Ralf Giering and Thomas Kaminski

# Applying TAF to generate efficient derivative code of Fortran 77-95 programs

*This paper features FastOpt's automatic differentiation (AD) tool Transformation of Algorithms in Fortran (TAF), a source to source translator for programs written in Fortran 77-95. TAF and its predecessor TAMC have a long record of successful large-scale applications. Here we give an overview on how TAF approaches typical challenges of AD such as handling of badly written program code, of large memory/disk requirements, of iterative solvers or of black box routines. We also point out, where the user is required to prepare his program code prior to invoking TAF.*

## 1. Introduction

FastOpt is a relatively young company which specialises in optimisation/inverse modelling, sensitivity analysis and the likes. This paper features Transformation of Algorithms in Fortran (TAF, [1]), our tool for Automatic Differentiation (AD, see [2]) of Fortran 77-95 programs. TAF operates as a source-to-source transformation tool: from a given Fortran program, which evaluates a function, TAF generates a second Fortran program, which evaluates the function's derivative.

In brief (see further [2, 3]), the basic approach of a source-to-source translator is as follows: First the (potentially long and complex) function code is decomposed into elementary functions. Each elementary function is then differentiated, which yields a so- called local Jacobian. The local Jacobians are multiplied, which, according to the chain rule, yields the derivative of the composite function. For the particular principles and rules that are implemented in TAF we refer to Talagrand [3] or Giering and Kaminski [4]. The generated code yields derivatives, which are exact up to machine precision. This is in contrast to approximating the derivative by divided differences of function evaluations, which is the traditional (and inaccurate) alternative to AD.

Applying the chain rule results in evaluating a product of the local Jacobians. If evaluation works in the order given by the function, the derivative code is referred to as operating in forward (or tangent linear) mode. If it works in the opposite order, the derivative code is referred to as operating in reverse (or adjoint) mode. Of course, both modes yield the same result. Depending on the ratio of the function's independent (input) to dependent (output) variables, there are large differences in the required computational effort, though. Similar to the finite difference approximation, the computational resources needed in forward mode increase with the number of independent variables. In reverse mode, they are roughly proportional to the number of dependent variables. For instance, the reverse mode, i.e. an adjoint model, is particularly well-suited for optimisation/control theory applications: It efficiently provides the gradient of a scalar valued objective function with respect to an arbitrary number of independent (control) variables to a gradient-based algorithm for minimisation.

TAF generates both forward and reverse mode derivative codes, i.e. tangent linear and adjoint models. Code that evaluates second derivatives (Hessian code) is generated by applying TAF twice. TAF and its predecessor TAMC have a long record of successful applications to large-scale models in the various fields of numerical modelling. Selected applications are presented by Giering et al. [5] and Giering and Kaminski [6]. Recent and relatively complete lists are available at www.FastOpt.com and at www.autodiff.org, together with information on further AD tools. For a scalar valued function, TAF generated adjoint models require only about 2-5 times the CPU time of a function evaluation.

An AD tool is challenged by a number of complexities, especially when it tackles the reverse mode. In the following sections we list a number of typical challenges, which TAF faces when handling these large-scale applications. We also include examples of code preparations we suggest to users prior to invoking TAF. Finally we present some conclusions.

## 2. Badly written code

TAF often is confronted with non-standard Fortran, i.e. language extensions of particular compilers. TAF can handle many of these but certainly does fail from time to time, in which case the user needs to remove the non-standard constructs from the function code.

TAF needs to analyse the control flow structure of the code. In addition, in reverse mode, the control flow must be reversed. Ordinary branches or loops don't pose any problem to TAF, see, e.g., [7] on how those are handled. Structures which repeatedly use statements like `cycle, exit`, and `goto` can, however, have extremely complex control flows. Some of these structures can be analysed and normalised by TAF, i.e. they are replaced by higher-level constructs. As an example of what we regard as bad code, take the following hidden `if then else` structure, which is expressed by two `goto`s:

```
c     compute sqrt(x), but avoid x = 0
      subroutine mysqrt( n, x, y )
      implicit none
      integer n
      real    x(n), y, epsilon
      parameter (epsilon=0.0001)
      if ( abs(x(1)) .lt. epsilon) goto 100
      y = sqrt (x(1))
      goto 200
 100  continue
      print*,' Attention: x close to zero (sqrt not differentiable)'
      y = 0.
 200  continue
      end
```

TAF first normalises this to an explicit `if then else` structure and then differentiates it (with declarations removed to save space):

```
      subroutine admysqrt( n, x, adx, ady )
... declarations removed by hand ...
      if (abs(x(1)) .lt. epsilon) then
      else
        adx(1) = adx(1)+ady*(1./(2.*sqrt(x(1))))
        ady = 0.
      endif
      end
```

The next bit of code shows a hidden `do while` structure that is expressed by two `goto`s.

```
C     for given x find y such that  0 = sqrt(y) - y + x
      subroutine model( n, x, y )
      implicit none
      integer n, i, imax
      real    x(n), y, yold, epsilon
      parameter (epsilon=0.0001, imax=100)
      y = 2.
      i = 1
 100  continue
      yold = y
      y = sqrt(y) + x(1)
      if ( i .gt. imax) goto 200
      if ( abs(y-yold) .gt. epsilon) goto 100
      return
 200  continue
      print*,'bad luck: no convergence'
      END
```

TAF cannot yet normalise this structure and reports an error. The user needs to modify the function code, e.g. by including the loop's termination condition into its entry condition:

```
 100  continue
      i = i + 1
      yold = y
      y = sqrt(y) + x(1)
      if ( abs(y-yold) .gt. epsilon .and. i .lt. imax) goto 100
      if (i .eq. imax) print*, 'bad luck: no convergence'
      END
```

In this form, TAF can normalise the structure to a `do while` structure and differentiate it. A better and much clearer solution, however, would be to manually replace this type of structure altogether by an explicit `do while` structure.

### 3. Large disk/memory requirements

To evaluate a local Jacobian, the derivative code typically requires values of particular variables (required values [7]) from the function evaluation. In adjoint code, required values can be provided by recomputation or by storing them during a preceding function evaluation and reading them back in during the derivative evaluation [7]. To generate most efficient recomputations, TAF uses a much refined version of the Efficient Recomputation Algorithm (ERA [8]). Furthermore, the user can trigger generation of a storing/reading scheme by inserting TAF store directives in the function code. Via an argument of the TAF init directive the user can choose, whether to store the values on dynamic or static memory or on disk. This gives large flexibility in trading-off CPU time against memory/disk resources. Furthermore, generation of a checkpointing scheme [9] can be easily triggered [6]. Implementation of a checkpointing scheme considerably reduces the required memory/disk space at the cost of almost an additional function evaluation.

### 4. Iterative solvers

It is common to use iterative solvers for differential or algebraic equations. TAF by default generates adjoint code, which uses the values of the required variables from each iteration in the function evaluation. Christianson [10, 11] has derived an alternative adjoint formulation, which uses only the values from the final iteration in the function evaluation. In TAF generation of this memory/disk-efficient alternative adjoint code can be triggered by inserting a directive in the function code [7, 5].

### 5. Inaccessible code

Often library routines are called in the function evaluation, but their code is not available. TAF, however, needs the data flow information for the entire function code. By providing the necessary information via TAF flow directives, the user can enable TAF to handle these black box routines. Let's suppose, for example, that the code of the subroutine `mysqrt` from section  that is called for the following function evaluation was not available:

```
c$taf subroutine mysqrt input = 1, 2
c$taf subroutine mysqrt output =       3
c$taf subroutine mysqrt depend = 1, 2
c$taf subroutine mysqrt active =    2, 3
c$taf subroutine mysqrt adname  = admysqrt
      subroutine model( n, x, y )
      implicit none
      integer n
      real    x(n), y
      call mysqrt(n,x,y)
      end
```

The above set of TAF flow directives provides all necessary information. The numbers in the first 4 directives refer to the position of variables in the subroutine's argument list. The first two determine input and output variables, the next the required variables, and the fourth the active variables, i.e. those for which derivative information has to be propagated by the derivative code. The last one sets the name of the corresponding adjoint subroutine to `admysqrt`. The generated adjoint code then calls `admysqrt` (again with declarations removed to save space):

```
      subroutine admodel( n, x, adx, y, ady )
... declarations removed by hand ...
      call admysqrt( n,x,adx,ady )
      end
```

The user has to add hand written code for `admysqrt` to the generated derivative code. TAF flow directives may also be used for routines of which the code is submitted to TAF. In this case, TAF ignores the routine's source code and uses the flow information. We exploit this feature whenever we want to use a routines's hand written derivative code instead of generating it. This is useful, e.g. to exploit self adjointness [5].

## 6. Further recommendations

In addition to the above mentioned user interventions and preparations, we would like to give a few more recommendations to TAF users: In time integrations often values such as boundary conditions are read in every time step from a sequential file. For generating the corresponding adjoint these should be replaced by direct access files, which allow access in arbitrary order. Also initialisations should be split off the function code, because they can complicate the dependency analysis. Furthermore, the allocation/deallocation structure of Fortran 90 allocatable arrays should be kept simple.

## 7. Conclusions

We have presented a number of typical challenges that an AD tool must face and have shown how TAF approaches these challenges. We have provided examples of situations in which the user needs to modify the function code or should insert TAF directives. A large number of successful applications demonstrate that the user can well handle all typical complexities with some preparation of the function code. Since we are rapidly progressing in TAF development, the number of required preparations is constantly decreasing. We are also going to transfer the TAF concepts from Fortran to C and build the AD tool Transformation of Algorithms in C (TAC).

## References

[1] FastOpt, Transformation of Algorithms in Fortran, http://www.FastOpt.com.
URL `http://www.FastOpt.com`

[2] A. Griewank, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM, Philadelphia, 2000.

[3] O. Talagrand, The use of adjoint equations in numerical modelling of the atmospheric circulation, in: A. Griewank, G. F. Corliss (Eds.), Automatic Differentiation of Algorithms: Theory, Implementation, and Application, SIAM, Philadelphia, Penn., 1991, pp. 169–180.

[4] R. Giering, T. Kaminski, Recipies for adjoint code construction, ACM Trans. Math. Software 24 (4) (1998) 437–474, also appeared as Max-Planck Institut für Meteorologie Hamburg, Technical Report No. 212, 1996.

[5] R. Giering, T. Kaminski, T. Slawig, Applying TAF to a Navier-Stokes solver that simulates an Euler flow around an airfoil (2002).

[6] R. Giering, T. Kaminski, On the performance of derivative code generated by TAMC, submitted to Optimization Methods and Software .

[7] R. Giering, T. Kaminski, Recipes for Adjoint Code Construction, ACM Trans. Math. Software 24 (4) (1998) 437–474.

[8] R. Giering, T. Kaminski, Generating recomputations in reverse mode AD, in: G. Corliss, A. Griewank, C. Fauré, L. Hascoet, U. Naumann (Eds.), Automatic Differentiation of Algorithms: From Simulation to Optimization, Springer Verlag, Heidelberg, 2002, Ch. 33, pp. 283–291.

[9] A. Griewank, Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation, Optimization Methods and Software 1 (1992) 35–54.

[10] B. Christianson, L. C. W. Dixon, S. Brown, Sharing storage using dirty vectors, in: M. Berz, C. Bischof, G. Corliss, A. Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, Penn., 1996, pp. 107–115.

[11] B. Christianson, Reverse accumulation and implicit functions, Optimization Methods and Software 9 (4) (1998) 307–322.

Ralf Giering, FastOpt, Martinistr. 21, 20251 Hamburg
Thomas Kaminski, FastOpt, Martinistr. 21, 20251 Hamburg