted arcs included, we have the original version associated with the left-hand side of Table 3.3. The dotted arcs can be replaced by the direct, dashed arcs if the computational procedure is modified according to the right-hand side of Table 3.3. The superscripts denote additional components of the intermediate values, which contain the partial derivatives needed on the way back. Apart from its aesthetic appeal, the symmetry of the Hessian graph promises at least a halving of the operation count

Table 3.3. Adjoint procedure = forward sweep + reverse sweep.

Nonsymmetric adjoint			Adjoint with symmetric graph		
(v_1, v_2)	$=$	(x_1, x_2)	(v_1, v_2)	$=$	(x_1, x_2)
v_3	$=$	$\exp(v_1)$	v_3	$=$	$\exp(v_1)$
v_4	$=$	$\sin(v_1{+}v_2)$	$(v_4^{(0)}, v_4^{(1)})$	$=$	$(\sin(v_1{+}v_2), \cos(v_1{+}v_2))$
v_5	$=$	$v_3 * v_4$	$(v_5^{(0)}, v_5^{(1)}, v_5^{(2)})$	$=$	$(v_3 * v_4^{(0)}, v_4^{(0)}, v_3)$
y	$=$	v_5	y	$=$	$v_5^{(0)}$
$\bar{v}_5$	$=$	$\bar{y}$	$\bar{v}_5^{(0)}$	$=$	$\bar{y}$
$(\bar{v}_3, \bar{v}_4)$	$\mathrel{+}=$	$\bar{v}_5 * (v_4, v_3)$	$(\bar{v}_3, \bar{v}_4)$	$\mathrel{+}=$	$\bar{v}_5 * (v_5^{(1)}, v_5^{(2)})$
$(\bar{v}_1, \bar{v}_2)$	$\mathrel{+}=$	$\bar{v}_4 * \cos(v_1{+}v_2) * (1,1)$	$(\bar{v}_1, \bar{v}_2)$	$\mathrel{+}=$	$\bar{v}_4 * v_4^{(1)} * (1,1)$
$\bar{v}_1$	$\mathrel{+}=$	$\bar{v}_3 * \exp(v_1)$	$\bar{v}_1$	$\mathrel{+}=$	$\bar{v}_3 * v_3$
$(\bar{x}_1, \bar{x}_2)$	$=$	$(\bar{v}_1, \bar{v}_2)$	$(\bar{x}_1, \bar{x}_2)$	$=$	$(\bar{v}_1, \bar{v}_2)$

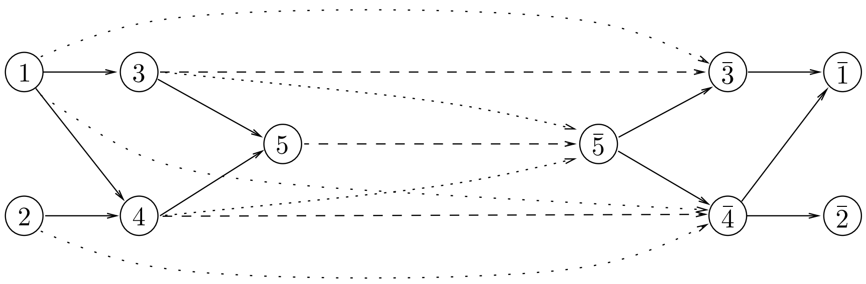

Figure 3.1. Graph corresponding to left $(\cdots)$ and right $(---)$ part of Table 3.3.

and storage requirement for the accumulation task discussed in Section 6. A different but related symmetrization of Hessian graphs was developed by Dixon (1991).

3.4. The memory issue

The fact that gradients and other adjoint vectors $\bar{\mathbf{x}} = \bar{F}(\mathbf{x}, \bar{\mathbf{y}}) = \bar{\mathbf{y}}F'(\mathbf{x})$ can, according to (3.14) and (3.15), be computed with essentially the same operation count as the underlying $\mathbf{y} = F(\mathbf{x})$ is certainly impressive, and possibly still a little bit surprising. However, its practical benefit is sometimes in doubt on account of its potentially large memory requirement. Our no-overwrite conditions (2.7) and (2.8) mean that, except for incremental assignments, each elemental function requires new storage for its value. As we

Table 3.4. Direct/adjoint statement pairs.

Case	On forward sweep			On reverse sweep		
(i)	v_i	$\longrightarrow$	`STACK`	v_i	$\longleftarrow$	`STACK`
	v_i	$=$	$\varphi_i(u_i)$	$\bar{u}_i$	$+=$	$\bar{\varphi}_i(u_i, \bar{v}_i)$
(ii)	v_i	$+=$	$\varphi_i(u_i)$	$\bar{u}_i$	$+=$	$\bar{\varphi}_i(u_i, \bar{v}_i)$
				v_i	$-=$	$\varphi_i(u_i)$
(iii)	v_i	$=$	$\varphi_i(u_i)$	$\bar{u}_i$	$+=$	$\bar{\varphi}_i(v_i, \bar{v}_i)$
	v_i	$\longrightarrow$	`STACK`	v_i	$\longleftarrow$	`STACK`

noted before, the composition (2.10) always represents a well-defined vector function $\mathbf{y} = F(\mathbf{x})$, no matter how the projections Q_i and P_i are defined, provided the state $\mathbf{v}$ is initialized to zero. As a consequence the tangent procedure listed in Table 3.1 can also be applied without any difficulties, yielding consistent results.

Things are very different for the reverse mode. Here the $\bar{\varphi}_i(u_i, \bar{v}_i)$ or $\bar{\varphi}_i(v_i, \bar{v}_i)$ require the old values of u_i or v_i from the time $\varphi_i(u_i)$ itself was originally evaluated. One simple way to ensure this is to save the $v_i = P_i^T \mathbf{v}$ on a value stack during the forward sweep, and then to recover them on the way back just before or just after the form $\bar{\varphi}_i(u_i, \bar{v}_i)$ or $\bar{\varphi}_i(v_i, \bar{v}_i)$ is used, respectively. Then we may modify the statements in the main loops of Table 2.1 and Table 3.2 by one of the three pairs listed in Table 3.4.

The first case (i) describes a copy-on-write strategy for nonincremental statements where all pre-values are saved on a stack just before they are overwritten. On the reverse sweep the value is read back from the stack to restore the state $\mathbf{v}^{(i-1)}$ that was valid just before the original call to φ_i. No extra storage is necessary in case (ii) of additively incremental statement as the previous state can be restored by the corresponding decremental operation. This mechanism allows, for example, the adjoining of an LU factorization procedure with a stack whose size grows only quadratically rather than cubically in the matrix dimension. This growth rate can be made linear if we also choose to invert multiplicatively incremental operations like $v *= u$ and $v /= u$, provided we can ensure $u \neq 0$, which is of course a key point of pivoting in matrix factorization. In general, we face a choice between restoration from a stack and various modes of recomputation. For a more thorough discussion of these issues see Faure and Naumann (2001) and Giering and Kaminski (2001*b*). The main difference between cases (i) and (iii) is that, in the latter, the new value of v_i is needed for the adjoint $\bar{\varphi}(v_i, \bar{v}_i)$ so that restoration of the old value takes place afterwards.

If all elementals are treated according to (i) in Table 3.4, we obtain the bound

$$\mathtt{MEM}\bigl(\bar{F}(\mathbf{x},\bar{\mathbf{y}})\bigr) \approx \mathtt{MEM}\bigl(F\bigr) + \sum_{i=1}^{l} m_i \sim \mathtt{OPS}\bigl(F(\mathbf{x})\bigr). \tag{3.16}$$

Here the proportionality relation assumes that $\mathtt{OPS}(\varphi_i) \sim m_i \sim \mathtt{MEM}(\varphi_i)$, which is reasonable under most circumstances. The situation is depicted schematically in Figure 3.2. The `STACK` of intermediates v_i is sometimes called a *trajectory* or *execution* log of the evaluation procedure.

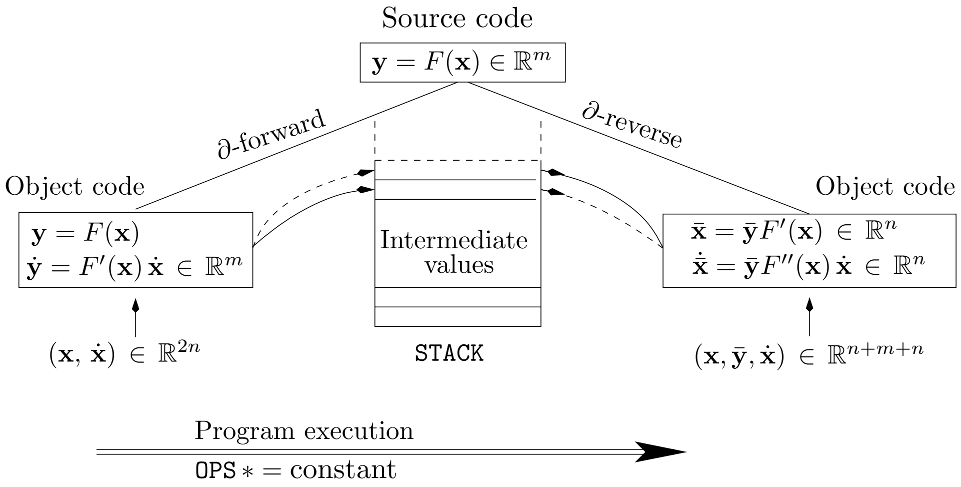
Source code
$\mathbf{y} = F(\mathbf{x}) \in \mathbb{R}^m$
∂-forward
∂-reverse
Object code
$\mathbf{y} = F(\mathbf{x})$
$\dot{\mathbf{y}} = F'(\mathbf{x})\,\dot{\mathbf{x}} \in \mathbb{R}^m$
Intermediate values
Object code
$\bar{\mathbf{x}} = \bar{\mathbf{y}}F'(\mathbf{x}) \in \mathbb{R}^n$
$\dot{\bar{\mathbf{x}}} = \bar{\mathbf{y}}F''(\mathbf{x})\,\dot{\mathbf{x}} \in \mathbb{R}^n$
$(\mathbf{x}, \dot{\mathbf{x}}) \in \mathbb{R}^{2n}$
STACK
$(\mathbf{x}, \bar{\mathbf{y}}, \dot{\mathbf{x}}) \in \mathbb{R}^{n+m+n}$
Program execution
$\mathtt{OPS}* = \text{constant}$

Figure 3.2. Practical execution of forward and reverse differentiation.

The situation is completely analogous to backward integration of the costate in the optimal control of ordinary differential equations (Campbell, Moore and Zhong 1994). More specifically, the problem

$$\min_{\mathbf{x}\in\mathcal{L}^\infty([0,T],\mathbb{R}^n)} \psi(\mathbf{v}(T)) \quad \text{such that} \quad \mathbf{v}(0) = \mathbf{v}_0 \in \mathbb{R}^n \tag{3.17}$$

and

$$\frac{\mathrm{d}}{\mathrm{d}t}\,\mathbf{v}(t) = \Phi(\mathbf{v}(t),\mathbf{x}(t)) \quad \text{for} \quad 0 \le t \le T \tag{3.18}$$

has the adjoint system

$$\frac{\mathrm{d}}{\mathrm{d}t}\,\bar{\mathbf{v}}(t) = -\bar{\mathbf{v}}(t)\,\Phi_{\mathbf{v}}(\mathbf{v}(t),\mathbf{x}(t)), \tag{3.19}$$

$$\bar{\mathbf{x}}(t) = \bar{\mathbf{v}}(t)\,\Phi_{\mathbf{x}}(\mathbf{v}(t),\mathbf{x}(t)), \tag{3.20}$$

with the terminal condition

$$\bar{\mathbf{v}}(T) = \nabla\psi(\mathbf{v}(T)). \tag{3.21}$$

Here the trajectory $\mathbf{v}(t)$ is a continuous solution path, whose points enter into the coefficients of the corresponding adjoint differential equation (3.19) for $\bar{\mathbf{v}}(t)$. Rather than storing the whole trajectory, we may instead store only certain checkpoints and repeat the forward simulation in pieces on the way back. This technique is the subject of the subsequent Section 4.

3.5. Propagating vectors or sparsity patterns

Rather than just evaluating one tangent $\dot{\mathbf{y}} = \dot{F}(\mathbf{x}, \dot{\mathbf{x}})$ or one adjoint $\bar{\mathbf{x}} = \bar{\mathbf{y}} F'(\mathbf{x})$, we frequently wish to evaluate several of them, or even the whole Jacobian $F'(\mathbf{x})$ simultaneously. While this bundling increases the memory requirement, it means that the general overhead, and especially the effort for evaluating the φ_i and their partial derivatives, may be better amortized. With $\dot{\mathbf{X}} \in X^p$ any set of p directions in X and $t \in \mathbb{R}^p$ a p-dimensional parameter, we may redefine for each intermediate v_i the derivative vector as

$$\dot{v}_i \equiv \nabla_t \, v_i\big(\mathbf{x} + \dot{\mathbf{X}}\, t\big)\bigg|_{t=0} \in \mathcal{H}_i^p.$$

Then the tangent procedure Table 3.1 can be applied without any formal change, though the individual products $\dot{v}_i = \varphi(u_i, \dot{u}_i) = \varphi'(u_i)\dot{u}_i$ must now be computed for $\dot{u}_i \in \mathcal{D}_i^p$. The overall result of this *vector* forward mode is the matrix

$$\dot{\mathbf{Y}} = \dot{F}(\mathbf{x}, \dot{\mathbf{X}}) \equiv F'(\mathbf{x})\dot{\mathbf{X}} \in Y^p.$$

In terms of complexity we can derive the upper bounds

$$\texttt{OPS}\big(\dot{F}(\mathbf{x}, \dot{\mathbf{X}})\big) \le (1 + 2\,p)\,\texttt{OPS}(F(\mathbf{x})) \le p\,\texttt{OPS}\big(\dot{F}(\mathbf{x}, \dot{\mathbf{x}})\big), \qquad (3.22)$$

$$\texttt{MEM}\big(\dot{F}(\mathbf{x}, \dot{\mathbf{X}})\big) \le (1 + p)\,\texttt{MEM}\big(F(\mathbf{x})\big),$$

and

$$\texttt{CLOCK}\big(\dot{F}(\mathbf{x}, \dot{\mathbf{X}})\big) \le (1 + 2\,p)\,\texttt{CLOCK}(F(\mathbf{x})).$$

With a bit of optimism we could replace the factor 2 in (3.22) by 1, which would make the forward mode as expensive as one-sided differences. In fact, depending on the problem and the computing platform, the forward mode tool ADIFOR 2.0 typically achieves a factor somewhere between 0.5 and 2. Unfortunately, this cannot yet be said for reverse mode tools.

There we obtain, for given $\bar{\mathbf{Y}} \in (Y^*)^q$, the adjoint bundle

$$\bar{\mathbf{X}} = \bar{F}\big(\mathbf{x}, \bar{\mathbf{Y}}\big) \equiv \bar{\mathbf{Y}}\, F'(\mathbf{x}) \in \big(X^*\big)^q$$

at the temporal complexity

$$\texttt{OPS}\big(\bar{F}(\mathbf{x}, \bar{\mathbf{Y}})\big) \le (1 + 2\,q)\,\texttt{OPS}(F(\mathbf{x})) \le q\,\texttt{OPS}\big(\bar{F}(\mathbf{x}, \bar{\mathbf{y}})\big),$$

and

$$\texttt{CLOCK}\big(\bar{F}(\mathbf{x},\bar{\mathbf{Y}})\big) \leq (1+2\,q)\,\texttt{CLOCK}(F(\mathbf{x})).$$

Since the trajectory size is independent of the adjoint dimension q, we obtain from (3.16) the spatial complexity

$$\texttt{MEM}(\bar{F}(\mathbf{x},\bar{\mathbf{Y}})) \sim q\,\texttt{MEM}(F(\mathbf{x})) + \texttt{OPS}(F(\mathbf{x})) \sim \texttt{OPS}(F(\mathbf{x})).$$

The most important application of the vector mode is probably the efficient evaluation of sparse matrices by matrix compression. Here $\dot{\mathbf{X}} \in X^p$ is chosen as a *seed matrix* for a given sparsity pattern such that the resulting *compressed* Jacobian $F'(\mathbf{x})\dot{\mathbf{X}}$ allows the reconstruction of all nonzero entries in $F'(\mathbf{x})$. This technique apparently originated with the grouping proposal of Curtis, Powell and Reid (1974), where each row of $\dot{\mathbf{X}}$ contains exactly one nonzero element and the p columns of $\dot{\mathbf{Y}}$ are approximated by the divided differences

$$\frac{1}{\varepsilon}\left[F\big(\mathbf{x}+\varepsilon\,\dot{\mathbf{X}}\,\mathbf{e}_j\big) - F(\mathbf{x})\right] = \dot{\mathbf{Y}}\mathbf{e}_j + 0(\varepsilon) \quad \text{for} \quad j = 1\dots p.$$

Here $\mathbf{e}_j$ denotes the jth Cartesian basis vector in $\mathbb{R}^p$. In AD the matrix $\dot{\mathbf{Y}}$ is obtained with working accuracy, so that the conditioning of $\dot{\mathbf{X}}$ is not quite so critical. The reconstruction of $F'(\mathbf{x})$ from $F'(\mathbf{x})\dot{\mathbf{X}}$ relies on certain submatrices of $\dot{\mathbf{X}}$ being nonsingular. In the CPR approach they are permutations of identity matrices and thus ideally conditioned. However, there is a price to pay, namely the number of columns p must be greater or equal to the chromatic number of the column-incidence graph introduced by Coleman and Moré (1984). Any such colouring number is bounded below by $\hat{n} \leq n$, the maximal number of nonzeros in any row of the Jacobian. By a degree of freedom argument, we see immediately that $F'(\mathbf{x})$ cannot be reconstructed from $F'(\mathbf{x})\dot{\mathbf{X}}$ if $p < \hat{n}$, but $p = \hat{n}$ suffices for almost all choices of the seed matrix $\dot{\mathbf{X}}$. The gap between the chromatic number and $\hat{n}$ can be as large as $n - \hat{n}$, as demonstrated by an example of Hossain and Steihaug (1998). Whenever the gap is significant we should instead apply dense seeds $\dot{\mathbf{X}} \in X^{\hat{n}}$, which were proposed by Newsam and Ramsdell (1983). Rather than using the seemingly simple choice of Vandermonde matrices, we may prefer the much better conditioned Pascal or Lagrange seeds proposed by Hossain and Steihaug (2002) and Griewank and Verma (2003), respectively. In many applications sparsity can be enhanced by exploiting partial separability, which sometimes even allows the efficient calculation of dense gradients using the forward mode.

The compression techniques discussed above require the *a priori* knowledge of the sparsity pattern, which may be rather complicated and thus tedious for the user to supply. Then we may prefer to execute the forward mode with $p = n$ and the v_i stored and manipulated as dynamically sparse vectors (Bischof, Carle, Khademi and Mauer 1996). Excluding exact cancellations, we may conclude that the operation count for computing the whole Jacobians in this sparse vector mode is also bounded above by $(1 + 2\,\hat{n})\texttt{OPS}(F(x))$. Unfortunately this bound may not be a very good indication of actual runtimes since the dynamic manipulation of sparse data structures typically incurs a rather large overhead cost. Alternatively we may propagate the sparsity pattern of the v_i as bit patterns encoded in $n/32$ integers (Giering and Kaminski 2001*a*). In this way the sparsity pattern of $F'(\mathbf{x})$ can be computed with about $n/32$ times the operation count of F itself and very little overhead. By so-called *probing algorithms* (Griewank and Mitev 2002) the cost factor $n/32$ can often be reduced to $O(\hat{n}\,\log n)$ for a seemingly large class of sparse matrices.

Throughout this subsection we have tacitly assumed that the sparsity pattern of $F'(\mathbf{x})$ is static, *i.e.*, does not vary as a function of the evaluation point $\mathbf{x}$. If it does, we have to either recompute the pattern at each new argument $\mathbf{x}$ or determine certain envelope patterns that are valid, at least in a certain subregion of the domain. All the techniques we have discussed here in their application to the Jacobian $F'(\mathbf{x})$ can be applied analogously to its transpose $F'(\mathbf{x})^T$ using the reverse mode (Coleman and Verma 1996). For certain matrices such as arrowheads, a combination of both modes is called for.

3.6. Higher-order adjoints

In Figure 3.2 we have already indicated that a combination of forward and reverse mode yields so-called *second-order adjoints* of the form

$$\dot{\bar{\mathbf{x}}} \equiv \dot{\bar{F}}(\mathbf{x},\bar{\mathbf{y}},\dot{\mathbf{x}},\dot{\bar{\mathbf{y}}}) \equiv \bar{\mathbf{y}}F''(\mathbf{x})\dot{\mathbf{x}} + \dot{\bar{\mathbf{y}}}F'(\mathbf{x}) \in X = X^*$$

for any given $(\mathbf{x},\dot{\mathbf{x}}) \in X^2$ and $(\bar{\mathbf{y}},\dot{\bar{\mathbf{y}}}) \in (Y^*)^2$. In Figure 3.2 the vector $\dot{\bar{\mathbf{y}}} \in Y^*$ was assumed to vanish, and the abbreviation

$$\dot{\bar{F}}(\mathbf{x},\bar{\mathbf{y}},\dot{\mathbf{x}},0) \equiv \bar{\mathbf{y}}F''(\mathbf{x})\dot{\mathbf{x}} \equiv \nabla_{\mathbf{x}}\left[\bar{\mathbf{y}}F'(\mathbf{x})\dot{\mathbf{x}}\right] \in X^*$$

is certainly stretching conventional matrix notation. To obtain an evaluation procedure for $\dot{\bar{F}}$ we simply have to differentiate the combination of Table 2.1 and Table 3.2 once more in the forward mode. Consequently, composing (3.6) and (3.14) we obtain the bound

$$\texttt{OPS}(\dot{\bar{F}}(\mathbf{x},\bar{\mathbf{y}},\dot{\mathbf{x}},\dot{\bar{\mathbf{y}}})) \leq 3\,\texttt{OPS}(\bar{F}(\mathbf{x},\bar{\mathbf{y}})) \leq 9\,\texttt{OPS}(F(\mathbf{x})).$$

Though based on a simple-minded count of multiplicative operations, this bound is, in our experience, not a bad estimate for actual runtimes.

In the context of optimization we may think of $F \equiv (f, c)$ as the combination of a scalar objective function $f : X \to \mathbb{R}$ and a vector constraint

$c : X \rightarrow \mathbb{R}^{m-1}$. With $\bar{\mathbf{y}}$ a vector of Lagrange multipliers $\bar{\mathbf{y}}_i$, the symmetric matrix $\bar{\mathbf{y}}F''(\mathbf{x}) = \sum_{i=1}^{m} \mathbf{y}_i \nabla^2 F_i$ is the Hessian of the Lagrangian function. One second-order adjoint calculation then yields exactly the information needed to perform one inner iteration step within a truncated Newton method applied to the KKT system

$$\bar{\mathbf{y}}F'(\mathbf{x}) = 0, \qquad c(\mathbf{x}) = 0.$$

Consequently the cost of executing one inner iteration step is roughly ten times that of evaluating $F = (f, c)$. Note in particular that a procedure for the evaluation of the whole constraint Jacobian ∇c need not be developed either automatically or by hand. Walther (2002) found that iteratively solving the linearized KKT system using exact second-order adjoints was much more reliable and also more efficient than the same method based on divided differences on the gradient of the Lagrangian function.

The brief description of second-order adjoints above begs the question of their relation to a nested application of the reverse mode. The answer is that $\dot{\bar{\mathbf{y}}} = \bar{\mathbf{y}}F''(\mathbf{x})\dot{\mathbf{x}}$ could also be computed in this way, but that the complication of such a nested reversal yields no advantage whatever. To see this we note first that, for a symmetric Jacobian $F'(\mathbf{x}) = F'(\mathbf{x})^T$, we obviously have $\bar{\mathbf{y}}F'(\mathbf{x}) = (F'(\mathbf{x})\,\dot{\mathbf{x}})^T$ if $\dot{\mathbf{x}} = \bar{\mathbf{y}}^T$. Hence the adjoint vector $\bar{\mathbf{x}} = \bar{\mathbf{y}}F'(\mathbf{x})$ can be computed by forward differentiation if $F'(\mathbf{x})$ is symmetric and thus actually the Hessian $\nabla^2\psi(\mathbf{x})$ of a scalar function $\psi(\mathbf{x})$ with the gradient $\nabla\psi(\mathbf{x}) = F(\mathbf{x})$. In other words, it never makes sense to apply the reverse mode to vector functions that are gradients. On the other hand, applying reverse differentiation to $\mathbf{y} = F(\mathbf{x})$ yields

$$\left[F(\mathbf{x})^T, \bar{\mathbf{y}}\,F'(\mathbf{x})\right]^T = \nabla_{\bar{\mathbf{y}},x}\,\bar{\mathbf{y}}\,F(\mathbf{x}).$$

This partitioned vector is the gradient of the Lagrangian $\bar{\mathbf{y}}F(\mathbf{x})$, so that a second differentiation may be carried out without loss of generality in the forward mode, which is exactly what we have done above. The process can be repeated to yield $\ddot{\bar{\mathbf{x}}} = \bar{\mathbf{y}}F'''(\mathbf{x})\dot{\mathbf{x}}\dot{\mathbf{x}}$ and so on by the higher forward differentiation techniques described in Chapter 10 of Griewank (2000).

3.7. Section summary

For functions $\mathbf{y} = F(\mathbf{x})$ evaluated by procedures of the kind specified in Section 2, we have derived procedures for computing corresponding tangents $F'(\mathbf{x})\dot{\mathbf{x}}$ and gradients $\bar{\mathbf{y}}F'(\mathbf{x})$, whose composition yields second-order adjoints $\bar{\mathbf{y}}F''(\mathbf{x})\dot{\mathbf{x}}$. All these derivative vectors are obtained with a small multiple of the operation count for $\mathbf{y}$ itself. Derivative matrices can be obtained by bundling these vectors, where sparsity can be exploited through matrix compression. By combining the well-known derivatives of elemental functions using the chain rule, the derived procedures can be generated line by line and, more generally, subroutine by subroutine. These simple observations form the core of AD, and everything else is questionable as to its relevance to AD and scientific computing in general.

4. Serial and parallel reversal schedules

The key difficulty in adjoining large evaluation programs is their reversal within a reasonable amount of memory. Performing the actual adjoint operations is a simple task compared to (re)producing all intermediate results in reverse order. The problem of reversing a program execution has received some perfunctory attention in the computer science literature (see, *e.g.*, Bennett (1973) and van der Snepscheut (1993)). The first authors to consider the problem from an AD point of view were apparently Volin and Ostrovskii (1985) and later Horwedel (1991). The problem of nested reversal that occurs in Pantoja's algorithm (Pantoja 1988) for computing Newton steps in discrete optimal control was first considered by Christianson (1999). No optimal reversal schedule has yet been devised for this discrete variant of the matrix Riccati equation. It might merely be a first example of computational schemes with an intricate, multi-directional information flow that can be treated by the checkpointing techniques developed in AD. In this survey we will examine only the case of discrete evolutions of the form (2.11). They may be interpreted as a chain of l subroutine calls. More generally we may consider the reversal of calling trees with arbitrary structure, whose depth in particular is a critical parameter for memory–runtime trade-offs (see Section 12.2 in Griewank (2000)).

Throughout this section we suppose that the transition functions Φ_i change all or at least most components of the state vector $\mathbf{v}^{(i-1)}$ so that l now represents the number of time-steps rather than elemental functions. Then the basic form of the reverse mode requires the storage of l times the full state vector $\mathbf{v}^{(i-1)}$ of dimension $N = \dim(\mathcal{H})$. Since both l and N may be rather large, their product may indicate an enormous memory requirement, which can however be avoided in one of two ways. First, without any conditions on the transition functions Φ_i we may apply checkpointing, as discussed in this section. The simple idea is to store only a selection of the intermediate state vectors on the first sweep through and then to recalculate the others using repeated partial forward sweeps. These alternative trade-offs between spatial and temporal complexities generate rather intricate serial or parallel reversal schedules, which are discussed in this section. For certain choices the resulting growth in memory and runtime or processor number is only logarithmically dependent on l.

Second, the dependence on l can be avoided completely if (2.12) is not a true iterative loop, but is instead either parallel or represents a contractive fixed-point iteration. In both cases the individual transitions Φ_i commute at least asymptotically and, as we will see in Section 5, they can be adjoined individually without the need for a global execution log.

4.1. Binary schedules for serial reversals

First we consider a simple reversal strategy, which already exhibits certain key characteristics, though it is never quite optimal. Suppose we have exactly $l = 16 = 2^4$ time-steps and enough memory to accommodate $s = 4$ states on a stack of checkpoints. On the first forward sweep we may then store the 0th, 8th, 12th, and 14th intermediate states in the checkpoints c_0, c_1, c_2, and c_3, respectively. This process is displayed in Figure 4.1, where the vertical axis represents *physical* time-steps and the horizontal axis *computational* time-steps or *cycles*.

After advancing to state 15 and then reversing the last step, *i.e.*, applying $\bar{\Phi}_{16}$, we may then restart the calculation at the last checkpoint c_3 containing the 14th state, and immediately perform $\bar{\Phi}_{15}$. The execution of these adjoint steps is represented by the little hooks at the end of each downward slanting run. They are assumed to take twice as long as the forward steps. Horizontal lines indicate checkpoint positions and slanted lines to the right represent forward transition steps, *i.e.*, applications of the Φ_i. After the successive reversals $\bar{\Phi}_{15}$, $\bar{\Phi}_{14}$, and $\bar{\Phi}_{13}$, the last two checkpoints can be abandoned and one of them subsequently overwritten by the 10th state, which can be reached by two steps from the 8th state, which resides in the second checkpoint.

Throughout we will let `REPS`(l) denote the *repetition count*, *i.e.*, the total number of repeated forward steps needed by some reversal schedule. Rather than formalizing the concept of a reversal schedule as a sequence of certain

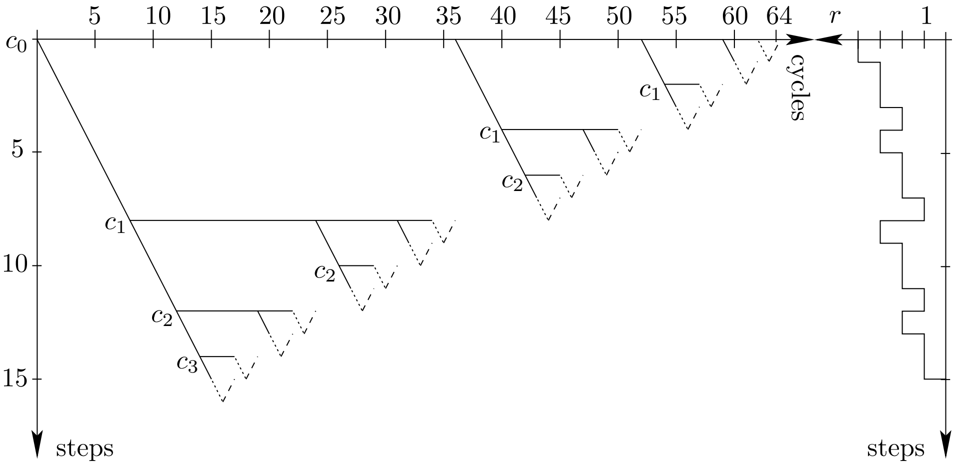

Figure 4.1. Binary serial reversal schedule for $l = 16 = 2^4$ and $s = 4$.

actions, we recommend thinking of them in terms of their graphical representation as depicted here in Figures 4.1 and 4.2. For a binary reversal of an even number of l time-steps we obtain the recurrence

$$\mathtt{REPS}(2\,l) = 2\,\mathtt{REPS}(l) + l \quad \text{with} \quad \mathtt{REPS}(1) = 0.$$

Here l represents the cost of advancing to the middle and $2\,\mathtt{REPS}(l)$ the cost of reversing the two halves. Consequently, we derive quite easily that, for all binary powers $l = 2^p$,

$$\mathtt{REPS}(l) = l\,p/2 = l\,\log_2 l/2.$$

For $l = 16$ we obtain a total of $16 \cdot 4/2 = 32$ simple forward steps plus 16 adjoint steps, which adds up to a total runtime of $32 + 2 \cdot 16 = 64$ cycles, as depicted in Figure 4.1. The profile on the right margin indicates how often each one of the physical steps is repeated, a number r that varies here between 0 and 4. The total operation count for evaluating the adjoint $\bar{F}(\mathbf{x}, \bar{\mathbf{y}})$ is by

$$\mathtt{REPS}(l)\;\mathtt{OPS}(\Phi_i) + l\,\mathtt{OPS}(\bar{\Phi}_i) \leq \bigl(\mathtt{REPS}(l)/l + 3\bigr)\,\mathtt{OPS}(F),$$

where the last inequality follows from the application of (3.14) to a single transition Φ_i. We also need $p = \log_2 l$ checkpoints, which is the maximal number of horizontal lines intersecting any imaginary vertical in Figure 4.1. Hence, both temporal and spatial complexity grow essentially by the same factor, namely,

$$2\,\frac{\mathtt{OPS}(\bar{F}(\mathbf{x}, \bar{\mathbf{y}}))}{\mathtt{OPS}(F(\mathbf{x}))} \approx \log_2 l \approx \frac{\mathtt{MEM}(\bar{F}(\mathbf{x}, \bar{\mathbf{y}}))}{\mathtt{MEM}(F(\mathbf{x}))}.$$

This common logarithmic factor must be compared with the ratios of 3 and l obtained for the operation count and memory requirement in basic reverse mode. Thus we have a drastic reduction in the memory requirement at the cost of a rather moderate increase in the operation count. On a sizable problem and a machine with a hierarchical memory the increase may not necessarily translate into an increased runtime, as data transfers to remote regions of memory can be avoided. Alternatively we may keep the wall clock time to its minimal value by performing the repeated forward sweeps using auxiliary processes on a parallel machine.

For example, if we have $\log_2 l$ processors as well as checkpoints we may execute the binary schedule depicted in Figure 4.2 for $l = 32 = 2^5$ in 64 cycles. We call such schedules *time-minimal* because both the initial forward and the subsequent reverse sweep proceed uninterrupted, requiring a total of $2l$ computational cycles. Here the maximal number of slanted lines intersecting an imaginary vertical line gives the number of processors required. There is a natural association between the checkpoint c_i and the

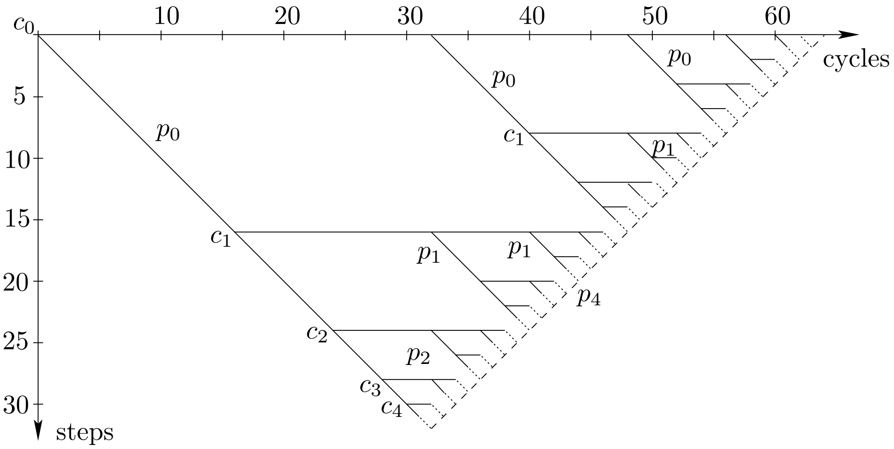

Figure 4.2. Binary parallel reversal schedule for $l = 32 = 2^5$.

processor p_i, which keeps restarting from the same state saved in c_i. For the optimal Fibonacci schedules considered below the task assignment for the individual processors is considerably more complicated. For the binary schedule executed in parallel we now have the complexity estimate

$$\frac{\texttt{CLOCK}(\bar{F}(\mathbf{x}, \bar{\mathbf{y}}))}{\texttt{CLOCK}(F(\mathbf{x}))} \approx 2 \quad \text{and} \quad \frac{\texttt{PROCS}(\bar{F}(\mathbf{x}, \bar{\mathbf{y}}))}{\texttt{PROCS}(F(\mathbf{x}))} \approx \log_2 l.$$

Here the original function evaluation may be carried out in parallel so that $\texttt{PROCS}(F(x)) > 1$, but then $\log_2 l$ times as many processors must be available for the reversal with minimal wall clock time. If not quite as many are available, we may of course compress the sequential schedule somewhat along the computing axis without reaching the absolute minimal wall clock time. Such hybrid schemes will not be elaborated here.

4.2. Binomial schedules for serial reversal

While elegant in its simplicity, the binary schedule (like other schemes that recursively partition the remaining simulation lengths by a fixed proportion) is not optimal in either the serial or parallel context. More specifically, given memory for $\log_2 l$ checkpoints, we can construct serial reversal schedules that involve significantly fewer than $l \log_2 l/2$ repeated forward steps or time-minimal parallel reversal schedules that get by with fewer than $\log_2 l$ processors. The binary schedules have another major drawback, namely they cannot be implemented online for cases where the total number of steps to be taken is not known *a priori*. This situation may arise, for example, when a differential equation is integrated with adaptive step-size, even if the period of integration is fixed beforehand. Optimal online schedules have so far only been developed for multi-processors.

The construction of serially optimal schedules is possible by variations of dynamic programming. The key observation for this decomposition into smaller subschedules is that checkpoints are *persistent*, in that they exist until all later steps to the right of them have been reversed. In Figures 4.1 and 4.2 this means graphically that all horizontal lines continue until they (almost) reach the upward slanting line representing the step reversal.

Lemma 4.1. (Checkpoint persistence) Any reversal schedule can be modified without a reduction of its length l or an increase in its repetition count such that, once established at a state j, a checkpoint stays fixed and is not overwritten until the $(j+1)$st step has been reversed, *i.e.*, $\bar{\Phi}_{j+1}$ applied. Moreover, during the 'life-span' of the checkpoint at state j, all actions occur to the right, *i.e.*, concern only states $k \geq j$.

This checkpoint persistence principle is easy to prove (Griewank 2000) under the crucial assumption that all state vectors $\mathbf{v}^{(i)}$ are indeed of the same dimension and thus require the same amount of storage. While this assumption could be violated due to the use of adaptive grids, it would be hard to imagine that the variations in size could be so large that taking the upper bound of the checkpoint size as a uniform measure would result in very significant inefficiencies. Another uniformity assumption that we will use, but which can be quite easily relaxed, is that the computational effort for all individual transitions Φ_i is the same. Thus the optimal reversal schedules depend only on the total number l of steps to be taken.

In Figure 4.1 we see that the setting of the second checkpoint c_1 effectively splits the reversal into two subproblems. After storing the initial state into c_0 and then advancing to state eight, the remaining eight physical steps are reversed using just the three checkpoints c_1, c_2 and c_3. Subsequently the first eight steps are reversed also using only three checkpoints, namely c_0, c_1, c_2, although c_3 is again available. In fact, it would be better to set c_1 at a state $\check{l}$ that solves, for $s = 4$ and $l = 16$, the dynamic programming problem

$$\texttt{REPS}(s,l) \equiv \min_{1<\check{l}<l} \big\{ \check{l} + \texttt{REPS}(s-1, l-\check{l}) + \texttt{REPS}(s, \check{l}) \big\}. \tag{4.1}$$

Here $\texttt{REPS}(s,l)$ denotes the minimal repetition count for a reversal of l steps using just s checkpoints. The three terms on the right represent the effort of, respectively, advancing $\check{l}$ steps, reversing the right subchain $[\check{l} \dots l]$ using $s-1$ checkpoints, and finally reversing the left subchain $[0 \dots \check{l}]$, again using all s checkpoints. To derive a nearly explicit formula for the function $\texttt{REPS}(s,l)$, we will consider values

$$l_r(s,l) \leq l \quad \text{for} \quad r \geq 0 < s.$$

They are defined as the number of steps $i = 1 \dots l$ that are repeated at most r

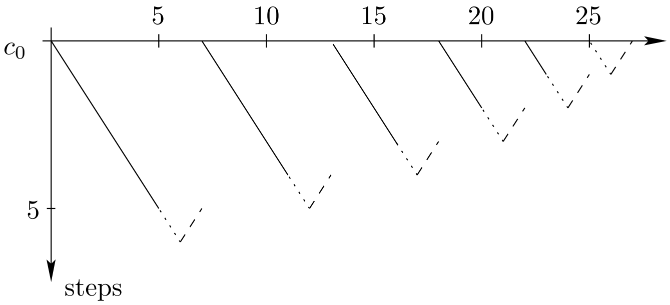

Figure 4.3. Reversal with a single checkpoint ($s = 1, l = 6$).

times, maximized over all reversals on the range $[0 \ldots l]$. By definition, the $l_r(s,l)$ are nondecreasing as functions of r, such that $l_{r+1}(s,l) \geq l_r(s,l)$. Moreover, as it turns out, these numbers have maxima

$$l_r(s) \equiv \max_{l \geq 0} l_r(s,l),$$

which are attained for all sufficiently large l.

For $s = 1$ this is easy to see. As there is just one checkpoint it must be set to the initial state, and the reversal schedule takes the trivial form depicted in Figure 4.3 for the case $l = 6$. Hence we see there is exactly one step that is never repeated, one that is repeated once, twice, and so on, so that

$$l_r(1) = r + 1 = l_r(1,l) \quad \text{for all} \quad 0 \leq r < l. \tag{4.2}$$

On the other hand, in any reversal schedule all but the final step must be repeated at least once, so that we have

$$l_0(s) = 1 = l_0(s,l) \quad \text{for all} \quad 0 \leq s < l. \tag{4.3}$$

The initial values (4.2) and (4.3) for $s = 1$ or $r = 0$ facilitate the proof of the binomial formula below by induction. The other assertions were originally established by Grimm, Pottier and Rostaing-Schmidt (1996).

Proposition 4.2.

(i) Lemma 4.1 implies that, for any reversal schedule,

$$l_r(s) \leq \beta(s,r) \equiv (s+r)!/(s!\,r!), \tag{4.4}$$

where l is the chain length and s the number of checkpoints.

(ii) The resulting repetition count is bounded below by

$$\texttt{REPS}(s,l) \geq l\, r_{\max} - \beta(s+1, r_{\max} - 1) \tag{4.5}$$

where $r_{\max} \equiv r_{\max}(s,l)$ is uniquely defined by

$$\beta(s, r_{\max} - 1) < l \leq \beta(s, r_{\max}).$$

(iii) $\mathtt{REPS}(s,l)$ does attain its lower bound (4.5), with (4.4) holding as equality for all $0 \leq r < r_{\max}$, and consequently

$$l_{r_{\max}}(s,l) = l - \beta(s, r_{\max}-1,) = l - \sum_{i=0}^{r_{\max}-1} l_r(s,l).$$

Proof. It follows from checkpoint persistence that

$$l_r(s,l) \leq \max_{1<\check{l}<l} \bigl\{ l_{r-1}(s,\check{l}) + l_r(s-1, l-\check{l}) \bigr\},$$

because all steps in the left half $[0 \ldots \check{l}]$ get repeated one more time during the initial advance to $\check{l}$. Now, by taking the maximum over l and $\check{l} < l$, we obtain the recursive bound

$$l_r(s) \leq l_{r-1}(s) + l_r(s-1).$$

Together with the initial conditions (4.2) and (4.3), we immediately arrive at the binomial bound

$$l_r(s) \leq \beta(s,r) \equiv (s+r)!/(s!\,r!),$$

which establishes (i). To prove the remaining assertions let us suppose we have a given chain length l and some serial reversal schedule using s checkpoints that repeats $\Delta\tilde{l}_r \geq 0$ steps exactly r times for $r = 0, 1 \ldots l$. Looking for the most efficient schedule, we then obtain the following constrained minimization problem:

$$\begin{aligned} &\mathrm{Min} \sum_{r=0}^{\infty} r\Delta\tilde{l}_r \quad \text{such that} \quad l = \sum_{r=0}^{\infty} \Delta\tilde{l}_r \quad \text{and} \\ &\tilde{l}_r \equiv \sum_{i=0}^{r} \Delta\tilde{l}_i \leq l_r(s) \quad \text{for all} \quad r \geq 0. \end{aligned} \tag{4.6}$$

By replacing the right sides of the inequality constraints by their upper bounds $\beta(s,r)$, we relax the problem so that the minimal value can only go down. Also relaxing the integrality constraint on the variables $\Delta\tilde{l}_r$ for $r \geq 0$ we obtain a standard LP whose minimum is obtained at

$$\begin{aligned} \Delta\tilde{l}_r &= \beta(s,r) - \beta(s,r-1) = \beta(s-1,r) \quad \text{for} \quad 0 \leq r < r_{\max} \quad \text{and} \\ \Delta\tilde{l}_{r_{\max}} &= l - \beta(s, r_{\max}-1) \geq 0 \quad \text{where} \quad \beta(s, r_{\max}-1) < l \leq \beta(s, r_{\max}). \end{aligned}$$

This solution is integral and the resulting lower bound on the cost is given by

$$\begin{aligned} \mathtt{REPS}(s,l) &\geq \bigl[l - \beta(s, r_{\max}-1)\bigr]\, r_{\max} + \sum_{j=0}^{r_{\max}-1} j\,\beta(s-1,j) \\ &= r_{\max}\, l - \beta(s+1, r_{\max}-1) \leq r_{\max}\, l, \end{aligned}$$

where the equation follows from standard binomial identities (Knuth 1973). That the lower bound $\mathtt{REPS}(s,l)$ is actually attained is established by the following construction of suitable schedules by recursion.

When $s=1$ we can only use the schedule displayed in Figure 4.3, where $\Delta\tilde{l}_r = 1$ for all $0 \le r < l$ so that $r_{\max} = l-1$ and $\mathtt{REPS} = l\,(l-1) - \beta(2, r_{\max}-1) = l\,(l-1)/2$. When $l=1$ no step needs to be repeated at all, so that $\Delta\tilde{l}_0 = 1$, $r_{\max} = 0$ and thus $\mathtt{REPS} = 0 = \beta(s+1, r_{\max}-1)$. As an induction hypothesis we suppose that a binomial schedule satisfying (4.4) and (4.5) as equalities exists for all pairs $\tilde{s} \ge 1 \le \tilde{l}$ that precede (s,l) in the lexicographic ordering. This induction hypothesis is trivially true for $s=1$ and $l=1$. There is a unique value $r_{\max}$ such that

$$\beta(s, r_{\max}-2) + \beta(s-1, r_{\max}-1) = \beta(s, r_{\max}-1) < l \tag{4.7}$$

$$\text{and} \quad l \le \beta(s, r_{\max}) = \beta(s, r_{\max}-1) + \beta(s-1, r_{\max}).$$

Then we can clearly partition l into $\check{l}$ and $l-\check{l}$ such that

$$\beta(s, r_{\max}-2) < \check{l} \le \beta(s, r_{\max}-1) \tag{4.8}$$

$$\text{and} \quad \beta(s-1, r_{\max}-1) < l-\check{l} \le \beta(s-1, r_{\max}), \tag{4.9}$$

which means that the induction hypothesis is satisfied for the two subranges $[0\ldots\check{l}]$ and $[\check{l}\ldots l]$. This concludes the proof by induction. □

Figure 4.4 displays the incremental counts

$$\Delta l_r(s) = l_r(s) - l_{r-1}(s) \approx \beta(s-1, r),$$

where the approximation holds as equality when $l_r(s)$ and $l_{r-1}(s)$ achieve their maximal values $\beta(s,r)$ and $\beta(s-1,r)$. The solid line represents an optimal schedule and the dashed line some other schedule for the given parameters $s=3$ and $l=68$. The area under the graphs is the same, as it must equal the number of steps, 68.

An optimal schedule for $l=16$ and $s=4$ is displayed in Figure 4.5. It achieves with $r_{\max}=3$ the minimal repetition count $\mathtt{REPS}(3,16) = 3 \cdot 16 - \beta(5,2) = 48 - 21 = 27$, which yields together with 16 adjoint steps a total number of 59 cycles. This compares with a total of 65 cycles for the binary schedule depicted in Figure 4.1, where the repetition count is 33 and the distribution amongst the physical steps less even. Of course, the discrepancy is larger on bigger problems.

The formula $\mathtt{REPS}(s,l) = l\,r_{\max} - \beta(s+1, r_{\max}-1)$ shows that the number $r_{\max}$, which is defined as the maximal number of times that any step is repeated, gives a very good estimate of the temporal complexity growth between $\bar{F}(\mathbf{x}, \bar{\mathbf{y}})$ and $F(\mathbf{x})$, given s checkpoints. The resulting complexity is piecewise linear, as displayed in Figure 4.6. Not surprisingly, the more

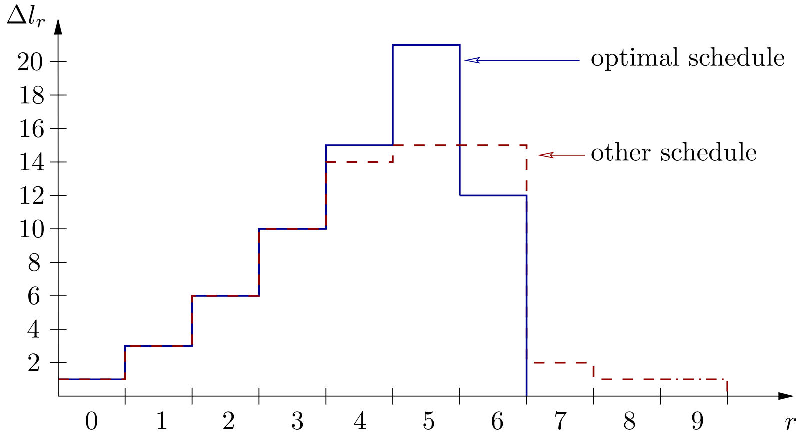

Figure 4.4. Repetition levels for $s = 3$ and $l = 68$.

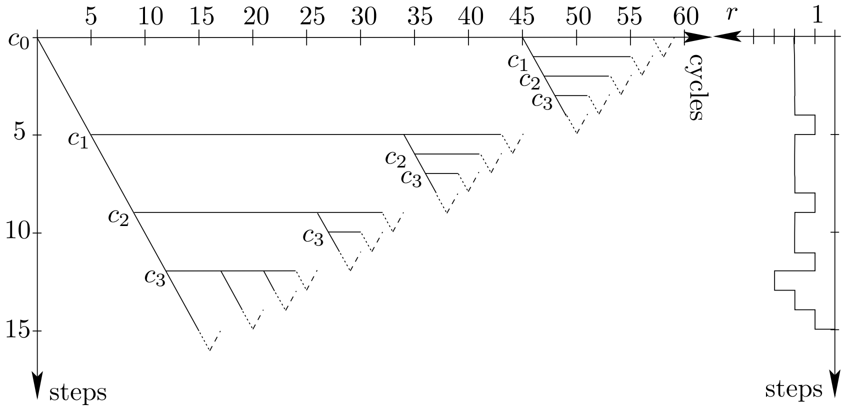

Figure 4.5. Binomial serial reversal schedule for $l = 16$ and $s = 4$.

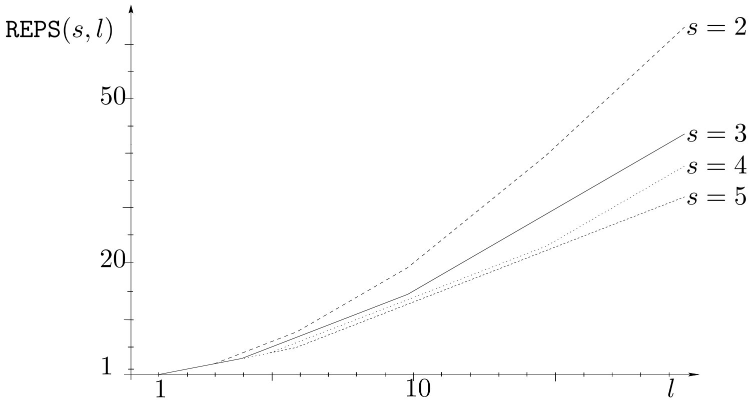

Figure 4.6. Repetition count for chain length l and checkpoint number s.

checkpoints s we have at our disposal the slower the cost grows. Asymptotically it follows from $\beta(s, r_{\max}) \approx l$ by Stirling's formula that, for fixed s, the approximate complexity growth is

$$r_{\max} \sim \frac{s}{\mathrm{e}} \sqrt[s]{l} \quad \text{and} \quad \texttt{REPS}(s,l) \sim \frac{s}{\mathrm{e}} l^{1+1/s}, \tag{4.10}$$

where e is Euler's number.

A slight complication arises when the total number of steps is not known *a priori*, so that a set of optimal checkpoint positions cannot be determined beforehand. Rather than performing an initial sweep just for the purpose of determining l, we may construct online methods that release earlier checkpoints whenever the simulation continues for longer than expected. Sternberg (2002) found that the increase of the repeated step numbers compared with the (in hindsight) optimal schemes is rarely more than 5%.

A key advantage of the binomial reversal schedule compared with the simpler binary one is that many steps are repeated exactly $r_{\max}(s,l)$ times. Since none of them are repeated more often than $r = r_{\max}$ times, we obtain the complexity bounds

$$\frac{\texttt{OPS}(\bar{F}(\mathbf{x}, \bar{\mathbf{y}}))}{\texttt{OPS}(F(\mathbf{x}))} \leq 3 + r \quad \text{and} \quad \frac{\texttt{MEM}(\bar{F}(\mathbf{x}, \bar{\mathbf{y}}))}{\texttt{MEM}(F(\mathbf{x}))} \leq 3 + s \tag{4.11}$$

provided $l \leq \beta(s,r) \equiv (s+r)!/(s!\,r!)$.

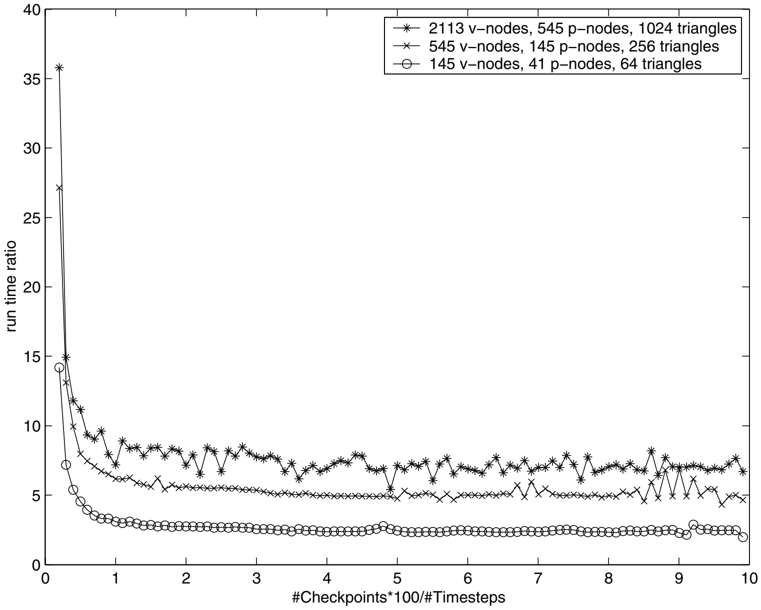

Figure 4.7. Runtime behaviour, 1000 time-steps.

The binomial reversal schedules were applied in an oceanographic study by Restrepo, Leaf and Griewank (1998), have been implemented in the software routine `revolve` (Griewank and Walther 2000), and are used in the Hilbert class library (Gockenbach, Petro and Symes 1999). Figure 4.7 reports the experimental runtime ratios on a linearized Navier–Stokes equation for the driven cavity problem. The problem was discretized using Taylor–Hood elements on 2113 velocity nodes and 545 pressure nodes. We need 38 kbytes to store one state: for details see Hinze and Walther (2002). Over 1000 time-steps the memory requirement for the basic approach of storing all intermediate states would be 38 Mbytes.

While this amount of storage may appear manageable it turns out that it can be reduced by a factor of about a hundred without any noticeable increase in runtime despite a growth factor of $r_{\max} = 8 \approx \frac{1}{\mathrm{e}}10(1000)^{1/10}$ in the repetition count. The total operation count ratio is somewhat smaller because of the constant effort for adjoint steps. By inspection of Figure 4.7 we also note that the runtime ratios are lower for coarser discretizations and thus smaller state space dimensions. This confirms the notion that the amount of data transfer from and to various parts of the memory hierarchy has become critical for the execution times.

The bound (4.11) is valid even when the assumption that $\mathtt{OPS}(\Phi_i)$ is the same for all i is not true. Only when these individual step costs vary by orders of magnitude is it worthwhile to minimize the temporal complexity of $\bar{F}(\mathbf{x}, \bar{\mathbf{y}})$ more carefully. The task of finding optimal reversal schedules for sequences of steps with nonuniform step costs is quite similar to the construction of optimal binary search trees considered by Knuth (1973). As in that setting, exact solutions can be computed with an effort of order l^2 and nearly optimal ones with an effort of order $l \log_2 l$.

The binomial reversal schedules can be generalized to the multi-step situation where

$$\mathbf{v}^{(i)} = \Phi_i(\mathbf{v}^{(i-1)}, \mathbf{v}^{(i-2)}, \ldots, \mathbf{v}^{(i-q)}) \quad \text{for some} \quad q > 1. \tag{4.12}$$

Then checkpoints consist of q consecutive states $[i-q, i)$ that are needed to advance repeatedly towards i and beyond. Under the critical uniformity assumption $\mathtt{MEM}(\Phi_i) = \mathtt{MEM}(\Phi_l)$ for $i = 1 \ldots l$, we can show that the checkpoint persistence principle still applies, so that dynamic programming is also applicable. More specifically, according to Theorem 3.1 in Walther (1999), we have for the minimal repetition count $\mathtt{REPS}(s, l, q)$

$$\lim_{l \to \infty} \frac{\mathtt{REPS}(s, l, q)}{l^{1+q/s}} = \left[\frac{(s/q)!}{q}\right]^{q/s} \approx \frac{s}{\mathrm{e}\, q^{(1+q/s)}}. \tag{4.13}$$

Exactly the same asymptotic complexity can be achieved by the one-step checkpointing schemes discussed above if they are adapted in the following way. Suppose the l original time-steps are interpreted as l/q *mega-time-*

steps between *mega-states* comprising q consecutive states $[qi-q, qi)$ for $i = 1 \dots l/q$. Here we may have to increase l formally to the next integer multiple of q. While the number of time-steps is thus divided by q, the complexity of a mega-step is of course q times that of a normal step. Moreover, since the dimension of the state space has also grown q-fold, we have to assume that the number of checkpoints s is divisible by q so that we can now keep up to s/q mega-states in memory. Then, replacing `REPS` by `REPS`$/q$, l by l/q and s by s/q in (4.10) yields asymptotically the right-hand side of (4.13). Hence we may conclude that directly exploiting multi-step structure is only worthwhile when l/q is comparatively small. Otherwise, interpreting multi-step evolutions as one-step evolutions on mega-states yields nearly optimal reversal results.

4.3. Fibonacci schedules for parallel reversals

Just as in the serial case the binary schedules discussed in Section 4.1 are also quite good but not really optimal in the parallel scenario. Moreover, they cannot be conveniently updated when l changes in the bidirectional simulation scenario discussed below. First let us briefly consider the scenario which is depicted for $l = 8$ in Figure 4.8.

By inspection, the maximal numbers of horizontal and slanted lines intersecting any vertical line are 2 and 3, respectively. This can be executed using 2 checkpoints and 3 processors. In contrast, the binary schedule for that problem defined by the lowest quarter of Figure 4.2 requires 3 checkpoints and processors each. On larger problems the gap is considerably bigger, with the numbers of checkpoints and processors each being reduced by the factor $1/\log_2[(1+\sqrt{5})/2] \approx 1.44$.

The construction of optimal schedules is again recursive, though no decomposition into completely separate subtasks is possible. For our example the optimal position for setting checkpoint c_1 is state 5, so that we have two subschedules of length 5 and 3, represented by the two shaded triangles in the second layer of Figure 4.8 from the top. However, they must be performed at last in parts simultaneously, with computing resources being passed from the bottom left triangle to the top right triangle in a rather intricate manner.

As it turns out, it is best to count both checkpoints and processors together as resources, whose number is given by ϱ. Obviously, a processor can always transmute into a checkpoint by just standing still. The maximal length of any chain that can be reverted without any interruption using ϱ resources will be denoted by l_ϱ. The resources in use during any cycle are depicted in the histograms at the right-hand side of Figure 4.8. These resource profiles are partitioned into the subprofiles corresponding to the two subschedules, and the shaded area caused by additional activities linking

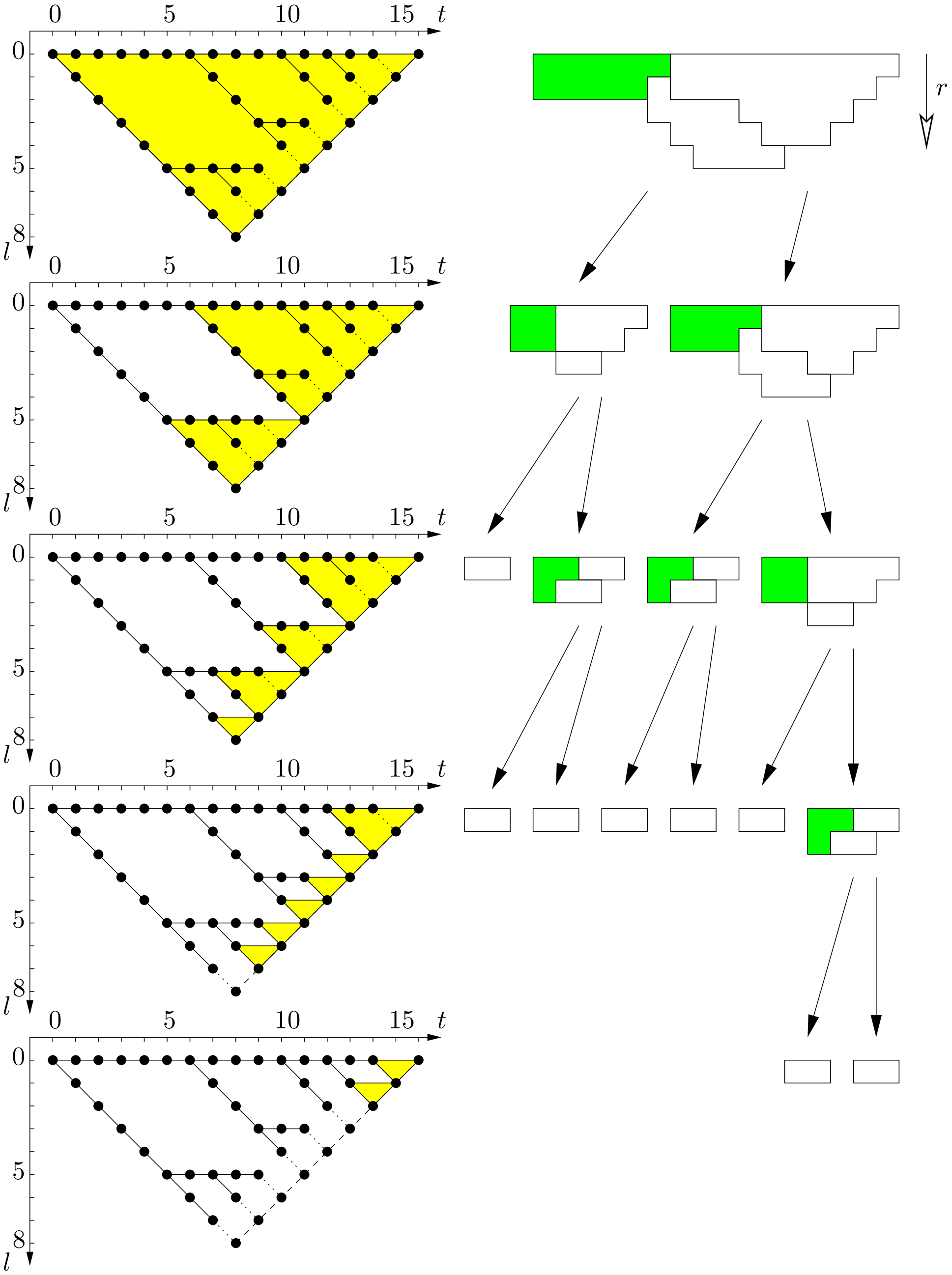

Figure 4.8. Optimal parallel reversal schedule with recursive decomposition into subschedules and corresponding resource profiles.

the two. At first, one checkpoint keeps the initial state, and one processor advances along the diagonals until it kicks off the bottom left subschedule, which itself needs at first two resources, and then for a brief period of the two cycles 8 and 9 it requires three. In cycle 10 that demand drops again to two, and one resource can be released to the top right subschedule. As we can see, the subprofiles fit nicely together and we always obtain a gradual increase toward the vertex cycles l and $l+1$, and subsequently a gentle decline.

Each subschedule in decomposed again into smaller schedules of sizes 3, 2 and ultimately 1. The fact that the schedule sizes (1, 2, 3, 5, and 8) are Fibonacci numbers is not coincidental. A rigorous derivation would require too much space, but we give the following argument, whose key observations were proved by Walther (1999).

- Not only checkpoints but also processors persist, *i.e.*, run without interruption until they reach the step to be reversed immediately (graphically that means that there are no hooks in Figures 4.2 and 4.8).
- The first checkpoint position $\check{l}$ partitions the original range $[0 \ldots l]$ into two subranges $[0 \ldots \check{l}]$ and $[\check{l} \ldots l]$ such that, if $l = l_\varrho$ is maximal with respect to a given number ϱ of resources, so are the two parts
$$\check{l} = l_{\varrho-1} \quad \text{and} \quad l - \check{l} = l_{\varrho-2} < l_{\varrho-1}.$$

As a consequence of these observations, we have

$$l_\varrho = l_{\varrho-1} + l_{\varrho-2} \quad \text{starting with} \quad l_\varrho = \varrho \quad \text{for} \quad \varrho \leq 3,$$

where the initial values l_1 and l_2 are obtained by the schedules depicted in the right corner of Figure 4.8.

Hence we obtain the Fibonacci number shifted in index by one so that according to the usual representation,

$$l_\varrho = \text{round}\Big(\big[(1+\sqrt{5})/2\big]^\rho/\sqrt{5}\Big) \quad \text{for} \quad \varrho = 1, 2 \ldots.$$

Conversely, the number of resources needed for given l is approximately

$$\varrho \approx \log_{(1+\sqrt{5})/2} l \approx 0.7202\,(2\log_2 l).$$

As was shown by Walther (1999), of the ϱ resources needed by the Fibonacci schedule at most one more than half must be processors. Consequently, compared with the binary schedule the optimal parallel schedule achieves a reduction in both resources by about 28%. While this gain is not really enormous, the difference between the binary and Fibonacci reversals becomes even more pronounced if we consider the scenario where l is not known *a priori*.

Imagine we have a simulation that runs forward in time with a fair bit of diffusion or energy dissipation, so that we cannot integrate the model equations backwards. In order to be able to run the 'movie' backwards nevertheless, we take checkpoints at selected intermediate states and then repeat partial forward runs using auxiliary processors. The reversal is supposed to start instantaneously and to run at exactly the same speed as the forward simulation. Moreover, the user can switch directions at any time. As it turns out, the Fibonacci schedules can be updated adaptively so that ϱ resources, with roughly half of them processors, suffice as long as l does not exceed l_ϱ. When that happens, one extra resource must be added so that $\varrho \mathrel{+}= 1$.

An example using maximally 5 processors and 3 checkpoints is depicted in Figure 4.9. First the simulation is advanced from state 0 to state $48 < 55 = l_9$. In the 48th transition step 6 checkpoints have been set and 2 processors are running. Then the 'movie' is run backwards at exactly the same speed until state $32 < 34 = l_8$, at which point 5 processors are running and 3 checkpoints are in memory. Then we advance again until state 40, reverse once more and so on. The key property of the Fibonacci numbers used in these bidirectional simulation schedules is that, by the removal of a checkpoint between gaps whose size equals neighbouring Fibonacci numbers, the size of the new gap equals the next-larger Fibonacci numbers.

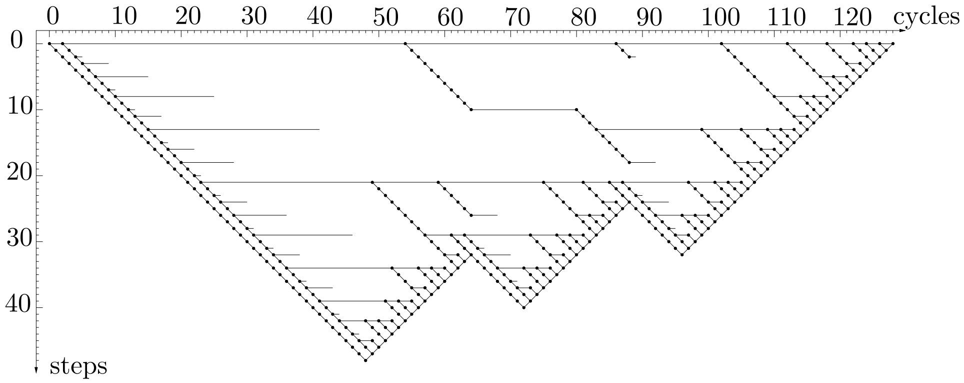

Figure 4.9. Bidirectional simulation using up to 9 processors and checkpoints.

4.4. Pantoja's algorithm and the Riccati equation

To compute a Newton step $\Delta x(t)$ of the control $x(t)$ for the optimality condition $\bar{x}(t) = 0$ associated with (3.17)–(3.21), we have to solve a linear–quadratic regulator problem that approximates the given nonlinear control problem along the current trajectory. Its solution requires the integration of the matrix Riccati equation (Maurer and Augustin 2001, Caillau and

Noailles 2001), backward from a terminal condition involving the objective Hessian $\nabla^2\psi$. After this intermediate, reverse sweep, a third, forward sweep yields the control correction $\Delta x(t)$. Pantoja's algorithm (Pantoja 1988) may be viewed as a discrete variant of these coupled ODEs (Dunn and Bertsekas 1989). It was originally derived as a clever way of computing the Newton step on the equality-constrained finite-dimensional problem obtained by discretizing (3.17)–(3.21). Here, we are interested primarily in the dimensions of the state spaces for the various evaluation sweeps and the information that flows between them. Assuming that the control dimension $p = \dim(\mathbf{x})$ is much smaller than the original state space dimension $q = \dim(\mathbf{v})$, we arrive at the scenario sketched in Figure 4.10. Here $B(t)$ is a $p \times q$ matrix path that must be communicated from the intermediate to the third sweep.

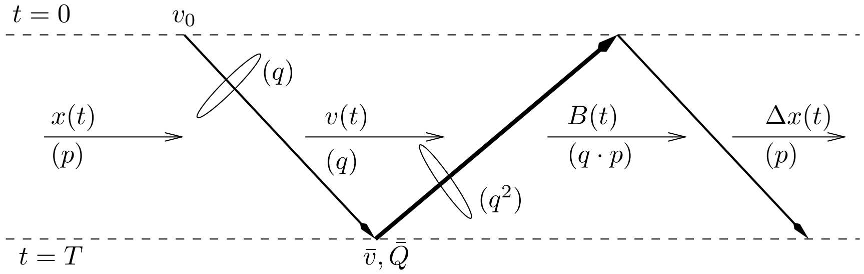

Figure 4.10. Nested reversal for Riccati/Pantoja computation of Newton step.

The horizontal lines represent information flow between the three sweeps that are represented by slanted lines. The two ellipses across the initial and intermediate sweep represent checkpoints, and are annotated by the dominant dimension of the states that need to be saved for a restart on that sweep. Hence we have thin checkpoints of size q on the initial, forward sweep, and thick checkpoints of size q^2 on the immediate, reverse sweep. A simple-minded 'record all' approach would thus require memory of order $l\,q^2$, where l is now the number of discrete time-steps between 0 and T. The intermediate sweep is likely to be by far the most computationally expensive, as it involves matrix products and factorizations.

The final sweep again runs forward in time and again has a state dimension $q+p$, which will be advanced in spurts as information from the intermediate sweep becomes available. Christianson (1999) has suggested a two-level checkpointing scheme, which reduces the total storage from order $q^2\,l$ for a simple log-all approach to $q^2\sqrt{l}$. Here checkpoints are placed every $\sqrt{l}$ time-steps along the initial and the intermediate sweep. The subproblems of length $\sqrt{l}$ are then treated with complete logging of all intermediate states. More sophisticated serial and parallel reversal schedules are currently under development and will certainly achieve a merely logarithmic dependence of the temporal and spatial complexity on l.

4.5. Section summary

The basic reverse mode makes both the operation count $\texttt{OPS}\big(\bar{F}(\mathbf{x},\bar{\mathbf{y}})\big)$ and the memory requirement $\texttt{MEM}\big(\bar{F}(\mathbf{x},\bar{\mathbf{y}})\big)$ for adjoints proportional to the operation count $\texttt{OPS}\big(F(\mathbf{x})\big) \geq \texttt{MEM}\big(F(\mathbf{x})\big)$. By serial checkpointing on explicitly time-dependent problems, both ratios

$$\texttt{OPS}\big(\bar{F}(\mathbf{x},\bar{\mathbf{y}})\big)/\texttt{OPS}\big(F(\mathbf{x})\big) \quad \text{and} \quad \texttt{MEM}\big(\bar{F}(\mathbf{x},\bar{\mathbf{y}})\big)/\texttt{MEM}\big(F(\mathbf{x})\big)$$

can be made proportional to the logarithm of the number of time-steps, which equals roughly $\texttt{OPS}\big(F(\mathbf{x})\big)/\texttt{MEM}\big(F(\mathbf{x})\big)$. A similar logarithmic number of checkpoints and processors suffices to maintain the minimal wall clock ratio

$$\texttt{CLOCK}\big(F(\mathbf{x},\bar{\mathbf{y}})\big)/\texttt{CLOCK}\big(F(\mathbf{x})\big) \in [1,2]$$

that is obtained theoretically when all intermediate results can be stored. In practice the massive data movements may well lead to significant slowdown. Parallel reversal schedules can be implemented online so that, at any point in time the direction of simulation can be reversed without any delay and repeatedly.

5. Differentiating iterative solvers

Throughout the previous sections we implicitly made the crucial assumption that the trajectory traced out by the forward simulation actually matters. Otherwise the taking of checkpoints, and multiple restarts, would not be worth the trouble at all.

Many large-scale computations involve pseudo-time-stepping for solving systems of equations. The resulting solution vector can then be used to evaluate certain performance measures whose adjoints might be required for design optimization. In that case, a mechanical application of the adjoining process developed in the previous sections may still yield reasonably accurate reduced gradients. However, the accuracy achieved is nearly impossible to control and the cost is unnecessarily high, especially in terms of memory space. With regard to derivative accuracy, we should first recall that the derivatives obtained in the forward mode are, up to rounding errors, identical to the ones obtained in the (mechanical) reverse mode, while the corresponding computational cost can differ widely, of course. Therefore we begin this section with an analysis of the application of the forward mode to a contractive fixed-point iteration.

5.1. *Black box differentiation in forward mode*

Let us consider a loop of the form (2.11) with the vector $\mathbf{v}^{(i)} = (\mathbf{x}, \mathbf{z}_i, \mathbf{y}) \in X \times Z \times Y = H$ partitioned into the vector of independent variables $\mathbf{x}$, the proper state vector $\mathbf{z}_i$ and the vector of dependent variables $\mathbf{y}$. While $\mathbf{x}$ is assumed constant throughout the iteration, each vector $\mathbf{y}$ is a simple output, or function, and may therefore not affect any of the other values. Then we may write

$$\mathbf{z}_0 = \mathbf{z}_0(\mathbf{x}), \quad \mathbf{z}_i = G(\mathbf{x}, \mathbf{z}_{i-1}) \text{ for } i = 1 \ldots l, \quad \mathbf{y} = F(\mathbf{x}, \mathbf{z}_l) \tag{5.1}$$

for suitable functions

$$G : X \times Z \mapsto Z \quad \text{and} \quad F : X \times Z \mapsto Y.$$

The statement $\mathbf{z}_0 = \mathbf{z}_0(\mathbf{x})$ is meant to indicate that $\mathbf{z}_0$ is suitably initialized given the value of $\mathbf{x}$. In many applications there might be an explicit dependence of G on the iterations counter i so that, in effect, $\mathbf{z}_i = G^{(i)}(\mathbf{x}, \mathbf{z}_{i-1})$, a situation that was considered in Campbell and Hollenbeck (1996) and Griewank and Faure (2002). For simplicity we will assume here that the iteration is stationary in that $G^{(i)} = G$, an assumption that was also made in the fundamental papers by Gilbert (1992) and Christianson (1994). To assure convergence to a fixed point we make the following assumption.

Contractivity Assumption. The equation

$$\mathbf{z} = G(\mathbf{x}, \mathbf{z})$$

has a solution $\mathbf{z}_* = \mathbf{z}_*(\mathbf{x})$, and there are constants $\delta > 0$ and $\rho < 1$ such that

$$\|\mathbf{z} - \mathbf{z}_*(\mathbf{x})\| \le \delta \quad \Rightarrow \quad \|G_{\mathbf{z}}(\mathbf{x}, \mathbf{z})\| \le \rho < 1, \tag{5.2}$$

where $G_{\mathbf{z}} \equiv \partial G / \partial \mathbf{z}$, represents the Jacobian of G with respect to $\mathbf{z}$ and $\|G_{\mathbf{z}}\|$ denotes a matrix norm that must be consistent with a corresponding vector norm $\|\mathbf{z}\|$ on Z.

According to Ostrowski's theorem (see Propositions 10.1.3 and 10.1.4 in Ortega and Rheinboldt (1970)), it follows immediately that the initial condition $\|\mathbf{z}_0 - \mathbf{z}_*\| < \delta$ implies Q-linear convergence in that

$$Q\{\mathbf{z}_i - \mathbf{z}_*\} \equiv \limsup_{i \to \infty} \|\mathbf{z}_i - \mathbf{z}_*\| / \|\mathbf{z}_{i-1} - \mathbf{z}_*\| \le \rho. \tag{5.3}$$

Another consequence of our contractivity assumption is that 1 is not an eigenvalue of $G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*)$, so that, by the implicit function theorem, the locally unique solution $\mathbf{z}_* = \mathbf{z}_*(\mathbf{x})$ has the derivative

$$\partial \mathbf{z}_* / \partial \mathbf{x} = [I - G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*)]^{-1} G_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_*). \tag{5.4}$$

For a given tangent $\dot{\mathbf{x}}$, we obtain the directional derivative $\dot{\mathbf{z}}_* \equiv [\partial \mathbf{z}_* / \partial \mathbf{x}] \dot{\mathbf{x}}$.

We continue to use such restrictions to one-dimensional subspaces of domain and (dual) range in order to stay within matrix-vector notation as much as possible. The resulting vector

$$\dot{\mathbf{y}}_* = \dot{F}(\mathbf{x}, \mathbf{z}_*, \dot{\mathbf{x}}, \dot{\mathbf{z}}_*) \equiv F_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_*)\dot{\mathbf{x}} + F_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*)\dot{\mathbf{z}}_*$$

represents a directional derivative $f'(\mathbf{x})\dot{\mathbf{x}}$ of the *reduced function*

$$f(\mathbf{x}) \equiv F(\mathbf{x}, \mathbf{z}_*(\mathbf{x})). \tag{5.5}$$

Now the crucial question is whether and how this implicit derivative can be evaluated at least approximately by AD. The simplest approach is to differentiate the loop (5.1) in black box fashion.

Provided not only G but also F is differentiable in some neighbourhood of the fixed point $(\mathbf{x}, \mathbf{z}_*(\mathbf{x}))$, we obtain by differentiation in the forward mode

$$\dot{\mathbf{z}}_0 = 0, \quad \dot{\mathbf{z}}_i = \dot{G}(\mathbf{x}, \mathbf{z}_{i-1}, \dot{\mathbf{x}}, \dot{\mathbf{z}}_{i-1}) \text{ for } i = 1 \ldots l, \quad \dot{\mathbf{y}} = \dot{F}(\mathbf{x}, \mathbf{z}_l, \dot{\mathbf{x}}, \dot{\mathbf{z}}_l), \tag{5.6}$$

where

$$\dot{G}(\mathbf{x}, \mathbf{z}, \dot{\mathbf{x}}, \dot{\mathbf{z}}) \equiv G_{\mathbf{x}}(\mathbf{x}, \mathbf{z})\dot{\mathbf{x}} + G_{\mathbf{z}}(\mathbf{x}, \mathbf{z})\dot{\mathbf{z}} \tag{5.7}$$

and $\dot{F}$ is defined analogously. Since the derivative recurrence (5.6) is merely a linearization of the original iteration (5.1), the contractivity is inherited and we obtain for any initial $\dot{\mathbf{z}}_0$ the R-linear convergence result

$$R\{\dot{\mathbf{z}}_i - \dot{\mathbf{z}}_*\} \equiv \limsup_{i\to\infty} \sqrt[i]{\|\dot{\mathbf{z}}_i - \dot{\mathbf{z}}_*\|} \leq \rho. \tag{5.8}$$

This result was originally established by Gilbert (1992) and later generalized by Griewank, Bischof, Corliss, Carle and Williamson (1993) to a much more general class of Newton-like methods. We may abbreviate (5.8) to $\dot{\mathbf{z}}_i = \dot{\mathbf{z}}_* + \tilde{O}(\rho^i)$ for ease of algebraic manipulation, and it is an immediate consequence for the reduced function $f(\mathbf{x})$ defined in (5.5) that

$$F(\mathbf{x}, \mathbf{z}_i) = f(\mathbf{x}) + \tilde{O}(\rho^i) \quad \text{and} \quad \dot{F}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i) = f'(\mathbf{x})\dot{\mathbf{x}} + \tilde{O}(\rho^i).$$

As discussed by Ortega and Rheinboldt (1970), R-linear convergence is a little weaker than Q-linear convergence, in that successive discrepancies $\|\dot{\mathbf{z}}_i - \dot{\mathbf{z}}_*\|$ need not go down monotonically but must merely decline on average by the factor ρ. This lack of monotonicity comes about because the leading term on the right-hand side of (5.6), as defined in (5.7), may also approach its limit $G_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_*)\dot{\mathbf{x}}$ in an irregular fashion. A very similar effect occurs for Lagrange multipliers in nonlinear programming. Since (5.3) implies by the triangle inequality

$$\limsup_{i\to\infty} \|\mathbf{z}_i - \mathbf{z}_*\| \big/ \|\mathbf{z}_i - \mathbf{z}_{i-1}\| \leq \frac{\rho}{1-\rho},$$

we may use the last step-size $\|\mathbf{z}_i - \mathbf{z}_{i-1}\|$ as an estimate for the remaining discrepancy $\|\mathbf{z}_i - \mathbf{z}_*\|$. In contrast, this is not possible for R-linearly converging sequences, so the last step-size $\|\dot{\mathbf{z}}_i - \dot{\mathbf{z}}_{i-1}\|$ does not provide a reliable indication of the remaining discrepancy $\|\dot{\mathbf{z}}_i - \dot{\mathbf{z}}_*\|$.

In order to gauge whether the current value $\dot{\mathbf{z}}_i$ is a reasonable approximation to $\dot{\mathbf{z}}_*$, we recall that the latter is a solution of the *direct sensitivity* equation

$$[I - G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*)]\dot{\mathbf{z}} = G_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_*)\dot{\mathbf{x}}, \tag{5.9}$$

which is a mere rewrite of (5.4). Hence it follows under our assumptions that, for any candidate pair $(\mathbf{z}, \dot{\mathbf{z}})$,

$$\big(\|\mathbf{z} - \mathbf{z}_*\| + \|\dot{\mathbf{z}} - \dot{\mathbf{z}}_*\|\big) = O\big(\|\mathbf{z} - G\,(\mathbf{x}, \mathbf{z})\| + \|\dot{\mathbf{z}} - \dot{G}\,(\mathbf{x}, \mathbf{z}, \dot{\mathbf{x}}, \dot{\mathbf{z}})\|\big), \tag{5.10}$$

with $\dot{G}$ as defined in (5.7). The directional derivative $\dot{G}(\mathbf{x}, \mathbf{z}, \dot{\mathbf{x}}, \dot{\mathbf{z}})$ can be computed quite easily in the forward mode of automatic differentiation so that the residual on the right-hand side of (5.10) is constructively available.

Hence we may execute (5.1) and (5.6) simultaneously, and stop the iteration when not only $\|\mathbf{z}_i - G(\mathbf{x}, \mathbf{z}_i)\|$ but also $\|\dot{\mathbf{z}}_i - \dot{G}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i)\|$ is sufficiently small. The latter condition requires a modification of the stopping criterion, but otherwise the directional derivative $\dot{\mathbf{z}}_i \approx \dot{\mathbf{z}}_* = \dot{\mathbf{z}}_*(\mathbf{x}, \dot{\mathbf{x}})$ and the resulting $\dot{\mathbf{y}}_i = \dot{F}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i) \approx \dot{\mathbf{y}}_*$ can be obtained by black box differentiation of the original iterative solver.

Sometimes G takes the form

$$G(\mathbf{x}, \mathbf{z}) = \mathbf{z} - P(\mathbf{x}, \mathbf{z}) \cdot H(\mathbf{x}, \mathbf{z}),$$

with $H(\mathbf{x}, \mathbf{z}) = 0$ the state equation to be solved and $P(\mathbf{x}, \mathbf{z}) \approx H_{\mathbf{z}}^{-1}(\mathbf{x}, \mathbf{z})$ a suitable preconditioner. The optimal choice $P(\mathbf{x}, \mathbf{z}) = H_{\mathbf{z}}^{-1}(\mathbf{x}, \mathbf{z})$ represents Newton's method but can rarely be realized exactly on large-scale problems. Anyway we find

$$\dot{\mathbf{z}} - \dot{G}(\mathbf{x}, \mathbf{z}, \dot{\mathbf{x}}, \dot{\mathbf{z}}) = \dot{\mathbf{z}} - P(\mathbf{x}, \mathbf{z})\left[H_{\mathbf{x}}(\mathbf{x}, \mathbf{z})\,\dot{\mathbf{x}} + H_{\mathbf{z}}(\mathbf{x}, \mathbf{z})\,\dot{\mathbf{z}}\right] - \dot{P}(\mathbf{x}, \mathbf{z})\,H(\mathbf{x}, \mathbf{z}),$$

where

$$\dot{P}(\mathbf{x}, \mathbf{z}) = P_{\mathbf{x}}(\mathbf{x}, \mathbf{z})\,\dot{\mathbf{x}} + P_{\mathbf{z}}(\mathbf{x}, \mathbf{z})\,\dot{\mathbf{z}},$$

provided $P(\mathbf{x}, \mathbf{z})$ is differentiable at all. As $H(\mathbf{x}, \mathbf{z}_i)$ converges to zero, the second term $\dot{P}(\mathbf{x}, \mathbf{z})\,H(\mathbf{x}, \mathbf{z}_i)$ could, and probably should, be omitted when evaluating $\dot{G}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i)$. This applies in particular when $P(\mathbf{x}, \mathbf{z})$ is not even continuously differentiable, for example due to pivoting in a preconditioner based on an incomplete LU factorization. Setting $\dot{P} = 0$ does not affect the R-linear convergence result (5.8), and may reduce the cost of the derivative iteration. However, it does require separating the preconditioner P from the residual H, which may not be a simple task if the whole iteration function G is incorporated in a legacy code. In Figure 5.1 the curves labelled 'state equation residual' and 'direct derivative residual' represent $\|\mathbf{z}_i - G(\mathbf{x}, \mathbf{z}_i)\|$ and $\|\dot{\mathbf{z}}_i - \dot{G}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i)\|$, respectively. The other two residuals are explained in the following subsection.

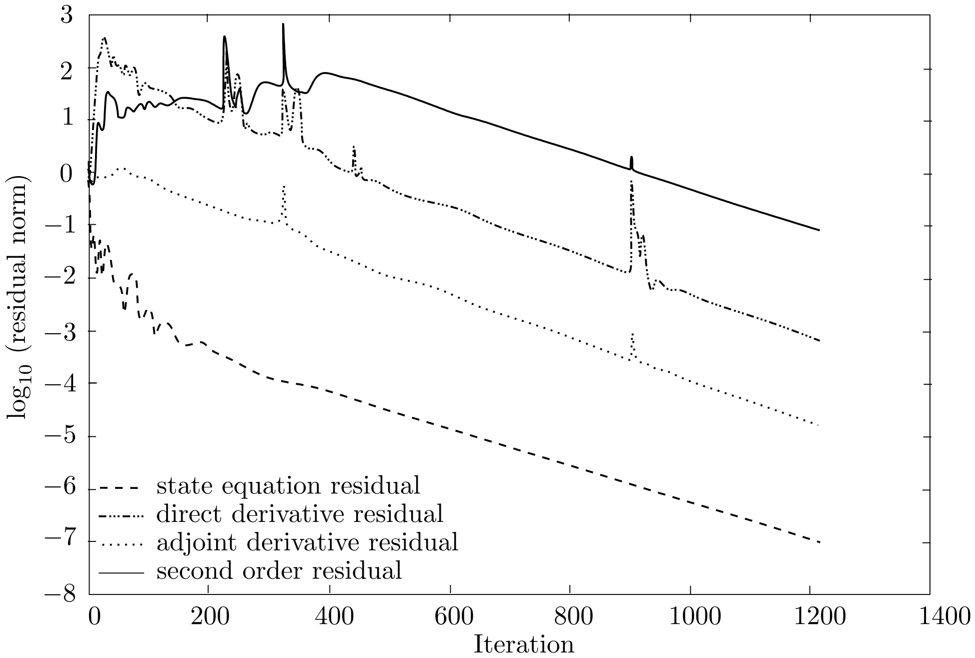

Figure 5.1. History of residuals on $2D$ Euler solver, from Griewank and Faure (2002).

While the contractivity assumption (5.2) seems quite natural, it is by no means always satisfied. For example, conjugate gradient and other general Krylov space methods cannot be written in the form (5.1) with an iteration function $G(\mathbf{x}, \mathbf{z})$ that has a bounded derivative with respect to $\mathbf{z}$. The problem is that the current residual $H(\mathbf{z}, \mathbf{x})$ often occurs as a norm or in inner products in denominators, so that $G_{\mathbf{z}}(\mathbf{z}, \mathbf{x})$ turns out to be unbounded in the vicinity of a fixed point $\mathbf{z}_* = \mathbf{z}_*(\mathbf{x})$. The same is also true for quasi-Newton methods based on secant updating, but a rather careful analysis showed that (5.8) is still true if (5.2) is satisfied initially (Griewank *et al.* 1993). In practice R-linear convergence of the derivatives has been observed for other Krylov subspace methods, but we know of no proof that this must occur under suitable assumptions.

5.2. Two-phase and adjoint differentiation

Many authors (Ball 1969, Cappelaere, Elizondo and Faure 2001) have advocated the following approach, for which directives are included in TAMC (Giering and Kaminski 2000) and possibly other AD tools. Rather than differentiating the fixed point iteration (5.1) itself, we may let it run undifferentiated until the desired solution accuracy has been obtained, and then solve the sensitivity equation (5.9) in a second phase. Owing to its linearity, this problem looks somewhat simpler than the original task of solving the nonlinear state equation $\mathbf{z} = G(\mathbf{x}, \mathbf{z})$. When G represents a Newton step we have asymptotically $\rho = 0$, and the solution of (5.9) can be achieved in a single step

$$\dot{\mathbf{z}} = \dot{G}(\mathbf{x}, \mathbf{z}, \dot{\mathbf{x}}, 0) = -P(\mathbf{x}, \mathbf{z})\, H_{\mathbf{x}}(\mathbf{x}, \mathbf{z})\, \dot{\mathbf{x}}$$

where $\mathbf{z} \approx \mathbf{z}_*$ represents the final iterate of the state vector. Also, the simplified iteration may be applied since $\dot{P}$ is multiplied by $H(\mathbf{x}, \mathbf{z}) \approx 0$. When G represents an inexact version of Newton's method based on an iterative linear equation solver, it seems a natural idea to apply exactly the same method to the sensitivity equation (5.9). This may be quite economical because spectral properties and other information, that is known *a priori* or gathered during the first phase iteration, may be put to good use once more. Of course, we must be willing and able to modify the code by hand unless AD provides a suitable tool (Giering and Kaminski 2000). The ability to do this would normally presume that a fairly standard iterative solver, for example of Krylov type, is in use. If nothing is known about the solver except that it is assumed to represent a contractive fixed point iteration, we may still apply (5.6) with $\mathbf{z}_{i-1} = \mathbf{z}$ fixed so that

$$\dot{\mathbf{z}}_i = G_{\mathbf{x}}(\mathbf{x}, \mathbf{z})\dot{\mathbf{x}} + G_{\mathbf{z}}(\mathbf{x}, \mathbf{z})\dot{\mathbf{z}}_i \quad \text{for} \quad i = 1 \ldots l. \tag{5.11}$$

Theoretically, the AD tool could exploit the constancy of $\mathbf{z}$ to avoid the repeated evaluations of certain intermediates. In effect we apply the linearization of the last state space iteration to propagate derivatives forward. The idea of just exploiting the linearization of the last step has actually been advocated more frequently for evaluating adjoints of iterative solvers (Giering and Kaminski 1998, Christianson 1994).

To elaborate on this we first need to derive the adjoint of the fixed point equation $\mathbf{z} = G(\mathbf{z}, \mathbf{x})$. It follows from (5.4) by the chain rule that the total derivative of the reduced response function defined in (5.5) is given by

$$f'(\mathbf{x}) = F_{\mathbf{x}} + F_{\mathbf{z}} \left[I - G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*)\right]^{-1} G_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_*).$$

Applying a weighting functional $\bar{\mathbf{y}}$ to $\mathbf{y}$, we obtain the adjoint vector

$$\bar{\mathbf{x}}_* = \bar{\mathbf{y}} f'(\mathbf{x}) = \bar{\mathbf{y}}\, F_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_*) + \bar{\mathbf{g}}_* G_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_*), \tag{5.12}$$

where

$$\bar{\mathbf{g}}_* \equiv \bar{\mathbf{z}}_* \left[I - G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*)\right]^{-1} \quad \text{with} \quad \bar{\mathbf{z}}_* \equiv \bar{\mathbf{y}}\, F_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*). \tag{5.13}$$

While the definition of $\bar{\mathbf{z}}_*$ follows our usual concept of an adjoint vector, the role of $\bar{\mathbf{g}}_* \in Z$ warrants some further explanation. Suppose we introduce an additive perturbation $\mathbf{g}$ of G so that we have the system

$$\mathbf{z} = G(\mathbf{z}, \mathbf{x}) + \mathbf{g}, \quad \mathbf{y} = F(\mathbf{z}, \mathbf{x}).$$

Then it follows from the implicit function theorem that $\bar{\mathbf{g}}_*$, as given by (5.13), is exactly the gradient of $\bar{\mathbf{y}}\,\mathbf{y}$ with respect to $\mathbf{g}$ evaluated at $\mathbf{g} = 0$. In other words $\bar{\mathbf{g}}_*$ is the vector of Lagrange multipliers of the constraint $\mathbf{z} = G(\mathbf{x}, \mathbf{z})$ given the objective function $\bar{\mathbf{y}}\,\mathbf{y}$. From (5.13) it follows directly that $\bar{\mathbf{g}}_*$ is the unique solution of the adjoint sensitivity equation

$$\bar{\mathbf{g}}\big[I - G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*)\big] = \bar{\mathbf{z}}_*. \tag{5.14}$$

Under our contractivity assumption (5.2) on $G_{\mathbf{z}}$, the corresponding fixed point iteration

$$\bar{\mathbf{g}}_{i+1} = \bar{\mathbf{z}}_* + \bar{\mathbf{g}}_i\, G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*) \tag{5.15}$$

is also convergent, whence

$$Q\big\{\bar{\mathbf{g}}_i - \bar{\mathbf{g}}_*\big\} = \limsup_{i\to\infty} \big\|\bar{\mathbf{g}}_i - \bar{\mathbf{g}}_*\big\| \big/ \big\|\bar{\mathbf{g}}_{i-1} - \bar{\mathbf{g}}_*\big\| \leq \rho. \tag{5.16}$$

In the notation of Section 3, the right-hand side of (5.15) can be evaluated as

$$\bar{\mathbf{g}}_i G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*) \equiv \bar{G}(\mathbf{x}, \mathbf{z}_*, \bar{\mathbf{g}}_i)$$

by a reverse sweep on the procedure for evaluating $G(\mathbf{x}, \mathbf{z}_*)$. In many applications, this adjoining of the iteration function $G(\mathbf{x}, \mathbf{z})$ is no serious problem, and requires only a moderate amount of memory space. When $G = I - P\,H$ with $P = H_{\mathbf{z}}^{-1}$ representing Newton's method, we have $G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_*) = 0$, and the equation (5.15) reduces to $\mathbf{g}_1 = \bar{\mathbf{z}}_* = \mathbf{g}_*$. The relation between (5.15) and mechanical adjoining of original fixed point iteration was analysed carefully in Giles (2001) for linear time-variant systems and their discretizations.

Both (5.11) and (5.15) assume that we really have a contractive, single-step iteration. If in fact $\mathbf{z}_i = G^{(i)}(\mathbf{x}, \mathbf{z}_i)$ and each individual $G^{(i)}$ only contracts the solution error on some subspace, as occurs for example in multi-grid methods, then the linearization of what happens to be the last iteration will not provide a convergent solver for the direct or adjoint sensitivity equation.

5.3. *Adjoint-based error correction*

Whenever we have computed approximate solutions $\mathbf{z}$ and $\bar{\mathbf{z}}$ to the state equation (5.2) and the adjoint sensitivity equation (5.14), we have in analogy to (5.10) the error bound

$$\big(\|\mathbf{z} - \mathbf{z}_*\| + \|\bar{\mathbf{z}} - \bar{\mathbf{z}}_*\|\big) = O\big(\|\mathbf{z} - G\,(\mathbf{x}, \mathbf{z})\| + \|\bar{\mathbf{z}} - \bar{G}\,(\mathbf{x}, \mathbf{z}, \bar{\mathbf{x}}, \bar{\mathbf{z}})\|\big). \tag{5.17}$$

Using the approximate adjoint solution $\bar{\mathbf{g}}$ we may improve the estimate for the weighted response $\bar{\mathbf{y}}\,F(\mathbf{x}, \mathbf{z}_*) = \bar{\mathbf{y}}\,f(\mathbf{x})$ by the correction $\bar{\mathbf{g}}(G(\mathbf{z}, \mathbf{x}) - \mathbf{z})$. More specifically, we have for Lipschitz continuously differentiable F and G the Taylor expansion

$$F(\mathbf{x}, \mathbf{z}_*) = F(\mathbf{x}, \mathbf{z}) + F_{\mathbf{z}}(\mathbf{x}, \mathbf{z})(\mathbf{z}_* - \mathbf{z}) + O\big(\|\mathbf{z} - \mathbf{z}_*\|^2\big)$$

and

$$0 = \mathbf{z}_* - G(\mathbf{x}, \mathbf{z}_*)) = \mathbf{z} - G(\mathbf{x}, \mathbf{z}) + \big[I - G_{\mathbf{z}}(\mathbf{x}, \mathbf{z})\big](\mathbf{z}_* - \mathbf{z}) + O\big(\|\mathbf{z} - \mathbf{z}_*\|^2\big).$$

By subtracting $\bar{\mathbf{g}}$ times the second equation from the first, we obtain the estimate

$$\begin{aligned}\bar{\mathbf{y}}\, F(\mathbf{x}, \mathbf{z}) &- \bar{\mathbf{g}}\big[\mathbf{z} - G(\mathbf{x}, \mathbf{z})\big] - \bar{\mathbf{y}}\, f(\mathbf{x}) \\ &= O\big(\|\bar{\mathbf{g}}\big[I - G_{\mathbf{z}}(\mathbf{x}, \mathbf{z})\big] - \bar{\mathbf{z}}\|\; \|\mathbf{z} - \mathbf{z}_*\| + \|\mathbf{z} - \mathbf{z}_*\|^2\big).\end{aligned}$$

Now, if $\mathbf{z} = \mathbf{z}_i$ and $\bar{\mathbf{g}} = \bar{\mathbf{g}}_i$ are generated according to (5.1) and (5.15), we derive from (5.3) and (5.16) that

$$\bar{\mathbf{y}}\, F(\mathbf{x}, \mathbf{z}_i) - \bar{\mathbf{g}}_i\big[\mathbf{z}_i - G(\mathbf{x}, \mathbf{z}_i)\big] = \bar{\mathbf{y}}\, f(\mathbf{x}) + \tilde{O}(\rho^{2i}).$$

Hence the corrected estimate converges twice as fast as the uncorrected one. It has been shown by Griewank and Faure (2002) that, for linear G and F and zero initializations $\mathbf{z}_0 = 0$ and $\bar{\mathbf{g}}_0 = 0$, the ith corrected estimate is exactly equal to the $2\,i$th uncorrected value $\bar{\mathbf{y}}\, F(\mathbf{x}, \mathbf{z}_i)$. The same effect occurs when the initial $\mathbf{z}_0$ is quite good compared to $\bar{\mathbf{z}}_0$, since the adjoint iteration (5.15) is then almost linear. This effect is displayed in Figure 5.2, where the normal (uncorrected) value lags behind the double (adjoint corrected) one by almost exactly the factor 2. Here the weighted response $\bar{\mathbf{y}}F$ was the drag coefficient of the NACA0012 airfoil.

In our setting the discrepancies $\mathbf{z} - \mathbf{z}_*$ and $\bar{\mathbf{g}} - \mathbf{g}_*$ come about through iterative equation solving. The same duality arguments apply if $\mathbf{z}_*$ and $\bar{\mathbf{g}}_*$ are solutions of operator equations that are approximated by solutions $\mathbf{z}$ and $\bar{\mathbf{g}}$ of corresponding discretizations. Under suitable conditions elaborated

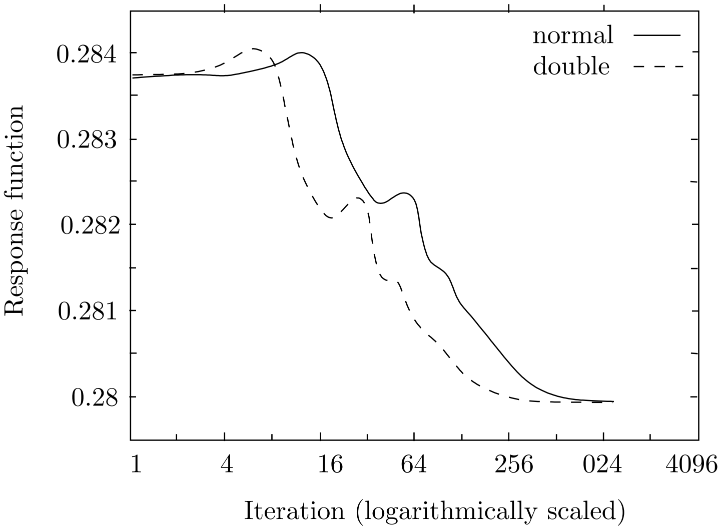

Figure 5.2. Normal and corrected values of drag coefficient.

in Giles and Pierce (2001), Giles and Süli (2002) and Becker and Rannacher (2001), the adjoint correction technique then doubles the order of convergence with respect to the mesh-width. In both scenarios, solving the adjoint equation provides accurate sensitivities of the weighted response with respect to solution inaccuracies. For discretized PDEs this information may then be used to selectively refine the grid where solution inaccuracies have the largest effect on the weighted response (Becker and Rannacher 2001).

5.4. Grey box differentiation and one-shot optimization

The two-phase approach considered above seems inappropriate in all situations where the design vector $\mathbf{x}$ is not constant, but updated repeatedly to arrive at some desirable value of the response $f(\mathbf{x}) = F(\mathbf{x}, \mathbf{z}_*(\mathbf{x}))$, for example a minimum. Then it makes more sense to apply simultaneously with the state space iteration (5.1) the adjoint iteration

$$\bar{\mathbf{g}}_{i+1} = \bar{\mathbf{z}}_i + \bar{\mathbf{g}}_i G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_i) = \bar{\mathbf{y}}\, F_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_i) + \bar{\mathbf{g}}_i G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_i). \tag{5.18}$$

As we observed for the forward derivative recurrence (5.6), the continuing variation in $\mathbf{z}_i$ destroys the proof of Q-linear convergence, but we have

$$R\{\bar{\mathbf{g}}_i - \bar{\mathbf{g}}_*\} \equiv \limsup_{i\to\infty} \sqrt[i]{\|\bar{\mathbf{g}}_i - \bar{\mathbf{g}}_*\|} \le \rho, \tag{5.19}$$

analogous to (5.8). The $\bar{\mathbf{g}}_i$ allow the computation of the approximate reduced gradients

$$\bar{\mathbf{x}}_i \equiv \bar{\mathbf{y}}\, F_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_i) + \bar{\mathbf{g}}_i\, G_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_i) = \bar{\mathbf{x}}_* + \tilde{O}(\rho^i). \tag{5.20}$$

Differentiating (5.18) once more, we obtain the second-order adjoint iteration

$$\dot{\bar{\mathbf{g}}}_{i+1} = \bar{\mathbf{y}} \dot{F}_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i) + \bar{\mathbf{g}}_i \dot{G}_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i) + \dot{\bar{\mathbf{g}}}_i G_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_i), \tag{5.21}$$

where

$$\dot{F}_{\mathbf{z}}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i) \equiv F_{\mathbf{zx}}(\mathbf{x}, \mathbf{z}_i)\dot{\mathbf{x}} + F_{\mathbf{zz}}(\mathbf{x}, \mathbf{z}_i)\dot{\mathbf{z}}_i$$

and $\dot{G}_{\mathbf{z}}$ is defined analogously. The vector $\dot{\bar{\mathbf{g}}}_i \in X^*$ obtained from (5.21) may then be used to calculate

$$\begin{aligned}\dot{\bar{\mathbf{x}}}_i &= \bar{\mathbf{y}} \dot{F}_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i) + \dot{\bar{\mathbf{g}}}_i G_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_i) + \bar{\mathbf{g}}_i \dot{G}_{\mathbf{x}}(\mathbf{x}, \mathbf{z}_i, \dot{\mathbf{x}}, \dot{\mathbf{z}}_i)\\ &= \dot{\bar{\mathbf{x}}}_* + \tilde{O}(\rho^i) = \bar{\mathbf{y}} f''(\mathbf{x})\dot{\mathbf{x}} + \tilde{O}(\rho^i),\end{aligned}$$

where $\dot{F}_{\mathbf{x}}$ and $\dot{G}_{\mathbf{x}}$ are also defined by analogy with $\dot{F}_{\mathbf{z}}$. The vector $\dot{\bar{\mathbf{x}}}$ represent a first-order approximation to the product of the reduced Hessian $\bar{\mathbf{y}} f''(\mathbf{x}) = \nabla^2_{\mathbf{x}}[\bar{\mathbf{y}} f(\mathbf{x})]$ with the direction $\dot{\mathbf{x}}$.

While the right-hand side of (5.21) looks rather complicated, it can be evaluated as a second-order adjoint of the vector function

$$E(\mathbf{x}, \mathbf{z}) \equiv [F(\mathbf{x}, \mathbf{z}), G(\mathbf{x}, \mathbf{z})],$$

whose first-order adjoint $\bar{E}$ appears on the right-hand side of (5.18).

Overall we have the following derivative enhanced iterations for $i = 0, 1 \ldots$:

$$\begin{aligned}
(\mathbf{y}, \mathbf{z}) &= \mathbf{E}\,(\mathbf{x}, \mathbf{z}) && \text{(original)},\\
(\bar{\mathbf{x}}, \bar{\mathbf{g}}) &= \bar{\mathbf{E}}\,(\mathbf{x}, \mathbf{z}, \bar{\mathbf{y}}, \bar{\mathbf{g}}) && \text{(adjoint)},\\
(\dot{\mathbf{y}}, \dot{\mathbf{z}}) &= \dot{\mathbf{E}}\,(\mathbf{x}, \mathbf{z}, \dot{\mathbf{x}}, \dot{\mathbf{z}}) && \text{(direct)},\\
(\dot{\bar{\mathbf{x}}}, \dot{\bar{\mathbf{g}}}) &= \dot{\bar{\mathbf{E}}}\,(\mathbf{x}, \mathbf{z}, \bar{\mathbf{y}}, \bar{\mathbf{g}}, \dot{\mathbf{x}}, \dot{\mathbf{z}}, \dot{\bar{\mathbf{g}}}) && \text{(second)}.
\end{aligned}$$

Here we have omitted the indices, because the new versions of $\mathbf{y}, \mathbf{z}, \bar{\mathbf{x}}, \bar{\mathbf{g}}, \dot{\mathbf{y}}, \dot{\mathbf{z}}$, $\dot{\bar{\mathbf{x}}}$ and $\dot{\bar{\mathbf{g}}}$ can immediately overwrite the old ones. Under our contractivity assumption all converge with the same R-factor ρ. Norms of the discrepancies between the left- and right-hand sides can be used to measure the progress of the iteration. Their plot in Figure 5.1 confirms the asymptotic result, but also suggests that higher derivatives lag behind lower derivatives, as was to be expected.

We have named this section 'grey box differentiation' because the generation of the nonincremental adjoint statement $(\bar{\mathbf{x}}, \bar{\mathbf{g}}) = \bar{E}(\mathbf{x}, \mathbf{z}, \bar{\mathbf{y}}, \bar{\mathbf{g}})$ from the original $(\mathbf{y}, \mathbf{z}) = E(\mathbf{x}, \mathbf{z})$ does not follow the usual recipe of adjoint generation, first and foremost because the outer loop need not be reversed, and consequently the old versions of the various variables need not be saved on a stack. Also, the adjoints are not updated incremently by += but properly assigned new values. There are many possible variations of the derived fixed point iterations. In 'one shot' optimization we try to solve simultaneously the stationarity condition $\bar{\mathbf{x}} = \nabla f(\mathbf{x}) = 0$, possibly by another fixed point iteration

$$\mathbf{x} = \mathbf{x} - Q^{-1}\bar{\mathbf{x}} \quad \text{with} \quad Q \approx \nabla^2 f(\mathbf{x}).$$

The reduced Hessian approximation Q might be based on $\mathtt{dim}(X)$ second-order adjoints of the form $\dot{\bar{\mathbf{x}}}$, with $\dot{\mathbf{x}}$ ranging over a basis in X. Alternatively, we may use secant updates or apply Uzawa-like algorithms. Some of these variants are employed in Newman, Hou, Jones, Taylor and Korivi (1992), Keyes, Hovland, McInnes and Samyono (2001), Forth and Evans (2001), Mohammadi and Pironneau (2001), Venditti and Darmofal (2000), Hinze and Slawig (2002) and Courty, Dervieux, Koobus and Hascoët (2002).

6. Jacobian matrices and graphs

The forward and reverse mode are nearly optimal choices when it comes to evaluating a single tangent $\dot{\mathbf{y}} = F'(\mathbf{x})\dot{\mathbf{x}}$ or a single gradient $\bar{\mathbf{x}} = \bar{\mathbf{y}}F'(\mathbf{x})$, respectively. In contrast, when we wish to evaluate a Jacobian with $m > 1$ rows and $n > 1$ columns, we observe in this section that the forward and reverse mode are just two extreme options from a wide range of possible ways to apply the chain rule. This generalization opens up the chance for a very significant reduction in operation count, although the search for an elimination ordering with absolutely minimal operation count is a hard combinatorial problem.

6.1. Nonincremental structure and Bauer's formula

In this subsection we will exclude all incremental statements so that there is a 1–1 correspondence between the φ_i and their results v_i. They must then, by (2.8), all be independent of each other in that $P_i^T\,P_j = 0$ if $i \neq j$. Theoretically any evaluation procedure with incremental statements can be rewritten by combining all contributions to a certain value in one statement. For example, the reverse sweep of the adjoint procedure on the right-hand side of Table 3.3 can be combined to the procedure listed in Figure 6.1.

$$\boxed{\begin{aligned}
\bar{v}_5 &= \bar{y}\\
\bar{v}_4 &= \bar{v}_5 * v_3\\
\bar{v}_3 &= \bar{v}_5 * v_4\\
\bar{v}_2 &= \bar{v}_4 * \cos(v_1 + v_2)\\
\bar{v}_1 &= \bar{v}_4 * \cos(v_1 + v_2) + \bar{v}_3 * v_3
\end{aligned}}$$

Figure 6.1. Nonincremental adjoint.

On larger codes this elimination of incremental statements may be quite laborious and change the structure significantly. Without incremental assignments we have the orthogonal decomposition

$$\mathcal{H} = \mathcal{H}_1 \oplus \mathcal{H}_2 \oplus \cdots \oplus \mathcal{H}_l \quad \text{and} \quad v_i \in \mathcal{H}_i \quad \text{for} \quad i = 1 \ldots l.$$

Consequently we have

$$Q_i = Q_i \sum_{j \prec i} P_j\,P_j^T = \sum_{j \prec i} Q_i\,P_j\,P_j^T$$

and may write

$$v_i = \varphi_i\left(\sum_{j \prec i} Q_i\,P_j\,v_j\right).$$

Thus we obtain the partial derivatives

$$C_{ij} = \frac{\partial \varphi_i}{\partial v_j} = \varphi_i'(u_i)\,Q_i\,P_j \in \mathbb{R}^{m_i \times m_j}.$$

The interpretation of the C_{ij} as $m_i \times m_j$ matrices again assumes a fixed orthonormal basis for all $\mathcal{H}_i \mathrel{\widehat{=}} \mathbb{R}^{m_i}$ and thus for their Cartesian product $\mathcal{H}$. Since C_{ij} is nontrivial exactly when $j \prec i$, we may attach these matrices as

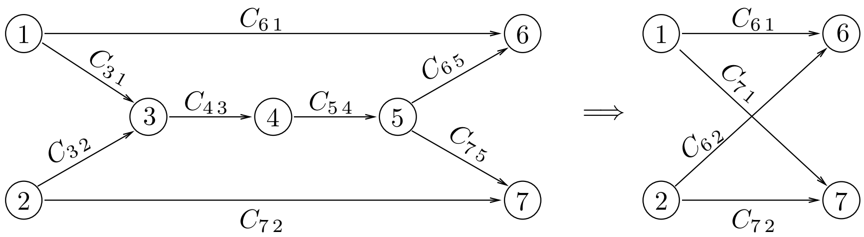

Figure 6.2. Jacobian vertex accumulation on a simple graph.

labels to the edges of the computation graph, as sketched in Figure 6.2 for a particular case with two independents and two dependents.

Obviously the local derivatives C_{ij} uniquely determine the overall Jacobian $F'(\mathbf{x})$, whose calculation from the C_{ij} we will call *accumulation*. Formally we may use the following explicit expression for the individual Jacobian blocks, which can be derived from the chain rule by induction on l.

Lemma 6.1. (Bauer's formula) The derivative of any dependent variable $y_i = v_{l-m+i}$ with respect to any independent variable $x_j = v_j$ is given by

$$\frac{\partial y_i}{\partial x_j} = \sum_{\mathcal{P}\in[j\to\hat{\imath}]} \prod_{(\tilde{\jmath},\tilde{\imath})\subset\mathcal{P}} C_{\tilde{\imath}\tilde{\jmath}}, \tag{6.1}$$

where $[j \to \hat{\imath}]$ denotes the set of all paths connecting j to $\hat{\imath} \equiv l - m + i$ and the pair $(\tilde{\jmath}, \tilde{\imath})$ ranges over all edges in $\mathcal{P}$ in descending order from left to right.

The order of the factors C_{ij} along any one of the paths matters only if some of the vertex dimensions $m_i = \dim(v_i)$ are greater than 1. For the example depicted in Figure 6.2 we find, according to Lemma 6.1,

$$\begin{aligned}
\frac{\partial y_1}{\partial x_1} &= \frac{\partial v_6}{\partial v_1} \equiv C_{61} = C_{61} + C_{65}\, C_{54}\, C_{43}\, C_{31},\\
\frac{\partial y_1}{\partial x_2} &= \frac{\partial v_6}{\partial v_2} \equiv C_{62} = C_{65}\, C_{54}\, C_{43}\, C_{32},\\
\frac{\partial y_2}{\partial x_1} &= \frac{\partial v_7}{\partial v_1} \equiv C_{71} = C_{75}\, C_{54}\, C_{43}\, C_{31},\\
\frac{\partial y_2}{\partial x_2} &= \frac{\partial v_7}{\partial v_2} \equiv C_{72} = C_{72} + C_{75}\, C_{54}\, C_{43}\, C_{32}.
\end{aligned}$$

In the scalar case the number of multiplications or additions needed to accumulate all entries of the Jacobian by Bauer's formula is given by

$$\mathtt{SIZE}\,(F(\mathbf{x})) = \sum_{j=1}^{n} \sum_{i=1}^{m} \sum_{\mathcal{P}\in[j\to\hat{\imath}]} |\mathcal{P}|,$$

where $|\mathcal{P}|$ counts the number of vertices in $\mathcal{P}$. It is not hard to see that this often enormous number is proportional to the length of the formulas that express each y_i directly in terms of all x_j without any named intermediates.

The expression (6.1) has the flavour of a determinant, which is no coincidence since in the scalar case it is in fact a determinant, as we shall see in Section 6.3. Naturally, an explicit evaluation is usually very wasteful for there are often common subexpressions, such as $C_{5\,4}\,C_{4\,3}$, that should probably be calculated first. Actually this is not necessarily so, unless we assume all intermediates v_i to be scalars. For instance, if the two independents v_1, v_2 and dependents v_6, v_7 were scalars but the three intermediates v_3, v_4 and v_5 were vectors of length d, then the standard matrix product $C_{5\,4}\,C_{43}$ of two $d \times d$ matrices would cost d^3 multiplications; whereas a total of 8 matrix-vector multiplications would suffice to compute the four Jacobian entries by bracketing their expressions from left to right or *vice versa*.

6.2. Jacobian accumulation by vertex elimination

In graph terminology we can interpret the calculation of the product $C_{5\,3} \equiv C_{5\,4}\,C_{4\,3}$ as the elimination of the vertex ④ in Figure 6.2. Then we have afterwards the simplified graph depicted on the left-hand side of Figure 6.3.

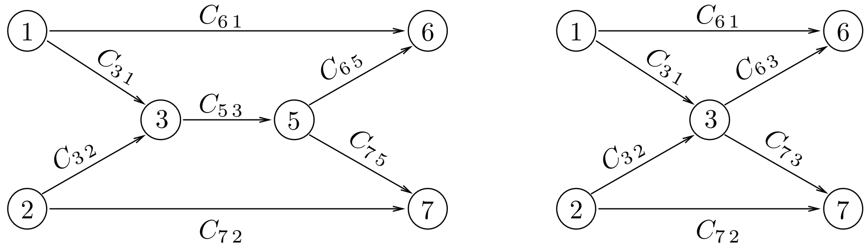

Figure 6.3. Successive vertex eliminations on problem displayed in Figure 6.2.

Subsequently we may eliminate vertex ⑤ by setting $C_{6\,3} \equiv C_{6\,5}\,C_{5\,3}$ and $C_{7\,3} = C_{7\,5}\,C_{5\,3}$. Finally, eliminating vertex ③ we arrive at the labels $C_{6\,1}$, $C_{6\,2}$, $C_{7\,1}$, and $C_{7\,2}$, which are exactly the (block) elements of the Jacobian $F'(\mathbf{x})$ represented as a bipartite graph in Figure 6.2.

In general, elimination of an intermediate vertex v_j from the computational graph requires the incrementation

$$C_{ik} \mathrel{+}= C_{ij}\,C_{jk} \in \mathbb{R}^{m_k \times m_i}$$

for all predecessor–successor pairs (k, i) with $k \prec j \prec i$. If a direct edge (i, k) did not exist beforehand it must now be introduced with the initial value $C_{ik} = C_{ij}\,C_{jk}$. In other words, the precedence relation $\prec$ and the graph structure must be updated such that all pairs (k, i) with $k \prec j \prec i$

are directly connected by an edge, afterwards the vertex j and its edges are deleted. Again assuming elementary matrix–matrix arithmetic, we have the total elimination cost

$$\mathtt{MARK}(v_j) \equiv \mathtt{OPS}\big(\mathtt{Elim}(v_j)\big) = m_j \left(\sum_{k \prec j \prec i} m_k \, m_i \right).$$

When all vertices are scalars $\mathtt{MARK}(v_j)$ reduces to the Markowitz degree familiar from sparse Gaussian elimination (Rose and Tarjan 1978). There as here, the overall objective is to minimize the vertex accumulation cost

$$\mathtt{VACC}\big(F'(\mathbf{x})\big) \equiv \sum_{n<j\leq l-m} \mathtt{MARK}(v_j),$$

where the degree $\mathtt{MARK}(v_j)$ needs to be computed just before the elimination of the jth vertex. This accumulation effort is likely to dominate the overall operation count compared to the effort for evaluating the elemental partials

$$\mathtt{OPS}\left(F'(\mathbf{x})\right) - \mathtt{VACC}\left(F'(\mathbf{x})\right) \equiv \sum_{i=1}^{l} \mathtt{OPS}\left(\{C_{ij}\}_{j \prec i}\right) \leq q \, \mathtt{OPS}\left(F(\mathbf{x})\right),$$

where

$$q \equiv \max_i \big\{ \mathtt{OPS}\left(\{C_{ij}\}_{j \prec i}\right) / \mathtt{OPS}\left(\varphi_i\right) \big\}.$$

Typically this maximal ratio q is close to 2 or some other small number for any given library of elementary functions φ_i. In contrast to sparse Gaussian elimination, the linearized computational graph stays acyclic throughout, and the minimal and maximal vertices, whose Markowitz degrees vanish trivially, are precluded from elimination. They form the vertex set of the final bipartite graph whose edges represent the nonzero (block) entries of the accumulated Jacobian, as shown on the right-hand side of Figure 6.2.

For the small example Figure 6.2 with scalar vertices, the elimination order ④, ⑤ and ③ discussed above requires $1+2+4=7$ multiplications. In contrast, elimination in the natural order ③, ④ and ⑤, or its exact opposite ⑤, ④ and ③, requires $2+2+4=8$ multiplications. Without too much difficulty it can be seen that, in general, eliminating all intermediate vertices in the orders $n+1, n+2, \ldots, (l-1), l$ or $l, l-1, \ldots, n+1, n$ is mathematically equivalent to applying the sparse vector forward or the sparse reverse mode discussed in Section 3.5, respectively. Thus our tiny example demonstrates that, already, neither of the two standard modes of AD needs to be optimal in terms of the operation count for Jacobian accumulation.

As in the case of sparse Gaussian elimination (Rose and Tarjan 1978), it has been shown that minimizing the fill-in during the accumulation of Jacobian is an NP-hard problem and the same is probably true for minimizing the operation count. In any case no convincing heuristics for selecting one of the $(l-m-n)!$ vertex elimination orderings have yet been developed. A good reason for this lack of progress may be the observation that, on one hand, Jacobian accumulation can be performed in more general ways, and on the other hand, it may not be a good idea in the first place. These two aspects are discussed in the remainder of this section.

Recently John Reid has observed that, even though Jacobian accumulation involves no divisions, certain vertex elimination orderings may lead to numerical instabilities. He gave an example similar to the one depicted in Figure 6.4, where the arcs are annotated by constant partials u and h, whose values are assumed to be rather close to 2 and -1, respectively.

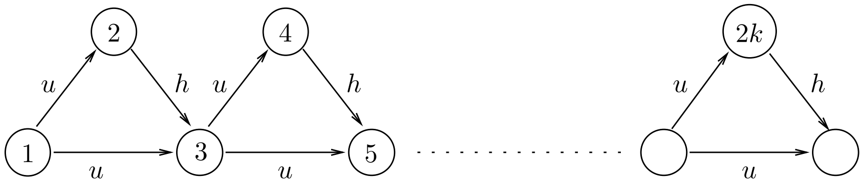

Figure 6.4. Reid's example of potential accumulation instability.

When the intermediate vertices $2, 3, 4, \ldots, 2k-1, 2k$ are eliminated forward or backward, all newly calculated partial derivatives are close to zero, as should be the final result $c_{2k+1,1}$. However, if we first eliminate only the odd vertices $3, 5, \ldots, 2k-1$, the arc between vertex 1 and vertex $2k+1$ temporarily reaches the value $u^k \approx 2^k$. The subsequent elimination of the even intermediate vertices $2, 4, 6, \ldots, 2k$ theoretically balances out this enormous value to nearly zero. However, in floating point arithmetic errors are certain to be amplified enormously. Hence, it is clear that numerical stability must be a concern in the study of suitable elimination orderings. In this particular case the numerically unstable, prior elimination of the odd vertices is also the one that makes the least sense in terms of minimizing fill-in and operation count. More specifically, the number of newly allocated and computed arcs grows quadratically with the depth k of the computational graph. Practically any other elimination order will result in a temporal and spatial complexity that is linear in k.

6.3. Jacobians as Schur complements

Because we have excluded incremental assignments and imposed the single assignment condition (2.7), the structure of the evaluation procedure Table 2.1 is exactly reflected in the nonlinear system

$$0 = E(\mathbf{x}; \mathbf{v}) = (\varphi_i(u_i) - v_i)_{i=1\ldots l},$$

where, as before, $\varphi_i(u_i) = \varphi_i(\,) = x_i$ for $i = 1 \ldots n$. By our assumptions on the data dependence $\prec$, the system is triangular and its $l \times l$ (block)

Jacobian has the structure

$$E'(\mathbf{x};\mathbf{v}) = \Big(C_{ij} - \delta_{ij} I\Big)_{i=1\ldots l}^{j=1\ldots l} \equiv C - I \tag{6.2}$$

$$= \begin{bmatrix}
-I & 0 & \cdots & 0 & 0 & \cdots & \cdots & 0 & 0 & \cdots & \cdots & 0\\
0 & \ddots & \ddots & \vdots & \vdots & & & \vdots & \vdots & & & \vdots\\
\vdots & \ddots & \ddots & 0 & \vdots & & & \vdots & \vdots & & & \vdots\\
0 & \cdots & 0 & -I & 0 & \cdots & \cdots & 0 & 0 & \cdots & \cdots & 0\\
\mathrm{x} & \cdots & \cdots & \mathrm{x} & -I & 0 & \cdots & 0 & 0 & \cdots & \cdots & 0\\
\vdots & & & \vdots & \mathrm{x} & \ddots & \ddots & \vdots & \vdots & & & \vdots\\
\vdots & & & \vdots & \vdots & \ddots & \ddots & 0 & \vdots & & & \vdots\\
\mathrm{x} & \cdots & \cdots & \mathrm{x} & \mathrm{x} & \cdots & \mathrm{x} & -I & 0 & \cdots & \cdots & 0\\
\mathrm{x} & \cdots & \cdots & \mathrm{x} & \mathrm{x} & \cdots & \cdots & \mathrm{x} & -I & 0 & \cdots & 0\\
\vdots & & & \vdots & \vdots & & & \vdots & 0 & \ddots & \ddots & \vdots\\
\vdots & & & \vdots & \vdots & & & \vdots & \vdots & \ddots & \ddots & 0\\
\mathrm{x} & \cdots & \cdots & \mathrm{x} & \mathrm{x} & \cdots & \cdots & \mathrm{x} & 0 & \cdots & 0 & -I
\end{bmatrix}
= \begin{bmatrix} -I & 0 & 0\\ B & L-I & 0\\ R & T & -I \end{bmatrix}.$$

Here δ_{ij} is the Kronecker delta and $C_{ij} \neq 0 \Longleftrightarrow j \prec i \Longrightarrow j < i$. The last relation implies in particular that the matrix L is strictly lower-triangular. Applying the implicit function theorem to $E(\mathbf{x};\mathbf{v}) = 0$, we obtain the derivative

$$\begin{aligned} F'(\mathbf{x}) &\equiv R + T(I-L)^{-1}B \\ &= R + T[(I-L)^{-1}B] = R + [T(I-L)^{-1}]B. \end{aligned} \tag{6.3}$$

The two bracketings on the last line represent two particular ways of accumulating the Jacobian $F'(\mathbf{x})$. One involves the solution of n linear systems in the unary lower-triangular matrix $(L-I)$ and the other requires the solution of m linear systems in its transpose $(L-I)^T$. As observed in Chapter 8 of Griewank (2000), the two alternatives correspond once more to the forward and reverse mode of AD, and their relative cost need not be determined by the ratio m/n nor even by the sparsity structure of the Jacobian $F'(\mathbf{x})$ alone. Instead, what matters is the sparsity of the extended Jacobian $E'(\mathbf{v},\mathbf{x})$, which is usually too huge to deal with in many cases. In the scalar case with $R = 0$ we can rewrite (6.3) for any two Cartesian basis vectors $e_i \in \mathbb{R}^m$ and $e_j \in \mathbb{R}^n$ by Cramer's rule as

$$e_i^T F'(\mathbf{x})\, e_j = \det\begin{bmatrix} I-L & B\mathbf{e}_j\,\mathbf{e}_i^T \\ -T & I \end{bmatrix}. \tag{6.4}$$

Thus we see that Bauer's formula given in Lemma 6.1 indeed represents the determinant of a sparse matrix. Moreover, we may derive the following alternative interpretation of Jacobian accumulation procedures.

6.4. Accumulation by edge elimination

Eliminating a certain vertex j in the computational graph, as discussed in the previous section, corresponds to using the $-I$ in the jth diagonal block of (6.2) to eliminate all other (block) elements in its row or column. This can be done for all $n > j \leq l-m$ in arbitrary order until the lower-triangular part L has been zeroed out completely, and the bottom left block R now contains the Jacobian $F'(\mathbf{x})$. Rather than eliminating all subdiagonal blocks in a row or column at once, we may also just zero out one of them, say C_{ij}. This can be interpreted as back- or front-elimination of the edge (j, i) from the computational graph, and requires the updates

$$C_{ik} \mathrel{+}= C_{ij}\, C_{jk} \quad \text{for all} \quad k \prec j \tag{6.5}$$

or

$$C_{hj} \mathrel{+}= C_{hi}\, C_{ij} \quad \text{for all} \quad h \succ i, \tag{6.6}$$

respectively. It is easy to check that, after the precedence relation $\prec$ has been updated accordingly, Bauer's formulas is unaffected. Front-eliminating all edges incoming to a node j or back-eliminating all outgoing edges is equivalent to the vertex elimination of j. The back-elimination (6.5) or the front-elimination (6.6) of the edge (j, i) may actually increase the number of nonzero (block-)elements in (6.2), which equals the number of edges in the computational graph. However, we can show that the sum over the length of all directed path in the graph and also the number of nonzeros in the subdiagonals of (6.2) are monotonically decreasing. Therefore, elimination of edges in any order must again lead eventually to the same bipartite graph representing the Jacobian.

Let us briefly consider edge elimination for the 'lion' example of Naumann displayed in Figure 6.5. It can be checked quite easily that eliminating the two intermediate vertices ① and ② in either order requires a total of 12 multiplications. However, if the edge c_{84} is first back-eliminated, the total effort is reduced by one multiplication. Hence we see that the generalization from vertex to edge elimination makes a further reduction in the operation count possible. However, it is not yet really known how to make good use of this extra freedom, and there is the danger that a poor choice of the order in which the edges are eliminated can lead to an exponential effort overall. This cannot happen in vertex elimination for which $\mathtt{VACC}(F'(\mathbf{x}))$ always has the cubic bound $\big[l \max_{1\leq i\leq l}\, (m_i)\,\big]^3$.

The same upper bound applies to edge elimination if we ensure that no edge is eliminated and reintroduced repeatedly through fill-in. To exclude this possibility, edge (j, i) may only be eliminated when it is final, *i.e.*, no other path connects the origin j to the destination i. Certain edge elimination orderings correspond to breaking down the product representation (3.1) further into a matrix chain, where originally each factor corresponds

Program:

$v_3 = c_{21} * v_1 + c_{32} * v_2$
$v_4 = c_{43} * v_3$
$v_5 = c_{54} * v_4$
$v_6 = c_{64} * v_4$
$v_7 = c_{74} * v_4$
$v_8 = c_{84} * v_4 + c_{83} * v_3$

Graph:

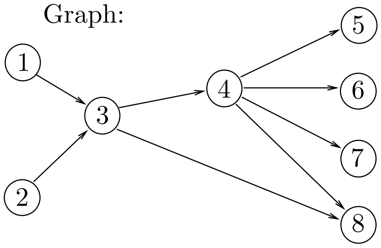

Figure 6.5. The ‘lion’ example from Naumann (1999) with $l = 8$.

to a single elemental partial C_{ij}. Many of these extremely sparse matrices commute and can thus be reordered. Moreover, the resulting chain can be bracketed in many different ways. The bracketing can be optimized by dynamic programming (Griewank and Naumann 2003*b*), which is classical when all matrix factors are dense (Aho, Hopcroft and Ullman 1974). Various elimination strategies were tested on the stencil generated by the Roe flux for hyperbolic PDEs (Tadjouddine, Forth and Pryce 2001).

6.5. Face elimination and its optimality

Rather than combining one edge with all its successors or with all its predecessors, we may prefer to combine only two edges at a time. In other words, we just wish to update $C_{ik} \mathrel{+}= C_{ij}\, C_{jk}$ for a single triple $k \prec j \prec i$, which is called a ‘face’ in Griewank and Naumann (2003*a*). Unfortunately, this kind of modification cannot be represented in a simple way on the level of the computational graph. Instead we have to perform *face eliminations* on the line graph whose vertices $\boxed{j,i}$ correspond initially to the edges (j,i) of the original graph appended by two extra sets of minimal vertices o_j for $j = 1 \ldots n$ and maximal vertices d_i for $i = 1 \ldots m$. They represent edges $o_j = (-\infty, j)$ connecting a common source $(-\infty)$ to the independent vertices of the original graphs and edges $d_i = (\hat{\imath}, \infty)$ connecting the dependent vertices $\hat{\imath} = l - m + i$ for $i = 1 \ldots m$ to a common sink $(+\infty)$. A line edge connects two line vertices $\boxed{k,i}$ and $\boxed{j,h}$ exactly when $j = i$. Now the vertices rather than the edges are labelled by the values C_{ij}. Without a formal definition we display in Figure 6.6 the line graph corresponding to the ‘lion’ example defined in Figure 6.5. It is easy to check that line graphs of DAGs (directed acyclic graphs) are also acyclic. They have certain special properties (Griewank and Naumann 2003*a*), which may, however, be lost for a while by face elimination, as discussed below.

Bauer’s formula (6.1) may be rewritten in terms of the line graph as

$$\frac{\partial y_i}{\partial x_j} = \sum_{\mathcal{P}\in[o_j\to d_i]} \; \prod_{\boxed{\tilde{\jmath},\tilde{\imath}}\,\subset\,\mathcal{P}} C_{\tilde{\imath}\tilde{\jmath}}, \tag{6.7}$$

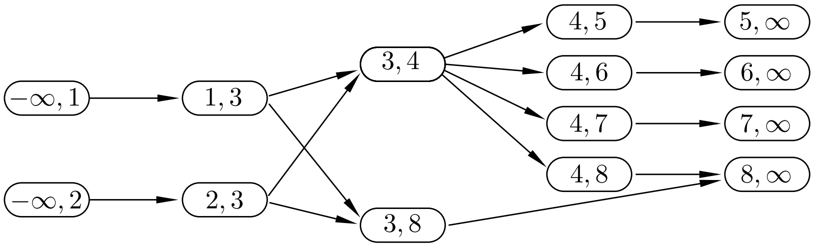

Figure 6.6. Line graph for Naumann's 'lion' example.

where $[o_j \to d_i]$ represents the set of all paths $\mathcal{P}$ connecting o_j to d_i, and the $\boxed{\tilde{j},\tilde{i}}$ range over all vertices belonging to path $\mathcal{P}$ in descending order. The elimination of a face $\boxed{k,j}$, $\boxed{j,i}$ with $k \neq -\infty$ and $i \neq +\infty$ can now proceed by introducing a new line vertex with the value $C_{ij}\,C_{jk}$, and connecting it to all predecessors of $\boxed{k,j}$ and all successors of $\boxed{j,i}$. It is very easy to see that this modification leaves (6.7) valid and again reduces the total sum of the length of all maximal paths in the line graph. After the successive elimination of all interior edges of the line graph whose vertices correspond initially to edges in the computational graph, we arrive at a tripartite graph. The latter can be interpreted as the line graph of a bipartite graph, whose edge values are the vertex labels of the central layer of the tripartite graph. Naumann (2001) has constructed an example where accumulation by face elimination reduces the number of multiplications below the minimal count achievable by edge elimination. It is believed that face elimination on the scalar line graph is the most general procedure for accumulating Jacobians.

Conjecture 6.2. Given a scalar computational graph, consider a fixed sequence of multiplications and additions to compute the Jacobian entries $\partial y_i/\partial x_j$ given by (6.1) and (6.7) for arbitrary real values of the C_{ij} for $j \prec i$. Then the number of multiplications required is no smaller than that required by some face elimination sequence on the computational graph, that is,

$$\mathtt{ACC}\big(F'(\mathbf{x})\big) = \mathtt{FAAC}\big(F'(\mathbf{x})\big) \leq \mathtt{EACC}\big(F'(\mathbf{x})\big) \leq \mathtt{VACC}\big(F'(\mathbf{x})\big)$$

where the inequalities hold strictly for certain example problems, and FAAC, EACC denote the minimal operation count achievable by face and edge elimination, respectively.

Proof. We prove the assertion under the additional assumptions that all $c_{ij} = C_{ij}$ are scalar, and that the computational graph is absorption-free in that any two vertices j and i are connected by at most one directed path. The property is inherited by the line graph. Then the accumulation procedure involves only multiplications, and must generate a sequence of partial products of the form

$$c_{\mathcal{P}} \equiv \prod_{(j,i) \subset \mathcal{P}} c_{ij}, \tag{6.8}$$

where $\mathcal{P} \subset \mathcal{P}^c \subset \mathcal{G}$ is a subset of a connected path $\mathcal{P}^c$. Multiplying coefficients c_{ij} that are not contained in a common path makes no sense, since the resulting products cannot occur in any one of the Jacobian entries. Now suppose we modify the original accumulation procedure by keeping all products in factored form, that is,

$$c_{\mathcal{P}_k} = \prod_{(j,i) \in \mathcal{P}_k} c_{ij} \quad \text{for} \quad k = 0, 1, \ldots, \bar{k},$$

where $\mathcal{P} = \bigcup_{k=0}^{\bar{k}} \mathcal{P}_k$ is the decomposition of $\mathcal{P}$ into its $\bar{k}+1$ maximal connected subcomponents. We show, by induction on the number of elements in $\mathcal{P}$, that this partitioned representation can be obtained by exactly $\bar{k}$ fewer multiplications than $c_{\mathcal{P}}$ itself. The assertion is trivially true for all individual arcs, which form a single element path with $\bar{k} = 0$. Now suppose we multiply two partial products $c_{\mathcal{P}}$ and $c_{\mathcal{P}'}$ with $\mathcal{P} \cap \mathcal{P}' = \emptyset$. Then the new partial path

$$\mathcal{P} \cup \mathcal{P}' = \bigcup_{k=0}^{\bar{k}} \mathcal{P}_k \cup \bigcup_{k'=0}^{\bar{k}'} \mathcal{P}'_{k'}$$

must be partitioned again into maximal connected components. Before these mergers we have $\bar{k}+\bar{k}'+2$ subpaths, and, by induction hypothesis, $\bar{k}+\bar{k}'+1$ saved multiplications. Whenever two subpaths $\mathcal{P}_k$ and $\mathcal{P}'_{k'}$ are adjacent we merge them, expending one multiplication and eliminating one gap. Hence the number of connected components minus one stays exactly the same as the number of saved multiplications. In summary, we find that we may rewrite the accumulation procedure such that all $\mathcal{P}$ successively occurring in (6.8) are, in fact, connected paths. Furthermore, without loss of generality we may assume that they are ordered according to their length. Now we will show that each of them can be interpreted as a face elimination on a line graph, whose initial structure was defined above. The first accumulation step must be of the form $c_{ijk} = c_{ij}\, c_{jk}$ with $k \prec j$ and $j \prec i$. This simplification may be interpreted as a face elimination on the line graph, which stays absorption-free. Hence we face the same problem as before, but for a graph of somewhat reduced total length. Consequently, the assertion follows by induction on the total length, *i.e.*, the sum of the lengths of all maximal paths. □

According to the conjecture, accumulating Jacobians with minimal multiplication count requires the selection of an optimal face elimination sequence of which there exists a truly enormous number, even for comparatively small initial line graphs. Before attacking such a difficult combinatorial problem we may pause to question whether the accumulation of Jacobians is really such an inevitable task as it may seem at first. The answer is in general 'no'. To demonstrate this we make two rather simple but fundamental observations. Firstly, Newton steps may sometimes be calculated more cheaply by a finite algorithm (of infinite precision) that does not involve accumulating the corresponding square Jacobian. Secondly, the representation of the Jacobian even as a sparse matrix, *i.e.*, a rectangular array of numbers, may be inappropriate, because it hides structure and wastes storage.

6.6. Jacobians with singular minors

Neither in Section 3 nor in Sections 6.4 and 6.5 have we obtained a definite answer concerning the cost ratio

$$\mathtt{OPS}\big([F(\mathbf{x}), F'(\mathbf{x})]\big)/\mathtt{OPS}(F(\mathbf{x})) \in \big[1, 3\min(\hat{n}, \hat{m})\big].$$

Here $\hat{n} \leq n$ and $\hat{m} \leq m$ denote, as in Section 3.5, the maximal number of nonzeros per row and column, respectively. A very simple dense example for which the upper bound is attained up to the constant factor 6 is

$$F(\mathbf{x}) = \frac{1}{2}\,(\mathbf{a}^T\mathbf{x})^2\mathbf{b} \quad \text{with} \quad \mathbf{a} \in \mathbb{R}^n, \quad \mathbf{b} \in \mathbb{R}^m. \tag{6.9}$$

Here we have $\mathtt{OPS}(F(\mathbf{x})) = n + m$, but $F'(\mathbf{x}) = \mathbf{b}(\mathbf{a}^T\mathbf{x})\mathbf{a}^T \in \mathbb{R}^{m\times n}$ has in general mn distinct entries so that

$$\mathtt{OPS}(F'(\mathbf{x}))/\mathtt{OPS}(F(\mathbf{x})) \geq mn/(m+n) \geq \min(m,n)/2,$$

for absolutely any method of calculating the Jacobians as a rectangular array of reals.

The troubling aspect of this example is that we can hardly imagine a numerical purpose where it would make sense to accumulate the Jacobian – quite the opposite. If we treat the inner product $z = \mathbf{a}^T\mathbf{x} = \sum_{i=1}^{n} a_i x_i$ as a single elemental operation, the computational graph of (6.9) takes for $n = 3$ and $m = 2$ the form sketched in Figure 6.7. Here z is also the derivative of $\frac{1}{2}z^2$.

The elimination of the two intermediate nodes in Figure 6.7 yields an $m \times n$ matrix with mn nonzero elements. Owing to rounding errors this matrix will typically have full rank, whereas the unaccumulated representation $F'(\mathbf{x}) = \mathbf{b}(\mathbf{a}^T\mathbf{x})\mathbf{a}^T$ reveals that $\operatorname{rank}(F'(\mathbf{x})) \leq 1$ for all $\mathbf{x} \in \mathbb{R}^n$. Thus we conclude that important structural information can be lost during the accumulation process. Even if we consider the $n + m$ components of $\mathbf{a}$ and $\mathbf{b}$ as free parameters, the set

$$\mathtt{Reach}\big\{F'(\mathbf{x})\big\} \equiv \big\{F'(\mathbf{x}) : \mathbf{x}, \mathbf{a} \in \mathbb{R}^n,\ \mathbf{b} \in \mathbb{R}^m\big\} \subset \mathbb{R}^{m\times n}$$

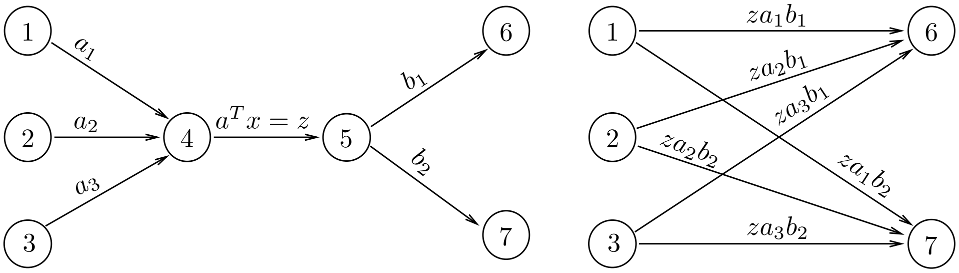

Figure 6.7. Vertex accumulation of a Jacobian on a simple graph.

is only an $(n+m-1)$-dimensional submanifold of the set of all real $m \times n$ matrices. We can check without too much difficulty that, in this case, $\texttt{Reach}\{F'(\mathbf{x})\}$ is characterized uniquely as the set of $m \times n$ matrices $A = (a_{ij})$ for which all 2×2 submatrices are singular, that is,

$$0 = \det \begin{pmatrix} a_{i\,k} & a_{i\,j} \\ a_{h\,k} & a_{h\,j} \end{pmatrix} \quad \text{for all} \quad 1 \le i < h \le m, \;\; 1 \le k < j \le n.$$

6.7. Generic rank

In the computational graph displayed in Figure 6.7, all 2×2 matrices correspond to a subgraph with the structure obtained overall for $n = 2 = m$. It is obvious that such matrices are singular since $z = \mathbf{a}^T \mathbf{x}$ is the single degree of freedom into which all the independent variables $x_1 \dots x_n$ are mapped. More generally, we obtain the following fundamental result.

Proposition 6.3. Let the edge values $\{c_{ij}\}$ of a certain linearized computational graph be considered as free parameters. Then the rank of the matrix A with elements

$$a_{ij} = \sum_{\mathcal{P} \in [j \to \imath]} \; \prod_{(\tilde{\jmath}, \tilde{\imath}) \in \mathcal{P}} c_{\tilde{\imath}\tilde{\jmath}}, \quad 1 \le i \le m, \;\; 1 \le j \le n,$$

is, for almost all combinations $\{c_{ij}\}_{j \prec i}$, equal to the size r of a maximal match, that is, the largest number of disjoint paths connecting the roots to the leaves of the graph.

Proof. Suppose we split all interior vertices into an input port and an output port connected by an internal edge of capacity 1. Furthermore we connect all minimal vertices by edges of unit capacity to a common source, and analogously all maximal ones to a common sink. All other edges are given infinite capacity. Then the solution of the max-flow (Tarjan 1983) problem is integral and represents exactly a maximal match as defined above. The corresponding min-cut solution consists of a set of internal edges of finite and thus unit capacity, whose removal would split the modified graph into two halves. The values v_j at these split vertices form a vector $\mathbf{z}$ of reals that depends differentiable on $\mathbf{x}$ and determines differentiably $\mathbf{y}$. Hence we have a decomposition

$$\mathbf{y} = F(\mathbf{x}) = H(\mathbf{z}) \quad \text{with} \quad \mathbf{z} = G(\mathbf{x}),$$

so that

$$\begin{aligned} \operatorname{rank}\big(F'(\mathbf{x})\big) &= \operatorname{rank}\big(H'(\mathbf{z})\, G'(\mathbf{x})\big) \\ &\leq \max\big\{\operatorname{rank}\big(H'(\mathbf{z})\big),\ \operatorname{rank}\big(G'(\mathbf{x})\big)\big\} \leq r \equiv \dim(\mathbf{z}). \end{aligned}$$

To attain the upper bound, we simply set $c_{ij} = 1$ for every edge (j, i) in the maximal match, all others being set to zero. Then the resulting matrix A is a permutation of a diagonal matrix with the number of nonzero elements being equal to its rank r. Because the determinant corresponding to the permuted diagonal has the value 1 for the special choice of the edge values given above, and it is a polynomial in the c_{ij}, it must in fact be nonzero for almost all combinations of the c_{ij}. This completes the proof. □

As we have seen in the proof, generic rank can be determined by a max flow computation for which very efficient algorithms are now available (Tarjan 1983). There are other structural properties which, like rank, can be deduced from the structure of the computational graph, such as the multiplicity of the zero eigenvalue (Röbenack and Reinschke 2001). All this kind of structural information is obliterated when we accumulate the Jacobian into a rectangular array of numbers. Therefore, in the remainder of this section we hope to motivate and initiate an investigation into the properties of Jacobian graphs and their optimal representation.

6.8. Scarce Jacobians and their representation

A simple criterion for the loss of information in accumulating Jacobians is a count on the number of degrees of freedom. In the graph Figure 6.3 of the rank-one example (6.8) we have $n + m + 1$ free edge values whose accumulation into dense $m \times n$ matrices cannot possibly reach all of $\mathbb{R}^{m\times n}$ when $n > 1 < m$. In some sense that is nothing special, since, for sufficiently smooth F, the set of reachable Jacobians

$$\big\{F'(\mathbf{x}) \in \mathbb{R}^{m\times n} : \mathbf{x} \in \mathbb{R}^n\big\}$$

forms by its very definition an n-dimensional manifold embedded in the $m\,n$ dimensional linear space $\mathbb{R}^{m\times n}$. Some of that structure may be directly visible as *sparsity*, where certain Jacobian entries are zero or otherwise constant. Then the set of reachable Jacobians is in fact contained in an affine subspace of $\mathbb{R}^{m\times n}$, which often has only a dimension of order $\mathcal{O}(n+m)$. It is well understood that such sparsity structure can be exploited for storing, factoring and otherwise manipulating these matrices economically, especially when the dimensions m and n are rather large. We contend here that a similar effect can occur when the computational graph rather than the Jacobian is sparse in a certain sense.

More specifically, we will call the computational graph and the resulting Jacobian *scarce* if the matrices $(\partial y_i/\partial x_j)_{j=1\dots n, i=1\dots m}$ defined by Bauer's formula (6.7) for arbitrary values of the elemental partials $C_{\tilde{\imath}\tilde{\jmath}}$ do not range over all of $\mathbb{R}^{m\times n}$. For the example considered in Section 6.7, we obtain exactly the set of matrices whose rank equals one or zero, which is a smooth manifold of dimension $m+n-1$. Naturally, this manifold need not be regular and, owing to its homogeneity, there are always bifurcations at the origin. The difference between $n\,m$ and the dimension of the manifold may be defined as the *degree of scarcity.* The dimension of the manifold is bounded above but generally not equal to the number of arcs in the computational graph. The discrepancy is exactly two for the example above.

A particularly simple kind of scarcity is sparsity, where the number of zeros yields the degree of sparsity, provided the other entries are free and independent of each other. It may then still make sense to accumulate the Jacobian, that is, represent it as set of partial derivatives. We contend that this is not true for Jacobians that are scarce but not sparse, such as the rank-one example.

In the light of Proposition 6.3 and our observation on the rank-one example, we might think that the scarcity structure can always be characterized in terms of a collection of vanishing subdeterminants. This conjecture is refuted by the following example, which is also representative for a more significant class of problems for which full accumulation seems a rather bad idea.

Suppose we have a 3×3 grid of values v_j for $j=1\dots 9$, as displayed in Figure 6.8, that are subject to two successive transformations. Each time the new value is a function of the old values at the same place and its neighbours to the west, south and southwest. At the southern and western boundary we continue the dependence periodically so that the new value of v_4, for example, depends on the old values of v_4, v_3, v_1, and v_6. This dependency is displayed in Figure 6.8. Hence the 9×9 Jacobians of each of the two transitions contain exactly four nonzero elements in each row. Consequently we have $4\cdot 9\cdot 2=72$ free edge values but $9\cdot 9=81$ entries in the product of the two matrices, which is the Jacobian of the final state with respect to the initial state.

Since by following the arrows in Figure 6.8 we can get from any grid point to any other point in two moves, the Jacobian is dense. Therefore the degree of scarcity defined above is at least $81-72=9$. However, the scarcity of this Jacobian cannot be explained in terms of singular submatrices. Any such square submatrix would be characterized by two sets $J_i\subset\{1,2,\dots,9\}$ for $i=0,2$ with J_0 selecting the indices of columns (= independent variables) and J_2 the rows (= dependent variables) of the submatrix. It is then not too difficult to find an intermediate set J_1 such that each element in J_1 is

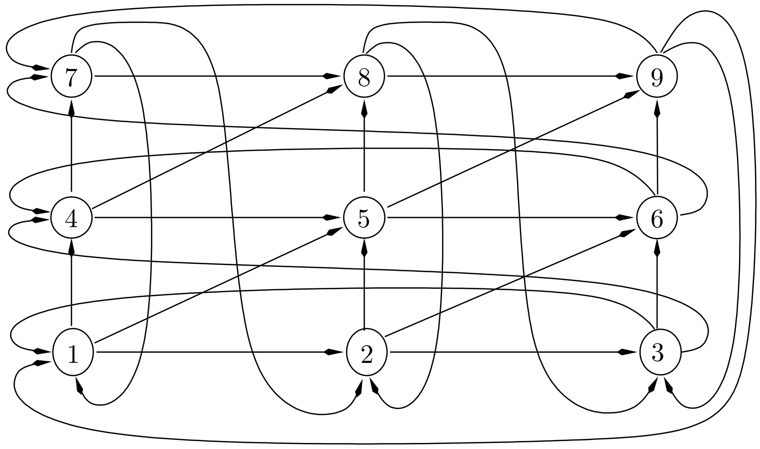

Figure 6.8. Update dependences on the upwinding example.

the middle of a directed path from an element in J_0 to an element in J_2 with all these paths being disjoint. Then it follows from Proposition 6.3 that the submatrix has generically full rank, and hence its determinant cannot vanish.

The example sketched in Figure 6.8 may be viewed as prototypical of upwind discretizations of a time-dependent PDE on a two-dimensional domain. There the Jacobian of the final state with respect to the initial state is the product of transformation matrices that are block-tridiagonal for reasonably regular discretizations. If there are just enough time-steps such that information can traverse the whole domain, the Jacobian will be dense but still scarce, as shown in Section 8.3 of Griewank (2000). It was also observed in that book that the greedy Markowitz heuristic has significantly worse performance than forward and reverse. We may view this as yet another indication that accumulating Jacobians may not be such a good idea in the first place.

That leaves the question of determining what is the best representation of scarce Jacobians whose full accumulation is redundant by definition of scarcity. Suppose we have evaluated all local partials c_{ij} and are told that a rather large number of Jacobian–vector or vector–Jacobian products must be computed subsequently. This requires execution of the forward loop

$$\dot{v}_i = \sum_{j \prec i} c_{ij} * \dot{v}_j \quad \text{for} \quad i = n+1 \ldots l \tag{6.10}$$

or the (nonincremental) reverse loop

$$\bar{v}_j = \sum_{i \succ j} \bar{v}_i * c_{ij} \quad \text{for} \quad j = l-m \ldots 1, \tag{6.11}$$

which follow from Tables 3.1 and 3.2, provided all v_i are scalar and disjoint. In either loop there is exactly one multiplication associated with each arc value c_{ij}, so that the number of arcs is an exact measure for the cost of computing Jacobian–vector products $\dot{\mathbf{y}} = F'(\mathbf{x})\dot{\mathbf{x}}$ and vector–Jacobian products $\bar{\mathbf{x}} = \bar{\mathbf{y}}F'(\mathbf{x})$.

In order to minimize that cost we may perform certain preaccumulations by eliminating edges or whole vertices. For example, in Figure 6.7 we can eliminate the central arc either at the front or at the back, and thus reduce the number of edge values from $n{+}m{+}1$ to $n{+}m$ without changing the reach of the Jacobian. This elimination requires here $\min(n, m)$ multiplications and will therefore be worthwhile even if only one Jacobian–vector product is to be computed subsequently. Further vertex, edge, or even face eliminations would make the size of the edge set grow again, and cause a loss of scarcity in the sense discussed above. The only further simplification respecting the Jacobian structure would be to normalize any one of the edge values to 1 and thus save additional multiplications during subsequent vector product calculations. Unfortunately, at least this last simplification is clearly not unique and it would appear that there is in general no unique minimal representation of a computational graph. As might have been expected, there is a rather close connection between the avoidance of fill-in and the maintenance of scarcity.

Proposition 6.4.

(i) If the front- or back-elimination of an edge in a computational graph does not increase the total number of edges, then the degree of scarcity remains constant.

(ii) If the elimination of a vertex would lead to a reduction in the total number of edges, then at least one of its edges can be eliminated via (i) without loss of scarcity.

Proof. Without loss of generality we may consider back-elimination of an edge (j, i), as depicted in Figure 6.9. The no fill-in condition requires that all predecessors of $\textcircled{j}$ but possibly one, say $\textcircled{k_0}$, are already directly connected to $\textcircled{i}$. Now we have to show that the values of the new arcs after the elimination can be chosen such that they reproduce any possible sensitivities in the subgraph displayed in Figure 6.9. Since the arc from $\textcircled{j}$ to $\textcircled{h}$ and possibly other successors of $\textcircled{j}$ remain unchanged, we must also keep all arcs c_{jk_t} for $t = 0 \ldots s$ unaffected. If also $k_0 \prec i$ there is no difficulty at all, as we may scale c_{ij} to one without loss of generality so that the new arc values are given by $\tilde{c}_{k_t} = c_{ik_t} + c_{jk_t}$, which is obviously reversible. Otherwise we have

$$\tilde{c}_{ik_0} = c_{ij} * c_{jk_0} \quad \text{and} \quad \tilde{c}_{ik_t} = c_{ik_t} + c_{ij} * c_{jk_t} \quad \text{for} \quad k = 1 \ldots s.$$

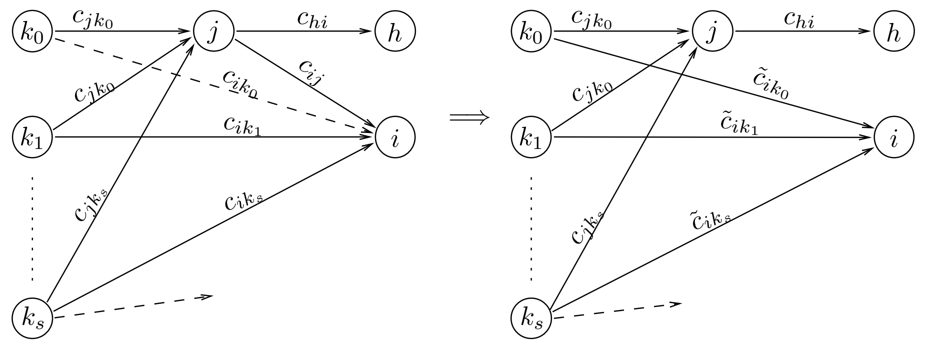

Figure 6.9. Scarcity-preserving back-elimination of (j, i).

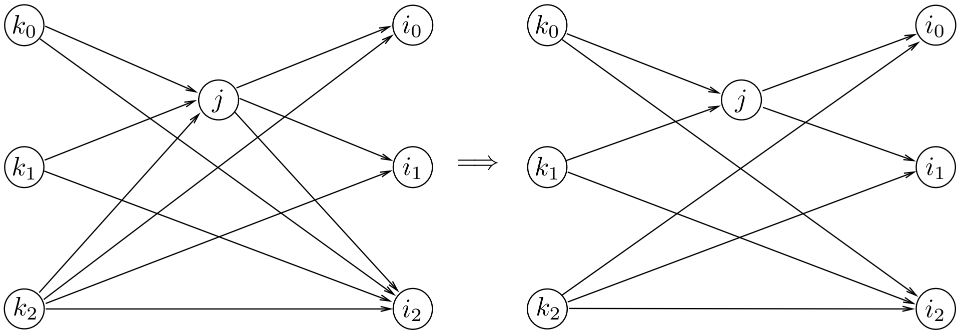

Figure 6.10. Scarcity-preserving elimination of (j, i_2) and (k_2, j).

Since we may assume without loss of generality that $c_{jk_0} \neq 0$, this relation may be reversed so that the degree of scarcity is indeed maintained as asserted.

To prove the second assertion, let us assume that the vertex $\textcircled{j}$ to be eliminated has $(p{+}1)$ predecessors and $(s{+}1)$ successors, which span together with $\textcircled{j}$ a subgraph of $p{+}s{+}3$ vertices. After the elimination of $\textcircled{j}$ they form a dense bipartite graph with $(s + 1)(p + 1)$ edges. Because of the negative fill-in, the number of direct arcs between the predecessors and successors of $\textcircled{j}$ is at least

$$(s + 1)(p + 1) - (s + 1 + p + 1) + 1 = s\, p.$$

From this it can easily be shown by contradiction that at least one predecessor of $\textcircled{j}$ is connected to all but possibly one of its successors, so that its link to $\textcircled{j}$ can be front-eliminated. □

The application of the proposition to the 3×3 subgraph depicted in Figure 6.10 shows that there are $6 + 5$ edges in the original graph on the left which would be reduced to 9 by the elimination of the central node.

However, this operation would destroy the property that the Jacobian of the vertices $\textcircled{i_0}$ and $\textcircled{i_1}$ with respect to $\textcircled{k_0}$ and $\textcircled{k_1}$ is always singular by Proposition 6.3. Instead we should front-eliminate (k_2, j) and back-eliminate (j, i_2), which reduces the number of arcs also to 9, while maintaining the scarcity.

Obviously Proposition 6.4 may be applied repeatedly until any additional edge elimination would increase the total edge count and no additional vertex elimination could reduce it further. However, there are examples where the resulting representation is still not minimal, and other, more global, scarcity-preserving simplifications may be applied. Also the issue of normalization, that is, the question of which arc values can be set to 1 to save extra multiplications, appears to be completely open. Hence we cannot yet set up an optimal Jacobian representation for the seemingly simple, and certainly practical, task of completing a large number of Jacobian–vector or vector–Jacobian products with the minimal number of multiplications.

6.9. Section summary

For vector functions defined by an evaluation procedure, each partial derivative $\partial y_i/\partial x_j$ of a dependent variable y_i, with respect to an independent variable x_j, can be written down explicitly as a polynomial of elemental partial derivatives $c_{\tilde{\imath}\tilde{\jmath}}$. However, naively applying this formula of Bauer is grossly inefficient, even for obtaining one $\partial y_i/\partial x_j$, let alone for a simultaneous evaluation of all of them. The identification of common subexpressions can be interpreted as vertex and edge elimination on the computational graph or face elimination on the associated line graph. The latter approach is believed to be the most general, and thus potentially most efficient, procedure. In contrast to vertex and edge elimination, face elimination can apparently *not* be interpreted in terms of linear algebra operations on the extended Jacobian (6.2). In any case, heuristics to determine good, if suboptimal, elimination sequences remain to be developed.

The accumulation of Jacobians as rectangular arrays of numbers appears to be a bad idea when they are scarce without being sparse, as defined in Section 6.8. Then there is a certain intrinsic relation between the matrix entries which is lost by accumulation. Instead we should strive to simplify the computational graph by suitable eliminations until a minimal representation is reached. This would, for example, be useful for the subsequent evaluation of Jacobian–vector or vector–Jacobian products with minimal costs. More importantly, structural properties like the generic rank should be respected. Thus we face the challenge of developing a theory of canonical representations of linear computational graphs, including in particular their composition.

Acknowledgements
The author is greatly indebted to many collaborators and co-authors, as well as the AD community as a whole. To name some would mean to disregard others, and quite a few are likely to disagree with my view of the field anyway. The initial typesetting of the manuscript was done by Sigrid Eckstein, who took particular care in preparing the many tables and figures.

REFERENCES

A. Aho, J. Hopcroft and J. Ullman (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.

R. Anderssen and P. Bloomfield (1974), 'Numerical differentiation proceedings for non-exact data', *Numer. Math.* **22**, 157–182.

W. Ball (1969), *Material and Energy Balance Computations*, Wiley, pp. 560–566.

R. Becker and R. Rannacher (2001), An optimal control approach to error control and mesh adaptation in finite element methods, in *Acta Numerica*, Vol. 10, Cambridge University Press, pp. 1–102.

C. Bennett (1973), 'Logical reversibility of computation', *IBM J. Research Development* **17**, 525–532.

M. Berz, C. Bischof, G. Corliss and A. Griewank, eds (1996), *Computational Differentiation: Techniques, Applications, and Tools*, SIAM, Philadelphia, PA.

C. Bischof, G. Corliss and A. Griewank (1993), 'Structured second- and higher-order derivatives through univariate Taylor series', *Optim. Methods Software* **2**, 211–232.

C. Bischof, A. Carle, P. Khademi and A. Mauer (1996), 'The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs', *IEEE Comput. Sci. Engr.* **3**, 18–32.

T. Braconnier and P. Langlois (2001), From rounding error estimation to automatic correction with AD, in Corliss *et al.* (2001), Chapter 42, pp. 333–339.

D. Cacuci, C. Weber, E. Oblow and J. Marable (1980), 'Sensitivity theory for general systems of nonlinear equations', *Nucl. Sci. Engr.* **88**, 88–110.

J.-B. Caillau and J. Noailles (2001), Optimal control sensitivity analysis with AD, in Corliss *et al.* (2001), Chapter 11, pp. 105–111.

S. Campbell and R. Hollenbeck (1996), Automatic differentiation and implicit differential equations, in Berz *et al.* (1996), pp. 215–227.

S. Campbell, E. Moore and Y. Zhong (1994), 'Utilization of automatic differentiation in control algorithms', *IEEE Trans. Automatic Control* **39**, 1047–1052.

B. Cappelaere, D. Elizondo and C. Faure (2001), Odyssée versus hand differentiation of a terrain modelling application, in Corliss *et al.* (2001), Chapter 7, pp. 71–78.

A. Carle and M. Fagan (1996), Improving derivative performance for CFD by using simplified recurrences, in Berz *et al.* (1996), pp. 343–351.

B. Christianson (1992), 'Automatic Hessians by reverse accumulation', *IMA J. Numer. Anal.* **12**, 135–150.

B. Christianson (1994), 'Reverse accumulation and attractive fixed points', *Optim. Methods Software* **3**, 311–326.

B. Christianson (1998), 'Reverse accumulation and implicit functions', *Optim. Methods Software* **9**, 307–322.

B. Christianson (1999), 'Cheap Newton steps for optimal control problems: Automatic differentiation and Pantoja's algorithm', *Optim. Methods Software* **10**, 729–743.

B. Christianson, L. Dixon and S. Brown (1997), Automatic differentiation of computer programs in a parallel computing environment, in *Applications of High Performance Computing in Engineering V* (H. Power and J. C. Long, eds), Computational Mechanics Publications, Southampton, pp. 169–178.

T. Coleman and J. Moré (1984), 'Estimation of sparse Jacobian matrices and graph coloring problems', *SIAM J. Numer. Anal.* **20**, 187–209.

T. Coleman and A. Verma (1996), Structure and efficient Jacobian calculation, in Berz *et al.* (1996), pp. 149–159.

A. Conn, N. Gould and P. Toint (1992), *LANCELOT, a Fortran Package for Large-Scale Nonlinear Optimization* (*Release A*), Vol. 17 of *Computational Mathematics*, Springer, Berlin.

G. Corliss (1991), Automatic differentiation bibliography, in Griewank and Corliss (1991), pp. 331–353.

G. Corliss, C. Faure, A. Griewank, L. Hascoët and U. Naumann, eds (2001), *Automatic Differentiation: From Simulation to Optimization*, series in *Computer and Information Science*, Springer, New York.

F. Courty, A. Dervieux, B. Koobus and L. Hascoët (2002), Reverse automatic differentiation for optimum design: From adjoint state assembly to gradient computation, Technical report, INRIA Sophia-Antipolis.

A. Curtis, M. Powell and J. Reid (1974), 'On the estimation of sparse Jacobian matrices', *J. Inst. Math. Appl.* **13**, 117–119.

L. C. W. Dixon (1991), Use of automatic differentiation for calculating Hessians and Newton steps, in Griewank and Corliss (1991), pp. 114–125.

J. Dunn and D. Bertsekas (1989), 'Efficient dynamic programming implementations of Newton's method for unconstrained optimal control problems', *J. Optim. Theory Appl.* **63**, 23–38.

Faa di Bruno (1856), 'Note sur une nouvelle formule de calcule differentiel', *Quar. J. Math.* **1**, 359–360.

C. Faure and U. Naumann (2001), Minimizing the tape size, in Corliss *et al.* (2001), Chapter 34, pp. 279–284.

S. A. Forth and T. Evans (2001), Aerofoil optimisation via AD of a multigrid cell-vertex Euler flow solver, in Corliss *et al.* (2001), Chapter 17, pp. 149–156.

Y. M. V. G. M. Ostrovskii and W. W. Borisov (1971), 'Über die Berechnung von Ableitungen', *Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie* **13**, 382–384.

R. Giering and T. Kaminski (1998), 'Recipes for adjoint code construction', *ACM Trans. Math. Software* **24**, 437–474.

R. Giering and T. Kaminski (2000), On the performance of derivative code generated by TAMC. Manuscript, FastOpt, Hamburg, Germany. Submitted to *Optim. Methods Software* See `www.FastOpt.de/papers/perftamc.ps.gz`.

R. Giering and T. Kaminski (2001*a*), 'Automatic sparsity detection'. Draft.

R. Giering and T. Kaminski (2001*b*), Generating recomputations in reverse mode AD, in Corliss *et al.* (2001), Chapter 33, pp. 271–278.

J. C. Gilbert (1992), 'Automatic differentiation and iterative processes', *Optim. Methods Software* **1**, 13–21.

M. Giles (2001), On the iterative solution of adjoint equations, in Corliss *et al.* (2001), pp. 145–151.

M. B. Giles and N. A. Pierce (2001), 'An introduction to the adjoint approach to design', *Flow, Turbulence and Combustion* **65**, 393–415.

M. B Giles and E. Süli (2002), Adjoint methods for PDEs: *a posteriori* error analysis and postprocessing by duality, in *Acta Numerica*, Vol. 11, Cambridge University Press, pp. 145–236.

M. Gockenbach, M. J. Petro and W. Symes (1999), 'C++ classes for linking optimization with complex simulations', *ACM Trans. Math. Software* **25**, 191–212.

A. Griewank (2000), *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*, Vol. 19 of *Frontiers in Applied Mathematics*, SIAM, Philadelphia.

A. Griewank and G. Corliss, eds (1991), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, PA.

A. Griewank and C. Faure (2002), 'Reduced functions, gradients and Hessians from fixed point iteration for state equations', *Numer. Alg.* **30**, 113–139.

A. Griewank and C. Mitev (2002), 'Detecting Jacobian sparsity patterns by Bayesian probing', *Math. Prog.* **93**, 1–25.

A. Griewank and U. Naumann (2003*a*), Accumulating Jacobians by vertex, edge or face elimination, in *Proceedings of the 6th African Conference on Research in Computer Science* (L. Andriamampianina, B. Philippe, E. Kamgnia and M. Tchuente, eds), Imprimerie Saint Paul, Camerun and INRIA, France, pp. 375–383.

A. Griewank and U. Naumann (2003*b*), 'Accumulating Jacobians as chained sparse matrix products', *Math. Prog.* **95**, 555–571.

A. Griewank and A. Verma (2003), 'On Newsam–Ramsdell-seeds for calculating sparse Jacobians'. In preparation.

A. Griewank and A. Walther (2000), 'Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation', *Trans. Math. Software* **26**, 19–45.

A. Griewank, C. Bischof, G. Corliss, A. Carle and K. Williamson (1993), 'Derivative convergence for iterative equation solvers', *Optim. Methods Software* **2**, 321–355.

A. Griewank, J. Utke and A. Walther (2000), 'Evaluating higher derivative tensors by forward propagation of univariate Taylor series', *Math. Comp.* **69**, 1117–1130.

J. Grimm, L. Pottier and N. Rostaing-Schmidt (1996), Optimal time and minimum space-time product for reversing a certain class of programs, in Berz *et al.* (1996), pp. 95–106.

S. Hague and U. Naumann (2001), Present and future scientific computation environments, in Corliss *et al.* (2001), Chapter 5, pp. 55–62.

L. Hascoët (2001), The data-dependence graph of adjoint programs, Research Report 4167, INRIA, Sophia-Antipolis, France. See `www.inria.fr/rrrt/rr-4167.html`.

L. Hascoët, S. Fidanova and C. Held (2001), Adjoining independent computations, in Corliss *et al.* (2001), Chapter 35, pp. 285–290.

M. Hinze and T. Slawig (2002), Adjoint gradients compared to gradients from algorithmic differentiation in instantaneous control of the Navier–Stokes equations, Preprint, TU Dresden.

M. Hinze and A. Walther (2002), An optimal memory-reduced procedure for calculating adjoints of the instationary Navier–Stokes equations, Preprint MATH-NM-06-2002, TU Dresden.

J. Horwedel (1991), GRESS: A preprocessor for sensitivity studies on Fortran programs, in Griewank and Corliss (1991), pp. 243–250.

A. S. Hossain and T. Steihaug (1998), 'Computing a sparse Jacobian matrix by rows and columns', *Optim. Methods Software* **10**, 33–48.

S. Hossain and T. Steihaug (2002), Sparsity issues in the computation of Jacobian matrices, Technical Report 223, Department of Computer Science, University of Bergen, Norway.

M. Iri, T. Tsuchiya and M. Hoshi (1988), 'Automatic computation of partial derivatives and rounding error estimates with applications to large-scale systems of nonlinear equations', *Comput. Appl. Math.* **24**, 365–392. Original Japanese version appeared in *J. Information Processing* **26** (1985), 1411–1420.

L. Kantorovich (1957), 'Ob odnoj matematicheskoj simvolike, udobnoj pri provedenii vychislenij na mashinakh', *Doklady Akademii Nauk. SSSR* **113**, 738–741.

G. Kedem (1980), 'Automatic differentiation of computer programs', *ACM Trans. Math. Software* **6**, 150–165.

D. Keyes, P. Hovland, L. McInnes and W. Samyono (2001), Using automatic differentiation for second-order matrix-free methods in PDE-constrained optimization, in Corliss *et al.* (2001), Chapter 3, pp. 33–48.

D. Knuth (1973), *The Art of Computer Programming*, Vol. 1 of *Computer Science and Information Processing*, Addison-Wesley, MI.

S. Linnainmaa (1976), 'Taylor expansion of the accumulated rounding error', *BIT* (*Nordisk Tidskrift for Informationsbehandling*) **16**, 146–160.

S. Linnainmaa (1983), 'Error linearization as an effective tool for experimental analysis of the numerical stability of algorithms', *BIT* **23**, 346–359.

R. Lohner (1992), Verified computing and programs in PASCAL-XSC, with examples, Habilitationsschrift, Institute for Applied Mathematics, University of Karlsruhe.

M. Mancini (2001), A parallel hierarchical approach for automatic differentiation, in Corliss *et al.* (2001), Chapter 27, pp. 225–230.

G. Marcuk (1996), *Adjoint Equations and Perturbation Algorithms in Nonlinear Problems*, CRC Press, Boca Raton, FL.

H. Maurer and D. Augustin (2001), Sensitivity analysis and real-time control of parametric optimal control problems using boundary value methods, in *Online Optimization of Large Scale Systems* (M. Grötschel, S. Krumke and J. Rambau, eds), Springer, Berlin/Heidelberg/New York, Chapter I, pp. 17–55.

B. Mohammadi and O. Pironneau (2001), *Applied Shape Optimization for Fluids*, series in *Numerical Mathematics and Scientific Computation*, Clarendon Press, Oxford.

R. Moore (1979), *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, PA.

U. Naumann (1999), Efficient calculation of Jacobian matrices by optimized application of the chain rule to computational graphs, PhD thesis, Institute of Scientific Computing, Germany.

U. Naumann (2001), Elimination techniques for cheap Jacobians, in Corliss *et al.* (2001), Chapter 29, pp. 241–246.

R. Neidinger (200x), Directions for computing multivariate Taylor series. Technical report, Davidson College, Davidson. To appear in *Math. Comp.*

P. Newman, G.-W. Hou, H. Jones, A. Taylor and V. Korivi (1992), Observations on computational methodologies for use in large-scale, gradient-based, multidisciplinary design incorporating advanced CFD codes, Technical Memorandum 104206, NASA Langley Research Center. AVSCOM Technical Report 92-B-007.

G. Newsam and J. Ramsdell (1983), 'Estimation of sparse Jacobian matrices', *SIAM J. Algebraic Discrete Methods* **4**, 404–417.

J. Ortega and W. Rheinboldt (1970), *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York.

C. Pantelides (1988), 'The consistent initialization of differential-algebraic systems', *SIAM J. Sci. Statist. Comput.* **9**, 213–231.

J. D. O. Pantoja (1988), 'Differential dynamic programming and Newton's method', *Internat. J. Control* **47**, 1539–1553.

J. Pryce (1998), 'Solving high-index DAEs by Taylor series', *Numer. Alg.* **19**, 195–211.

L. Rall (1983), Differentiation and generation of Taylor coefficients in Pascal-SC, in *A New Approach to Scientific Computation* (U. Kulisch and W. L. Miranker, eds), Academic Press, New York, pp. 291–309.

J. Restrepo, G. Leaf and A. Griewank (1998), 'Circumventing storage limitations in variational data assimilation studies', *SIAM J. Sci. Comput.* **19**, 1586–1605.

K. Röbenack and K. Reinschke (2000), 'Graph-theoretic characterization of structural controllability for singular DAE', *Z. Angew. Math. Mech.* **8 Suppl. 1**, 849–850.

K. Röbenack and K. Reinschke (2001), Nonlinear observer design using automatic differentiation, in Corliss *et al.* (2001), Chapter 15, pp. 133–138.

D. Rose and R. Tarjan (1978), 'Algorithmic aspects of vertex elimination on directed graphs', *SIAM J. Appl. Math.* **34**, 177–197.

J. Sternberg (2002), Adaptive Umkehrschemata für Schrittfolgen mit nichtuniformen Kosten, Diploma Thesis, Institute of Scientific Computing, Germany.

F. Stummel (1981), 'Optimal error estimates for Gaussian elimination in floating-point arithmetic', *GAMM, Numerical Analysis* **62**, 355–357.

M. Tadjouddine, S. A. Forth and J. Pryce (2001), AD tools and prospects for optimal AD in CFD flux Jacobian calculations, in Corliss *et al.* (2001), Chapter 30, pp. 247–252.

R. Tarjan (1983), 'Data structures and network algorithms', *CBMS-NSF Reg. Conf. Ser. Appl. Math.* **44**, 131.

J. van der Snepscheut (1993), *What Computing Is All About*, Suppl. 2 of *Texts and Monographs in Computer Science*, Springer, Berlin.

D. Venditti and D. Darmofal (2000), 'Adjoint error estimation and grid adaptation for functional outputs: application to quasi-one-dimensional flow', *J. Comput. Phys.* **164**, 204–227.

Y. Volin and G. Ostrovskii (1985), 'Automatic computation of derivatives with the use of the multilevel differentiating technique, I: Algorithmic basis', *Comput. Math. Appl.* **11**, 1099–1114.

A. Walther (1999), Program reversal schedules for single- and multi-processor machines, PhD thesis, Institute of Scientific Computing, Germany.

A. Walther (2002), 'Adjoint based truncated Newton methods for equality constrained optimization'.

X. Zou, F. Vandenberghe, M. Pondeca and Y.-H. Kuo (1997), Introduction to adjoint techniques and the MM5 adjoint modeling system, NCAR Technical Note NCAR/TN-435-STR, Mesocale and Microscale Meteorology Division, National Center for Atmospheric Research, Boulder, Colorado.