Numerical Algorithms **30:** 259–301, 2002.

Automatic differentiation for solving nonlinear partial differential equations: an efficient operator overloading approach

E. Tijskens [a], D. Roose [b], H. Ramon [a] and J. De Baerdemaeker [a]

[a] *Laboratorium Landbouwwerktuigkunde, K.U. Leuven, Kasteelpark Arenberg 30, B-3001, Leuven, Belgium*
E-mail: engelbert.tijskens@agr.kuleuven.ac.be
[b] *Departement Computerwetenschappen, K.U.Leuven, Celestijnenlaan 200A, B-3001, Leuven, Belgium*

Received 14 September 2000; revised 12 April 2000
Communicated by C. Brezinski

By resorting to Automatic Differentiation (AD) users of nonlinear PDE solvers can be relieved from the extra work of linearising a nonlinear PDE system and at the same time improve on the computational efficiency. This paper describes the main AD techniques and discusses how the operator overloading approach of AD can be extended to eliminate the overhead generally incurred with operator overloading. A recent AD system FastDer++, specially designed for this purpose, is integrated into a Least Squares solver. The necessary modifications to the general FEM algorithms. Code fragments and timing results demonstrate that (1) integrating AD with nonlinear PDE solvers leads to highly flexible code with a close resemblance to the mathematical expression of the problem, (2) coding and debugging efforts are greatly reduced, and (3) the computational efficiency is improved.

Keywords: automatic differentiation, scientific computing, mathematical modeling, nonlinear PDE

1. Introduction

The maturity of the mathematical and numerical aspects of Finite Element Methods (FEM) and the recent advances in object oriented software engineering has today led to the point where the construction of solvers for general systems of partial differential equations (PDE) has become feasible for users without a strong mathematical, numerical or programming background in a time efficient manner. General FEM packages, as, e.g., Diffpack® [25,28], have dramatically reduced the required efforts from the user to build such solvers. Ideally, of course, the user would only have to enter the PDE system at hand in a symbolic form together with initial and boundary conditions and perhaps some constraining algebraic equations. This is still beyond reach and in practice, the user has to derive discretised equations himself and provide details about the grid and

the elements used. In general, this is not a major obstacle. For nonlinear systems the user also has to carry out a linearisation of the PDE system or provide the Jacobian of the discretised system. Although a trivial task in principle, it is time-consuming, error-prone and potentially very hard or even prohibitive, depending on the constitutive equations for the model parameters in the system [13]. In addition, on linearising the equations, the user looses its intuition about the physics of the underlying problem. This obstructs the debugging process. The situation certainly does not encourage the exploration of PDE-based mathematical models. In the last decade also Automatic Differentiation techniques (AD) have become mature [14]. Contrary to finite difference schemes, automatic differentiation is able to provide partial derivatives which are exact to within rounding error, as opposed to finite difference schemes which show truncation error [29]. This is not the only advantage that comes with AD. It can in principle be used to relieve the user from the task of linearizing a nonlinear PDE system. An example of where this is already done is in the field of semiconductor equations where new models emerge all the time [35,40,41]. It will be demonstrated in this paper that the integration of AD in nonlinear PDE solvers leads to highly efficient and flexible code, with a strong resemblance to the original mathematical expression of the problem, and that coding and debugging efforts are greatly reduced. Consequently, its major asset is believed to be a dramatic reduction in the cost of solver development, so that the exploration of mathematical PDE-based models is thereby facilitated. In section 2 it will be demonstrated with a simple Poisson equation why general FEM packages can be made so general and user friendly in the linear case, how this practicality is destroyed in the nonlinear case, and how it can be restored by applying AD-principles. In section 3 we will present a practical and didactical overview of existing AD techniques. These are extended in section 4 to vectorised AD to become particularly efficient in the framework of nonlinear PDEs. Although the extension is general and also applicable in the case of finite differences and finite volume discretisation, it will be explored here in the framework of finite element discretisation. The extended techniques are implemented in C++ in a software library called FastDer++. The general FEM algorithm is modified slightly to allow for the integration of vectorised AD. Section 5 discusses code fragments and timing results for two test problems in order to demonstrate the potential of AD integration in nonlinear solvers. Finally, conclusions are summarised in section 6.

2. The finite element method

Consider the simple linear Poisson equation in d dimensions:

$$-\nabla \cdot \left[\lambda(\mathbf{x})\nabla u\right] = f(\mathbf{x}) \quad \text{on } \Omega \subset \mathbb{R}^d. \tag{1}$$

The parameter λ is a scalar or a second order tensor and may be a function of $\mathbf{x} \in \Omega$. The right hand side $f(\mathbf{x})$ represents the forcing function. Following standard practice,

the unknown function $u(\mathbf{x})$ is approximated in an element e by a linear combination of shape functions $N_i^{(e)}(\mathbf{x})$:

$$u^{(e)}(\mathbf{x}) = \sum_i u_i N_i^{(e)}(\mathbf{x}), \quad i = 1, \ldots, N_{\text{nodes}}^{(e)},$$

with $N_{\text{nodes}}^{(e)}$ the number of nodes in element e, and i is called a local numbering index. The Galerkin weak formulation [8] of the PDE (1) results in the element system:

$$\sum_j A_{ij}^{(e)} u_j^{(e)} = b_i^{(e)}, \tag{2}$$

with element matrix and vector, respectively

$$\begin{aligned} A_{ij}^{(e)} &= \int_{\Omega^{(e)}} \nabla N_i^{(e)}(\mathbf{x}) \cdot \lambda(\mathbf{x}) \nabla N_j^{(e)}(\mathbf{x}) \, \mathrm{d}\mathbf{x} \quad \text{and} \\ b_i^{(e)} &= \int_{\Omega^{(e)}} f(\mathbf{x}) N_i^{(e)}(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \end{aligned} \tag{3}$$

Assembly of the element matrices and vectors leads to a linear algebraic system:

$$\mathbf{A}\mathbf{u} = \mathbf{b}, \tag{4}$$

where

$$\mathbf{A}_{IJ} = \sum_{e_{IJ}} \mathbf{A}_{l(I,e),l(J,e)}^{(e)} \quad \text{and} \quad \mathbf{b}_I = \sum_{e_I} b_{l(I,e)}^{(e)}. \tag{5}$$

Here, the indices I and J denote a global numbering of the nodes, and $l(I, e)$ is a mapping which gives the local node number in element e of the node with global number I. By convention, local node numbers will be denoted by lowercase indices and global node numbers by uppercase indices. The indices e_{IJ}, respectively e_I, run over all elements containing nodes I and J, respectively node I.

The integrals in the element matrix and vector (2) are usually evaluated numerically with a suitable quadrature formula, e.g., of Gauss–Legendre type. If parametric elements are used the integration domain has a fixed size and shape. The element's shape and size are then captured by the Jacobian of the coordinate transformation between parametric coordinates ξ and physical coordinates $\mathbf{x}$:

$$\left| \frac{\partial \mathbf{x}}{\partial \xi} \right|_{\xi_q}, \qquad \mathbf{x} = \sum_i \mathbf{x}_i^{(e)} N_i^{(e)}(\xi), \tag{6}$$

so that the element matrix and vector can be generally expressed as

$$\begin{aligned}\mathbf{A}_{ij}^{(e)} &= \sum_{q}^{N_{ip}^{(e)}} W_p \nabla N_i^{(e)}(\xi_q) \cdot \lambda\big(\mathbf{x}(\xi_q)\big) \nabla N_j^{(e)}(\xi_q) \left|\frac{\partial \mathbf{x}}{\partial \xi}\right|_{\xi_q} \quad \text{and} \\ \mathbf{b}_i^{(e)} &= \sum_{q}^{N_{ip}^{(e)}} W_q f\big(\mathbf{x}(\xi_q)\big) N_i^{(e)}(\xi_q) \left|\frac{\partial \mathbf{x}}{\partial \xi}\right|_{\xi_q}.\end{aligned} \tag{7}$$

The index q runs over the number of integration points $N_{ip}^{(e)}$. Equations (5) and (7) lie at the heart of the general FEM algorithm. Obviously, only the functions $\lambda(\mathbf{x})$ and $f(\mathbf{x})$ in equations (7) do depend directly on the PDE, while all other factors depend on the element type and quadrature formula applied. Consequently, the details about element type and quadrature formula can be hidden from the user and in fact only $\lambda(\mathbf{x})$ and $f(\mathbf{x})$ need to be specified. Everything else, including assembly, linear solver, etc., can be handled by problem-independent library routines. This is a very convenient situation since a strong mathematical, numerical or programming background is no longer a prerequisite to use the FE method and much of its success is due to this fact. Advances in software engineering – object oriented programming in particular – have contributed to the development of libraries which handles issues such as element type selection, grid definition, adaptive meshing, linear solvers, etc. in a flexible and transparent way. A typical example of this is Diffpack® [25,28].

The situation changes dramatically, when the PDE becomes nonlinear. E.g., consider the quasilinear Poisson equation

$$-\nabla \cdot \big[\lambda(\mathbf{x}; u)\nabla u\big] = f(\mathbf{x}), \quad \text{on } \Omega \subset \mathbb{R}^d, \tag{8}$$

where now $\lambda(\mathbf{x}; u)$ depends on the unknown function $u(\mathbf{x})$. The nonlinearity is immaterial to the FE method itself, and equations (5) and (7) remain valid. Its only effect is that the matrix A depends on the unknown nodal values u, and hence the resulting algebraic system is nonlinear:

$$\mathbf{F}(\mathbf{u}) = \mathbf{A}(\mathbf{u})\mathbf{u} - \mathbf{b} = 0. \tag{9}$$

In case the forcing function $f = f(\mathbf{x}; u)$ too becomes a function of $u(\mathbf{x})$, the element vector $\mathbf{b}$ will also depend on $\mathbf{u}$. Equation (9) can be linearised by putting $\mathbf{u}^{s+1} = \mathbf{u}^s + \delta\mathbf{u}^s$ and approximating the function $\mathbf{F}(\mathbf{u})$ as a Taylor series truncated after the linear term:

$$\frac{\partial \mathbf{F}}{\partial \mathbf{u}}\big(\mathbf{u}^s\big)\delta\mathbf{u}^s = -\mathbf{F}\big(\mathbf{u}^s\big). \tag{10}$$

This is solved iteratively by taking an initial guess for $\mathbf{u}^{s=0}$, solving the linear system (10) for $\delta\mathbf{u}^s$, and restarting the computation with $\mathbf{u}^{s+1} = \mathbf{u}^s + \delta\mathbf{u}^s$ until convergence is reached. This method is generally referred to as the Newton–Raphson method [22]. The method effectively linearises the nonlinear algebraic system (9) and breaks it down to a series of linear algebraic systems (10).

As an alternative approach, one can reverse the order of discretisation and linearisation and first linearise the PDE and then discretise the resulting linear PDE. In order to do so one puts $u^{s+1}(\mathbf{x}) = u^s(\mathbf{x}) + \delta u^s(\mathbf{x})$. Approximating all functions of u as a Taylor series truncated after the linear term and dropping all terms containing products of δu^s one obtains a linear PDE:

$$-\nabla \cdot \left[\lambda\left(u^s\right)\nabla \delta u + \lambda'\left(u^s\right)\nabla u^s \delta u\right] = f + \nabla \cdot \left[\lambda\left(u^s\right)\nabla u^s\right]. \tag{11}$$

The system is again solved iteratively by taking an initial guess for $u^{s=0}(\mathbf{x})$, solving the linear PDE (11) for $\delta u^s(\mathbf{x})$, and restarting the computation with $u^{s+1}(\mathbf{x}) = u^s(\mathbf{x}) + \delta u^s(\mathbf{x})$ until convergence is reached. This, in fact, implies the application of the Newton–Raphson method at the level of the PDE. Both methods are formally equivalent, and lead to the same equations. The element matrix and vector are now found to be, respectively:

$$\begin{aligned}
\mathbf{A}_{ij}^{(e)} &= \sum_{q}^{N_{ip}^{(e)}} W_p \nabla N_i^{(e)}(\xi_q)\left\{\lambda\left(\mathbf{x}(\xi_q); u^s\right)\nabla N_j^{(e)}(\xi_q) + \lambda'\left(\mathbf{x}(\xi_q); u^s\right)\nabla u^s N_j^{(e)}(\xi_q)\right\}\left|\frac{\partial \mathbf{x}}{\partial \xi}\right|_{\xi_q}, \\
\mathbf{b}_i^{(e)} &= \sum_{q}^{N_{ip}^{(e)}} W_q\left\{f\left(\mathbf{x}(\xi_q)\right)N_i^{(e)}(\xi_q) - \nabla N_i^{(e)}(\xi_q)\lambda\left(\mathbf{x}(\xi_q); u^s\right)\nabla u^s\right\}\left|\frac{\partial \mathbf{x}}{\partial \xi}\right|_{\xi_q}.
\end{aligned} \tag{12}$$

From the standpoint of the programmer of the general FEM library linearising the PDE is very attractive and elegant, as it results in a series of linear PDEs for which the general FEM library can already compute the solution. The only changes to the general FEM library that are required, have to do with setting up the series, running it, and terminating it when it has converged. Modern FEM libraries, e.g., Diffpack® [25,28], have chosen this approach to nonlinear PDEs. The user of the library is less lucky, as he now faces the extra work of linearising the PDE, programming the linearised element matrix and vector (12), and providing the derivative

$$\lambda'(\mathbf{x}; u) = \frac{\partial \lambda(\mathbf{x}; u)}{\partial u}. \tag{13}$$

Although a trivial task in principle, this is time-consuming, error-prone and potentially very hard or even prohibitive, depending on the constitutive equations for the model parameters in the system [13]. Clearly, the exploration of different constitutive equations is not encouraged in this way. Applying the Newton–Raphson method on the nonlinear algebraic level according to equation (10) is not really an alternative, as then the user needs to provide the equivalent of the jacobian $\partial \mathbf{F}/\partial \mathbf{u}$ at the element level, i.e.

$$\left[\frac{\partial}{\partial \mathbf{u}}\left(\sum_{j=1}^{N_{\text{nodes}}^{(e)}} \mathbf{A}_{ij}^{(e)}\left(\mathbf{u}_j^{(e)}\right)\mathbf{u}_j^{(e)} - \mathbf{b}_i^{(e)}\right)\right]_{(\mathbf{u}_j^{(e)})^s}, \tag{14}$$

thus facing the same problem, as well as additional bookkeeping. From the standpoint of the user, the linearised equations (11), and the partial derivative of λ, given by (13),

are a nuisance, distracting his attention from the real problem, being the PDE (8) and the function $\lambda(u)$. Programming $\lambda(u)$ is in fact as far as he really wants to go. This not an unreasonable desire, neither is it unrealistic, as the rules for differentiation are in fact straightforward to apply and the recipe to evaluate a function can be mapped in a unique way to a recipe to evaluate its derivatives.[1] Consequently, the evaluation of derivatives is a task that in principle could be hidden from the user. Algorithms and techniques that are committed to this task, belong to the field generally addressed as Automatic Differentiation (AD). The next section explains the AD in some detail and how it can be used in a FEM framework. For more detail the reader is referred to the recent book by Griewank [14].

3. Automatic differentiation

Automatic Differentiation (AD) is based on the mathematical concept of differential algebras [3,4]. Alternatively, and conceptually much simpler, it can be envisaged as an implementation of the chain rule. Let w be a dependent variable depending on n_I independent variables u_i:

$$w = w(\mathbf{u}) = w(u_{1-n_I}, u_{2-n_I}, \ldots, u_1, u_0) : \mathbb{R}^{n_I} \to \mathbb{R}. \tag{15}$$

The particular way of numbering the independent variables u_i will turn out the be quite practical in describing AD algorithms. As we are interested in evaluating first order derivatives of w with respect to the independent variables $\mathbf{u}$ we define the following structure:

$$\widehat{w} \equiv [w \quad \nabla_{\mathbf{u}} w] \in \mathbb{R}^{1+n_I}. \tag{16}$$

Throughout this paper the gradient vector w.r.t. $\mathbf{u}$ will be considered to be a row vector for convenience. Structures like (16) are found to be part of a mathematical structure, called a Differential Algebra (DA) [3,4] and hence will be termed DA elements. Operators acting on them will be termed DA operators. The independent variables u_i ($i = 1 - n_I, \ldots, 1, 0$) are also DA elements:

$$\widehat{u}_i = [u_i \quad \nabla_{\mathbf{u}} u_i] = [u_i \quad \mathbf{e}_{i+n_I}] \in \mathbb{R}^{1+n_I}, \tag{17}$$

where $\mathbf{e}_{i+n_I}$ is the ith unit vector of $\mathbb{R}^{n_I}$. Thus

$$\begin{aligned} \widehat{u}_{1-n_I} &= [u_{1-n_I} \quad 1 \quad 0 \quad \ldots \quad 0], \\ \widehat{u}_{2-n_I} &= [u_{2-n_I} \quad 0 \quad 1 \quad \ldots \quad 0], \quad \text{etc.} \end{aligned}$$

[1] Exceptionally, however, when this recipe involves loops which are terminated according to some convergence criterion, the derivative may not have converged. This will only rarely be the case in the framework of nonlinear PDEs.

A general k-ary operator φ acting on $\mathbf{v} = [v_1 \quad v_2 \quad \dots \quad v_k]$ can be extended to a DA operator $\widehat{\varphi}$ acting on the DA elements $\widehat{\mathbf{v}} = [\widehat{v}_1 \quad \widehat{v}_2 \quad \dots \quad \widehat{v}_k]$ by applying the chain rule:

$$\widehat{\varphi}(\widehat{\mathbf{v}}) = \widehat{\varphi}(\widehat{v}_1, \widehat{v}_2, \dots, \widehat{v}_k) = \left[\varphi(v_1, v_2, \dots, v_k) \quad \sum_{j=1}^{k} \frac{\partial \varphi}{\partial v_j} \nabla_{\mathbf{u}} v_j\right]. \tag{18}$$

It is assumed that the operator φ is differentiable at all points of interest. This precludes value dependent branches. E.g., for the unary operator $\sin(v)$ one has:

$$(\widehat{v}) = \left[\sin(v) \quad \frac{\partial \sin(v)}{\partial v} \nabla_{\mathbf{u}} v\right] = \left[\sin(v) \quad \cos(v) \nabla_{\mathbf{u}} v\right]$$

and for the binary multiplication operator:

$$\widehat{v}_1 \widehat{\cdot} \widehat{v}_2 = \left[v_1 v_2 \quad \frac{\partial (v_1 v_2)}{\partial v_1} \nabla_{\mathbf{u}} v_1 + \frac{\partial (v_1 v_2)}{\partial v_2} \nabla_{\mathbf{u}} v_2\right] = [v_1 v_2 \quad v_2 \nabla_{\mathbf{u}} v_1 + v_1 \nabla_{\mathbf{u}} v_2].$$

Obviously, by successively applying DA operators $\widehat{\varphi}$ on the DA elements $\widehat{u}_i$ in the same way as one would apply the ordinary operators φ on the independent variables u_i, any function and its partial derivatives with respect to u_i can be evaluated automatically. A straightforward implementation of this idea is the Forward Mode algorithm, described in section 3.1.

So far, we have tacitly assumed that there is only one dependent variable w. In many practical cases, however, multiple dependent variables will be needed. The restriction is harmless, as each dependent variable can be computed along the same lines. This may give rise to a performance problem when there are common subexpressions in the dependent variables. E.g.,

$$w_1 = w_1(u_1, u_2, u_3) = (u_1 + u_2)^2,$$
$$w_2 = w_2(u_1, u_2, u_3) = (u_1 + u_2) u_3.$$

According to common programming practice, and not only for performance reasons, the two-fold evaluation of the common subexpression $u_1 + u_2$ should be avoided by storing it as an intermediary result:

$$t = u_1 + u_2; \qquad w_1 = t^2; \qquad w_2 = t u_3.$$

In DA formulation, this becomes:

$$\widehat{t} = \widehat{u}_1 \widehat{+} \widehat{u}_2; \qquad \widehat{w}_1 = \widehat{t}^{\,\widehat{2}}; \qquad \widehat{w}_2 = \widehat{t} \widehat{\cdot} \widehat{u}_3,$$

where $\widehat{+}$, $\widehat{2}$, and $\widehat{\cdot}$ are the DA operators for addition, square and multiplication, respectively. Clearly, this kind of optimisation is easily achieved and making it the responsibility of the programmer is not a problem.

In order to rigorously describe the AD algorithms we will assume that the dependent variable, $w = w(\mathbf{u}) : \mathbb{R}^{n_I} \to \mathbb{R}$, whose value and partial derivatives with respect

to **u** we want to evaluate, is defined by a sequential program in the following form by composing m elementary functions φ_i like sin, +, ...:

$$\begin{aligned} &\text{for } i = 1, 2, \ldots, m \\ &\quad u_i = \varphi_i(v_1, v_2, \ldots, v_{k_i}), \\ &\qquad v_j \in \{u_{1-n_I}, u_{2-n_I}, \ldots, u_{-1}, u_0, u_{-1}, \ldots, u_{i-1}\}, \quad j = 1, 2, \ldots, k_i, \\ &\quad w = u_m. \end{aligned} \tag{19}$$

Thus, the elementary functions φ_i are k_i-ary operators, whose arguments are selected among the independent variables u_i ($i = 1 - n_I, 2 - n_I, \ldots, -1, 0$) and the already computed quantities u_l ($l = 1, 2, \ldots, i - 1$). The algorithm (19) can perhaps best be visualised as a computational graph with the vertex set $\{i = 1-n_I, 2-n_I, \ldots, -1, 0, 1, \ldots, m\}$. Each vertex or node i is associated with an elementary function φ_i taking k_i arguments and its resulting value u_i. Arcs relate the arguments v_j ($j = 1, 2, \ldots, k_i$) of φ_i with already computed quantities u_l ($l < i$).

To illustrate this concept, consider the example function:

$$w = w(u_{-1}, u_0) = \left(u_{-1}^2 + u_0^2\right) \exp\left(-\mu\left(u_{-1}^2 + u_0^2\right)\right). \tag{20}$$

The sequential program for evaluating w requires the composition of $m = 6$ elementary operators in the following obvious way:

$$\begin{array}{lll} u_1 = u_{-1}^2, & \varphi_1 = \varphi_1(v_1) = v_1^2, & v_1 = u_{-1}, \\ u_2 = u_0^2, & \varphi_2 = \varphi_2(v_1) = v_1^2, & v_1 = u_0, \\ u_3 = u_1 + u_2, & \varphi_3 = \varphi_3(v_1, v_2) = v_1 + v_2, & v_1 = u_1, v_2 = u_2, \\ u_4 = -\mu u_3, & \varphi_3 = \varphi_3(v_1) = -\mu v_1, & v_1 = u_3, \\ u_5 = \exp(u_4), & \varphi_3 = \varphi_3(v_1) = \exp(v_1), & v_1 = u_4, \\ u_6 = u_3 u_5, & \varphi_6 = \varphi_6(v_1, v_2) = v_1 v_2, & v_1 = u_3, v_2 = u_5. \\ w = u_6, & & \end{array}$$

The computational graph for the example function (20) is depicted in figure 1. It contains the same information as the above scheme in a somewhat condensed form. We will now extend the algorithm (19) for the evaluation of w to evaluate also its partial derivatives with respect to the independent variables u_i ($i = 1 - n_I, 2 - n_I, \ldots, -1, 0$).

3.1. Forward mode

The forward mode mode is a straightforward implementation of the chain rule as applied in equation (18). The gradients of the independent variables u_i ($i = 1 - n_I$, $2 - n_I, \ldots, -1, 0$) are initialized to the ith Cartesian basis vector of $\mathbb{R}^{n_I}$, $\mathbf{e}_{i+n_I}$, as expressed in equation (17) and derivatives are accumulated as the program proceeds:

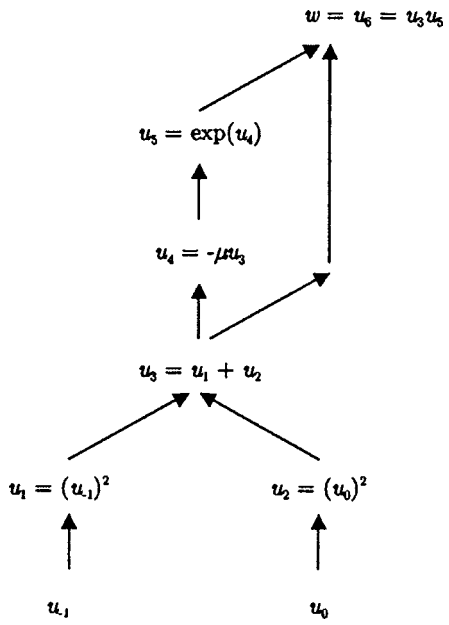

Figure 1. Computational tree of the example function (20).

for $i = 1 - n_I, 2 - n_I, \ldots, -1, 0$
$\quad \nabla_{\mathbf{u}} u_i = \mathbf{e}_{i+n_I}$
for $i = 1, 2, \ldots, m$

$$\left.\begin{aligned} u_i &= \varphi_i(v_1, v_2, \ldots v_{k_i}) \\ \nabla_{\mathbf{u}} u_i &= \sum_{j=1}^{k_i} \frac{\partial \varphi_i}{\partial v_j} \nabla_{\mathbf{u}} v_j \end{aligned}\right\} \quad \begin{aligned} v_j &\in \{u_{1-n_I}, u_{2-n_I}, \ldots, u_{-1}, u_0, u_{-1}, \ldots, u_{i-1}\}, \\ & j = 1, 2, \ldots, k_i, \end{aligned} \tag{21}$$

$w = u_m$.
$\nabla_{\mathbf{u}} w = \nabla_{\mathbf{u}} u_m$.

The body of the main for loop can be seen to correspond to the application of a DA operator $\widehat{\varphi}$ (see equation (18)). For forward mode AD the computational tree can be extended by adding the gradient computation to the nodes, and associating the local partial derivatives $\partial \varphi_i / \partial v_j$ with the arcs. The assumption that the operator φ is differentiable at all points of interest, excludes the possibility of code branches where the branching condition depends on the independent variables. As it is our intention to use AD to compute partial derivatives in a Newton–Raphson process for solving a nonlinear system, where the nonlinear system is always assumed to be differentiable, this restriction has to be met anyway.

The sequential program for our example function (22) extended for forward mode AD is thus:

$$
\begin{aligned}
&\nabla_{\mathbf{u}} u_{-1} = [1 \quad 0], \\
&\nabla_{\mathbf{u}} u_0 = [0 \quad 1], \\
&u_1 = u_{-1}^2, && \varphi_1 = \varphi_1(v_1) = v_1^2, && v_1 = u_{-1}, \\
&\nabla_{\mathbf{u}} u_1 = 2u_{-1}\nabla_{\mathbf{u}} u_{-1}, \\
&u_2 = u_0^2, && \varphi_2 = \varphi_2(v_1) = v_1^2, && v_1 = u_0, \\
&\nabla_{\mathbf{u}} u_2 = 2u_0\nabla_{\mathbf{u}} u_0, \\
&u_3 = u_1 + u_2, && \varphi_3 = \varphi_3(v_1, v_2) = v_1 + v_2, && v_1 = u_1, v_2 = u_2, \\
&\nabla_{\mathbf{u}} u_3 = \nabla_{\mathbf{u}} u_1 + \nabla_{\mathbf{u}} u_2, \\
&u_4 = -\mu u_3, && \varphi_3 = \varphi_3(v_1) = -\mu v_1, && v_1 = u_3, \\
&\nabla_{\mathbf{u}} u_4 = -\mu\nabla_{\mathbf{u}} u_3, \\
&u_5 = \exp(u_4), && \varphi_3 = \varphi_3(v_1) = \exp(v_1), && v_1 = u_4, \\
&\nabla_{\mathbf{u}} u_5 = \exp(u_4)\nabla_{\mathbf{u}} u_4 = u_5\nabla_{\mathbf{u}} u_4, \\
&u_6 = u_3 u_5, && \varphi_6 = \varphi_6(v_1, v_2) = v_1 v_2, && v_1 = u_3, v_2 = u_5, \\
&\nabla_{\mathbf{u}} u_6 = u_5\nabla_{\mathbf{u}} u_3 + u_3\nabla_{\mathbf{u}} u_5, \\
&w = u_6, \\
&\nabla_{\mathbf{u}} w = \nabla_{\mathbf{u}} u_6.
\end{aligned}
$$

The corresponding computational tree is depicted in figure 2. It was demonstrated [13] that the work ratio,

$$q\{w\} \equiv \frac{\text{work}\{w, \nabla_{\mathbf{u}} w\}}{\text{work}\{w\}} \propto n_I, \tag{22}$$

grows approximately linearly with n_I, as the computation of $\nabla_{\mathbf{u}} u_i$ involves at least one multiplication of a scalar and a n_I-dimensional vector. Also it is rather economical with memory since once all arcs leaving from a given node in the computational tree have been prosecuted, the information stored in the node can be forgotten.

3.2. *Reverse mode*

In the reverse mode mode a scalar derivative $\bar{u}_i$, a so called *adjoint variable*, is associated with each u_i ($i = 1 - n_I, 2 - n_I, \ldots, -1, 0$) rather than the gradient $\nabla_{\mathbf{u}} u_i$:

$$\bar{u}_i \equiv \frac{\partial u_m}{\partial u_i}. \tag{23}$$

By definition we have

$$\bar{u}_m = \frac{\partial u_m}{\partial u_m} = 1, \tag{24}$$

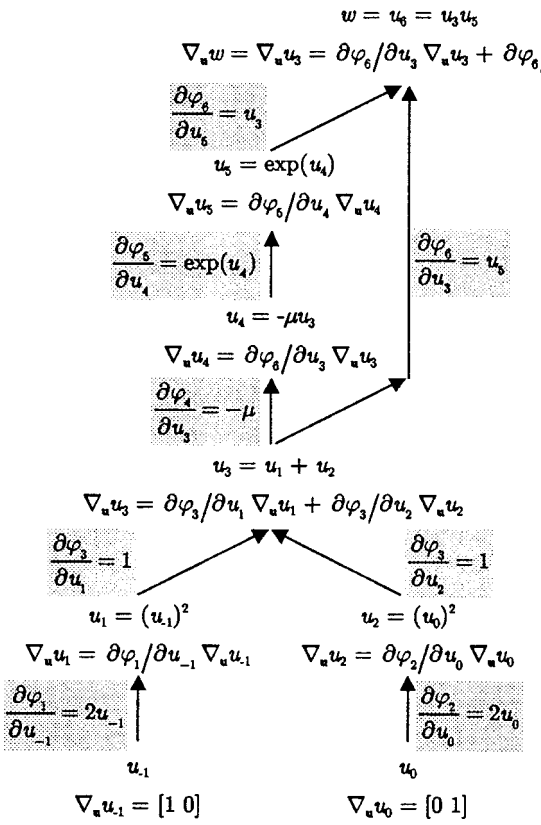

Figure 2. Computational tree of the example function (20) augmented with forward mode AD.

and the elements of the gradient vector $\nabla_{\mathbf{u}} w$ we are actually interested in are simply these adjoint variables associated with the independent variables:

$$\nabla_{\mathbf{u}} w = [\bar{u}_{1-n_I} \quad \bar{u}_{2-n_I} \quad \ldots \quad \bar{u}_{-1} \quad \bar{u}_0]. \tag{25}$$

Looking at the computational tree of our example function (20) (see figure 3), one can easily see that after traversing the tree in the usual order (bottom-up) to evaluate the values u_i of the nodes and setting $\bar{u}_m = 1$ (see equation (24)), one can start to build the adjoint variables $\bar{u}_i$ ($i = m-1, m-2, \ldots, 1, 0, -1, \ldots, 1-n_I$) by traversing the tree in the reverse direction (hence the name of the algorithm) using the relation

$$\bar{u}_i = \sum_{p_i} \frac{\partial \varphi_{p_i}}{\partial u_i} \bar{u}_{p_i}. \tag{26}$$

Here, p_i runs over the parent nodes of node i. (In case the computational tree is extended to account for multiple dependent variables, care must be taken that only parent nodes p_i are selected which are related to the dependent variable.) For a proof of relation (26) the reader is referred to [14,23].

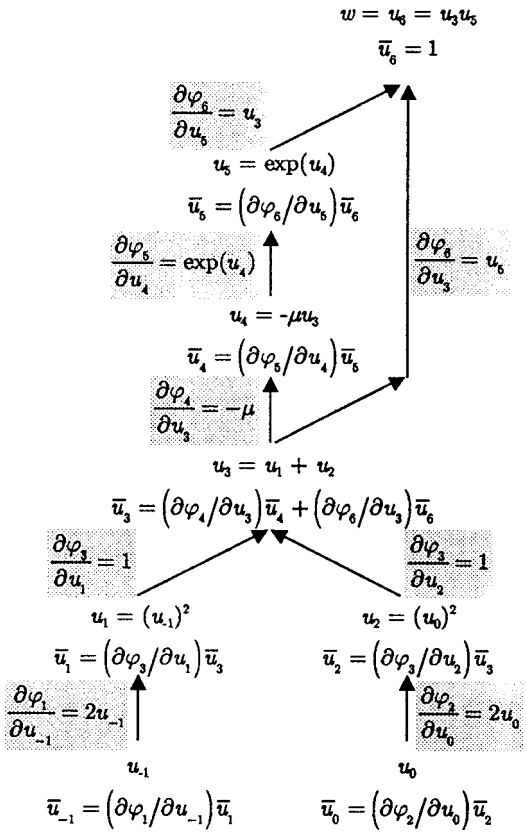

Figure 3. Computational tree of the example function (20) augmented with reverse mode AD.

Equation (28) allows us to extend algorithm (19) for obtaining the partial derivatives $\nabla_{\mathbf{u}} w$ as follows:

for $i = 1, 2, \ldots, m$ // forward sweep
  $u_i = \varphi_i(v_1, v_2, \ldots, v_{k_i})$,
    $v_j \in \{u_{1-n_I}, u_{2-n_I}, \ldots, u_{-1}, u_0, u_{-1}, \ldots, u_{i-1}\}$, $j = 1, 2, \ldots, k_i$
  $\bar{u}_i = 0$ // initialisation
$w = u_m$ // result
$\bar{u}_m = 1$ // initialisation
for $i = m - 1, \ldots, 2, 1$ // reverse sweep
  for all parent nodes p_i of node i

$$\bar{u}_i = \bar{u}_i + \frac{\partial \varphi_{p_i}}{\partial u_i} \bar{u}_{p_i} \tag{27}$$

$\nabla_{\mathbf{u}} w = [\bar{u}_{1-n_I} \quad \bar{u}_{2-n_I} \quad \ldots \quad \bar{u}_0]$ // result

For the example function (20) the above algorithm is translated as:

$$
\begin{aligned}
&u_1 = u_{-1}^2, && \bar{u}_1 = 0,\\
&u_2 = u_0^2, && \bar{u}_2 = 0,\\
&u_3 = u_1 + u_2, && \bar{u}_3 = 0,\\
&u_4 = -\mu u_3, && \bar{u}_4 = 0,\\
&u_5 = \exp(u_4), && \bar{u}_5 = 0,\\
&u_6 = u_3 u_5, && \bar{u}_6 = 1,\\
&w = u_6,\\
&\bar{u}_5 += \frac{\partial \varphi_6}{\partial u_5}\bar{u}_6 = u_3 1,\\
&\bar{u}_4 += \frac{\partial \varphi_5}{\partial u_4}\bar{u}_5 = \exp(u_4)u_3,\\
&\bar{u}_3 += \frac{\partial \varphi_4}{\partial u_3}\bar{u}_4 = -\mu \exp(u_4)u_3,\\
&\bar{u}_3 += \frac{\partial \varphi_6}{\partial u_3}\bar{u}_6 = -\mu \exp(u_4)u_3 + u_5 1,\\
&\bar{u}_2 += \frac{\partial \varphi_3}{\partial u_2}\bar{u}_3 = 1\big(-\mu \exp(u_4)u_3 + u_5\big),\\
&\bar{u}_1 += \frac{\partial \varphi_3}{\partial u_1}\bar{u}_3 = 1\big(-\mu \exp(u_4)u_3 + u_5\big),\\
&\bar{u}_0 += \frac{\partial \varphi_2}{\partial u_0}\bar{u}_2 = 2u_0\big(-\mu \exp(u_4)u_3 + u_5\big),\\
&\bar{u}_{-1} += \frac{\partial \varphi_1}{\partial u_{-1}}\bar{u}_1 = 2u_{-1}\big(-\mu \exp(u_4)u_3 + u_5\big),\\
&\frac{\partial w}{\partial u_0} = \bar{u}_0,\\
&\frac{\partial w}{\partial u_{-1}} = \bar{u}_{-1}.
\end{aligned}
$$

One can easily verify that, after filling in the values for u_{-1}, u_0, u_3, u_4, and u_5, one indeed obtains the desired partial derivatives of w:

$$\frac{\partial w}{\partial u_i} = 2u_i\big(1 - \mu\big(u_{-1}^2 + u_0^2\big)\big)\exp\big(-\mu\big(u_{-1}^2 + u_0^2\big)\big), \quad i \in \{-1, 0\}.$$

For reverse mode, the work ratio $q\{w\}$ does not grow linearly with n_I, but has an upper bound [13]:

$$q\{w\} = \frac{\text{work}\{w, \nabla_{\mathbf{u}} w\}}{\text{work}\{w\}} \leqslant 5. \tag{28}$$

On the other hand, reverse mode may require a lot more storage than for the forward mode since the entire computational tree must be stored to allow for the reverse traversal

and the computation of the derivatives. A node may be only be forgotten during the reverse sweep when all its child nodes have been processed. This potential drawback can be cured by trading storage for recalculation, e.g., by checkpointing [14]. Furthermore, it may be computationally advantageous to compute the partial derivatives $\partial\varphi_i/\partial v_j$ together with the intermediate quantity u_i, which still increases the storage needed for each link. In addition it must be said that implementation details may incur a varying overhead on the basic AD algorithms so that the reverse mode approach is not necessarily the optimal approach.

3.3. Hybrid schemes

Ideas from both modes may be combined to obtain more efficient schemes by splitting the computational tree into subtrees and selecting the most appropriate algorithm for each subtree. Consider, e.g., a dependent variable $w = w(\mathbf{u})$, $\mathbf{u} \in \mathbb{R}^{n_I}$, which can be decomposed as

$$w(\mathbf{u}) = y\big(\mathbf{x}(\mathbf{u})\big) = (y \circ \mathbf{x})(\mathbf{u}) \tag{29}$$

for some vector-valued function $\mathbf{x}$:

$$\begin{aligned} \mathbf{x} &= \mathbf{x}(\mathbf{u}) : \mathbb{R}^{n_I} \to \mathbb{R}^{n_x}, \\ y &= y(\mathbf{x}) : \mathbb{R}^{n_x} \to \mathbb{R}. \end{aligned}$$

This corresponds to splitting the computational tree of $w(\mathbf{u})$ in a lower part, associated with $\mathbf{x}(\mathbf{u})$ and an upper part associated with $y(\mathbf{x})$. Their gradients

$$\nabla_{\mathbf{x}} y = \begin{bmatrix} \dfrac{\partial y}{\partial x\_1 - n_x} & \dfrac{\partial y}{\partial x\_2 - n_x} & \cdots & \dfrac{\partial y}{\partial x\_0} \end{bmatrix},$$

and

$$\nabla_{\mathbf{u}} \mathbf{x} = \begin{bmatrix} \dfrac{\partial x\_1 - n_x}{\partial u_{1-n_I}} & \dfrac{\partial x\_1 - n_x}{\partial u_{2-n_I}} & \cdots & \dfrac{\partial x\_1 - n_x}{\partial u_0} \\ \dfrac{\partial x\_2 - n_x}{\partial u_{1-n_I}} & \dfrac{\partial x\_2 - n_x}{\partial u_{2-n_I}} & \cdots & \dfrac{\partial x\_2 - n_x}{\partial u_0} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial x\_0}{\partial u_{1-n_I}} & \dfrac{\partial x\_0}{\partial u_{2-n_I}} & \cdots & \dfrac{\partial x\_0}{\partial u_0} \end{bmatrix}$$

may be computed by forward or reverse mode AD, whichever is more convenient, while by applying the chain rule,

$$\nabla_{\mathbf{u}} w^{\mathrm{T}} = \nabla_{\mathbf{u}} (y \circ \mathbf{x})^{\mathrm{T}} = \nabla_{\mathbf{u}} \mathbf{x}^{\mathrm{T}} \nabla_{\mathbf{x}} y^{\mathrm{T}} \tag{30}$$

the two subtrees can be glued back together. (Note that transpose signs are required since we agreed on gradients being row vectors rather than column vectors.) E.g., consider

again the example function (20) which we generalise slightly for a arbitrary number of independent variables:

$$w(\mathbf{u}) = \left(\sum_{i=1-n_I}^{0} u_i^2 \right) \exp\left(-\mu \left(\sum_{i=1-n_I}^{0} u_i^2 \right) \right).$$

To split the computational tree, we decompose $w(\mathbf{u}) = y(\mathbf{x}(\mathbf{u}))$, where

$$\mathbf{x}(\mathbf{u}) = x(\mathbf{u}) = \sum_{i=1-n_I}^{0} u_i^2, \qquad n_x = 1,$$

and

$$y(\mathbf{x}) = y(x) = x \exp(-\mu x).$$

In the case $n_I = 2$, as in (20), this corresponds to splitting at node $i = 3$. When n_I is rather large, a forward approach would have to carry n_I-dimensional gradient vectors all along the computational tree, although its upper part (corresponding to $y(x)$) is a concatenation of unary operators. By splitting the computational tree, the upper part can be evaluated most efficiently in forward mode since it has only one independent variable ($n_x = 1$). The computations with n_I-dimensional gradient vectors are thereby restricted to the lower subtree (corresponding to $x(\mathbf{u})$). The concatenation of both subtrees requires the evaluation of the matrix product (30). The splitting becomes more favorable when the number of independent variables u_i increases, or when the height of the computational subtree for $y(x)$ increases.

3.4. Implementation of AD algorithms

Basically, there are two ways to implement the above algorithms in a computer program. Perhaps the most straightforward way is to implement the DA operators (18). This can be most elegantly carried out in a language that supports operator overloading in which case the symbols used for the DA operators can be taken to be the same as for the usual arithmetic operators (exp(·), unary minus, sqrt(·), etc., addition, subtraction, multiplication, division, etc.) and the symbols used for the operands have a type corresponding to the DA element structures (16). In this way the difference between a program like (19) to evaluate a function and the corresponding program that also evaluate its derivatives is minimal, so that a user programming a function does not even have to know at all that his program will compute these derivatives. If the language does not support operator overloading different symbols have to be used for the operators. Examples of operator overloading are ADOL-C (C/C++, [15]), FADBAD (C/C++, [2]) and `autodiff::Fad<T_float>` (C/C++, [1]), KDPACK (FORTRAN-77, [39]) and FastDer (FORTRAN-77, [35]), IMAS (FORTRAN-90, [30]), and ADMAT (Matlab, [38]).

An alternative way is to build a source code translator which reads the source code for a subprogram `W` that computes the value of a function and generates the source code

for a subprogram `W_dW` that evaluates its partial derivatives as well. Examples of this approach are Jake-F (FORTRAN-77, [18]), GRESS (FORTRAN-77, [19]) and PADRE2 (FORTRAN-77, [20]) and, more recently, ADIC (C/C++, [5]), ADIFOR (FORTRAN-77, [7]). This approach has important advantages regarding computational efficiency because the translation of the source code `W` and its execution are separated in time and the translator has a global view on the computational tree. Consequently, high level code optimisations can be applied. The advent of advanced C++ techniques, such as expression templates [37], opens up similar advantages for operator overloading approaches [1]. Its implementation is, however, extremely complicated as it requires complete knowledge of the programming language in which the source code for the subprogram `W` is written. A more complete overview of existing AD systems is presented in [12].

The application of AD techniques is very attractive from the user's point of view as it can relieve him from the time-consuming, error-prone and tedious task of linearising the PDE. Just as in the linear case the user would then need to specify only the parameter $\lambda(\mathbf{x}(\xi); u)$ and the forcing function $f\mathbf{x}(\xi); u)$. The trade-off is payed by the developer of PDE solver who has to implement the linearised equations and assemble its terms from, and their derivatives with respect to $\mathbf{u}$, which are evaluated by some AD-tool, hidden from the user.

In the next section we will highlight some issues concerning computational efficiency in the framework of nonlinear PDE systems, discretised by FEM, FVM, or FDM. The key observation, here, is the fact that the numerical solution of nonlinear PDE systems by means of Newton–Raphson techniques requires the evaluation of some quantities and their derivatives in a large number of integration points.

4. Vectorisation of forward and reverse mode

An important question is whether AD can also be computationally efficient. In principle reverse mode (27) has the better complexity characteristics for the computation of gradients. It requires, however, that the computational tree be stored in memory during the forward sweep and this can only be released gradually during the reverse sweep, unless specialised techniques are applied, trading off storage for recalculation, e.g., checkpointing [14]. In an operator overloading implementation the computational tree must be stored in dynamically allocated data structures as the tree is constructed at runtime. This incurs some non-negligible overhead on reverse mode operator overloading implementations. A source translator can avoid this by hardcoding, because the translation of the algorithm and its execution are separated in time. The forward mode (21) does not have this problem: there is no need to store the computational tree. However, often quite some unnecessary operations are being carried out. Consider the following function evaluating the length of a vector $\mathbf{u} \in \mathbb{R}^{n_I}$:

$$w(\mathbf{u}) = \sum_{i=1-n_I}^{0} u_i^2, \tag{31}$$

whose gradient will be evaluated as:

$$\nabla_{\mathbf{u}} w(\mathbf{u}) = \sum_{i=1-n_I}^{0} 2u_i \nabla_{\mathbf{u}} u_i = \sum_{i=1-n_I}^{0} 2u_i \mathbf{e}_{i+n_I}. \tag{32}$$

In this simple function the number of unnecessary operations by far exceeds the number of useful operations. It has $n_I \times (n_I - 1)$ multiplications by 0, $n_I \times 1$ multiplications by 1, and $(n_I - 1) \times (n_I - 1)$ additions of 0, totaling $2n_I^2 - 2n_I + 1$ useless computations. The number of useful computations is n_I multiplications by 2, and $n_I - 1$ additions, totaling only $2n_I - 1$. Clearly, only if the gradient of the arguments is fully populated (i.e. has only nonzero elements), there will be no unnecessary operations. Example (31) is not a worst case, neither is it a typical. Any independent variable which is introduced near the top of a computational subtree or which occurs only in some of the dependent variables and not in the other ones, implies a lot of useless computations. This problem has been noticed before and led to the introduction of the concept of partially separable functions [6,16,17]. Expressed in this framework, the gradient (32) can be computed more efficiently as

$$\nabla_{\mathbf{u}} w(u) = \begin{bmatrix} 2u_{1-n_I} \\ 2u_{2-n_I} \\ \vdots \\ 2u_0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}, \tag{33}$$

rather than as

$$\nabla_{\mathbf{u}} w(u) = \begin{bmatrix} 2u_{1-n_I} \\ 2u_{2-n_I} \\ \vdots \\ 2u_0 \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}. \tag{34}$$

In general, a function $w(\mathbf{u})$ is called *partially separable* if it can be represented in the form:

$$w(\mathbf{u}) = \sum_{j=1}^{m} w^j(\mathbf{u}), \tag{35}$$

where w^j depends on $n^j \ll n_I$ variables. Its gradient can be computed as:

$$\nabla_{\mathbf{u}} w(u) = \mathbf{w}'\mathbf{S}, \qquad \mathbf{w}' = [\nabla_{\mathbf{u}} w^1 \quad \nabla_{\mathbf{u}} w^2 \quad \dots \quad \nabla_{\mathbf{u}} w^m]^{\mathrm{T}}, \tag{36}$$

where $\mathbf{S} \in \mathbb{R}^{m \times \max\{n^j\}}$ is a called the seed matrix. It contains only ones and zeros, and its columns are orthogonal. In (34) $\mathbf{S}$ is the unit matrix, and in (34) it is a single column of ones. By selecting a good seed matrix useless computations with zeros are effectively avoided, exploiting the sparsity of $w(\mathbf{u})$. Of course, the determination of the seed matrix has its own cost, but if a single seed matrix can used for many evaluations of $\nabla w(\mathbf{u})$, there can be huge profits.

A further source of overhead in operator overloading implementations, is that there is a function call for every algebraic operator in the program since every operator will have to be replaced by a DA operator. This gives rise to as many function calls as there are vertices in the computational tree. A source translator implementation can also avoid this by hardcoding.

Finally, all the overhead just described, will occur every time the dependent variable $w(\mathbf{u})$ is evaluated, and hence it will be proportional to the total number of integration points in the discretised PDE system. One possible way to try to eliminate some of the overhead is to make use of expression templates [37]. This is a C++ technique by which the work to build the computational tree is carried out in compile time rather than in runtime. This approach has been taken by [1]. Here, a different approach is taken, exploiting the nature of the solver.

Fortunately, in the framework of FEM[2] we can extend the overloading operator approach in a way that almost completely eliminates the overhead in both forward and reverse mode. The key observation is that by the nature of the discretisation derivatives of only a limited number of functions are needed in a large number of points, i.e. the total number of integration points (this is found by counting each element with a multiplicity equal to the number of integration points in the element). In addition, in this setting, the number of independent variables is usually small. At most, it is equal to the number of unknown fields in the PDE system. So, typically $n_I < 10$. In the example problem discussed in section 2 only the derivative (13), $\lambda'(\mathbf{x}; u) = \partial\lambda(\mathbf{x}; u)/\partial u$, would be needed in every integration. If the above DA operators are extended to evaluate the function and its derivatives not in a single point but in an array of n_{ip} points the overhead would be incurred only once rather than n_{ip} times. We will address this technique as vectorised AD.

In a classical FEM application the algorithm for computing and assembling element matrices and vectors, reads:

```
for e = 1, 2, ..., N_elements                    // loop over elements
  for q = 1, 2, ..., N_ip^(e)                    // loop over integration points
    u_q^(e) = sum_i^(N_nodes^(e)) u_i N_i(ξ_q)   // current integration point
    compute λ(u_q^(e)) and λ'(u_q^(e))
    for i = 1, 2, ..., N_nodes^(e)               // loop over element nodes
      compute contribution of integration point u_q^(e) to b_i^(e) (equation (12))
      for j = 1, 2, ..., N_nodes^(e)             // loop over element nodes
        compute contribution of integration point u_q^(e) to A_ij^(e) (equation (12))
    assemble the contribution of the current integration point to b and A.
```

(37)

[2] The idea applies equally well to FDM and FVM.

In order to apply vectorised AD, the computations of $\lambda(u_q^{(e)})$ and $\lambda'(u_q^{(e)})$ for all integration points should be grouped:

$$
\begin{array}{l}
\alpha = 0 \\
\text{for } e = 1, 2, \ldots, N_{\text{elements}} \qquad \text{// loop over elements} \\
\quad \text{for } q = 1, 2, \ldots, N_{ip}^{(e)} \qquad \text{// loop over integration points} \\
\qquad \alpha = \alpha + 1 \\
\qquad u^{\alpha} = \sum_i^{N_{\text{nodes}}^{(e)}} u_i N_i(\xi_q) \qquad \text{// current integration point} \\
\text{compute } \lambda(u^{\alpha}) \text{ and } \lambda'(u^{\alpha}) \text{ for all } \alpha \text{ using vectorised AD} \\
\alpha = 0 \\
\text{for } e = 1, 2, \ldots, N_{\text{elements}} \qquad \text{// loop over elements} \\
\quad \text{for } q = 1, 2, \ldots, N_{ip}^{(e)} \qquad \text{// loop over integration points} \\
\qquad \alpha = \alpha + 1 \\
\qquad \text{for } i = 1, 2, \ldots, N_{\text{nodes}}^{(e)} \qquad \text{// loop over element nodes} \\
\qquad\quad \text{compute contribution of integration point } u^{\alpha} = u_q^{(e)} \text{ to } \mathbf{b}_i^{(e)} \\
\qquad\quad \text{for } j = 1, 2, \ldots, N_{\text{nodes}}^{(e)} \qquad \text{// loop over element nodes} \\
\qquad\qquad \text{compute contribution of integration point } u^{\alpha} = u_q^{(e)} \text{ to } \mathbf{A}_{ij}^{(e)} \\
\qquad \text{assemble the contribution of the current integration point to } \mathbf{b} \text{ and } \mathbf{A}.
\end{array}
\tag{38}
$$

Vectorised AD allows an efficient computation of $\lambda(u^{\alpha})$ and $\lambda'(u^{\alpha})$. The user has only to write code for $\lambda(u^{\alpha})$ as the overloaded operators take care of the derivative evaluations behind the scene. The extension to vectorised AD is effected by replacing the arguments u_i of the function $w(u_1, \ldots, u_n)$ (see equation (15)) by column vectors $[u_i]$ of length n:

$$
[u_i] = \begin{bmatrix} u_i^1 \\ u_i^2 \\ \vdots \\ u_i^n \end{bmatrix}. \tag{39}
$$

If we define:

$$
[w] = w\big([u_1], [u_2], \ldots, [u_{n_I}]\big) \equiv \begin{bmatrix} w\big(u_1^1, u_2^1, \ldots, u_{n_I}^1\big) \\ w\big(u_1^2, u_2^2, \ldots, u_{n_I}^2\big) \\ \vdots \\ w\big(u_1^n, u_2^n, \ldots, u_{n_I}^n\big) \end{bmatrix}, \tag{40}
$$

and

$$
[\nabla_{\mathbf{u}} w] = \nabla_{\mathbf{u}} w\big([u_1], [u_2], \ldots, [u_{n_I}]\big) \equiv \begin{bmatrix} \nabla_{\mathbf{u}} w\big(u_1^1, u_2^1, \ldots, u_{n_I}^1\big) \\ \nabla_{\mathbf{u}} w\big(u_1^2, u_2^2, \ldots, u_{n_I}^2\big) \\ \vdots \\ \nabla_{\mathbf{u}} w\big(u_1^n, u_2^n, \ldots, u_{n_I}^n\big) \end{bmatrix}, \tag{41}
$$

we can express the extension as:

$$
\begin{aligned}
\widehat{[u_i]} &\equiv \begin{bmatrix} [u_i] & [\nabla_{\mathbf{u}} u_i] \end{bmatrix} = \begin{bmatrix} [u_i] & [\mathbf{e}_i] \end{bmatrix} = \begin{bmatrix} u_i^1 & \mathbf{e}_i \\ u_i^2 & \mathbf{e}_i \\ \vdots & \vdots \\ u_i^n & \mathbf{e}_i \end{bmatrix}, \\
\widehat{[w]} &\equiv \begin{bmatrix} [w] & [\nabla_{\mathbf{u}} w] \end{bmatrix} = \begin{bmatrix} w\big(u_1^1, u_2^1, \ldots, u_{n_I}^1\big) & \nabla_{\mathbf{u}} w\big(u_1^1, u_2^1, \ldots, u_{n_I}^1\big) \\ w\big(u_1^2, u_2^2, \ldots, u_{n_I}^2\big) & \nabla_{\mathbf{u}} w\big(u_1^2, u_2^2, \ldots, u_{n_I}^2\big) \\ \vdots & \vdots \\ w\big(u_1^n, u_2^n, \ldots, u_{n_I}^n\big) & \nabla_{\mathbf{u}} w\big(u_1^n, u_2^n, \ldots, u_{n_I}^n\big) \end{bmatrix}.
\end{aligned}
\tag{42}
$$

Objects of the type (42) will still be termed DA elements as the distinction with the case where $n = 1$ will be clear from the context. If the columns of $\widehat{[w]}$ are numbered from 0 to n_I, the 0th column is called the *value column*, and the remaining columns are called *derivative columns*. A k-ary operator $\widehat{[\varphi]}$ on objects of type (42) is then expressed as:

$$
\widehat{[\varphi]}\big(\widehat{[v_1]}, \ldots, \widehat{[v_k]}\big) \equiv \begin{bmatrix} \big[\varphi(v_1, \ldots, v_k)\big] & \left[\sum_{j=1}^{k} \frac{\partial \varphi}{\partial v_j} \nabla_{\mathbf{u}} v_j\right] \end{bmatrix}. \tag{43}
$$

In this way the number of function calls is reduced by a factor n as one function call handles n integration points at a time. The extension also offers other interesting perspectives. If the data structure corresponding to a DA element is implemented in terms of dynamically allocated columns (of length n), columns with a constant value (most often 0 and 1, in the gradient columns) can be replaced by a scalar. In the Forward Mode algorithm a simple test will allow to reduce an elementwise computation on two columns to a computation on a column and a scalar, or even a computation on two scalars. Thereby, the above mentioned unnecessary computations with 0 and 1 in the forward mode can be eliminated efficiently. In the framework of partially separable functions, this means that the optimal seed matrix $\mathbf{S}$ (36) is determined on the fly, and without user intervention. However, as the same seed matrix will be used in n evaluations of $\nabla w(\mathbf{u})$ its cost is amortized effectively.

The extension to vectorised AD has also advantages in reverse mode as the overhead of dynamically building and storing the computational graph is now incurred only once. Thus in vectorised AD all sources of overhead are reduced to a minimum. As usually, this comes at the cost of increased memory requirements by a factor proportional to the column vector length n.

An efficient implementation of vectorised AD, FastDer++, is described in [34]. In this C++ class library the vectorised DA elements are represented by objects of class `DaElement`. All the usual operators acting on doubles are overloaded by vectorised DA operators. As a consequence the difference between a code to evaluate some quan-

tity $w(\mathbf{u})$ of type double and a code to evaluate its vectorised DA element ($\mathbf{u}$) is only visible in the declaration of the variables (DaElements rather than doubles) and in the initialisation of the variables. All subsequent examples are carried out with FastDer++.

A final remark regarding the term vectorised AD is appropriate. The quotation *vectorised* not only refers to the fact that the scalars in the DA elements $\widehat{w}$ have been replaced by column vectors to yield the vectorised DA elements $\widehat{[w]}$. It was intentionally chosen because the data structures and code organisation of FastDer++ bare heavily on the – now quite obsolete – concept of code vectorisation for vector computers (see, e.g., [31]). As much of the principles of the old vector processors have survived in current superscalar processors, these concepts are still very valuable for producing efficient code.

5. Example problems

Two example problems will be discussed in detail. First, a test problem based on a quasilinear Poisson equation, is investigated for didactical purposes, and for demonstrating the potential of the concept of vectorised AD in FEM/FDM/FVM applications. Next, a real life example from unsteady one-dimensional flow in a pipe is solved.

Both examples will be solved in the framework of least-squares FEM for first order systems of PDEs. The method is known as LSFEM [21] or FOSLS (First Order System – Least Squares [9,10]). It relies on a first order system formulation of the PDE system. Consider a problem defined by the set of PDEs:

$$\begin{aligned} \mathbf{Au} &= \mathbf{f} \quad \text{in } \Omega \subset \mathbb{R}^d, \\ \mathbf{Bu} &= \mathbf{0} \quad \text{on } \partial\Omega, \end{aligned} \tag{44}$$

where $\mathbf{A}$ is the differential operator, $\mathbf{B}$ the boundary operator, $\mathbf{u}(\mathbf{x})$ is the vector of unknown fields over the problem domain Ω and its boundary $\partial\Omega$, and $\mathbf{f}$ is the forcing vector. If we restrict ourselves for a moment to general linear systems of first-order PDEs, i.e. systems where the differential operator $\mathbf{A}$ in equation (44) of the form:

$$\mathbf{Au} = \sum_{k=1}^{d} \mathbf{A}_k \frac{\partial \mathbf{u}}{\partial x\_k} + \mathbf{A}_0\mathbf{u}, \quad \mathbf{A}_0, \mathbf{A}_k \in \mathbb{R}^{N_{\text{eq}} \times N_{\text{un}}} \ (k = 1, \ldots, d), \quad \mathbf{f} \in \mathbb{R}^{N_{\text{eq}}}. \tag{45}$$

Here, N_{un} and N_{eq} are the number of unknowns, respectively the number of equations. Since equations (44), (45) define a first-order system, the boundary conditions are necessarily essential and the boundary operator $\mathbf{B}$ is algebraic. They constitute the standard matrix form of a general linear system of first-order PDEs. The LSFEM amounts to a minimisation of the squared residual:

$$\|\mathbf{R}\|_0^2 = \int_\Omega (\mathbf{Av} - \mathbf{f})^2 \, \mathrm{d}\Omega \geqslant 0, \tag{46}$$

over the space of trial functions **v**. This is equivalent to to finding **u** such that:

$$\int_\Omega (\mathbf{Av})^{\mathrm{T}}(\mathbf{Au} - \mathbf{f})\,\mathrm{d}\Omega = 0, \quad \forall \mathbf{v}. \tag{47}$$

Expanding **u** in terms of FEM basis functions $N_j(\mathbf{x})$ with equal order elements for all unknown variables leads to a linear system:

$$\mathbf{KU} = \mathbf{F}, \tag{48}$$

where **U** is the global vector of nodal values. The global matrix **K** and global vector **F** are assembled in the usual way from the element matrices and vectors:

$$\begin{aligned}\mathbf{K}^{(e)} &= \int_\Omega (\mathbf{A}N_1 \quad \mathbf{A}N_2 \quad \ldots \quad \mathbf{A}N_{n^{(e)}})^{\mathrm{T}}(\mathbf{A}N_1 \quad \mathbf{A}N_2 \quad \ldots \quad \mathbf{A}N_{n^{(e)}})\,\mathrm{d}\Omega \\ \mathbf{F}^{(e)} &= \int_\Omega (\mathbf{A}N_1 \quad \mathbf{A}N_2 \quad \ldots \quad \mathbf{A}N_{n^{(e)}})^{\mathrm{T}}\mathbf{f}\,\mathrm{d}\Omega,\end{aligned} \tag{49}$$

where $n^{(e)}$ is the number of nodes in the element and

$$\mathbf{A}N_j = \sum_{k=1}^{d} \mathbf{A}_k \frac{\partial N_j}{\partial x\_k} + \mathbf{A}_0 N_j. \tag{50}$$

It can be demonstrated [21] that for the system defined by (44)–(45), under the usual conditions that the system is well posed and the solution **u** is sufficiently smooth, LSFEM gives a convergent solution, independent of the mathematical type of the PDE system. For strictly elliptical systems, the convergence can be proven [21] to be optimal for all variables, i.e. the error of the LSFEM solution is of the same order as the interpolation error. Least-squares FEM has some important advantages comparing to more conventional FEM practices. It has become standard practice to employ different numerical schemes for different types of differential equations. To the contrary, the LSFEM has a uniform formulation for the numerical solution of all types of PDEs and avoids special problem dependent treatments such as upwinding, artificial dissipation, staggered grids, or non-equal order elements, the LBB condition, artificial compressibility, operator-splitting, operator preconditioning and the selection of user tuneable parameters. In addition, it always leads to symmetric positive-definite matrices, which can be efficiently solved by matrix methods, such as the preconditioned conjugate gradient method. The speculation that the system of linear algebraic equations (48) generated by the least-squares method has a bad condition number is a common misconception [21]. If a second-order differential equation is approximated using the Galerkin finite element method with a uniform mesh, the condition number of the resulting global matrix is $\mathrm{O}(h^{-2})$. The LSFEM method, however is based on first order differential equations and is therefore also $\mathrm{O}(h^{-2})$. It is also suited for concurrent simulation of multiple physics, as most coupled physics problems can be easily cast into a system of first-order PDEs. These properties provide a way to construct a solver that is able to treat any problem that can be cast into a system of first-order PDEs. Since the user only needs to provide

the coefficients of the equation, the load vector, and the boundary conditions, the cost of solver development for new PDE based problems is drastically reduced. A disadvantage of LSFEM is probably that the number of equations – and hence also the size of the global matrix – increases. E.g., in three dimensions, the incompressible Navier–Stokes equations normally comprise four equations and four unknowns (velocity $\mathbf{v} \in \mathbb{R}^3$ and the pressure $p \in \mathbb{R}$). The corresponding optimal [21] LSFEM formulation, however, has eight equations and seven unknowns. Consequently, the global matrix is twice as large as in a classical formulation based on discretisation of the second-order system. Also, the different first order equations may have different contributions to the residual if they are not appropriately scaled. This may cause some difficulties in multiphysics problems.

The concept of AD becomes very attractive if the problem is relaxed to quasilinear systems of first-order PDEs, where the coefficients matrices $\mathbf{A}_k$ $(k = 0, 1, \ldots, d)$ and forcing vector $\mathbf{f}$ in (44)–(45) are allowed to depend the unknown field vector $\mathbf{u}(\mathbf{x})$:

$$\begin{aligned} \mathbf{A}_k &= \mathbf{A}_k(\mathbf{u}), \\ \mathbf{f} &= \mathbf{f}(\mathbf{u}). \end{aligned} \tag{51}$$

Equations (44)–(45), and (51) define a very general class of quasilinear PDE systems in which many PDE based problems from science and engineering can be formulated. It is a straightforward task to linearise these quasilinear partial differential equations using a Newton–Raphson scheme and develop a solver for the linearised equations. These linearised equations will depend not only on the $\mathbf{A}_k(\mathbf{u})$ $(k = 0, 1, \ldots, d)$ and $\mathbf{f}(\mathbf{u})$, but also on their partial derivatives with respect to $\mathbf{u}$, $\nabla_{\mathbf{u}}\mathbf{A}_k(\mathbf{u})$ and $\nabla_{\mathbf{u}}\mathbf{f}(\mathbf{u})$. The latter, however, can be evaluated by an AD system. So the user only needs to program the quasilinear coefficients $\mathbf{A}_k(\mathbf{u})$ and $\mathbf{f}(\mathbf{u})$, to solve his problem. From his point of view, the solver achieves automatic linearisation of the PDE problem.

5.1. A quasilinear Poisson equation

Consider the quasilinear Poisson equation in two dimensions:

$$\begin{aligned} \nabla \cdot \big(-\lambda(u)\nabla u\big) &= f(x, y) \quad \text{in } \Omega = [0, 1] \times [0, 1], \\ u &= 0 \qquad\qquad \text{on } \partial\Omega, \end{aligned} \tag{52}$$

where

$$\begin{aligned} f(x, y) &= -2\lambda\big(\big(x^2 - x\big) + \big(y^2 - y\big)\big) - \lambda'\big((2x - 1)^2\big(y^2 - y\big)^2 + \big(x^2 - x\big)^2(2y - 1)^2\big), \\ \lambda(u) &= \big(1 + u - 17u^2\big)\exp(3u), \\ \lambda'(u) &= \big(1 - 31u - 51u^2\big)\exp(3u). \end{aligned} \tag{53}$$

The analytical solution of this problem is:

$$u = \big(x^2 - x\big)\big(y^2 - y\big). \tag{54}$$

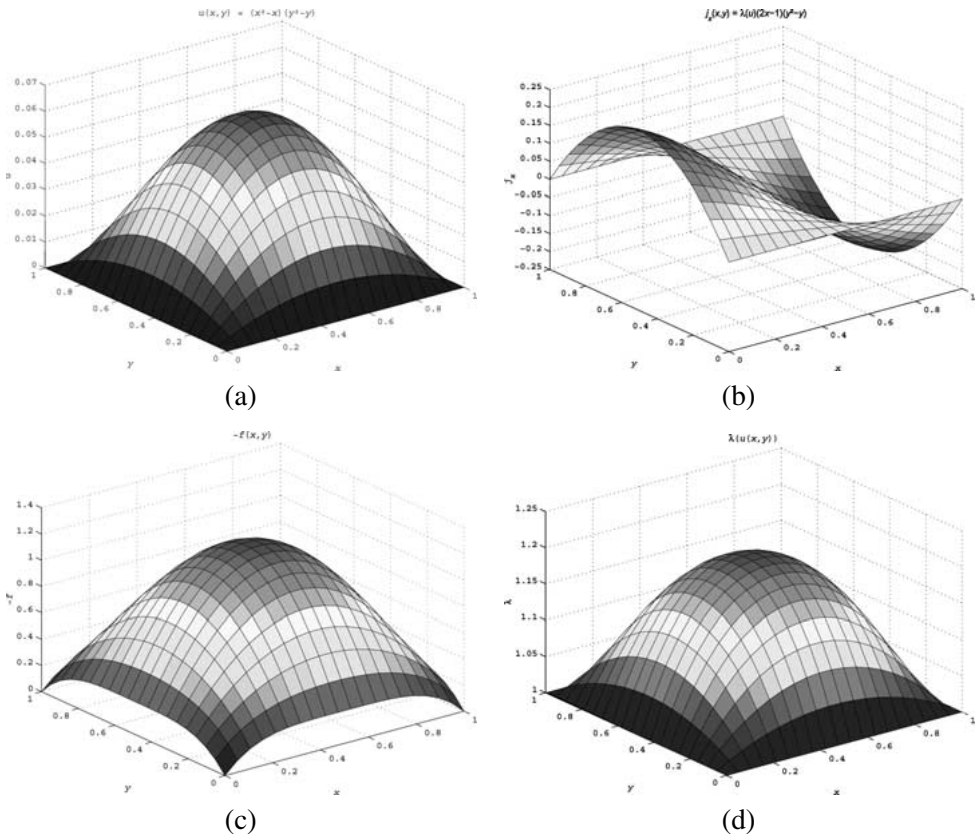

Figure 4. Analytical solution of the Poisson problem (52)–(53): (a) $u(x, y)$, (b) j_x, (c) $-f(x, y)$, (d) $\lambda(u(x, y))$.

Note that we have considered the forcing function $f(x, y)$ as a function of spatial variables only although through λ and λ' it depends also on u. This dependence is eliminated by evaluating λ and λ' for the analytical solution (54) in the evaluation of $f(x, y)$.

The flux components are:

$$\begin{aligned} J_x &= -\lambda(u)\frac{\partial u}{\partial x} = -\lambda(u)(2x-1)\big(y^2-y\big), \\ J_y &= -\lambda(u)\frac{\partial u}{\partial y} = -\lambda(u)\big(x^2-x\big)(2y-1). \end{aligned} \tag{55}$$

The analytical solution to the Poisson problem (52)–(53) is depicted graphically in figure 4. If the PDE (52) is formulated as a first order system:

$$\begin{aligned} \mathbf{j} - \nabla u &= 0 && \text{in } \Omega, \\ \nabla\cdot(-\lambda\mathbf{j}) &= f(\mathbf{x}) && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned} \tag{56}$$

In two dimensions this is a system with three equations and three unknowns (j_x, j_y and u). Consequently, it cannot be elliptical. Although this system can be perfectly solved using conventional LSFEM, a simple modification can cure this problem. In order to do so, one supplements equations (56) with an extra equation and appropriate boundary conditions:

$$\begin{aligned} \nabla \times \mathbf{j} &= 0 \quad \text{in } \Omega, \\ \mathbf{n} \times \mathbf{j} &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{57}$$

where $\mathbf{n}$ is a unit vector normal to the boundary $\partial\Omega$. Intuitively, we can accept that this does not change the problem since $\mathbf{j}$ is a gradient field and the curl of any gradient field is zero. Reference [21] gives a rigorous proof that the solutions of the systems (52) and (56)–(57) are equivalent and that the latter indeed is elliptical, thereby guaranteeing optimality of the error of the LSFEM solution. The supplemented system (56)–(57) is referred to as the optimal LSFEM formulation of system (52). For the two-dimensional case, equations (56)–(57) can be rewritten as

$$\begin{aligned} \frac{\partial j_x}{\partial x} + \frac{\partial j_y}{\partial y} &= -f(x, y) && \text{in } \Omega, \\ \frac{\partial j_y}{\partial x} - \frac{\partial j_x}{\partial y} &= 0 && \text{in } \Omega, \\ \lambda(u)\frac{\partial u}{\partial x} - j_x &= 0 && \text{in } \Omega, \\ \lambda(u)\frac{\partial u}{\partial y} - j_y &= 0 && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega, \\ j_x n_y - j_y n_x &= 0 && \text{on } \partial\Omega. \end{aligned} \tag{58}$$

Hence, the standard matrix form is given by equations (44)–(45) with $d = 2$ and:

$$\begin{aligned} \mathbf{A}_0(\mathbf{u}) &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}, \qquad \mathbf{f}(\mathbf{u}) = \begin{bmatrix} f \\ 0 \\ 0 \\ 0 \end{bmatrix}, \qquad \mathbf{u} = \begin{bmatrix} j_x \\ j_y \\ u \end{bmatrix}, \\ \mathbf{A}_1(\mathbf{u}) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \lambda(u) \\ 0 & 0 & 0 \end{bmatrix}, \qquad \mathbf{A}_2(\mathbf{u}) = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \lambda(u) \end{bmatrix}. \end{aligned} \tag{59}$$

The automatically linearising solver is implemented as a C++ class `LSFEM_nlAD` and makes extensive use of the Diffpack® [25,28]. This is a C++ class library for developing FEM/FDM/FVM solvers. A solver for a particular problem is constructed by creating a class which inherits publicly from class `LSFEM_nlAD`. E.g., for the above Poisson problem (only relevant member functions are shown) one has:

```
class Poisson2D_LSFEM_nlAD : public LSFEM_nlAD{
  virtual void computeEssBC(
    const int bi,       // boundary indicator (input argument)
    const Ptv(real)& x, // some x ∈ ∂Ω (input argument)
    Ptv(real)& uBC);    // u(x) for x ∈ ∂Ω (output argument)
  virtual bool setPDECoefficientTypes();
  virtual void computeNonlinearA_f();
//...
}
```
(60)

The class is used as in the following code fragment:

```
Poisson2D_LSFEM_nlAD mySimulator; // create simulator object
mySimulator.solveProblem();       // run simulation
mySimulator.reportProblem();      // produce report
```
(61)

The member functions solveProblem and reportProblem called in the above code fragment are inherited from the base class `LSFEM_nlAD`.

The member functions shown in code fragment (60) are user defined member functions to specify problem dependent issues. The function computeEssBC computes the boundary condition in for some point `x` on some part of the boundary identified by the boundary indicator `bi`, and stores it in a vector `uBC` ($\mathbf{u}(\mathbf{x}) = [u_1 \quad u_2 \quad \ldots \quad u_{N_{\text{un}}}]$). If there is no boundary condition for some $u_i,\ i \in \{1, 2, \ldots, N_{\text{un}}\}$, a predefined constant `NOBC` is assigned. According to the last two equations of (58) one has:

```
void Poisson2D_LSFEM_nlAD::
computeEssBC(const int bi, const Ptv(real)& x, Ptv(real)& uBC) {
  switch (bi) {
  case 1:                          // x=1
  case 3:                          // x=0
    uBC(1) = NOBC;                 // no BC
    uBC(2) = 0.0;                  // BC: jy=0
    uBC(3) = 0.0;                  // BC: u =0
      break;
  case 2:                          // y=1
  case 4:                          // y=0
    uBC(1) = 0.0;                  // BC: jx=0
    uBC(2) = NOBC;                 // no BC
    uBC(3) = 0.0;                  // BC: u =0
      break;
  }
}
```
(62)

The four sides of the problem domain $\Omega = [0, 1] \times [0, 1]$ are numbered counter clockwise starting with the rightmost side. The two other functions are central to the LSFEM approach with integrated AD. The member function `setPDECoefficient-`

`Types` serves to specify the nature of the coefficients in $\mathbf{A}_k(\mathbf{u})$ ($k = 0, 1, \ldots, d$) and $\mathbf{f}(\mathbf{u})$.

```
void Poisson2D_LSFEM_nlAD::setPDECoefficientTypes() {
  static bool types_set = false;
  if (!types_set) {
    setType_f(1,SPATIAL);          // f_1 depends on x
    setType_A(0,3,1,CONSTANT);     // (A_0)_31 is a constant coefficient
    setType_A(0,4,2,CONSTANT);     // (A_0)_42 is a constant coefficient
    setType_A(1,1,1,CONSTANT);     // (A_1)_11 is a constant coefficient
    setType_A(1,2,2,CONSTANT);     // (A_1)_22 is a constant coefficient
    setType_A(1,3,3,3);            // (A_1)_33 depends on u_3
    setType_A(2,1,2,CONSTANT);     // (A_2)_12 is a constant coefficient
    setType_A(2,2,1,CONSTANT);     // (A_2)_21 is a constant coefficient
    setType_A(2,4,3,3);            // (A_2)_43 depends on u_3
    types_set = true;
    return true;
  }
  return false;
}
```
(63)

The solver uses this information to find out which independent variables and spatial variables will actually be needed in the computation, and initialises the corresponding DA element data structures. In the above code fragment the solver will detect that it needs the spatial variables x and y to compute the forcing function $f(x, y)$ in (52), and the independent variable $u_3 = u$, but not the independent variables $u_1 = j_x$ and $u_2 = j_y$. Hence, no memory space and runtime will be wasted for initialising $[\widehat{u_1}]$ and $[\widehat{u_2}]$. Note that x and y are independent of $\mathbf{u}$, hence the DA element corresponding to, e.g., x will be represented as:

$$[\widehat{x}] = \begin{bmatrix} x^1 & 0 & 0 & \ldots & 0 \\ x^2 & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x^n & 0 & 0 & \ldots & 0 \end{bmatrix}.$$

FastDer++ is designed to deal efficiently with DA elements like this. The DA operators detect the presence of a vanishing gradient part and do not waste time computing all the multiplications with zero implied by the chain rule in equation (43). In fact, the vanishing columns are not even stored (see section 4). Before each Newton–Raphson iteration, the independent variables of course need to be updated. The solver will also verify if the underlying grid remains unchanged. If not, also $[\widehat{x}]$ and $[\widehat{y}]$ will be updated. Coefficients remaining unspecified in `setPDECoefficientTypes` are considered constant by default, i.e. they are considered independent of $\mathbf{u}$. The static `bool` variable `types_set` assures that the body of the function `setPDECoefficientTypes` is executed only once. The return value signals to its caller whether or not the coefficient

type information has changed. In this case it has changed only after the first call to function `setPDECoefficientTypes`. In time dependent problems the nature of the coefficients may change during the course of the computation. By returning `true` the `setPDECoefficientTypes` function can signal this to its caller, and the solver can undertake appropriate actions.

The member function `computeNonlinearA_f` is the hart of the solver. It computes the coefficients $\mathbf{A}_k(\mathbf{u})$ $(k = 0, 1, \ldots, d)$ and $\mathbf{f}(\mathbf{u})$, as well their partial derivatives with respect to $\mathbf{u}$, $\nabla_{\mathbf{u}}\mathbf{A}_k(\mathbf{u})$ and $\nabla_{\mathbf{u}}\mathbf{f}(\mathbf{u})$.

```
void Poisson2D_LSFEM_nlAD::computeNonlinearA_() {
  static bool const_coeff_computed = false;
  double lambda[3] = -2, -2, 34;  // polynomial coeff in −2λ
  double dlambda[3] = -4, 31, 51; // and −λ′, see eq. (53)
  if (!const_coeff_computed) {
    Au(0,3,1) = -1;  // Compute constant coefficients only once
    Au(0,4,2) = -1;
    Au(1,1,1) =  1;
    Au(1,2,2) =  1;
    Au(2,1,2) =  1;
    Au(2,2,1) = -1;
    // compute f(x,y) according to (53):
    DaElement xx_min_x = x(1)*(x(1) - 1);  // x^2 − x
    DaElement yy_min_y = x(2)*(x(2) - 1);  // y^2 − y
    DaElement u_anal = xx_min_x*yy_min_y; // analytical solution
    fu(1) = exp(3*u_anal)
      *(pol(u_anal,3, lambda)*(xx_min_x+yy_min_y)
      + pol(u_anal,3,dlambda)
      *((xx_min_x*(2*x(2) - 1))^2
      +((2*x(1) - 1)*yy_min_y)^2)));
  }
    // Compute quasilinear coefficients:
    Au(1,3,3) = pol(uu(3),3,lambda)*exp(3*uu(3));
    Au(2,4,3) = Au(1,3,3);
}
```

(64)

In the above code fragment the function

```
DaElement& pol(DaElement& x,       // variable of the polynomial
               const int noCoef,   // order of the polynomial + 1
               double* coef);      // coefficient array
```

evaluates a polynomial using the algorithm:

$$\begin{aligned} P(x) &= c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \\ &= c_0 + c_1 x\big(\cdots + x\big(c_{n-2} + x(c_{n-1} + x c_n)\big)\cdots\big). \end{aligned}$$

Note that if the above computation (64) had not declared any variables to store intermediate results, like `xx_min_x`, `yy_min_y` and `u_anal`, there would be no sign at all that the user is actually computing with objects of type `DaElement`, rather than `double`. This is due to the careful design of the AD library FastDer++, using operator overloading. Every operator acting on `doubles` has a `DaElement` equivalent, so that any computation with `doubles` can be easily carried over in a computation on `DaElements`.

The last statement in code fragment (64) shows a potential performance problem. In the problem at hand, there are two matrix elements actually storing the same quantity:

$$\mathbf{A}_{1,33}(\mathbf{u}) = \mathbf{A}_{2,43}(\mathbf{u}) = \lambda(u).$$

In the current FastDer++ implementation the statement

```
Au(2,4,3) = Au(1,3,3);
```

means that a physical copy is made, so that the DA element ($\mathbf{u}$) is actually stored twice. This is a waste of memory space and runtime. The problem can be alleviated by using reference counted objects implementing copy on write (see, e.g., [24], chapter 7).

Remember that in the problem definition (52)–(53) we choose to regard the forcing function $f(x, y)$ as a function of spatial variables only, which we can since the analytical solution is known. This choice is arbitrary, however, and we could equally well have considered it as a function of u as well. In that case the computation of coefficient `fu(1)` would have to be moved out of the `if (!const_coeff_computed)` body and `u_anal` would have to be replaced by `uu(3)`. The factors depending on spatial variables only can be kept inside the condition body for efficiency.

```
...// as before in (64)
DaElement lambda_factor_xy, dlambda_factor_xy;
if (!const_coeff_computed) {
  ...// as before in (64)
  // compute factors in f(x,y,u) depending on x and y only :
  DaElement xx_min_x = x(1)*(x(1) - 1);   // x^2 - x
  DaElement yy_min_y = x(2)*(x(2) - 1);   // y^2 - y
   lambda_factor_xy = xx_min_x + yy_min_y
  dlambda_factor_xy = ((2*x(1)-1)*yy_min_y)^2
                        +(xx_min_x*(2*x(2)-1))^2;
}
// Compute quasilinear coefficients:
fu(1) = exp(3*uu(3))
       *(pol(uu(3),3, lambda)* lambda_factor_xy
         +pol(uu(3),3,dlambda)* dlambda_factor_xy);
...// as before in (64)
```
(65)

The conventional way of dealing with nonlinear partial differential equations, however, is to linearise the system by hand (or using a symbolic algebra package), and to

program the linearised equations. Class `LSFEM_nlAD` is also capable of dealing with manually linearised equations. For the problem at hand (52)–(53) the standard matrix formulation of the Newton–Raphson linearised PDE system is given by:

$$
\begin{aligned}
\bar{\mathbf{A}}_1^{NR,s} &= \mathbf{A}_1\left(\mathbf{u}^{s-1}\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \lambda(u^{s-1}) \\ 0 & 0 & 0 \end{bmatrix}, \\
\bar{\mathbf{A}}_2^{NR,s} &= \mathbf{A}_2\left(\mathbf{u}^{s-1}\right) = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \lambda(u^{s-1}) \end{bmatrix}, \\
\bar{\mathbf{A}}_0^{NR,s} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & \left.\dfrac{\partial \lambda}{\partial u}\right|_{u^{s-1}} \left.\dfrac{\partial u}{\partial x}\right|_{u^{s-1}} \\ 0 & -1 & \left.\dfrac{\partial \lambda}{\partial u}\right|_{u^{s-1}} \left.\dfrac{\partial u}{\partial y}\right|_{u^{s-1}} \end{bmatrix}, \\
\bar{\mathbf{f}}^{NR,s} &= \begin{bmatrix} -f(x,y) - \left(\left.\dfrac{\partial j_x}{\partial x}\right|_{u^{s-1}} + \left.\dfrac{\partial j_y}{\partial y}\right|_{u^{s-1}} \right) \\ -\left(-\left.\dfrac{\partial j_y}{\partial x}\right|_{u^{s-1}} + \left.\dfrac{\partial j_x}{\partial y}\right|_{u^{s-1}} \right) \\ -\left(\lambda(u^{s-1}) \left.\dfrac{\partial u}{\partial x}\right|_{u^{s-1}} - j_x^{s-1} \right) \\ -\left(\lambda(u^{s-1}) \left.\dfrac{\partial u}{\partial y}\right|_{u^{s-1}} - j_y^{s-1} \right) \end{bmatrix},
\end{aligned}
\tag{66}
$$

where the overbar denotes the linearised coefficient matrices and vector, which now contain only constants, and $\mathbf{u}^{s-1}$ is the vector of unknown fields at the previous iteration. Obviously, the linearised coefficient matrices and vector are more densely populated and programming and debugging will take longer.

A solver for manually linearised equations, i.e. `Poisson2D_LSFEM_linearised`, must define a `computeLinearA_f` member function rather than the `computeNonlinearA_f` member function in the code fragment for the automatically linearising solver (60).

The task of this member function is to evaluate the linearised coefficients (66) at the current integration point:

```
void Poisson2D_LSFEM_linearised::
computeLinearA_f(const FiniteElement& fe) {
  // Diffpack function calls to retrieve the position
  // of the current integration point:
  Ptv(real) ptv;                    // work data structure
  fe.getGlobalEvalPt(ptv);
  real x = ptv(1);
  real y = ptv(2);
  // Diffpack function calls to retrieve field values at the
  // current integration point:
  double jx_prev = (*u)(1).valueFEM(fe);
  double jy_prev = (*u)(1).valueFEM(fe);
  double u_prev = (*u)(3).valueFEM(fe);
  // Diffpack function calls to retrieve the gradient of the
  // fields at the current integration point:
  (*u)(1).derivativeFEM(ptv,fe);    // j_x-field
  double djx_dx_prev = ptv(1);
  double djx_dy_prev = ptv(2);
  (*u)(2).derivativeFEM(ptv,fe);    // j_y-field
  double djy_dx_prev = ptv(1);
  double djy_dy_prev = ptv(2);
  (*u)(3).derivativeFEM(ptv,fe);    // u-field
  double du_dx_prev = ptv(1);
  double du_dy_prev = ptv(2);                                    (67)
  double lambda_prev, dlambda_du_prev;
  computeLambda(u_prev,             // u^prev, input argument
                lambda_prev,        // λ(u^prev), output argument
                dlambda_du_prev);   // ∂λ/∂u|_{u^prev}, output argument
  // computation of linearised coefficients:
  A0(3,1) = -1.0;
  A0(3,3) = dlambda_du_prev*du_dx_prev;
  A0(4,2) = -1.0;
  A0(4,3) = dlambda_du_prev*du_dy_prev;
  A1(1,1) = 1.0;
  A1(2,2) = 1.0;
  A1(3,3) = lambda_prev;
  A2(1,2) = 1.0;
  A2(2,1) = -1.0;
  A2(4,3) = lambda_prev;
  f(1) = forcingFunction(x,y)       // f(x,y) according to (53)
         - (djx_dx_prev + djy_dy_prev);
  f(2) = - (-djx_dy_prev + djy_dx_prev);
  f(3) = - (lambda_prev*du_dx_prev-jx_prev);
  f(4) = - (lambda_prev*du_dy_prev-jy_prev);
}
```

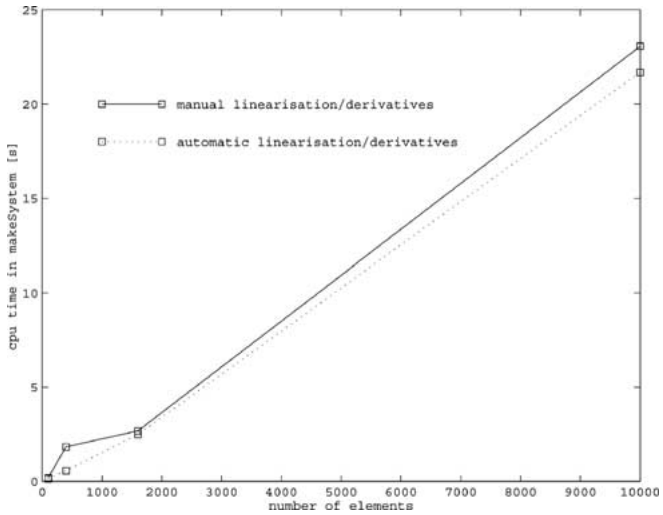

Figure 5. CPU time to build the linear algebraic system (`makeSystem` member function) for the Poisson problem with automatic linearisation and derivative computation (Poisson2D_LSFEM_nlAD solver, upper line), vs. manual linearisation and derivative computation (Poisson2D_LSFEM_linearised solver, lower line) on four different regular grids of bilinear square elements with, respectively.

The interesting part is, of course, the comparison of the computational efficiency of the AD-solver `Poisson2D_LSFEM_nlAD` (60), which does automatic linearisation, and the solver `Poisson2D_LSFEM_linearised`, which relies on manually linearised equations. Figure 5 represents the CPU time these solvers spend in the Diffpack® `makeSystem` member function, i.e. the time they need to build (but not solve) the linear algebraic systems. Computations have been carried out for four different regular grids of bilinear square elements: respectively with 10×10, 20×20, 40×40, and 100×100 elements, with one integration point per element. Note that the solution scheme in both solvers is exactly the same, i.e. Newton–Raphson iteration. The only difference is that the AD-solver `Poisson2D_LSFEM_nlAD` (60) evaluates the quasi-linear coefficient matrices $\mathbf{A}_k(\mathbf{u})$ $(k = 0, 1, \ldots, d)$ and vector $\mathbf{f}(\mathbf{u})$ using the AD system FastDer++, and from these, it derives the linearised coefficient matrices $\bar{\mathbf{A}}_k(\mathbf{u})$ $(k = 0, 1, \ldots, d)$ and vector $\bar{\mathbf{f}}(\mathbf{u})$, whereas the solver `Poisson2D_LSFEM_linearised` directly evaluates the linearised coefficient matrices $\bar{\mathbf{A}}_k(\mathbf{u})$ $(k = 0, 1, \ldots, d)$ and vector $\bar{\mathbf{f}}(\mathbf{u})$. Both solvers obtain exactly the same result after three nonlinear iterations. The timing results in figure 5 are most encouraging. The AD-solver `Poisson2D_LSFEM_nlAD` (60) performs slightly better on all four grids. This is surprising as from tests comparing FastDer++ with some other AD systems and hand coded derivative evaluation in a simple loop, a decrease in performance would expected [33]. The main advantage in using AD based solvers is, of course not this performance increase, but the fact that the user only needs to program the nonlinear system and is no longer bothered with linearising equations, and computing and programming derivatives.

5.2. A real life example

One of the advantages of LSFEM is that it provides a universal solution scheme irrespective of the mathematical type of the equations and convergence is guaranteed [21]. Optimality, however, is not always guaranteed, but then, in many cases the cost of solver development is more important than computational efficiency. This motivated us to use the `LSFEM_nlAD` class to numerically solve the stationary and transient 1D flow in elastic pipes. The pipe to be modeled is a section of a agricultural spraying machine. Consider a circular pipe of perfectly elastic material with a length l, radius r_0, and wall thickness e. Spatial and temporal variations of the fluid pressure in the pipe make the pipe expand and contract. This influences the cross section of the pipe, which on its turn influences the flow. The model equations are derived by considering mass and momentum balances together with the relation between the transmural pressure p and the mean circumferential stress σ in the pipe wall [36]:

$$\sigma = \frac{pr}{e}, \tag{68}$$

and the expression of linear elasticity of the wall material:

$$\sigma = E\varepsilon. \tag{69}$$

Here, ε is the strain defined by

$$\varepsilon = \frac{\delta r}{r}.$$

Friction was accounted for by the Blasius formula [11]

$$f = 0.316\,\mathrm{Re}^{-1/4}, \tag{70}$$

with the Reynolds number defined by

$$\mathrm{Re} = \frac{\rho|V|2r}{\mu}.$$

Here, ρ, μ and V are the mass density, viscosity and velocity of the fluid, respectively. In addition the pipe is assumed to be circular and horizontal, and of constant length l and wall thickness e.

In standard matrix form the model equations read:

$$\begin{bmatrix} 0 & A\left(\frac{1}{K}+\frac{1}{L}\right) \\ \frac{1}{A} & -\frac{Q}{A}\frac{1}{L} \end{bmatrix} \begin{bmatrix} \frac{\partial Q}{\partial t} \\ \frac{\partial p}{\partial t} \end{bmatrix} + \begin{bmatrix} 1 & \frac{Q}{K} \\ \frac{Q}{A^2} & \frac{1}{\rho} - \left(\frac{Q}{A}\right)^2 \frac{1}{L} \end{bmatrix} \begin{bmatrix} \frac{\partial Q}{\partial x} \\ \frac{\partial p}{\partial x} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q \\ p \end{bmatrix} = \begin{bmatrix} 0 \\ -f^* \mathrm{sign}(Q) \left|\frac{Q}{r}\right|^{7/4} r^{-3} \end{bmatrix}. \tag{71}$$

Here, the system has been written with the volume flux $Q = VA$ and the transmural pressure p as unknowns. The first equation corresponds to the mass balance, the second represents the momentum balance. The radius $r = r(p)$ is a function of the transmural pressure defined implicitly by the following relation:

$$p = \frac{1}{r} Ee \ln \frac{r}{r_0}, \tag{72}$$

which derives from equations (68), (69). $A = \pi\,(r(p))^2$ is the cross section. K and ρ are the bulk compressibility and the mass density, respectively, of the fluid in the pipe. Considering diluted aqueous solutions, the fluid density ρ is assumed to be pressure dependent according to:

$$\rho(p) = \rho_0 \exp\left(\frac{p}{K}\right). \tag{73}$$

The quantity $L(p)$ in (71) is given by:

$$\frac{1}{L(p)} = \frac{2r(p)}{Ee - pr(p)}. \tag{74}$$

Finally, the friction factor,

$$f^* = 0.00896\left(\frac{\mu}{\rho}\right)^{1/4}$$

was derived from (70) by factoring out the dependency on volume flux and wall radius. The problem domain is the rectangular domain:

$$[0, l] \times [0, t_{\text{end}}] \in \mathbb{R}^2, \tag{75}$$

which is discretised with a regular grid of bilinear square elements. Note that time is not discretised by some finite difference formula as is common in FEM practice. Instead it is considered as a dimension in the space of the problem domain. Time stepping was achieved by splitting the problem domain (75) into subdomains $[0, l] \times [t_i, t_{i+1}]$, where t_i is the ith discretision point in the time direction. This is to take advantage from the fact that the residual in the LSFEM method provides a natural criterion for adaptive meshing [21]. So, in principle the residual can be used for adaptive time stepping in this way [26,27].

The following boundary conditions are applied to the system. At the upstream end of the pipe a constant pressure is applied:

$$p(x = 0, t) = p_{\text{appl}}(t). \tag{76}$$

The downstream end of the pipe ($x = l$) is connected to another pipe with a series of spray nozzles. This part, including the nozzles, is considered as a lumped system, having the effect of an impedance:

$$Q(x = l, t) = Q_{x=l}(p) = cA\sqrt{\frac{2p_{x=l}(t)}{\rho}}. \tag{77}$$

This boundary condition is considerably more involved as it relates the volume flux and the pressure at the end of the pipe in a nonlinear way. Solving the quasilinear problem (71) and the nonlinear boundary condition (77) simultaneously with a Newton–Raphson scheme proved difficult, and instead the problem was solved using an outer successive substitution loop for the nonlinear boundary condition (77):

guess $p^{\text{prev}} = p_{x=l}$
while $|Q(x = l) - Q_{x=l}(p^{\text{prev}})| \geqslant tol$
  compute $Q^{\text{prev}} = Q(p^{\text{prev}})$ according to equation (77)
  solve the quasilinear problem (71) with B.C. (76) and $Q(x = l) = Q_{x=l}$.

Obviously, the problem (71) has several quasilinear coefficients and linearising the problem is a tedious, time-consuming and error-prone task. Applying the `LSFEM_nlAD` class and computing the quasilinear coefficients using FastDer++, the `computeNonlinearA_f` member function can be programmed:

```
void ElasticPipe_Transient_nlbc::computeNonlinearA_f() {
  static bool constant_coefficients_computed = false;
  if(!constant_coefficients_computed) {
    A1_11 = 1;                                  // cfr. eq. (71)
    constant_coefficients_computed = true;
  }
  DaElement& Q = uu(1);                         // aliasing for
  DaElement& p = uu(2);                         // convenience
  DaElement rho = rho0*exp(p/K);                // ρ(p), cfr. eq. (73)
  DaElement r = compute_r_from_p(p);            // r(p), cfr. eq. (72)
  DaElement A = pi*(r^2);                       // A
  DaElement Q_over_A = Q/A;                     // Q/A
  DaElement one_over_L = 2.0*r/(E*e-p*r);       // cfr. eq. (74)
  A2_12 = A*(1.0/K + one_over_L);               // cfr. eq. (71)
  A2_21 = 1/A;                                  // cfr. eq. (71)
  A2_22 = -Q_over_A*one_over_L;                 // cfr. eq. (71)
  A1_12 = Q/K;                                  // cfr. eq. (71)
  A1_21 = Q_over_A/A;                           // cfr. eq. (71)
  A1_22 = 1/rho - (Q_over_A^2)*one_over_L;      // cfr. eq. (71)
  f2 = -fstar*(r^(-3))*((Q/r)^1.75);            // cfr. eq. (71)
}
```
(78)

The member function `compute_r_from_p` solves equation (72) for r for given p by applying two Newton–Raphson iterations to the function

$$F(r) = \frac{1}{r} Ee \ln \frac{r}{r_0} - p = 0 \tag{79}$$

with

$$r = \frac{r_0}{1 - pr_0/Ee} \tag{80}$$

as an initial guess. We show the code to illustrate the flexibility of FastDer++.

```
DaElement& ElasticPipe_Transient_nlbc::
compute_r_from_p(DaElement& p) {
  DaElement r = r0/(1 - p*r0/(E*e));     // initial guess, (82)
  DaElement temp;
  for (int i=1;i<=2;++i) {
    temp = (E*e)*log(r/r0);
    r += = r*(temp - p*r)/(temp - E*e); // r^{i+1} = r^i - F(r^i)/∂_r F(r^i)
  }
  return returnAsTemp(r);
}
```
(81)

The return statement perhaps deserves some comment. In C++ a function cannot return a reference to a local variable, like `r`. Also, we do not want to return a `DaElement` because that would imply a costly call to the `DaElement` copy constructor. The `returnAsTemp` function converts the local variable into a temporary `DaElement` which is managed at a higher level. This is a cheap operation as it involves only a transfer of ownership of data (copying a few pointers). For details the reader is referred to [34].

The above code assumes that both Q and p are in S.I. units. In practice a scaling is applied to both unknowns as to obtain well conditioned linear algebraic systems. This is not shown in code fragment (78). Figure 6 shows the solution of a problem defined by equations (71), (75)–(77) which goes from one steady state to another through a step in the applied pressure at the downstream end of the pipe. The shape of the step is given by

$$p(x = 0, t) = \big(3 + 0.03 \exp(-t/1\text{ s})\big)10^5\text{ Pa}.$$

The volume flux and transmural pressure are shown as a function of space and time.

Linearising this system (71), (75)–(77) by hand and obtaining partial derivatives, although straightforward, is tedious, time-consuming and error-prone. It also produces inflexible code. Changing the model requires changes in at least two places in the code: for the function value and for the partial derivatives. The AD approach greatly facilitates the implementation of changes in the model. In order to illustrate what we have gained we provide the manually linearised equations together with the needed partial derivatives below. They were derived with the aid of a symbolic algebra package. The help of such a package is of course useful, but it does not do the entire job as its results are often not in a form that ensures optimal computational efficiency.

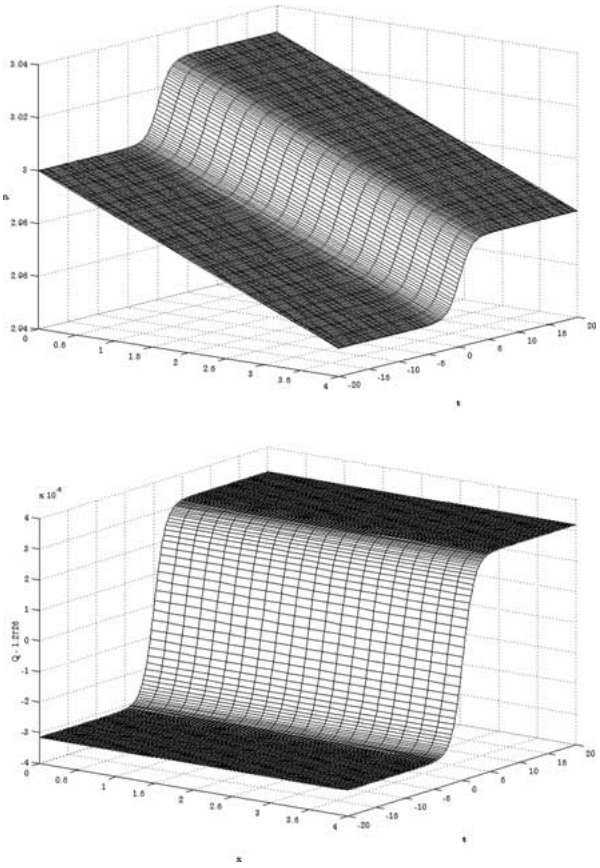

Figure 6. LSFEM solution of an elastic pipe problem defined by equations (71), (75)–(77) going from one steady state to another through a step in the applied pressure at the downstream end of the pipe. The volume flux and transmural pressure p are shown as a function of space and time (x, t).

The Newton–Raphson linearised equations for the elastic pipe problem in standard matrix form are given by (only the nonzero coefficients are given):

$$
\begin{aligned}
&\bar{\mathbf{A}}_k^{NR,s} = \mathbf{A}_k\big(\mathbf{u}^{s-1}\big), \quad k = 1, 2,\\
&\bar{A}_{0,11}^{NR,s} = \frac{\partial A_{1,12}}{\partial Q}\bigg|_{\mathbf{u}^{s-1}} \frac{\partial p}{\partial x}\bigg|_{\mathbf{u}^{s-1}}, \qquad \bar{A}_{0,12}^{NR,s} = \frac{\partial A_{2,12}}{\partial p}\bigg|_{\mathbf{u}^{s-1}} \frac{\partial p}{\partial t}\bigg|_{\mathbf{u}^{s-1}},\\
&\bar{A}_{0,21}^{NR,s} = \frac{\partial f_2}{\partial Q}\bigg|_{\mathbf{u}^{s-1}} + \frac{\partial A_{1,21}}{\partial Q}\bigg|_{\mathbf{u}^{s-1}} \frac{\partial Q}{\partial x}\bigg|_{\mathbf{u}^{s-1}} + \frac{\partial A_{1,22}}{\partial Q}\bigg|_{\mathbf{u}^{s-1}} \frac{\partial p}{\partial x}\bigg|_{\mathbf{u}^{s-1}} + \frac{\partial A_{2,22}}{\partial Q}\bigg|_{\mathbf{u}^{s-1}} \frac{\partial p}{\partial t}\bigg|_{\mathbf{u}^{s-1}},\\
&\bar{A}_{0,22}^{NR,s} = \frac{\partial f_2}{\partial p}\bigg|_{\mathbf{u}^{s-1}} + \frac{\partial A_{1,21}}{\partial p}\bigg|_{\mathbf{u}^{s-1}} \frac{\partial Q}{\partial x}\bigg|_{\mathbf{u}^{s-1}} + \frac{\partial A_{2,21}}{\partial p}\bigg|_{\mathbf{u}^{s-1}} \frac{\partial Q}{\partial t}\bigg|_{\mathbf{u}^{s-1}}\\
&\qquad + \frac{\partial A_{1,22}}{\partial p}\bigg|_{\mathbf{u}^{s-1}} \frac{\partial p}{\partial x}\bigg|_{\mathbf{u}^{s-1}} + \frac{\partial A_{2,22}}{\partial p}\bigg|_{\mathbf{u}^{s-1}} \frac{\partial p}{\partial t}\bigg|_{\mathbf{u}^{s-1}},
\end{aligned}
\tag{82}
$$

$$\bar{f}_1^{NR,s} = -A_{1,11}\left(\mathbf{u}^{s-1}\right)\frac{\partial Q}{\partial x}\bigg|_{\mathbf{u}^{s-1}} - A_{1,12}\left(\mathbf{u}^{s-1}\right)\frac{\partial p}{\partial x}\bigg|_{\mathbf{u}^{s-1}} - A_{2,12}\left(\mathbf{u}^{s-1}\right)\frac{\partial p}{\partial t}\bigg|_{\mathbf{u}^{s-1}},$$

$$\bar{f}_2^{NR,s} = f_2\left(\mathbf{u}^{s-1}\right) - A_{1,21}\left(\mathbf{u}^{s-1}\right)\frac{\partial Q}{\partial x}\bigg|_{\mathbf{u}^{s-1}} - A_{2,21}\left(\mathbf{u}^{s-1}\right)\frac{\partial Q}{\partial t}\bigg|_{\mathbf{u}^{s-1}} - A_{1,22}\left(\mathbf{u}^{s-1}\right)\frac{\partial p}{\partial x}\bigg|_{\mathbf{u}^{s-1}} - A_{2,22}\left(\mathbf{u}^{s-1}\right)\frac{\partial p}{\partial t}\bigg|_{\mathbf{u}^{s-1}}.$$

The needed partial derivatives of the coefficients are:

$$\begin{aligned}
\frac{\partial A_{1,12}}{\partial Q} &= \frac{1}{K},\\
\frac{\partial A_{1,21}}{\partial Q} &= \frac{1}{A^2},\\
\frac{\partial A_{1,21}}{\partial p} &= \frac{-2Q}{A^3}\frac{\partial A}{\partial p},\\
\frac{\partial A_{1,22}}{\partial Q} &= \frac{-2Q}{A^2 L},\\
\frac{\partial A_{1,22}}{\partial p} &= \frac{-1}{\rho^2}\frac{\partial \rho}{\partial p} + \frac{2Q^2}{A^3 L}\frac{\partial A}{\partial p} - \frac{Q^2}{A^2}\frac{\partial (1/L)}{\partial p},\\
\frac{\partial A_{2,12}}{\partial p} &= \left(\frac{1}{K} + \frac{1}{L}\right)\frac{\partial A}{\partial p} + A\frac{\partial (1/L)}{\partial p},\\
\frac{\partial A_{2,21}}{\partial p} &= \frac{-1}{A^2}\frac{\partial A}{\partial p},\\
\frac{\partial A_{2,22}}{\partial Q} &= \frac{-1}{AL},\\
\frac{\partial A_{2,22}}{\partial p} &= -\frac{Q}{A}\frac{\partial (1/L)}{\partial p} + \frac{Q}{A^2 L}\frac{\partial A}{\partial p},\\
\frac{\partial f_2}{\partial Q} &= \frac{-(7/4)\,fstar}{r^4}\left(\frac{Q}{r}\right)^{3/4},\\
\frac{\partial f_2}{\partial p} &= \frac{(19/4)\,fstar}{r^4}\left(\frac{Q}{r}\right)^{7/4}\frac{\partial r}{\partial p},
\end{aligned} \tag{83}$$

where

$$\begin{aligned}
\frac{\partial \rho}{\partial p} &= \frac{\rho}{K},\\
\frac{\partial A}{\partial p} &= 2\pi r\frac{\partial r}{\partial p},\\
\frac{\partial (1/L)}{\partial p} &= -2\frac{Ee(\partial r/\partial p) + r^2}{(Ee - pr)^2} - 2\frac{Ee(\partial r/\partial p) + r^2}{(Ee - pr)^2}.
\end{aligned} \tag{84}$$

The derivative of r with respect to p is rather complicated since it is defined iteratively as the solution of equation (79) after 2 Newton–Raphson iterations with (80) as an initial guess. Thus, denoting r^s as the approximation to r by the sth Newton–Raphson iteration and r^0 being the initial guess (80), one has:

$$r^s = r^{s-1} + C\big(r^{s-1}\big), \quad s = 1, 2, \ldots, \tag{85}$$

where

$$C(r) \equiv -\frac{F(r)}{F'(r)} = r\frac{Ee\ln(r/r_0) - pr}{Ee\ln(r/r_0) - Ee}. \tag{86}$$

Hence, we have

$$r \equiv r^2 = r^1 + C\big(r^1\big) = \big\{r^0 + C\big(r^0\big)\big\} + C\big(\big\{r^0 + C\big(r^0\big)\big\}\big).$$

The desired derivative is finally obtained as:

$$\frac{\partial r}{\partial p} = \frac{\partial r}{\partial r^0}\frac{\partial r^0}{\partial p}, \tag{87}$$

where

$$\begin{aligned}
\frac{\partial r}{\partial r^0} &= 1 + C'\big(r^0\big) + C'\big(r^0 + C\big(r^0\big)\big) + \big(1 + C'\big(r^0\big)\big)C'\big(r^0 + C\big(r^0\big)\big),\\
C'(r) &= \frac{\partial C}{\partial r} = -\frac{Ee\ln(r/r_0) + Ee + 2pr\ln(r/r_0) - 3pr - Ee(\ln(r/r_0))^2}{Ee(\ln(r/r_0) - 1)^2},\\
\frac{\partial r^0}{\partial p} &= -\frac{Eer_0^2}{(Ee - pr_0)^2}.
\end{aligned} \tag{88}$$

Obviously, programming the equations (82)–(88) is not a pleasant task. Furthermore, it is error-prone and inflexible. To the contrary, the code fragments (78) and (81) excel in simplicity, conciseness and flexibility. The integration of AD into our solver, has lead to a system which allows to express the mathematical equations of a PDE-based model in C++ code with a very strong resemblance to the original mathematical equations. This not only leads to a much better readability of the code, but also to dramatic reduction of programming and debugging work. This is considered a major asset of AD integration.

In order to investigate the computational efficiency of the `ElasticPipe_Transient_nlbc` solver a test program was written to evaluate the nonzero coefficients in equation (71) and their partial derivatives with FastDer++ as well as with two finite difference schemes (one-sided and two-sided).

Timing results are shown in figure 7 for different vector lengths $n = 2^m$ ($m = 1, 2, \ldots, 15$). The results for different vector lengths have been normalised by measuring the execution time for a vector length of 2^m with a multiplicity of 2^{16-m}, so that every test run effectively measures 2^{16} function and gradient evaluations. In addition, the timing results have been normalised relative to the execution time to evaluate the function value only (and not the gradient). Thus, the reported values are the work ratios $q\{w\}$ as defined in equation (22). Figure 7 shows that automatic differentiation again

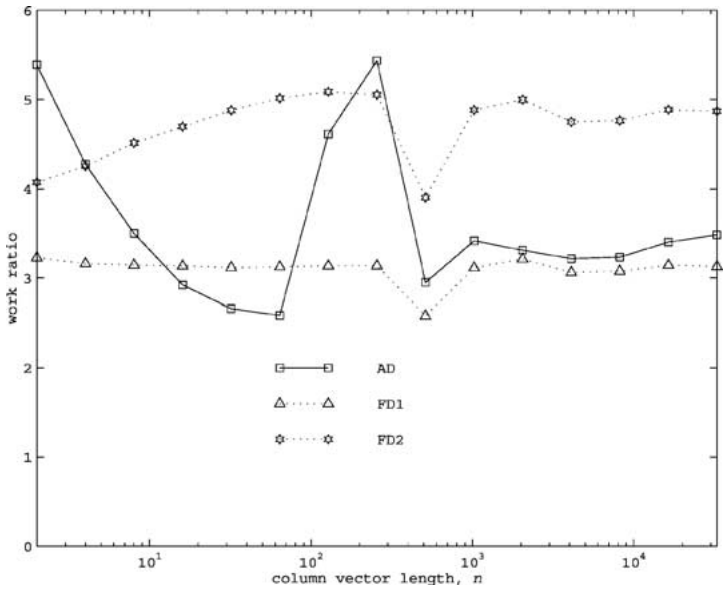

Figure 7. Timing results for automatic derivative computation (AD) of the nonzero coefficients in equation (71) vs. one-sided (FD1) and two-sided (FD2) finite difference derivative computation.

does the job very well. The performance of the AD-version is as comparable to that of the one-sided finite difference scheme

$$\frac{\partial w(\mathbf{u})}{\partial u_i} \approx \frac{w(\mathbf{u} + h\mathbf{e}_i) - w(\mathbf{u})}{h}, \tag{89}$$

and by 30% better than the two-sided finite difference scheme

$$\frac{\partial w(\mathbf{u})}{\partial u_i} \approx \frac{w(\mathbf{u} + h\mathbf{e}_i) - w(\mathbf{u} - h\mathbf{e}_i)}{2h}. \tag{90}$$

The peak around $n = 256$ is caused by caching problems and will probably disappear by improving on the locality of data in the FastDer++ data structures [34]. Of course, the AD version has the advantage of providing exact partial derivatives to within rounding error while the finite difference scheme faces truncation error. This can be a problem for functions with sharp corners. In principle, finite differencing schemes could be incorporated into the FastDer++ library, in order to provide the nearly the same flexibility and ease of use as the current AD versions. Still, the finite differencing requires careful selection of the h parameter. Tadjouddine et al. [32] have reported the use of AD tools to compute CFD flux Jacobian calculation of Roe's numerical flux. They conclude that operator overloading AD (not vectorised!) approaches are typically slower by a factor 5 to 10 than source translator approaches, and that a finite differencing scheme is still better by 10%. The results reported here, prove that vectorised is able to beat finite differencing schemes, especially taking into account that tests with a row-wise storage scheme in FastDer++ showed a further potential performance increase by approximately 30% [34].

6. Conclusion

In this paper we have demonstrated how by resorting to automatic differentiation, the user can be relieved from the extra work of linearising a nonlinear PDE and providing partial derivatives with respect to the unknown. An extension to common AD techniques, called vectorised AD has been developed that eliminates the overhead in both forward and reverse mode by taking profit of the fact that the dependent variables and their gradients have to be evaluated for a large number of values of the independent variables. Necessary modifications to the general algorithm for building the linear algebraic system are shown. A C++ implementation is discussed in detail in [34]. The direct consequence of the integration of AD in solvers for nonlinear PDE systems is that now the solver rather than the user takes care of the linearisation and the derivative computation. The user can directly program the nonlinear problem and is thereby relieved from the tedious, error-prone and time-consuming details of linearisation and derivative computation. The ease of use and flexibility of general FEM/FVM/FDM packages for solving nonlinear PDE systems can thus be put on the same footing for as for linear PDEs. This dramatically reduces the cost of solver development, and greatly facilitates the exploration of different models in a given setting. Code fragments and timing results of the discussed examples show clearly that

(1) integrating AD in nonlinear PDE solvers leads to highly flexible code with a close resemblance to the mathematical expression of the problem,

(2) coding and debugging efforts are greatly reduced, and

(3) the computational efficiency is not sacrificed.

In addition, comparison with results from literature clearly demonstrate that a performance improvement is to be expected over current methods for derivative evaluation in the framework of nonlinear PDE-solvers.

Acknowledgements

This research was initiated by author E.T. at IMEC in 1989 where it was sponsored by the National Impulse Programme on Information Technology (contract IT/SC/03). It is currently continued at the Faculty of Agricultural and Applied Biological Sciences, Katholieke Universiteit Leuven, Belgium, during a Postdoctoral Fellowship (OT/98/21) of author E.T.

References

[1] P. Aubert, N. Di Césaré and O. Pironneau, Automatic differentiation in C++ using espression templates and application to a flow control problem, SIAM J. Numer. Anal. 3 (2001) 197–208.

[2] C. Bendtsen and O. Stauning, Fadbad, a flexible C++ package for automatic differentiation using the forward and backward methods, Technical Report IMM-REP-1996-17, Technical University of Denmark, IMM, Department of Mathematical Modeling, Lyngby (1996).

[3] M. Berz, Differential algebraic description and analysis of trajectories in vacuum electronic devices including space-charge effects, IEEE Trans. Electron. Devices 35 (1988) 2002–2009.
[4] M. Berz, Arbitrary order description of arbitrary particle optical systems, Nuclear Instrum. Methods Phys. Res. A 298 (1990) 426–440.
[5] C. Bischof, L. Roh and A. Mauer-Oats, ADIC: An extensible automatic differentiation tool for ANSI-C, Technical Report, Argonne Preprint ANL/MCS-P626-1196, Mathematics and Computer Science division, Argonne National Laboratory, Argonne, IL (1997).
[6] C.H. Bischof, A. Bouaricha, P.M. Khademi and J.J. Moré, Computing gradients in large-scale optimization using automatic differentiation, INFORMS J. Comput. 9 (1997) 185–194.
[7] C.H. Bischof, A. Carle, P.M. Khademi and A. Mauer, The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs, IEEE Comput. Sci. Engrg. 3 (1996) MCS-P 481–1194
[8] D.S. Burnett, *Finite Element Applications, from Concepts to Applications* (Addison-Wesley, Reading, MA, 1987).
[9] Z. Cai, R.D. Lazarov, T.A. Manteuffel and S.F. McCormick, First-order least squares for partial differential equations: Part I, SIAM J. Numer. Anal. 31 (1994) 1785–1799.
[10] Z. Cai, T.A. Manteuffel and S.F. McCormick, First-order least squares for partial differential equations: Part II, SIAM J. Numer. Anal. 34 (1997) 425–454.
[11] H.M. Chaudry, *Applied Hydraulic Transients* (Van Nostrand Rheinold, New York, 1979).
[12] C. Faure and U. Naumann, AD2000: From simulation to optimization, in: *Third Internat. Conf. on Automatic Differentiation*, 19–23 June 2000, Nice, France, 2000.
[13] A. Griewank, On automatic differentiation, in: *Mathematical Programming: Recent Developments and Applications*, eds. M. Iri and K. Tanabe (Kluwer Academic Publishers, Dordrecht, 1989) pp. 83–108.
[14] A. Griewank, *Evaluatig Derivatives, Principles and Techniques of Algorithmic Differentiation*, Vol. 19 (SIAM, Philadelphia, PA, 2000).
[15] A. Griewank, D. Juedes and J. Utke, A package for the automatic differentiation of algorithms written in C/C++. User manual, Technical Report, Institute of Scientific Computing, Technical University of Dresden, Dresden, Germany (1996).
[16] A. Griewank and P.L. Toint, On the unconstrained optimization of partially separable functions, in: *Nonlinear Optimization*, ed. M.J.D. Powell (Academic Press, London, 1982).
[17] A. Griewank and P.L. Toint, Partitioned variable metric updates for large structured optimization problems, Numer. Math. 39 (1982) 119–137.
[18] K.E. Hillstrom, Users guide for JAKEF, Technical Report, Technical Memorandum ANL/MCS-TM-16, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (1985).
[19] J.E. Horwedel, B.A. Worley, E.M. Oblow and F.G. Pin, GRESS version 0.0, Users manual, Technical Report, ORNL/TM 10835, Oak Ridge National Laboratory, Oak Ridge, TN (1988).
[20] M. Iri and K. Kubota, Methods of fast automatic differentiation and applications, Technical Report, Research memorandum RMI 87-0, Department of Mathematical Engineering and Instrumantation Physics, Faculty of Engineering, University of Tokyo (1987).
[21] B.N. Jiang, *The Least-Squares Finite Element Method, Theory and Applications in Computational Fluid Dynamics and Electromagnetics* (Springer, Berlin, 1998).
[22] C.T. Kelley, *Iterative Methods for Linear and Nonlinear Equations* (SIAM, Philadelphia, PA, 1995).
[23] K.V. Kim, I.E. Nesterov, V.A. Skokov and B.V. Cherkasskii, An efficient algorithm for computing derivatives and extremal problems, English translation of: Effektivnyi algorithm vychisleniia proizvodnykh i ekstremal'nye zadachi, Ekonom. Matemat. Metody 20(2) (1984) 309–318.
[24] A. Koenig and B. Moo, *Ruminations on C++, a Decade of Programming Insight and Experience* (Addison-Wesley, Reading, MA, 1997).
[25] H.P. Langtangen, *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*, Lecture Notes in Computational Science and Engineering, Vol. 2 (Springer, Berlin, 1999).

[26] M. Majidi and G. Starke, Least-squares Galerkin methods for parabolic problems I: Semi-discretisation in time, SIAM J. Numer. Anal. (submitted).
[27] M. Majidi and G. Starke, Least-squares Galerkin methods for parabolic problems II: The fully discrete case and adaptive algorithms, SIAM J. Numer. Anal. (submitted).
[28] Numerical Objects AS, Diffpack World Wide Web home page, URL http://www.nobjects.com.
[29] W.H. Press and S.A. Teukolsky, Numerical calculation of derivatives, Comput. Phys. 1991 (1991) 68–69.
[30] A. Rhodin, IMAS – Integrated modeling and analysis system for the solution of optimal control problems, Comput. Phys. Comm. 107(1–3) (1997) 21–38.
[31] C.F. Schofield, *Optimising FORTRAN Programs* (Horwood, Chichester, 1989).
[32] M. Tadjouddine, S.A. Forth and J.D. Pryce, Ad tools and prospects for optimal ad in cfd flux Jacobian computations, in: *Automatic Differentiation of Algorithms – From Simulation to Optimization*, eds. G. Corliss, C. Faure, A. Griewank, L. Hascot and U. Naumann (Springer, Berlin, 2002).
[33] E. Tijskens, H. Ramon and J. De Baerdemaeker, Efficient operator overloading ad for solving non-linear PDEs, in: *Automatic Differentiation of Algorithms – From Simulation to Optimization*, eds. G. Corliss, C. Faure, A. Griewank, L. Hascot and U. Naumann (Springer, Berlin, 2002).
[34] E. Tijskens, D. Roose, H. Ramon and J. De Baerdemaeker, FastDer++, efficient automatic differentiation for nonlinear solvers, Math. Comput. Simulation (submitted).
[35] E. Tijskens, W. Schoenmaker and K. De Meyer, Automatic numerical evaluation of derivatives and its use in device simulators, in: *IEEE Workshop on Numerical Modelling of Processes and Devices for Integrated Circuits, NUPAD IV*, eds. M. Iri and K. Tanabe, 1992, pp. 251–254.
[36] S. Timoshenko and J.N. Goodier, *Theory of Elasticity* (McGraw-Hill, New York, 1970).
[37] T. Veldhuizen, Expression templates, C++ Report 7(4) (1996) 36–43.
[38] A. Verma, ADMAT: Automatic differentiation for MATLAB using object-oriented methods (2000).
[39] S.A. Wexler, Automatic evaluation of derivatives, Appl. Math. Comput. 24 (1987) 19–46.
[40] D.W. Yergeau, A dial-an-operator approach to simulation of impurity diffusion in semiconductors, Ph.D. thesis, Stanford University (1999).
[41] D.W. Yergeau, R.W. Dutton, A.H. Gencer and S. Dunham, A model implementation environment to support rapid prototyping of new TCAD models: A case study for dopant diffusion (1997).