Jyväskylän yliopiston julkaisuarkisto
Jyväskylä University Digital Archive
UNIVERSITY OF JYVÄSKYLÄ

This is an electronic reprint of the original article.
This reprint *may differ* from the original in pagination and typographic detail.

Author(s): Toivanen, Jukka; Mäkinen, Raino

Title: Implementation of sparse forward mode automatic differentiation with application to electromagnetic shape optimization

Year: 2011

Version:

Please cite the original version:
Toivanen, J., & Mäkinen, R. (2011). Implementation of sparse forward mode automatic differentiation with application to electromagnetic shape optimization. Optimization Methods and Software, 26(4-5), 601-616. https://doi.org/10.1080/10556781003642305

All material supplied via JYX is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.

RESEARCH ARTICLE

Implementation of Sparse Forward Mode Automatic Differentiation with Application to Electromagnetic Shape Optimization

JUKKA I. TOIVANEN* and RAINO A. E. MÄKINEN

Department of Mathematical Information Technology, P.O.Box 35, FI-40014 University of Jyväskylä, Finland

(*January 2010*)

In this paper we present the details of a simple lightweight implementation of so called sparse forward mode automatic differentiation (AD) in the C++ programming language. Our implementation and the well known ADOL-C tool (which utilizes taping and compression techniques) are used to compute Jacobian matrices of two nonlinear systems of equations from the MINPACK-2 test problem collection. Timings of the computations are presented and discussed. Moreover, we perform the shape sensitivity analysis of a time-harmonic Maxwell equation solver using our implementation and the tapeless mode of ADOL-C, which implements the dense forward mode AD. It is shown that the use of the sparse forward mode can save computation time even though the total number of independent variables in this example is quite small. Finally, numerical solution of an electromagnetic shape optimization problem is presented.

Keywords: automatic differentiation, shape sensitivity analysis, shape optimization

AMS Subject Classification: 65D25;65Y20;49Q10

1. Introduction

Derivatives are important in many fields of scientific computing, including solution of nonlinear equations, sensitivity analysis, and optimization. For example the solution of computationally expensive shape optimization problems [**?**] can often be made more robust and efficient by the use of good quality derivative information.

Automatic (or algorithmic) differentiation (AD) is a technique to numerically evaluate the derivatives of a function defined as a computer program exactly and with minimal user intervention. It exploits the fact that the computer program can be represented as a sequence of elementary arithmetic operations, and systematically applies the chain rule of differentiation to these operations. An introduction to both theoretical and implementational aspects of AD can be found in [**?**].

Automatic differentiation can be implemented using either operator overloading or source transformation. Operator overloading exploits the possibility provided by certain programming languages to redefine all arithmetic operators for user defined types. One then redefines (overloads) all required operations such that they implement not only the operation itself, but also the computation of necessary derivative information.

*Corresponding author. Email: jukka.i.toivanen@jyu.fi

Original code	AD version
`double v1 = 1.0;`	`addouble v1 = 1.0;`
`double v2 = 2.0;`	`addouble v2 = 2.0;`
`double f;`	`addouble f;`
	`v1.declareIndependent();`
	`v2.declareIndependent();`
`f = sin(v1*v1 + v2*v2);`	`f = sin(v1*v1 + v2*v2);`

Table 1. Original and differentiated versions of a simple calculation.

An example of this approach in C++ programming language is shown in Table 1, where the original and the AD version of a simple computation are shown. One must identify so called *independent variables*, i.e. variables with respect to which one wishes to differentiate the code. Variables that depend on the independent variables directly or indirectly are called *active variables*. They must be represented with a special AD type, in this example called `addouble`. The part of the code performing the actual computation remains exactly the same from the user point of view, since the compiler takes care of calling the appropriate functions implementing the derivative computation. This is an important feature of this approach when large simulation codes are considered.

The operator overloading approach typically has some computational overhead, since regular floating point variables have to be replaced by much more complicated objects throughout the code. Some implementational tricks that can reduce the overhead include the use of expression templates and traits [**?**] and the use of vectorization [**?**].

Source transformation, on the other hand, requires running the original program code through a preprosessor, which augments the code with the derivative computation routines. The augmented code is then compiled in a standard fashion. This approach may produce somewhat more efficient code, and it can also be applied in context with programming languages such as FORTRAN 77 that do not support operator overloading [**?**]. However, such tool is much more complicated to implement, which means that a typical end user in general has to rely on existing tools.

1.1. *Forward and reverse modes*

Let us consider the differentiation of a computer program implementing a (vector) function $\boldsymbol{\beta} = F(\boldsymbol{\alpha})$, where $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_n)$ and $\boldsymbol{\beta} = (\beta_1, \dots, \beta_m)$. The execution of the program can be considered as a sequence of assignments $v_i = \phi_i(\text{all relevant } v_j)$, where ϕ_i is some elementary operation having typically one or two input variables and $\boldsymbol{v}$ is a vector containing all variables present in the computation:

$$\boldsymbol{v} = (\underbrace{v_1, \dots v_n}_{\boldsymbol{\alpha}}, v_{n+1}, \dots, v_{n+p}, \underbrace{v_{n+p+1}, \dots, v_{n+p+m}}_{\boldsymbol{\beta}}).$$

The variables $v_{n+1}, \dots, v_{n+p}$ are the intermediate variables created during the computation. In the left of Table 2 the partition of the program shown in Table 1 into elementary operations is presented.

Alternatively, the execution process can be seen as the solution of the system of nonlinear equations

$$0 = \boldsymbol{E}(\boldsymbol{v}(\boldsymbol{\alpha}), \boldsymbol{\alpha}) \equiv (\phi_i(v_1, \dots v_{i-1}, \boldsymbol{\alpha}) - v_i)_{i=1,\dots,n+p+m} \tag{1}$$

where ϕ_i is the ith elementary operation of the user's code. For notational purposes we assume that the first n elementary operations are the initializations $\phi_i = \alpha_i$, and the remaining $p + m$ operations then depend on $\boldsymbol{\alpha}$ only indirectly. Moreover, we assume that the last m operations represent the assignment of the results of $F(\boldsymbol{\alpha})$ to the dependent variables $\boldsymbol{\beta}$.

We also assume that each variable is affected by exactly one assignment, i.e. no variables are ever overwritten, and that the right hand side of the assignment $v_i = \phi_i$ does not depend on v_i, i.e. there are no iterative assignments. However, in the actual computer code such cases can still be allowed as long as proper care is taken in the implementation of automatic differentiation. See [**?**] for details.

Performing implicit differentiation to (1) we obtain that

$$\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{\alpha}} = -\left(\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{v}}\right)^{-1} \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{\alpha}} = -\begin{pmatrix} -\boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{B} & \boldsymbol{L} - \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{R} & \boldsymbol{T} & -\boldsymbol{I} \end{pmatrix}^{-1} \begin{pmatrix} \boldsymbol{I} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{pmatrix}. \tag{2}$$

The so called extended Jacobian can be written as

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{v}} = \boldsymbol{C} - \boldsymbol{I}, \text{ where } C_{ij} = \frac{\partial \phi_i(v_1, \ldots, v_{i-1})}{\partial v_j}. \tag{3}$$

Since variables are always evaluated before they are used as arguments, the operation ϕ_i only takes arguments having an index lower than i, and $\boldsymbol{C}$ is thus strictly lower triangular. Since the elementary operations ϕ_i usually take at most two arguments, the matrix $\boldsymbol{C} - \boldsymbol{I}$ is also extremely sparse. For example the extended Jacobian related to the code in Table 1 is shown in the right of Table 2.

v1	=	1	-1	0	0	0	0	0
v2	=	2	0	-1	0	0	0	0
v3	=	v1*v1	2v1	0	-1	0	0	0
v4	=	v2*v2	0	2v2	0	-1	0	0
v5	=	v3+v4	0	0	1	1	-1	0
v6	=	sin(v5)	0	0	0	0	cos(v5)	-1

Table 2. The elementary operations (left) and the resulting extended Jacobian (right) for the code of Table 1.

It follows that the derivatives of the final variables of interest can be obtained as

$$\frac{\partial \boldsymbol{\beta}}{\partial \boldsymbol{\alpha}} = \boldsymbol{R} + \boldsymbol{T}(\boldsymbol{I} - \boldsymbol{L})^{-1}\boldsymbol{B}. \tag{4}$$

In the so called forward mode automatic differentiation one solves the lower triangular system

$$(\boldsymbol{I} - \boldsymbol{L})\boldsymbol{Z} = \boldsymbol{B} \tag{5}$$

by forward substitutions, and computes

$$\frac{\partial \boldsymbol{\beta}}{\partial \boldsymbol{\alpha}} = \boldsymbol{R} + \boldsymbol{T}\boldsymbol{Z}. \tag{6}$$

Notice that in forward mode AD there is no need to explicitly assemble (2) as a linear system. Instead the derivative information can be computed "on the fly", i.e. the derivatives of each intermediate variable can be computed when the generating operation is executed. When some intermediate variable is no longer needed the derivatives of that variable can also be deleted from memory.

Alternatively, one may use the so called reverse mode, and solve the upper triangular system

$$(\boldsymbol{I}-\boldsymbol{L})^T \widetilde{\boldsymbol{Z}} = \boldsymbol{T}^T \tag{7}$$

and compute

$$\frac{\partial \boldsymbol{\beta}}{\partial \boldsymbol{\alpha}} = \boldsymbol{R} + \widetilde{\boldsymbol{Z}}^T \boldsymbol{B}. \tag{8}$$

The reverse mode has theoretical advantages over the forward mode if $m < n$, since (7) involves fewer right hand sides than (5). However, to solve (7) by backward substitution, one must go through the computation in reverse order. One way to implement this is to record the necessary information about the operations on a so called "tape" during the execution of the program, and then traverse the tape in reverse order.

In principle one can also consider other ways to solve the system (2). Such methods are not very common in practice since the size of the system is often extremely large, which presents obvious problems regarding the storage of the system. Such approaches may still be feasible when applied to relatively small systems arising from the differentiation of some subroutine or code block inside a larger code. For example LU factorization approach to the solution of the system (2) is discussed in [**?**].

1.2. *Exploitation of sparsity*

In (5) there are n and in (7) there are m right hand sides, both of which can be extremely large numbers. This is the case especially when automatic differentiation is used to compute the Jacobian of a nonlinear system of equations. Thus, solving for all the right hand sides simultaneously presents some difficulties. However, in many cases the Jacobian matrices are sparse, which should be somehow exploited.

One way to do this is the computation of Jacobians using compression [**?** , Chap. 8]. The idea is to first find out the sparsity pattern of the Jacobian. Using e.g. graph coloring methods [**?**] structurally orthogonal columns of the Jacobian (columns that do not have non-zero entries on the same row) can then be grouped together and represented by only one independent variable. In this way, the number of independent variables, and thus the number of right hand sides in (5), can sometimes be greatly reduced.

Another possibility is to exploit sparsity of the derivatives using sparse forward propagation [**?** , Chap. 7]. The idea is to solve (5) for all right hand sides simultaneously, but to take advantage on the potential sparsity of the vectors $\partial \boldsymbol{v}/\partial \boldsymbol{\alpha}$. The solution of (5) through forward substitutions can be written as

$$\frac{\partial v_i}{\partial \boldsymbol{\alpha}} = \sum_j \frac{\partial \phi_i}{\partial v_j} \frac{\partial v_j}{\partial \boldsymbol{\alpha}}. \tag{9}$$

Here the summation goes through only those v_j's on which the operation ϕ_i actually depends (usually one or two variables).

In the sparse forward propagation approach the sum (9) is implemented as a linear combination of sparse vectors. This approach is often called dynamic exploitation of sparsity, since there is no need to have any a priori sparsity information or to perform any separate sparsity pattern detection phase. Instead the sparsity pattern of each intermediate variable is determined at the run time.

An implementation of such approach using the SparsLinC library and the ADIFOR tool is presented in [?]. A MATLAB implementation exploiting the sparse matrix class for the storage of sparse derivative vectors is presented in [?].

The sparse forward mode is sometimes criticized for having too much computational overhead for practical purposes [?]. On the other hand, the approach offers an elegant and user friendly way to exploit sparsity of the derivatives. We claim that with a suitable implementation this approach has only a modest computational overhead, and it is indeed suitable for practical purposes in many applications. The rest of this paper deals with the verification of our claim.

2. Simple implementation of sparse forward mode

In this section we present our implementation of sparse forward mode automatic differentiation. The implementation is simple and does not require any external libraries for the handling of the sparse derivative vectors.

2.1. *Index domain propagation*

Consider an elementary binary operation $w(\boldsymbol{\alpha}) = \phi(u(\boldsymbol{\alpha}), v(\boldsymbol{\alpha}))$, where $\boldsymbol{\alpha}$ are the independent variables. The chain rule of differentiation applied to the operation gives

$$\frac{\partial w}{\partial \alpha_j} = \frac{\partial \phi}{\partial u}\frac{\partial u}{\partial \alpha_j} + \frac{\partial \phi}{\partial v}\frac{\partial v}{\partial \alpha_j}. \tag{10}$$

The partial derivatives $\partial\phi/\partial u$ and $\partial\phi/\partial v$ of elementary operations are known. Therefore the partial derivatives $\partial w/\partial\alpha_j$ can be computed as long as the partial derivatives of the arguments are known, which is always the case in forward mode automatic differentiation.

The essence of the sparse forward mode is that the computation of $\nabla w := (\partial w/\partial\boldsymbol{\alpha})^T$ does not (necessarily) require going through all the independent variables. Simplifying the analysis of Griewank and Walther [?], let us define the index domain χ of a variable x as

$$\chi(x) = \{i \mid \partial x/\partial\alpha_i \neq 0\}. \tag{11}$$

From (10) we immediately notice that $\chi(w) \subset \chi(u) \cup \chi(v)$. The partial derivatives rarely cancel out, and in the practical implementation we assume that $\chi(w) = \chi(u) \cup \chi(v)$. The sets $\chi(u)$ and $\chi(v)$ are in general not the same, which means that in order to evaluate the gradient ∇w we must form the union $\chi(u) \cup \chi(v)$ and compute the corresponding partial derivatives.

Our implementation of sparse forward mode automatic differentiation is in C++, and is based on the operator overloading technique. We have defined a class `addouble`, which holds the value of one real variable and its partial derivatives with respect to the independent variables. The `addouble` objects carry along the index domain information, which enables the computation and storage of only non-zero partial derivatives.

We save the partial derivatives of a variable x in two vectors, $x.ind$ and $x.der$. The vector $x.ind$ holds the indices $\chi(w)$, while $x.der$ holds the derivatives $x.der(i) = \partial x/\partial\alpha_{x.ind(i)}$. The technique is illustrated in Table 3, where the *ind* and *der* vectors of the intermediate variables related to the code example in Table 1 are shown.

With a proper implementation, the computational complexity of the evaluation

Variable	Value	Ind	Der
v1	1.0	{1}	{1.0}
v2	2.0	{2}	{1.0}
v3	1.0	{1}	{2.0}
v4	4.0	{2}	{4.0}
v5	5.0	{1, 2}	{2.0, 4.0}
v6	-0.959	{1, 2}	{0.567, 1.135}

Table 3. Values of the ind and der vectors.

of the gradient ∇w (with $w = \phi(u, v)$ and ϕ being an elementary operation) is of the order $\mathcal{O}(|\chi(w)|)$. Here $|\cdot|$ denotes the size of a set. In our implementation we keep the elements of the vector *ind* in increasing order. This allows us to form the union of two index domains in a time proportional to the size of the union set, without the need to use temporary arrays of size n, such as the expanded real accumulator used in [**?**].

An example implementation of the evaluation of the gradient $\nabla w = \phi(u, v)$ in pseudo-code is shown as Algorithm *1*. In the code we use the notation: x.ind = set $\chi(x)$ in increasing order, x.der(i) = $\partial x / \partial \alpha_j$ with j = x.ind(i), x.N = size of the set $\chi(x)$, and the vectors are assumed to be indexed starting from 1. The counters up, vp and wp refer to the current index in u.ind, v.ind and w.ind respectively.

Algorithm 1 Gradient evaluation

```
up ← 1
vp ← 1
wp ← 0
while ((up ≤ u.N) or (vp ≤ v.N))
        if (vp > v.N) then
                ind ← u.ind(up)
                ud ← u.der(up)
                vd ← 0
                up ← up + 1
        else if (up > u.N) then
                ind ← v.ind(vp)
                vd ← v.der(vp)
                ud ← 0
                vp ← vp + 1
        else if (u.ind(up) < v.ind(vp)) then
                ind ← u.ind(up)
                ud ← u.der(up)
                vd ← 0
                up ← up + 1
        else if (v.ind(vp) < u.ind(up)) then
                ind ← v.ind(vp)
                vd ← v.der(vp)
                ud ← 0
                vp ← vp + 1
        else
                ind ← u.ind(up)
                ud ← u.der(up)
                vd ← v.der(vp)
                up ← up + 1
                vp ← vp + 1
        end if
```

wp ← wp + 1
w.ind(wp) ← ind
w.der(wp) ← $\frac{\partial\phi}{\partial u}$ * ud + $\frac{\partial\phi}{\partial v}$*vd
end while
w.N ← wp

The algorithm produces the vector w.ind, which is automatically in increasing order, and computes the corresponding partial derivatives. This is a simple reference implementation and probably has room for optimization, but there indeed are only $\mathcal{O}(1)$ operations for each non-zero partial derivative of w.

The implementation of unitary operations such as `sin` or `exp` is easier than binary operations, as no union of index domains has to be formed. Considering an elementary unitary operation $w = \phi(u)$ we can simply assume that $\chi(w) = \chi(u)$, and the partial derivatives are obtained as

$$\frac{\partial w}{\partial \alpha_j} = \frac{\partial \phi}{\partial u}\frac{\partial u}{\partial \alpha_j} \tag{12}$$

for all $j \in \chi(u)$.

2.2. *Memory allocation*

Next we present three possible approaches for the allocation of memory for the `addouble` objects.

(1) In the simplest (static) implementation each `addouble` object has a fixed space for the vectors *ind* and *der*. This space has to be larger than the absolute maximum size of the index set of any active variable, and this maximum has to be known at compile time. Exceptions are variables that are known to have dense derivative vectors. For an example of such variables see the note on the handling of partially separable functions in Section 2.3. Fortunately there are usually only few of these kind of variables, and it is often known in advance which variables belong to this class. Therefore these variables can be represented using objects of a different type.

(2) In a completely dynamic approach the space needed for the derivatives of the variable w could be allocated dynamically when assignment to the variable w is made. One problem with this approach is that the size of the index set $\chi(w)$ of a result variable of a computation $w = \phi(u, v)$ is not known in advance, but only after the Algorithm *1* is executed. One could first compute *ind* and *der* vectors of the results variable in large temporary arrays and then allocate only the amount of space actually needed for the derivatives of the result variable. This approach would completely avoid the allocation of extra space, but has the drawback of having to perform an extra phase of copying the information to the result variable.

(3) Alternatively, one can use the dynamic approach and overestimate the size of the index set $|\chi(w)|$ by $|\chi(u)| + |\chi(v)|$. One can then allocate the space before executing Algorithm *1* and store the derivative information directly into the result variable.

In our tests we have found out that dynamic memory allocation often causes significant overhead, and therefore we decided to use the static approach. Before the compilation of the code, the user must specify an overestimate to the maximum size of the index domain of any active variable. All active variables then have that much

room for the derivatives. Fortunately, the maximum number of active variables alive at each moment during the computation is usually quite modest, since most of the intermediate variables are used only as arguments to following elementary operations, and are destroyed quite soon after they are created. Therefore allocating the derivative arrays of the active variables based on a relatively rough overestimate is usually feasible.

2.3. *Partially separable functions*

A class of functions that deserves special attention are partially separable functions [?]. Let a function f be represented as a sum

$$f = \sum_{i=1}^{m} f_i \tag{13}$$

where each gradient ∇f_i is sparse. Then f is called partially separable. Quite often, however, the gradient ∇f of a partially separable variable is dense, i.e. has $\mathcal{O}(n)$ non-zero components.

A typical example of such a variable in the finite element setting is an integral type functional depending on the discrete solution:

$$f = \int_\Omega g(u_h) \quad \text{where } u_h = \sum_i^n q_i \varphi_i, \tag{14}$$

$\boldsymbol{q}$ are the degrees of freedom and $\boldsymbol{\varphi}$ are the basis functions. Such functional depends on all the degrees of freedom whose associated basis functions have support belonging to the domain of integration Ω. However, in the finite element setting integrals are computed elementwise:

$$f = \int_\Omega g(u_h)\mathrm{d}x = \sum_i \underbrace{\int_{K_i} g(u_h)\mathrm{d}x}_{f_i}, \tag{15}$$

where the summation goes through the elements K_i constituting Ω. In the element K_i the function u_h depends only on a few degrees of freedom, and therefore the gradient of f_i with respect to $\boldsymbol{q}$ is sparse.

The derivatives of each f_i in (13) can be efficiently computed using the sparse forward mode AD. On the other hand, it is more efficient to represent the gradient of the left hand side f as a dense vector. Namely, adding an entry to a dense vector can be implemented as an $\mathcal{O}(1)$ operation, whereas adding a new entry to our sparse vector may require moving all existing entries. To this end we have implemented a separate class to represent variables having a dense derivative vector, and overloaded the += operator to enable efficient summation of sparse gradients.

2.4. *Code modification*

We conclude this section with a summary of the steps that are required to modify the original code into a one that is able to differentiate the result variables with respect to given independent variables.

The independent variables must be represented with the `addouble` type, and they must be declared to be independent before any actual computations take

place. Throughout the code, each variable that depends on the independent variables must be defined to be of the `addouble` type instead of the regular `double` type. For the simplicity of implementation, other `double`s may also be represented by `addouble`s, but unnecessary replacements will result in some decrease in computational performance.

Compiler errors can be utilized to detect which variables need to be represented by `addouble`s, since the compiler will give an error if an AD type variable is assigned to a regular `double` variable. This is why one should not define automatic conversions from `addouble` type to a regular `double`, because such definitions make it easy to accidentally lose some derivative information. Instead, a special routine returning the value of the variable should be used in cases when a AD variable must be converted into a regular `double`, for example to interact with external IO routines.

Since `addouble` is a user defined type, all operations involving this type must be explicitly defined (overloaded) before they can be used. Fortunately, the number of different operations present in a typical code is quite modest. The header file containing the definitions of the overloaded operations must be included into the compilation of the program.

3. Differentiation test cases

In this section we compare the performance of our sparse forward mode implementation to the ADOL-C tool [**?**]. Computations were performed on a HP ProLiant DL585 server with 4 AMD Opteron 885 2.6 GHz dual core processors and 64 GB memory. No parallelization was exploited in the codes. GCC compiler version 4.0.2 with the optimization flag O4 was used to compile all codes.

In all the examples fixed size arrays (see Subsection 2.2) were used for the storage of the derivative information. A rather coarse overestimate 100 was used as the amount of space that was allocated for the derivatives.

3.1. *MINPACK-2 test problems*

First we consider problems defined in the MINPACK-2 test problem collection [**?**]. Codes in the collection are originally written in FORTRAN, but for this purpose a few of them were rewritten in C by the authors. The test problems considered here represent systems of nonlinear equations. Each example code takes as input the solution candidate vector $\boldsymbol{x}$, and computes the corresponding residual vector. In this test the Jacobian matrices related to these problems are computed using our implementation of the sparse forward propagation and the ADOL-C tool. Components of the input vector $\boldsymbol{x}$ are defined to be the independent variables, and components of the residual vector are the dependent variables.

Let $T(F)$ be the wall clock time consumed to the computation of the residual vector with the original code that uses regular `double` variables. Furthermore, let $T(J)$ be the wall clock time consumed to the computation of the Jacobian using the AD version of the code. The so called runtime ratio of the Jacobian computation is then defined to be $T(J)/T(F)$. Thus the runtime ratio is affected by the extra work that is needed to compute the residual vector and its derivatives (i.e. the Jacobian) instead of just the residual vector, and the computational overhead associated to the specific AD implementation. Wall clock time is used instead of CPU seconds because it takes into account also the time consumed to IO operations. To get more accurate timings, the computations were executed 20 times, and the average computation time was used to calculate the runtime ratio.

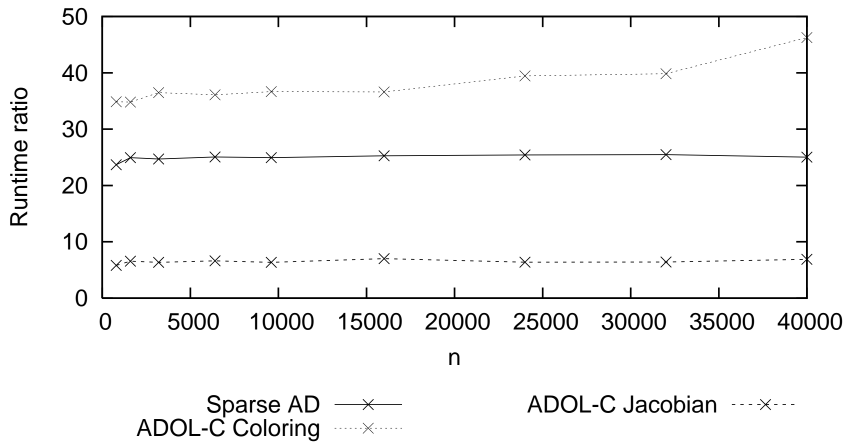

Figure 1. Runtime ratios for the FIC test problem.

We utilize the `sparse_jac` driver of ADOL-C, which computes the Jacobian by compression. Default options were used for the sparsity pattern computation. We refer to [**?**] for details on the ADOL-C tool. The compression technique involves the detection of the sparsity pattern and a coloring phase, after which the evaluation of the Jacobian can be performed. For subsequent Jacobian computations the sparsity pattern detection and coloring phases are not required anymore, if the sparsity pattern of the Jacobian remains the same. Therefore the ADOL-C timings are presented as two different values: the time consumed for the sparsity pattern detection and coloring (denoted by ADOL-C Coloring in Figures 1 and 2), and the time consumed on the evaluation of the Jacobian (ADOL-C Jacobian in the Figures). The runtime ratio of the sparsity pattern detection and coloring phase is defined similarly than in the case of the Jacobian computation.

Runtime ratios of MINPACK-2 problems called Flow in a Channel (FIC) and Swirling Flow between Disks (SFD) are shown in Figures 1 and 2, respectively. As explained in [**?**], from the sparsity structure of the Jacobians related to these problems it follows that the runtime ratios should be approximately constant with respect to n when sparse derivative propagation or compression techniques are used. Based on the Figures we can conclude that this really is the case with the implementations considered here. In these examples the number of non-zeros per row of the Jacobian does not grow with the number of independent variables. For the FIC test case the number of non-zeros per row of the Jacobian was between 6 and 9, except for two rows representing boundary conditions and having only 1 non-zero. The number of colors (columns in the compressed Jacobian) was 9. For the SFD test case the number of non-zeros per row was between 6 and 14, except for three boundary condition rows, and the number of colors was 14.

Results of similar differentiation tests using the FORTRAN versions of the same test problems are reported in [**?**]. In that article the ADIFOR tool [**?**] is used as a version implementing the sparse forward propagation with the aid of the SparsLinC library. Tests were run on two different platforms, and the runtime ratios 32.3 and 161.0 for the FIC problem and 25.2 and 156.0 for the SFD problem are reported.

From the Figures 1 and 2 we can see that the sparsity pattern detection and coloring phase is a rather computationally expensive operation. However, it has to performed only once in the case of consecutive Jacobian evaluations with a fixed sparsity pattern. From the runtime rations we can calculate that performing three or more Jacobian evaluations in the case of the FIC problem is already cheaper using ADOL-C and the compression technique than using our implementation of the sparse forward mode. In the case of the SFD problem four or more Jacobian

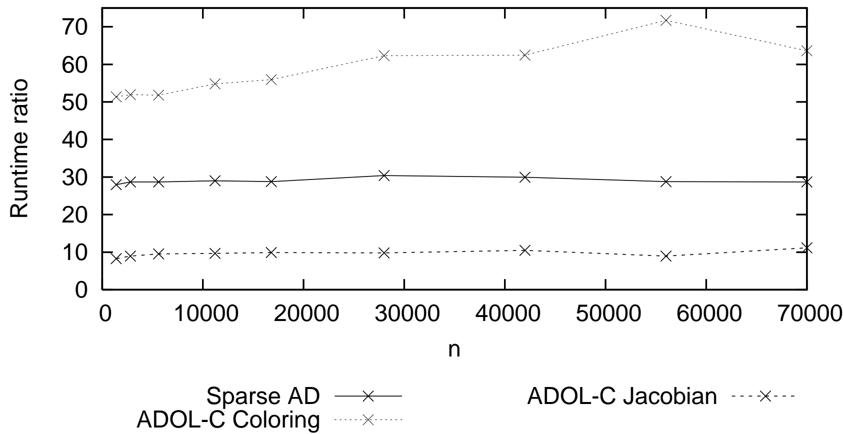

Figure 2. Runtime ratios for the SFD test problem.

evaluations becomes cheaper using ADOL-C.

These test programs involve only very simple computations, and do not therefore take very long to execute. The average function evaluation times varied from 0.00032 to 0.015 seconds for the FIC test problem, and from 0.00047 to 0.022 seconds for the SFD problem. In these cases the tape created by the ADOL-C tool involves only relatively small amount of operations, and can easily fit into the memory. Next we will consider the differentiation of a much more complicated code, in which case the tape becomes considerably larger.

3.2. *Electromagnetic simulator*

In this section we discuss the differentiation of a complete electromagnetic simulator with respect to geometrical changes. The simulator is based on the so called method of moments (MoM), which is a boundary element type method. For more details on the techniques employed in the solver we refer to [**?**], and discussion on the sensitivity analysis of the solver and some optimization examples can be found in [**?**].

We consider a metallic perfectly conducting antenna in a homogeneous medium. The antenna simulation problem is formulated as the electric field integral equation (EFIE) for the unknown electric current density $\boldsymbol{J}$ as

$$\left(\frac{-1}{\mathrm{i}\omega\varepsilon_0}\nabla\boldsymbol{\mathcal{S}}(\nabla_s\cdot\boldsymbol{J})(\boldsymbol{r})+\mathrm{i}\omega\mu_0\boldsymbol{\mathcal{S}}(\boldsymbol{J})(\boldsymbol{r})\right)_{\tan}=-\boldsymbol{E}^p_{\tan}(\boldsymbol{r}),\qquad \boldsymbol{r}\in S. \tag{16}$$

Here tan denotes tangential field component on the surface of the antenna S, $\nabla_s\cdot$ denotes the surface divergence, $\boldsymbol{E}^p$ is the primary electric field due to an excitation, and $\boldsymbol{\mathcal{S}}$ is the single-layer integral operator given by

$$\boldsymbol{\mathcal{S}}(\boldsymbol{F})(\boldsymbol{r})=\int_S G_0(\boldsymbol{r},\boldsymbol{r}')\,\boldsymbol{F}(\boldsymbol{r}')\,dS', \tag{17}$$

where G_0 is the free space Green's function.

Using the method of moments, the EFIE (16) can be reduced to a matrix equation

$$\boldsymbol{A}\,\boldsymbol{u}=\boldsymbol{b}, \tag{18}$$

where $\boldsymbol{A}$ is a full $N\times N$ complex valued system matrix, $\boldsymbol{u}$ is a $N\times 1$ complex vector

containing the coefficients of the basis function representation of the unknown current $\boldsymbol{J}$, and $\boldsymbol{b}$ is a $N \times 1$ excitation vector. In the numerical implementation of the solver the surface is divided into planar triangular elements and the basis functions are the Rao-Wilton-Glisson (RWG) functions [?], denoted by $\varphi_1, \ldots, \varphi_N$. The elements of the matrix $\boldsymbol{A}$ are given by

$$A_{mn} = \frac{1}{\mathrm{i}\omega\varepsilon_0} \int_S \int_S \nabla_s \cdot \varphi_m(\boldsymbol{r})\, G_0(\boldsymbol{r}, \boldsymbol{r}')\, \nabla_s \cdot \varphi_n(\boldsymbol{r}')\, dS'\, dS + \mathrm{i}\omega\mu_0 \int_{\mathrm{S}} \int_{\mathrm{S}} \varphi_{\mathrm{m}}(\boldsymbol{r}) \cdot \varphi_{\mathrm{n}}(\boldsymbol{r}')\, \mathrm{G}_0(\boldsymbol{r}, \boldsymbol{r}')\, \mathrm{dS}'\, \mathrm{dS}. \quad (19)$$

Using a voltage gap feed (delta generator), the elements of the excitation vector become

$$b_m = -V\, l_m, \quad (20)$$

where V is the voltage at the port and l_m is the length of the edge at the excitation port. Here m runs through all edges at the port.

After solving the equation (18) one can calculate the response of the system from the vector $\boldsymbol{u}$. One might be interested for example in the optimization of the antenna for a given input impedance [?] or maximization of the antenna gain [?]. Our aim is to find the sensitivity of the objective function with respect to variations in the antenna geometry parametrized by means of some set of design variables $\boldsymbol{\alpha}$.

Since the objective functions usually depend on the design $\boldsymbol{\alpha}$ implicitly through the solution vector $\boldsymbol{u}$, straightforward application of the automatic differentiation to the whole evaluation procedure would require differentiation of the linear system solver. This can be avoided by utilizing the well known adjoint sensitivity analysis technique as follows.

The derivative of a real valued objective function $\mathcal{J}$ with respect to design variable α_k is given by

$$\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\alpha_{\mathrm{k}}} = \frac{\partial \mathcal{J}}{\partial \alpha_k} + \Re\left[(\boldsymbol{p}^T \left(\frac{\partial \boldsymbol{b}}{\partial \alpha_k} - \frac{\partial \boldsymbol{A}}{\partial \alpha_k}\boldsymbol{u}\right)\right] \quad (21)$$

where the derivative $\partial\mathcal{J}/\partial\alpha_k$ reflects the explicit dependence of $\mathcal{J}$ on the design variable, and $\boldsymbol{p}$ is solved from the adjoint equation

$$\boldsymbol{A}^T \boldsymbol{p} = \overline{(\nabla_{\boldsymbol{u}}\mathcal{J})}, \quad \text{with} \quad \nabla_{\boldsymbol{u}}\mathcal{J} := \nabla_{\Re\boldsymbol{u}}\mathcal{J} + \mathrm{i}\nabla_{\Im\boldsymbol{u}}\mathcal{J}. \quad (22)$$

Here $\Re$ and $\Im$ denote the real and imaginary parts of a complex variable respectively, i is the imaginary unit, bar over a vector denotes complex conjugation and T is the transpose without a complex conjugation. In this approach we only need to differentiate the part of the code that computes the system matrix $\boldsymbol{A}$ and the right hand side vector $\boldsymbol{b}$ with respect to the design variables.

As a differentiation test case we consider the so called Yagi-Uda antenna system [?], a sketch of which is shown in Figure 3. Parts of the antenna are, from left to right, the reflector element, the driven element and a variable number of director elements. Each of the elements is modeled as a three dimensional cylinder, and discretized using a boundary mesh like the one shown in Figure 4.

First we consider differentiation with respect to the lengths of the antenna elements $l_1, l_2, \ldots, l_n$. The nodal coordinates of the mesh are related to the design

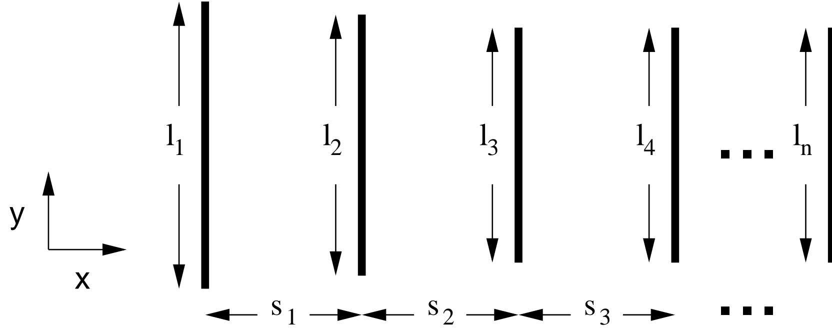

Figure 3. Sketch of the Yagi-Uda antenna system.

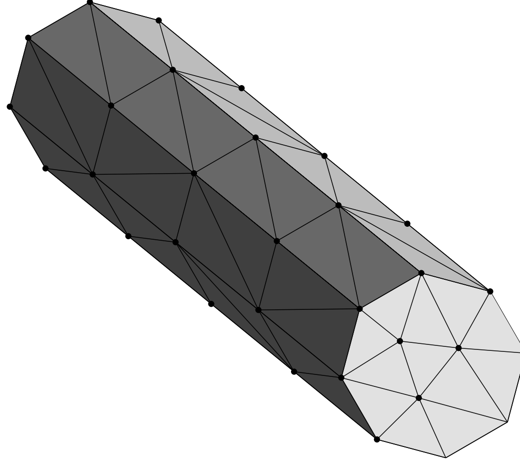

Figure 4. Boundary mesh of one of the antenna elements.

variable vector $\boldsymbol{\alpha}$ by the relation

$$y_j = y_j^{ref} * \alpha_k \tag{23}$$

where y_j and y_j^{ref} are the y coordinates of the jth mesh node in the deformed mesh and in the reference mesh respectively, and k is the index of the antenna element that contains the mesh node j. In other words, each design variable affects the y coordinates of all nodes belonging to one particular antenna element.

In terms of the previous notations, the dependent variables are now

$$\boldsymbol{\beta} = \{A_{11}, A_{12}, \ldots, A_{NN}, b_1, \ldots, b_N\}^T, \tag{24}$$

F is the assembly procedure, and the dependence of $\boldsymbol{\beta}$ on $\boldsymbol{\alpha}$ is through the mesh nodal coordinates.

Certain sparsity properties follow from the expression of the system matrix elements (19). Namely, the system matrix entry A_{mn} depends only on the geometry of the elements belonging to the support of φ_m or φ_n. Since the support of each basis function belongs to exactly one antenna element, it follows that each system matrix entry depends on either one or two design variables.

If the whole assembly process to calculate $\boldsymbol{A}$ and $\boldsymbol{b}$ is differentiated at once, the disadvantage of the tape based approach becomes clear. Namely, taping the whole process is extremely memory consuming. We performed a test with two antenna elements (reflector and driven element). In this case there were only 264 basis functions, but still the taping included a massive number of 188,736,243 operations,

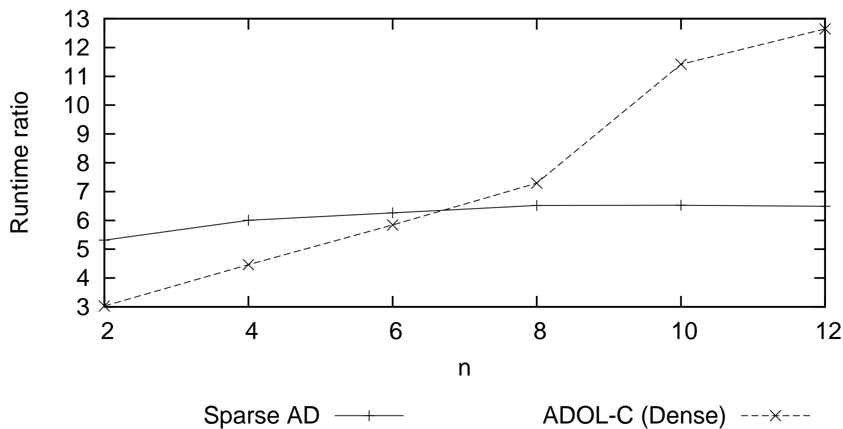

Figure 5. Runtime ratios for the electromagnetic simulator.

and as a result the ADOL-C tool wrote three files of sizes 69 MB, 180 MB, and 1.6 GB to the disk. The runtime ratio of this approach was 160.7, which can be considered quite large since there are only two design variables. This was probably due to the fact that the tape could no longer fit into the memory, but had to be written to the disk.

Aside from the tape based approach we tested the tapeless mode offered by the ADOL-C. It is basically an implementation of the traditional forward mode automatic differentiation without exploitation of sparsity. This approach is actually a quite natural choice for this example, since the total number of independent variables is relatively small.

Our aim is to show that despite of the small total number of design variables exploitation of the sparsity in the forward propagation still pays off. Figure 5 shows the runtime ratios of the tapeless (dense) mode ADOL-C and our implementation of the sparse forward mode as the number of antenna elements in the model is increased. The runtime ratios are the averages over five different computations.

As expected due to the aforementioned sparsity structure of the variables, the runtime ratio of the sparse forward mode remains approximately constant independently of n. On the other hand, the tapeless mode of ADOL-C does not exploit sparsity, but differentiates all active variables with respect to all independent variables. Therefore the runtime ratio of this approach grows with n. We can see, that when the number of design variables exceeds seven, the sparse forward mode is more efficient than the dense approach.

Notice that this is not in any way an extreme example. One can easily imagine for example shape optimization problems where the design variables affect only a very small portion of the computation domain. One could of course exploit such sparsity manually by excluding the system matrix entries that are not affected by the design variables from the differentiation, but this kind of approach would require extra bookkeeping and be prone to errors. Utilizing the sparse forward mode one can simply define the relations of the mesh nodes to the design variables, and let the sparsity capturing technique make sure that only non-zero derivatives are computed. This happens at the price of a relatively small computational overhead.

4. Optimization example

In this section we consider optimization of a Yagi-Uda array that has six elements. The optimization problem is the minimization of the square of the absolute value

of the so called scattering parameter S_{11} over a set of frequencies:

$$\min_{\boldsymbol{\alpha}} \max_{\boldsymbol{f}} |S_{11}(\boldsymbol{f}, \boldsymbol{\alpha})|^2 \tag{25}$$

where $\boldsymbol{f} = \{0.26 \text{ GHz}, 0.28 \text{ GHz}, 0.3 \text{ GHz}, 0.32 \text{ GHz}\}$.

The scattering parameter is defined as follows:

$$S_{11} = \frac{Z - Z_0}{Z + Z_0} \text{ with } Z = Y^{-1} \text{ and } Y = -\frac{1}{V^2} \boldsymbol{b}^T \boldsymbol{u}, \tag{26}$$

where Z_0 is the characteristic impedance of the feeding line $Z_0 = 50\ \Omega$, T denotes transpose of a column vector, V is the voltage of the feed, $\boldsymbol{b}$ is the excitation vector, and $\boldsymbol{u}$ is the solution vector.

We take the design variables $\boldsymbol{\alpha}$ to be the dimensions l_1, l_2, s_1 and s_2 (see Figure 3). The rest of the dimensions are fixed. Initial dimensions of the array are $s_1 = s_2 = s_3 = s_4 = s_5 = s_6 = 0.25$, $l_1 = l_2 = 0.45$, and $l_3 = l_4 = l_5 = l_6 = 0.406$.

Our implementation of the sparse automatic differentiation was used to differentiate the system matrix and excitation vector elements with respect to the design parameters. Using that information, the differentiation of the scattering parameter was easy due to the self-adjoint nature of the admittance Y (see [**?**] for details).

First we verify the correctness of our sensitivity analysis technique, and compute the sensitivity of $|S_{11}|^2$ with respect to the design parameters in the frequency 0.26 GHz. The following sensitivities were obtained:

$$\frac{\partial |S_{11}|^2}{\partial \boldsymbol{\alpha}} = \{0.8249, -5.9565, -0.3443, -0.1534\}.$$

These results were compared against response level forward finite differences using various step lengths. In Figure 6 are shown the gaps between the finite difference approximations and the sensitivities computed using automatic differentiation in the context of the adjoint approach. As we expect, the gaps diminish linearly as the finite difference step length is decreased, since the truncation errors in the finite difference approximations become smaller. For step lengths shorter than about 10^{-7} the round-off errors become significant in the finite difference computations, and the gaps between the derivatives increase again. Around the step length 10^{-7} the gaps between the derivatives are very small, and we can conclude that the sensitivities produced using the AD approach are correct.

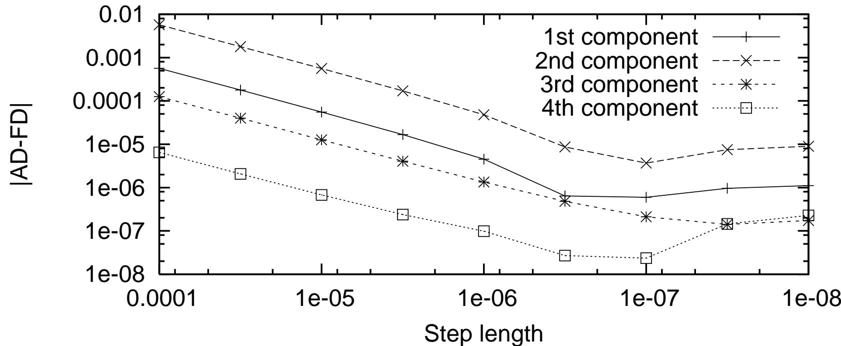

Figure 6. Gaps between derivatives produced using AD and the finite difference approximations.

Function `fminimax` of the MATLAB optimization toolbox was used to perform the optimization. We used the medium scale version, which utilizes sequential quadratic programming (SQP) as an optimization method. All parameters of the

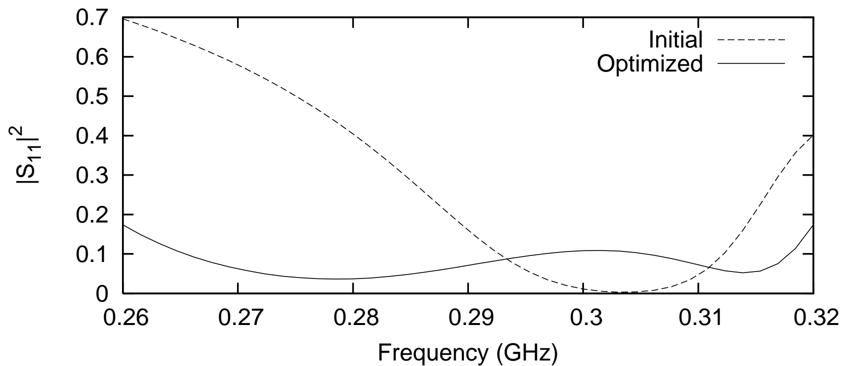

Figure 7. Initial and optimized performance of the antenna system over the frequency band.

optimizer were left to their default values. Constraints were posed such that l_1, $l_2 \in [0.406, 1]$, $l_1 \geq l_2$ and s_1, $s_2 \in [0.1, 1]$. The optimizer took 15 iterations, and needed 35 function evaluations. The optimal design was as follows: $l_1 = 0.549$, $l_2 = 0.494$, $s_1 = 0.278$ and $s_2 = 0.100$.

In Figure 7 are shown the initial and optimized values of $|S_{11}|^2$ over the consider frequency band. We see that a clear improvement is obtained: the optimized configuration exhibits much better performance in terms of the maximum absolute value of the scattering parameter over the frequency band.

5. Conclusions

In this paper we have presented our implementation of the so called sparse forward mode automatic differentiation. The approach exploits the sparsity by computing only the non-zero partial derivatives of each intermediate variable created during the computation. This approach does not require the so called taping, i.e. storing information of all operations performed during the computation, nor does it need any a priori sparsity information.

The sparse forward mode is sometimes criticized for having too much computational overhead for practical purposes. In this paper we have shown by numerical examples that for our implementation of the approach the computational overhead is somewhere around a factor of three, which can be considered quite reasonable.

We compared our implementation to the well known ADOL-C tool, which uses taping and compression techniques. Comparisons show that ADOL-C is more efficient if several evaluations of the Jacobian with a fixed sparsity pattern are performed, and if the tape created by ADOL-C can fit into the memory. However, when differentiation of a complex electromagnetic solver was considered, the tape based approach turned out to be impractical. Namely, the resulting huge tape had to be written to the disk, causing a large run time overhead. On the other hand, our implementation of the sparse forward mode completely avoids disk access, and can be used for the differentiation of the electromagnetic solver without problems. With this example we also showed that compared to dense forward mode automatic differentiation, exploitation of the sparsity can pay off even in cases where the total number of independent variables is quite small.

The automatic index domain capturing used in the sparse forward mode makes the technique easy to use for the developer of the code, since it has very few restrictions. For example, the developer does not have to manually keep track of the dependencies of the variables, the evaluation of the function does not have to be organized into a single subroutine, and the intermediate variables can be saved in any kind of data structure.

Acknowledgements

The electromagnetic simulator has been developed by Seppo Järvenpää, Pasi Ylä-Oijala, and others, from the Helsinki University of Technology. The authors would also like to thank Dr. Jussi Rahola for introducing us to interesting electromagnetic design problems.