Structure in Optimization: Factorable Programming and Functions

Laurent Hascoët, Shahadat Hossain and Trond Steihaug

Abstract It is frequently observed that effective exploitation of problem structure plays a significant role in computational procedures for solving large-scale nonlinear optimization problems. A necessary step in this regard is to express the computation in a manner that exposes the exploitable structure. The formulation of large-scale problems in many scientific applications naturally give rise to "structured" representation. Examples of computationally useful structures arising in large-scale optimization problems include unary functions, partially separable functions, and factorable functions. These structures were developed from 1967 through 1990. In this paper we closely examine commonly occurring structures in optimization with regard to efficient and automatic calculation of first- and higher-order derivatives. Further, we explore the relationship between source code transformation as in algorithmic differentiation (AD) and factorable programming. As an illustration, we consider some classical examples.

Keywords Algorithmic differentiation · Source code transformation · Factorable programming

T. Steihaug (✉)
Department of Informatics, University of Bergen, Box 7803,
5020 Bergen, Norway
e-mail: trond.steihaug@ii.uib.no

S. Hossain
Department of Mathematics and Computer Science, University of Lethbridge,
Lethbridge, AB, Canada

L. Hascoët
INRIA, Sophia-Antipolis, France

E. Gelenbe and R. Lent (eds.), *Computer and Information Sciences III*,
DOI: 10.1007/978-1-4471-4594-3_46,

1 Introduction

For simplicity, we will consider the unconstrained optimization problem

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

where $f : \mathbb{R}^n \mapsto \mathbb{R}$ is sufficiently smooth. Methods of interest are those that require derivatives up to order three.
Let $e^{(i)}$ be the ith row of the identity matrix. A function f is separable if it can be written as

$$f(x) = \sum_{i=1}^{n} \phi_i(e^{(i)}x),$$

and can be decomposed into user-defined scalar functions ϕ_i. Given m matrices $U_i \in \mathbb{R}^{n_i \times n}$, $n_i \leq n$ where row k, $1 \leq k \leq n_i$ is a row of the identity matrix, a partially separable function [1] is given by

$$f(x) = \sum_{i=1}^{m} \phi_i(U_i x),$$

where each function $\phi_i : \mathbb{R}^{n_i} \mapsto \mathbb{R}$ is provided by the user. The functions $\phi_i, i = 1, \ldots, m$ are called element functions [1] and the variables $v^{(i)} \in \mathbb{R}^{n_i}$, $v^{(i)} = U_i x$ are called elemental variables [2]. Linear combinations of elemental variables are called internal variables [2–4], $u^{(i)} = W_i U_i x$. If W_i has more columns than rows, the element function ϕ_i will be functions of fewer than n_i variables. Bouaricha and Moré [5] describe software ELSO that computes the gradient of functions provided in partially separable form. To take advantage of partially separable structure one defines $\phi(x) = (\phi_1(x)\ \phi_2(x)\ \ldots\ \phi_m(x))^T$, then $f(x) = \phi(x)^T e$ where e is the vector of all ones. By employing algorithmic differentiation forward mode, the sparse Jacobian $\phi'(x)$ is computed yielding the gradient $\nabla f(x) = \phi'(x)^T e$. Gay [6] describes a method for detecting partially separable form of AMPL expressions which is then utilized in Hessian computations. Partially separable function minimization with AD on distributed memory parallel computing system has been considered in [7].
Let $u^{(i)} \in \mathbb{R}^n$ be m given vectors. A unary function [8] is given by

$$f(x) = \sum_{i=1}^{m} \phi_i({u^{(i)}}^T x), \quad \phi_i : \mathbb{R} \mapsto \mathbb{R}. \tag{2}$$

Let $g : \mathbb{R}^m \mapsto \mathbb{R}$ be a separable functions given by $g(v) = \sum_{i=1}^{m} \phi_i(v_i)$. Each function ϕ_i is provided by the user. Denoting the ith row of U by ${u^{(i)}}^T$, the unary function (2) becomes, $f(x) = g(Ux)$. The gradient and the Hessian matrix of f at

TF		EF	
$c_1u + c_2$	$\frac{c_1}{u} + c_2$	$u + v$	$u * v$
$c_2 e^{c_1 u}$	$c_2 \log(c_1 u)$	$-u$	c
$c_2 \sin(c_1 u)$	$c_2 \cos(c_1 u)$	$\frac{1}{u}$	
$c_2 \min\{u, c_1\}$	$c_2 \max\{u, c_1\}$	e^u	$\log(u)$
		$\sin(u)$	$\cos(u)$
c_1, c_2 are constants and u is a variable		u, v are variables and c is a constant	

Fig. 1 TF transformation function [11]. EF elemental function [20]

x are obtained in special forms,

$$\nabla f(x) = U^T \nabla g(Ux), \quad \nabla^2 f(x) = U \nabla^2 g(Ux) U^T. \tag{3}$$

Since g is separable, the Hessian matrix $\nabla^2 g$ is a diagonal matrix. Of particular interest is the case when $m = n$ and U has full rank [9]. In this case the unary function is a change of variables in g.

The notion of factorable functions predates that of partially separable functions and unary functions in optimization. A function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is a factorable function [10] if it can be represented as the last function in a finite sequence of functions $\{\phi_i\}_{i=1}^{L}$ where $\phi_i : \mathbb{R}^n \mapsto \mathbb{R}$:

$$\begin{aligned}
&\phi_i(x) \equiv {u^{(i)}}^T x \text{ where } u^{(i)} \text{ are constant vectors}, \quad i = 1, \ldots, \ell \\
&\phi_i(x) \equiv \phi_{j<i}(x) \circ \phi_{k<i}(x), \quad i = \ell + 1, \ldots, L, \quad \circ \in \{\times, +, -, /, \hat{}\} \\
&\qquad \text{or} \\
&\phi_i(x) \equiv \tau_i(\phi_{j<i}(x)), \quad \tau_i : \mathbb{R} \mapsto \mathbb{R} \\
&f(x) = \phi_L(x)
\end{aligned}$$

The sequence $\{\phi_1(x), \ldots, \phi_\ell(x), \ldots, \phi_L(x)\}$ is called a factored sequence. The notation $\phi_{j<i}(x)$ means that there exists $j < i$ so that $\phi_j(x)$ is an element of the factored sequence defined above. The function τ_i is called a transformation function such as exponential, trigonometric and logarithm, but may also be user defined functions. In [11–14] the initialization is given by $\phi_i(x) \equiv x_i$, $i = 1, \ldots, \ell$, and $\ell = n$. Figure 1 shows examples of transformation functions. It is pointed out by Kedem [12] that the notion of factorable functions corresponds to a simple Fortran subroutine that consists of expression evaluations without "IF" and "GOTO" statements and with very limited loops. In the book on automatic differentiation Rall in 1981 [15] points out that what is called codeable functions in [15] are in fact factorable functions. In [16] a "nonlinear" factorable form that includes bilinear terms is used to solve mixed-integer nonlinear programming optimization problems. Methods for these classes of functions using a partial update Newton are considered in [17].

Almost any function used for computational purposes can be put into a factorable form. Examples of functions which cannot be are given in [18, Chap. 3] and [19]. The remainder of the paper is organized as follows.

Section 2 discusses factorable programming problems and functions. A factorable programming problem, discussed in Sect. 2 is a nonlinear optimization problem where the objective and the constraint functions are factorable functions. The definition of a factorable function from 1967 introduces the concept of structure in an optimization problem. The definition that is used by most authors is a recursive definition, with the initialization given by $\phi_i(x) \equiv x_i$, which is reviewed in the preceding paragraphs. The final part of Sect. 2 is a discussion of the relationship between a factorable function and algorithmic differentiation. Section 3 provides illustration of source transformation of selected factorable and generic unary functions from the literature.

2 Factorable Programming Problems

Factorable programming problems were introduced by McCormick [19, 21] in 1974. A factorable programming problem is of the form

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} \quad X^L(x) \\ &\text{Subject to} \quad l_i \le X^i(x) \le u_i, \quad \text{for} \quad i = 1, \ldots, L-1, \end{aligned}$$

where $X^i : \mathbb{R}^n \mapsto \mathbb{R}$. Here $X^i(x) = x_i$ for $i = 1, \ldots, n$ and for given $X^p(x)$, $p = 1, \ldots, i-1$ function X^i is defined recursively as

$$X^i(x) = \sum_{p=1}^{i-1} T_p^i(X^p(x)) + \sum_{p=1}^{i-1}\sum_{q=1}^{p} V_{q,p}^i(X^p(x)) \cdot U_{p,q}(X^q(x)), \tag{4}$$

where T's, U's, and V's are transformation functions of a single variable. The lower and upper bounds $l_i \le u_i$ are given constants (may include $\pm\infty$.) It follows immediately that functions $X^i(x), i = 1, \ldots, L$ in (4) can be written as factorable functions.

A factorable programming language combined with the program SUMT [22] for the general nonlinear optimization problem was derived by McCormick in 1974 in [21] and extended by McCormick and Ghaemi [11]. The functions $X^i(x), i = 1, \ldots, L$ in [11, 21] are called concomitant variable functions (cvfs). The cvfs consist of two terms: the first term is separable and the second is a quadratic term. The inputs to the program [11, 21] are split between these two terms. The input is line based and, for the separable part of cvf number i, each line of the input is element p in the sum together with the type of transformation T_p^i and the index p of the cvf X^p. Similarly, for the quadratic term, two transformations and the two cvfs need to be specified for each element in the (double) sum. A modeling language for nonlinear programming problems for factorable functions of the form (4) and the use of SUMT was developed by Pugh [23] in 1972. In this modeling language one can also specify sums and products, $\sum_{i=m_1}^{m_2} \cdot$ and $\prod_{i=m_1}^{m_2} \cdot$, in addition to transformations of a single

variable. The double sum $\sum_{i=1}^{5}\sum_{j=1}^{5} x_{ij}$, for example, can be represented as the string `SI(I,1,5,SI(J,1,5,X(I;J))).`
In a technical report from 1967 McCormick introduces the term "factorable nonlinear convex programming" for a class of problems whose nonlinear function have second partial derivatives given in a special form. The technical report is published in Fiacco and McCormick [24, pp. 184–188]. The point taken in [24, 25] is that a factorable function is one where the analytic derivation of the Hessian matrix directly yields this form. The processing of the modeling language [11, 21, 23] assembles the Hessian matrix on this special form.

2.1 Extending Factorable Functions

A somewhat more general definition of factorable functions will allow the transformations τ_i to be functions of several variables [12], i.e. $\tau_i(\phi_{i_1}, \ldots, \phi_{i_s})$, $i_1, \ldots, i_s < i$. In [12] Fortran programs are augmented with non standard data types and operators and the non standard constructs are translated by a pre-compiler into standard Fortran. The gradient of a factorable function given on the form [10] can be shown by a minor modification of the proof in [13], to be of the form

$$\nabla f(x) = \sum_{i=1}^{\ell} u^{(i)} \alpha_i(x),$$

where $\alpha_i(x) \in \mathbb{R}$ is composed of product of factored-sequence functions and the first derivative of the transformations. The Hessian matrix is of the form

$$\sum_{i=1}^{\ell}\sum_{j=1}^{\ell} u^{(i)} \alpha_{ij}(x) u^{(j)^T},$$

where $\alpha_{ij}(x) \in \mathbb{R}$ are composed of factored-sequence functions and, the first and second derivative of the transformations in the sequence. Jackson and McCormick [13] show that the higher derivatives too, will have a polyadic structure (the gradient will be a sum of monads and the Hessian matrix be a sum of dyads.)
We would like to emphasize that the polyadic structure of factorable function is preserved also in the case when $\phi_i(x)$ themselves are factorable functions for $i = 1, \ldots, \ell$. It follows immediately that for unary functions, the gradient and Hessian of f are given by (3) for $m = \ell$. The approach taken in this paper is that $\phi_i, i = 1, \ldots, \ell$, in general, are *user-defined functions given as computer programs*. An extension of partial separability introduced in [2] is to write the function as $f(x) = \sum_{i=1}^{\ell} \tau_i(\phi_i(x))$, where $\tau_i : \mathbb{R} \mapsto \mathbb{R}, i = 1, \ldots, \ell$ are called group functions and ϕ_i are partially separable functions. This is again an example of factorable functions.

2.2 The General Evaluation Procedure in AD

For a given value x the general evaluation procedure in automatic differentiation is given by

$$\begin{aligned} v_{n-i} &= x_i, && i = 1, \ldots, n \\ v_i &= \widehat{\phi}_i(v_j)_{j \prec i}, && i = 1, \ldots, \ell, \text{ where } \widehat{\phi}_i \text{ is an elemental function} \\ y &= v_\ell. \end{aligned}$$

Examples of elemental functions in AD are displayed in Fig. 1.
Each value v_i can be interpreted as an intermediate function $v_i(x)$ of the independent variable $x \in \mathbb{R}^n$ [20]. This interpretation exposes the relationship between AD and factorable functions. The transformations in [11, 21] are just combinations of elemental functions used in AD. Importantly, utilizing the fact that the derivatives have a polyadic structure, is not an alternative in AD since the number of elemental functions will be very high.
Distinct from the view in AD, the factored–sequence functions ϕ_i are user-specified functions and a source code transformation tool will naturally yield the derivatives of function f in a structure-preserving (e.g., polyadic) form. To illustrate this point, we consider an example by Jackson and McCormick [13] and unary functions using AD source transformation tool Tapenade [26].

3 Examples of Source Code Transformations

The following example is from Jackson and McCormick [13]. The function is given by

$$f(x) = a^T x \sin(b^T x) e^{c^T x}. \tag{5}$$

To make an efficient hand-coded evaluation of the gradient and the Hessian we rewrite the function. Let $a, b, c \in \mathbb{R}^n$ and let A be a $n \times 3$ matrix and $\phi : \mathbb{R}^3 \mapsto \mathbb{R}$:

$$A = [a\ b\ c], \quad \phi = (\phi_1, \phi_2, \phi_3), \quad g(\phi) = \phi_1 \sin(\phi_2) e^{\phi_3}, \quad \text{then } f(x) = g(A^T x).$$

The gradient and the Hessian matrix of f at x are:

$$\nabla f(x) = A \nabla_\phi g(A^T x), \qquad \nabla^2 f(x) = A \nabla^2_\phi g(A^T x) A^T,$$

where $\nabla_\phi g(\phi) = (\sin(\phi_2) e^{\phi_3}, \phi_1 \cos(\phi_2) e^{\phi_3}, \phi_1 \sin(\phi_2) e^{\phi_3})^T$ and

$$\nabla^2_\phi g(\phi) = \begin{pmatrix} 0 & \cos(\phi_2) e^{\phi_3} & \sin(\phi_2) e^{\phi_3} \\ \cos(\phi_2) e^{\phi_3} & -\phi_1 \sin(\phi_2) e^{\phi_3} & \phi_1 \cos(\phi_2) e^{\phi_3} \\ \sin(\phi_2) e^{\phi_3} & \phi_1 \cos(\phi_2) e^{\phi_3} & \phi_1 \sin(\phi_2) e^{\phi_3} \end{pmatrix}.$$

```
DO nd=1,nbdirs
  .......
  phi6bd(nd) = yb*phi5d(nd); phi5bd(nd) = yb*phi6d(nd)
  phi1bd(nd) = phi4d(nd)*phi6b + phi4*phi6bd(nd)
  phi4bd(nd) = phi1d(nd)*phi6b + phi1*phi6bd(nd)
  phi3bd(nd) = phi3d(nd)*EXP(phi3)*phi5b + EXP(phi3)*phi5bd(nd)
  phi2bd(nd) = COS(phi2)*phi4bd(nd) - phi2d(nd)*SIN(phi2)*phi4b
  xbd(nd, :) = b(:)*phi2bd(nd) + a(:)*phi1bd(nd) + c(:)*phi3bd(nd)
END DO
```

Fig. 2 Source transformation: forward mode $\nabla^2 f(x)$ based on the gradient code

For a given x the numbers of arithmetic operations are approximately $6n$ to compute the function and $11n$ to compute the gradient and the function.
As a factorable function, (5) can be decomposed into

$$\begin{aligned}
&\phi_1(x) = a^T x,\ \phi_2(x) = b^T x,\ \phi_3(x) = c^T x,\\
&\phi_4(x) = \sin(\phi_2(x)), \phi_5(x) = \exp(\phi_3(x)), \phi_6(x) = \phi_1(x) * \phi_4(x), \text{and}\\
&f(x) = \phi_6(x) * \phi_5(x).
\end{aligned}$$

The gradient code produced in source transformation reverse AD of a Fortran 90 implementation of the function shows that the numbers of arithmetic operations are approximately the same for the source transformation and the hand-coded gradient [10].

3.1 The Hessian Matrix with the Source Transformation Tool Tapenade

The Hessian matrix is computed using n matrix vector product $He^{(i)}, i = 1, \ldots, n$ where $e^{(i)}$ is the ith column of the identity matrix. In Fig. 2 we only show the innermost loop. The hand-coded second derivative requires approximately $\frac{5}{2}n^2$ arithmetic operations utilizing the symmetry. The number of arithmetic operations for the code in Fig. 2 is approximately $5n^2$. As a further illustration of the use of AD for structured problems we consider computing the gradient of a unary function (2). Hand-coded derivatives will be computed using (3). Assuming, for simplicity, that U is a square matrix, the number of arithmetic operations to compute the gradient will be roughly $4n^2$ plus the n function calls g_i'. From Fig. 3 it follows that the number of arithmetic operations is about the same as in hand-coded calculation for the function and the gradient in source transformation. The user must either use a source transformation tool of the user-specified functions or a hand-coded version of the scalar functions ϕ_i. In Fig. 3 we illustrate the use of source transformed user-specified functions.

Looking for further similarities between the AD adjoint in Fig. 3 and the mathematical gradient of a unary function (3), one may wonder what became of the transposition U^T. A closer look at the generated code `unary_b` reveals that it does

```
subroutine UNARY(x, n, U, y)
  integer n,i ; real x(n), U(n,n) v(n),y
  do i = 1,n
     v(i) = SUM(U(i,:)*x(:))
  enddo
  y = 0.0
  do i = 1,n
     y = y + F(i,v(i))
  enddo
end subroutine UNARY
```

```
subroutine UNARY_B(x, xb, n, u, y, yb)
  do i=1,n
     v(i) = SUM(u(i, :)*x(:))
  end do
  yb = 0.0
  do i=n,1,-1
     result1b = yb
     call F_B(i, phi(i), vb(i), result1b)
  end do
  xb = 0.0
  do i=n,1,-1
     xb = xb + u(i, :)*vb(i)
     vb(i) = 0.0
  end do
  yb = 0.0
end subroutine unary_b
```

Fig. 3 Example of a unary function and the source transformation using reverse mode

compute the correct gradient, but with no explicit transposition. Actually, from a computer science point of view, an AD tool has no idea that there is a matrix product, and even less that it should be transposed. What makes things work is data-flow reversal [27], which we can sketch as follows:

$$\texttt{x(:)} \xrightarrow{\texttt{U(i,:)}} \texttt{phi(i)} \qquad \Rightarrow \qquad \texttt{phib(i)} \xrightarrow{\texttt{U(i,:)}} \texttt{xb(:)}$$

In the original code, data flow leads from `x(:)` to `phi(i)`, using the constant `U(i,:)` on the way. The adjoint code, by reversing the data flow, leads from `phib(i)` to `xb(:)`, still using `U(i,:)` on the way. No transposition nor index manipulation is involved but the effect is the same. More generally, we observe that data flow reversal plays, in the AD adjoint code, the role played by transposition in the expression of the mathematical gradient.

4 Concluding Remarks

The relationship between algorithmic differentiation on elemental function level and factorable function is well known. However, the use of more computationally intensive elementals or user-defined elementals are usually not a part of the evaluation procedure in AD. The point taken here is that source transformation can be an attractive tool for a user when the functions are composed of a factored-sequence of known user-specified functions.

References

1. Griewank, A., Toint, Ph.L.: On the unconstrained optimization of partially separable functions. In: Powell, M.J.D. (ed.) Nonlinear Optimization 1981, pp. 301–312. Academic Press, New York (1982)
2. Conn, A.R., Gould, N.I.M., Toint, Ph.L.: An introduction to the structure of large scale nonlinear optimization problems and the LANCELOT project. In: Glowinski, R., Lichnewsky, A. (eds.) Computing Methods in Applied Sciences and Engineering, pp. 42–51. SIAM, Philadelphia (1990)
3. Conn, A.R., Gould, N.I.M., Toint, Ph.L.: LANCELOT: A Fortran Package for Large-Scale Nonlinear Optimization (Release A), 1st edn. Springer, Berlin (1992)
4. Conn, A.R., Gould, N.I.M., Toint, Ph.L.: Improving the decomposition of partially separable functions in the context of large-scale optimization: a first approach. In: Hager, W.W., Hearn, D.W., Pardalos, P.M. (eds.) Large Scale Optimization: State of the Art, pp. 82–94. Kluwer Academic Publishers, Amsterdam (1994)
5. Bouaricha, A., Morè, J.J.: Impact of partial separability on large-scale optimization. Comput. Optim. Appl. **7**, 27–40 (1997)
6. Gay, D.M.: More AD of nonlinear AMPL models: computing Hessian information and exploiting partial separability. In: Berz, M., Bischof, C., Corliss, G., Griewank, A. (eds.) Computational Differentiation: Techniques, Applications, and Tools, pp. 173–184. SIAM, Philadelphia (1996)
7. Conforti, D., De Luca, L., Grandinetti, L., Musmanno, R.: A parallel implementation of automatic differentiation for partially separable functions using PVM. Parallel Comput. **22**, 643–656 (1996)
8. McCormick, G.P., Sofer, A.: Optimization with unary functions. Math. Program. **52**(1), 167–178 (1991)
9. Steihaug, T., Suleiman, S.: Global convergence and the Powell singular function. J. Glob. Optim. 1–9 (2012). doi: 10.1007/s10898-012-9898-z. http://www.dx.doi.org/10.1007/s10898-012-9898-z
10. Hascoët, L., Hossain, S., Steihaug, T.: Structured computation in optimization and algorithmic differentiation. ACM Commun. Comput. Algebra **46**(3) (2012)
11. Ghaemi, A., McCormick, G.P.: Symbolic factorable SUMT: What is it? How is it used? Technical Report T-402. Institute for Management Science and Engineering, The George Washington University, Washington DC (May 1979)
12. Kedem, G.: Automatic differentiation of computer programs. ACM Trans. Math. Softw. **6**(2), 150–165 (1980)
13. Jackson, R.H.F., McCormick, G.P.: The polyadic structure of factorable function tensors with application to high-order minimization techniques. J. Optim. Theory Appl. **51**(1), 63–94 (1986)
14. Jackson, R.H.F., McCormick, G.P.: Second-order sensitivity analysis in factorable programming: theory and applications. Math. Program. **41**(1–3), 1–27 (1988)
15. Rall, L.B.: Automatic Differentiation: Techniques and Applications. Lecture Notes in Computer Science, vol. 120. Springer, Berlin (1981)
16. Smith, E.M., Pantelides, C.C.: Global optimisation of nonconvex minlps. Comput. Chem. Eng. **21**(Suppl.), S791–S796 (1997)
17. Goldfarb, D., Wang, S.Y.: Partial-update Newton methods for unary, factorable, and partially separable optimization. SIAM J. Optim. **3**(2), 382–397 (1993)
18. McCormick, G.P.: Nonlinear Programming: Theory, Algorithms and Applications. Wiley, New York (1983)
19. McCormick, G.P.: Computability of global solutions to factorable nonconvex programs: part I convex underestimating problems. Math. Program. **10**(1), 147–175 (1976)
20. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. 2nd edn, Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia (2008)

21. McCormick, G.P.: A mini-manual for use of the SUMT computer program and the factorable programming language. Technical Report SOL 74–15. Department of Operations Research, Stanford University, Stanford (August 1974)
22. Mylander, W.C., Holmes, R., McCormick, G.P.: A Guide to SUMT-Version 4: The Computer Program Implementing the Sequential Unconstrained Minimization Technique for Nonlinear Programming. RAC-P-63, Research Analysis Corporation, McLean (1971)
23. Pugh, R.E.: A language for nonlinear programming problems. Math. Program. **2**, 176–206 (1972)
24. Fiacco, A.V., McCormick, G.P.: Nonlinear Programming: Sequential Unconstrained Minimization Techniques. Wiley, New York (1968)
25. McCormick, G.P.: Minimizing structured unconstrained functions. Technical Paper RAC-TP-277. Research Analysis Corporation, McLean, Virginia (October 1967)
26. Hascoët, L., Pascual, V.: Tapenade 2.1 user's guide. Technical Report 0300, INRIA (2004)
27. Hascoët, L.: Reversal strategies for adjoint algorithms. In: Bertot, Y., Huet, G., Lévy, J.-J., Plotkin, G. (eds.) From Semantics to Computer Science. Essays in Memory of Gilles Kahn, pp. 487–503. Cambridge University Press, New York (2009)