Development and first applications of TAC++

Michael Voßbeck, Ralf Giering, and Thomas Kaminski

FastOpt, Schanzenstr. 36, 20357 Hamburg, Germany, http://www.FastOpt.com

Summary. The paper describes the development of the software tool Transformation of Algorithms in C++ (TAC++) for automatic differentiation (AD) of C(++) codes by source-to-source translation. We have transferred to TAC++ a subset of the algorithms from its well-established Fortran equivalent, Transformation of Algorithms in Fortran (TAF). TAC++ features forward and reverse as well as scalar and vector modes of AD. Efficient higher order derivative code is generated by multiple application of TAC++. High performance of the generated derivate code is demonstrated for five examples from application fields covering remote sensing, computer vision, computational finance, and aeronautics. For instance, the run time of the adjoints for simultaneous evaluation of the function and its gradient is between 1.9 and 3.9 times slower than that of the respective function codes. Options for further enhancement are discussed.

Key words: automatic differentiation, reverse mode, adjoint, Hessian, source-to-source transformation, C++, remote sensing, computational finance, CFD

1 Introduction

Automatic Differentiation (AD [15], see also http://autodiff.org) is a technique that yields accurate derivative information for functions defined by numerical programmes. Such a programme is decomposed into elementary functions defined by operations such as addition or division and intrinsics such as cosine or logarithm. On the level of these elementary functions, the corresponding derivatives are derived automatically, and application of the chain rule results in an evaluation of a multiple matrix product, which is automated, too.

The two principal implementations of AD are operator overloading and source-to-source transformation. The former exploits the overloading capability of modern object-oriented programming languages such as Fortran-90 [23] or C++ [16, 1]. All relevant operations are extended by corresponding derivative operations. Source-to-source transformation takes the function code as input and generates a second code that evaluates the function's derivative.

This derivative code is then compiled and executed. Hence, differentiation and derivative evaluation are separated. The major disadvantage is that any code analysis for the differentiation process has to rely exclusively on information that is available at compile time. On the other hand, once generated, the derivative code can be conserved, and the derivative evaluation can be carried out any time on any platform, independently from the AD-tool. Also, extended derivative code optimisations by a compiler (and even by hand) can be applied. This renders source-to-source transformation the ideal approach for large-scale and run-time-critical applications.

The forward mode of AD propagates derivatives in the execution order defined by the function evaluation, while the reverse mode operates in the opposite order. The AD-tool Transformation of Algorithms in Fortran (TAF, [10]) has generated highly efficient forward and reverse mode derivative codes of a number of large (5,000 - 375,000 lines excluding comments) Fortran 77-95 codes (for references see, e.g., [12] and `http://www.fastopt.com/references/taf.html`).

Regarding source-to-source transformation for C, to our knowledge, ADIC [3] is the only tool that is currently available. However, ADIC is restricted to the forward mode of AD. Hence, ADIC is not well-suited for differentiation of functions with a large number of independent and a small number of dependent variables, a situation typical of unconstrained optimisation problems. In this context, the restriction to the forward mode usually constitutes a serious drawback and requires an artificial reduction of the number of control variables.

This paper describes the development of our source-to-source translation tool TAC++ that features both forward (tangent) and reverse (adjoint) modes. As with TAF, we chose an application-oriented development approach and started from a simple, but non-trivial test code, which is introduced in sect. 2. Next, sect. 3 describes TAC++ and its application to the test code. Section 4 then discusses the performance of the generated code. Since this initial test development went on, and TAC++ has differentiated a number of codes, which are briefly described in sect. 5. Finally, sect. 6 draws conclusions.

2 Test codes

As starting point for our test code we selected the Roe Solver [24] of the CFD code EULSOLDO [5]. As a test object Roe's solver has become popular with AD-tool developers [25, 6]. EULSOLDO's original Fortran code has been transformed to C code (141 lines without comments and one statement per line) by means of the tool f2c [8] with command-line options `-A` (generate ANSI-C89), `-a` (storage class of local variables is automatic), and `-r8` (promote `real` to `double precision`). f2c also uses pointer types for all formal parameters, in order to preserve Fortran subroutine properties (call by reference). The f2c generated code also contains simple pointer arithmetics, as a consequence of different conventions of addressing elements of arrays in C and Fortran: While Fortran addresses the first element of the array `x` containing the independent variables by `x(1)`, the C version addresses this element by `x[0]`. In order to use the index values from the Fortran version, f2c includes a shift operation on the pointer to the array `x`, i.e. the statement `--x` is inserted. The transformed code is basic in the sense that it consists of the following language elements:

- Selected datatype: `int` and `double` in scalar, array, typedef, and pointer form
- Basic arithmetics: addition, subtraction, multiplication, division
- One intrinsic: `sqrt`
- A few control flow elements: `for`, `if`, `comma-expr`

```
void model(int *n, double x[], double *fc) {
    const int size = *n;
    const double weight = sin(3.);
    struct S { int cnt; double val; } loc[size], *loc_ptr;
    int i;
    double sum = 0.;
    for(i=0; i<size; i++)
      loc[i].val = x[i]*x[i];
    for(i=0; i<size; i++) {
      int m = size - 1 - i;
      double con = x[m] * weight;
      loc_ptr = &loc[i];
      sum += loc_ptr->val + con;
    }
    *fc = sum / size;
}
```

File 1: Second test code.

- A function call

The second test code (see file 1), with `x` as independent and `fc` as dependent variable, is taken from the TAC++ test environment. It belongs to the tests for correct handling of an active `struct` datatype, scoping, and access to a pointer.

3 TAC++

TAC++ is invoked via a script that establishes a secure-shell connection to the FastOpt servers. As TAC++ accepts preprocessed ANSI C89 code, the access script runs a preprocessor such as cpp before transferring the function code to the servers. It is, hence, advisable to regenerate the derivative code after porting the modelling system to a new platform.

```
l[0] = (d__1 = uhat - ahat, ((d__1) >= 0 ? (d__1) : -(d__1)));
```

File 2: Comma-expression in the C version of EULSOLDO

```
d__1=uhat-ahat;
l[0]=(d__1 >= 0 ? d__1 : -d__1);
```

File 3: file 2 in normalised form

In the design process of TAC++, our approach has been to implement well-proven and reliable TAF algorithms. When the front end has translated the C source code into an internal representation, a normalisation replaces certain language constructs by equivalent canonical code that is more appropriate to the transformation phase. For example the comma-expression in the C version of EULSOLDO that is shown in file 2 is normalised to the code segment shown in file 3.

TAC++ then performs an activity analysis to determine those functions and variables, that are active in the sense, that they depend on the input variables and affect the output variables [2, 10], which both have to be specified by the user. The main challenge in reverse mode AD

```
/* Absolute eigenvalues, acoustic waves with entropy fix. */
l[0] = (d__1 = uhat - ahat, abs(d__1));
dl1 = qrn[0] / qr[0] - ar - qln[0] / ql[0] + al;
/* Computing MAX */
d__1 = dl1 * 4.;
dl1star = max(d__1,0.);
if (l[0] < dl1star * .5) {
   l[0] = l[0] * l[0] / dl1star + dl1star * .25;
}
```

File 4: `if`-statement including code the if-clause depends on (from C version of EULSOLDO)

```
/* RECOMP============== begin */
d__1=uhat-ahat;
l[0]=(d__1 >= 0 ? d__1 : -d__1);
/* RECOMP============== end */
if( l[0] < dl1star*0.500000 ) {
  dl1star_ad+=l_ad[0]*(-(l[0]*l[0]/(dl1star*dl1star))+0.250000);
  l_ad[0]=l_ad[0]*(2*l[0]/dl1star);
}
```

File 5: Recomputations for adjoint statement of `if`-statement from File 4

is to provide required values, i.e. values from the function evaluation that are needed in the derivative code (for details see [7, 17, 10, 11]). By default TAC++ uses recomputation for providing required values, instead of recording them on disk/in memory. Hence, the generated code has similar disk/memory requirements than the function code. As in TAF, the Efficient Recomputation Algorithm (ERA [11]) avoids unnecessary recomputations, which is essential for generating efficient derivative code. For instance the adjoint statement (see file 5) of the `if`-statement from file 4 has the required variables `d__1` and `l[0]`. While `dl1star` is still available from an earlier recomputation (not shown), `l[0]` may be overwritten by the `if`-statement itself. Hence, only recomputations for `l[0]` have to be generated.

For the first test code, TAC++ generates an adjoint code comprising 560 lines in well readable format, with one statement or declaration per line. This excludes comments and the code generated from the include file. The complete processing chain is depicted in the right branch of Fig. 1.

File 6 shows the adjoint of our second test code from file 1. Note the declaration of the adjoint struct `S_ad`. As the field `cnt` is passive, `S_ad` contains `val_ad`, the adjoint of `val`, as its single component. The declaration and the initialisation blocks are followed by a forward sweep for the function evaluation, which also provides required values to the adjoint block. As the loop kernel overwrites the required values of `m`, the adjoint loop kernel contains its recomputation before the block of adjoint assignments that uses `m`. `loc_ptr_ad` is also provided. The scope of the variables `m` and `con` is the loop kernel. Since `con` is active, its adjoint variable `con_ad` is added to the set of variables whose scope is the adjoint loop kernel.

4 Performance

We have tested the performance of the generated code in a number of test environments, i.e. for different combinations of processor, compiler, and level of compiler optimisation.

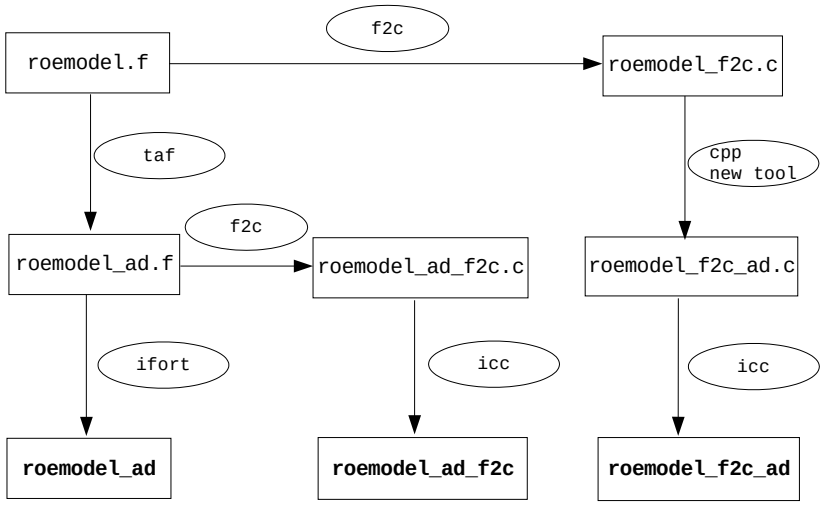

Fig. 1. Processing chain for test code. Oval boxes denote the stages of the processes. Rectangular boxes contain the files that are input/output to the individual stages of the process. Names of executables are printed in bold face letters. The right branch shows processing with f2c and new AD tool, the middle branch shows processing with TAF and f2c, and the left branch shows processing with TAF.

Our first test environment consists of a 3GHz Intel Core(TM)2 Duo processor and the Intel compiler (icc, Version 9.1) with flags "-fast -static". This environment achieves the fastest CPU-time for the function code. We have called it *standard* as it reflects the starting point of typical users, i.e. they are running a function code in production mode (as fast as possible) and need fast derivative code. In an attempt to isolate the impact of the individual factors processor, compiler, and optimisation level, further test environments have been constructed:

- The environment *gcc* differs from *standard* in that it uses the GNU C/C++ compiler (gcc, Version 4.2.1) with option "-O3 -static"
- The environment *AMD* differs from *standard* in that it uses another processor, namely the 1800 MHz Athlon64 3000+ and the corresponding fast compiler flags "-O3 -static".
- The environment *lazy* differs from *standard* in that it does not use compiler flags at all. In terms of compiler optimisation this is equivalent to the icc-flag "-O2".

For each environment, Table 1 lists the CPU time for a function evaluation ("Func"), a gradient and function evaluation ("ADM"), and their ratio. We used the timing module provided by ADOL-C, version 1.8.7 [16]. Each code has been run three times, and the fastest result has been recorded.

Compared to our Fortran tool TAF, TAC++ is still basic. To estimate the scope for performance improvement, we have applied TAF (with command-line options "-split -replaceintr")

```
void model_ad(int *n, double x[], double x_ad[], double *fc, double *fc_ad) {
    struct S;
    struct S_ad;
    const int size = *n;
    const double weight = sin(3.);
    struct S { int cnt; double val; };
    struct S_ad { double val_ad; };
    struct S loc[size];
    struct S *loc_ptr;
    int i;
    double sum;
    struct S_ad loc_ad[size];
    struct S_ad *loc_ptr_ad;
    double sum_ad;
    int ip1;
    for( ip1 = 0; ip1 < size; ip1++ )
      loc_ad[ip1].val_ad=0.;
    loc_ptr_ad=0;
    sum_ad=0.;
    sum=0.;
    for( i=0; i < size; i++ )
      loc[i].val=x[i]*x[i];
    for( i=0; i < size; i++ ) {
      int m;
      double con;
      m=size-1-i;
      con=x[m]*weight;
      loc_ptr=&loc[i];
      sum+=loc_ptr->val+con;
    }
    *fc=sum/size;
    sum_ad+=*fc_ad*(1F/size);
    *fc_ad=0;
    for( i=size-1; i >= 0; i-- ) {
      int m;
      double con;
      double con_ad;
      con_ad=0.;
      m=size-1-i;
      loc_ptr_ad=&loc_ad[i];
      loc_ptr_ad->val_ad+=sum_ad;
      con_ad+=sum_ad;
      x_ad[m]+=con_ad*weight;
      con_ad=0;
    }
    for( i=size-1; i >= 0; i-- ) {
      x_ad[i]+=loc_ad[i].val_ad*(2*x[i]);
      loc_ad[i].val_ad=0;
    }
    sum_ad=0;
}
```

File 6: Adjoint of File 1

to EULSOLDO's initial Fortran-version. A previous study on AD of EULSOLDO [6] identified this combination of TAF command-line options for generating the most efficient adjoint code. The TAF-generated adjoint has then been compiled with the Intel Fortran compiler (ifort, Version 9.1) and flags "-fast -static". This process is shown as left branch in Fig. 1. Table 2 compares the performance of TAC++ generated adjoint (in the environment *standard*, first row) with that of the TAF-generated adjoint (second row). Our performance ratio is in the range reported by [6] for a set of different environments. Besides the better performance of ifort-generated code, the second row also suggests that TAF-generated code is more efficient

Table 1. Performance of code generated by TAC++ (CPUs: seconds of CPU time)

Environment	Compiler	Options	Func[CPUs]	ADM[CPUs]	Ratio
standard	icc	-fast -static	2.2e-07	7.1e-07	3.2
gcc	gcc	-O3 -static	4.1e-07	2.0e-06	4.9
AMD	icc	-O3 -static	7.3e-07	2.4e-06	3.3
lazy	icc	–	4.7e-07	2.3e-06	4.9

by about 10%. In this comparison, both the AD tool and the compiler differ. To isolate their respective effects on the performance, we have carried out an additional test: We have taken the TAF-generated adjoint code, have applied f2c, and have compiled in the environment *standard* as depicted by the middle branch in Fig. 1. The resulting performance is shown in row 3 of Table 2. The value of 2.9 suggests that the superiority of the Fortran branch (row 2) over the C branch (row 1) cannot be attributed to the difference in compilers. It rather indicates some scope for improvement of the current tool in terms of performance of the generated code.

Two immediate candidates for improving this performance are the two TAF command-line options identified by [6]. The option "-replaceintr" makes TAF's normalisation phase replace intrinsics such as `abs, min, max` by `if-then-else` structures. In the C branch (row 1 of Table 2) this is already done by f2c, i.e. EULSOLDO's C version does not use any of these intrinsics. The TAF command-line option "-split", which introduces auxiliary variables to decompose long expressions to binary ones, is not available in TAC++ yet. Here might be some potential for improving the performance of the generated code.

Table 2. Performance of function and adjoint codes generated by TAC++ and TAF

Version	Func[CPUs]	ADM[CPUs]	Ratio
f2c → TAC++→ icc	2.2e-07	7.1e-07	3.2
TAF → ifort	2.2e-07	6.6e-07	2.9
TAF → f2c → icc	2.3e-07	6.7e-07	2.9

5 First TAC++ applications

Encouraged by the fast adjoint for our test code, we went on with our application-oriented development and tackled a set of native C codes of enhanced complexity from a variety of application areas. Table 3 gives an overview on these codes.

Two-stream [19] (available via `http://fapar.jrc.it`) simulates the radiative transfer within the vegetation canopy. In close collaboration with the Joint Research Centre (JRC) of the European Commission, we have constructed the inverse modelling package JRC-TIP [18, 21, 20]. JRC-TIP infers values and uncertainties for seven parameters such as the leaf

Table 3. Performance of derivatives of C codes generated by TAC++

Model	Application Area	#lines	Func[CPUs]	TLM/Func	ADM/Func	HES/Func
2stream	Remote Sensing	330	5.5e-6	1.7	3.8	23/7
ROF	Computer Vision	60	2.5e-6	1.6	1.9	yes
LIBOR	Comp. Finance	210	7.0e-5	1.3	3.7	
TAU-ij	Aerodynamics	130	1.1e-3	–	2.3	
Roeflux	Aero	140	2.2e-7	3.3	3.2	

area index (LAI) and the leaf radiative properties, which quantify the state of the vegetation from remotely sensed radiative fluxes and their uncertainties. The adjoint of two-stream is used to minimise the misfit between modelled and observed radiant fluxes. The inverse of the misfit's Hessian provides an estimate of the uncertainty range in the optimal parameter values. The full Hessian is generated in forward over reverse mode, meaning that the adjoint code is redifferentiated in (vector) forward mode. Table 3 lists the CPU times for the derivative codes in multiples of the CPU time of model code they are generated from. The timing has been carried out in the environment *standard* (see sect. 4). TLM (tangent linear model, i.e. scalar forward) and ADM (adjoint model, i.e. scalar reverse) values refer to the evaluation of both function and derivative. An evaluation of the 7 columns of two-stream's full Hessian requires the CPU time of 23 two-stream runs. For differentiation of the code, TAC++ was extended to handle further intrinsics ('cos', 'asin', 'exp', and 'sqrt') as well as nested function calls and nested 'for'-loops.

The ROF code maps the unknown structure of an image onto the misfit to the observed image plus a regularisation term that evaluates the total energy. The total variation denoising approach for image reconstruction uses the ROF adjoint for minimisation of that function. Our test configuration uses only 120 (number of pixels) independent variables. Hessian times vector code, again generated in forward over reverse mode, is used as additional information for the minimisation algorithm. The generated derivative code is shown in [22], who also present details on the application. Differentiation of this code required to extend TAC++ so as to handle nested loops with pointer arithmetics.

The LIBOR market model [4] is used to price interest derivative securities via Monte Carlo simulation of their underlying. Giles and Glasserman [14] present the efficient computation of price sensitivites with respect to 80 forward rates (so-called Greeks) with a hand-coded pathwise adjoint. For a slightly updated version of his model code, Giles [13] compares the performance of hand-coded and two AD-generated tangent and adjoint versions. On Intel's icc compiler, with highest possible code optimisations, the TAC++-generated adjoint is about a factor 2.5 slower than the hand-coded one and more than a factor of 10 faster than an operator overloading version derived with FADBAD [1]. The challenge for tool development were nested 'for' loops. The generated code is available at `http://www.fastopt.com/liborad-dist-1.tgz`.

TAU-ij is the Euler version of TAU, the German aeronautic community's solver for simulations on unstructured grids [9]. As a test for TAC++, we have selected a routine (`calc_inner_fluxes_mapsp`) from the core of the solver. The challenge of this application is the handling of the `struct` datatype.

In its current state, TAC++ does not cover C++, nor the full ANSI C89 standard. For example, dynamic memory allocation, `while`–loops, `union`s, function pointers, and functions with non-void return value are not handled yet. But this is rapidly changing.

6 Conclusions

We described the early development steps of TAC++ and gave an overview on recent applications, which use forward and reverse as well as scalar and vector modes of AD. Efficient higher order derivative code is generated by multiple application of TAC++, as demonstrated by Hessian codes for two of the applications. Although the generated derivative code is highly efficient, we identified scope for further improvement. The ongoing development is application-driven, i.e. we will tackle challenges as they arise in applications. Hence, TAC++'s functionality will be enhanced application by application. Fortunately, many of the C++ challenges occur also in Fortran-90. Examples are handling of *dynamic memory, operator overloading, overloaded functions*, or accessing of *private variables*. This allows us to port well-proved TAF algorithms to TAC++. Other challenges such as handling of classes, inheritance, and templates are specific to C++ and require a solution that is independent from TAF.

Acknowledgement. The authors thank Paul Cusdin and Jens-Dominik Müller for providing EULSOLDO in its original Fortran 77 version, Michael Pock for providing the ROF code, and Nicolas Gauger, Ralf Heinrich, and Norbert Kroll for providing the TAU-ij code. We enjoyed the joint work with Bernard Pinty and Thomas Lavergne on the two-stream inversion package and with Mike Giles on differentiation of the LIBOR code.

References

1. Bendtsen, C., Stauning, O.: FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark (1996)
2. Bischof, C.H., Carle, A., Khademi, P., Mauer, A.: ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. IEEE Computational Science & Engineering **3**(3), 18–32 (1996)
3. Bischof, C.H., Roh, L., Mauer, A.: ADIC — An extensible automatic differentiation tool for ANSI-C. Software–Practice and Experience **27**(12), 1427–1456 (1997)
4. Brace, A., Gatarek, D., Musiela, M.: The Market Model of Interest Rate Dynamics. Mathematical Finance **7**(2), 127–155 (1997)
5. Cusdin, P., Müller, J.D.: EULSOLDO. Tech. Rep. QUB-SAE-03-02, QUB School of Aeronautical Engineering (2003)
6. Cusdin, P., Müller, J.D.: Improving the performance of code generated by automatic differentiation. Tech. Rep. QUB-SAE-03-04, QUB School of Aeronautical Engineering (2003)
7. Faure, C.: Adjoining strategies for multi-layered programs. Optimisation Methods and Software **17**(1), 129–164 (2002). To appear. Also appeared as INRIA Rapport de recherche no. 3781, BP 105-78153 Le Chesnay Cedex, FRANCE, 1999.
8. Feldman, S.I., Weinberger, P.J.: A portable Fortran 77 compiler. In: UNIX Vol. II: research system (10th ed.), pp. 311–323. W. B. Saunders Company (1990)

9. Gerhold, T., Friedrich, O., Evans, J., Galle, M.: Calculation of Complex Three-Dimensional Configurations Employing the DLR-TAU-Code. AIAA Paper **167** (1997)
10. Giering, R., Kaminski, T.: Recipes for Adjoint Code Construction. ACM Trans. Math. Software **24**(4), 437–474 (1998)
11. Giering, R., Kaminski, T.: 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, chap. 33, pp. 283–291. Springer Verlag, Heidelberg (2002)
12. Giering, R., Kaminski, T., Slawig, T.: Generating Efficient Derivative Code with TAF: Adjoint and Tangent Linear Euler Flow Around an Airfoil. Future Generation Computer Systems **21**(8), 1345–1355 (2005)
13. Giles, M.: Monte Carlo evaluation of sensitivities in computational finance. In: E.A. Lipitakis (ed.) to appear in HERCMA 2007 (2007)
14. Giles, M., Glasserman, P.: Smoking adjoints: fast Monte Carlo Greeks. Risk pp. 92–96 (2006)
15. Griewank, A.: On automatic differentiation. In: M. Iri, K. Tanabe (eds.) Mathematical Programming: Recent Developments and Applications, pp. 83–108. Kluwer Academic Publishers, Dordrecht (1989)
16. Griewank, A., Juedes, D., Utke, J.: ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Software **22**(2), 131–167 (1996)
17. Hascoët, L., Naumann, U., Pascual, V.: TBR analysis in reverse-mode automatic differentiation. Future Generation Computer Systems **21**(8), 1401–1417 (2005)
18. Lavergne, T., Voßbeck, M., Pinty, B., Kaminski, T., Giering, R.: Evaluation of the two-stream inversion package. EUR 22467 EN, European Commission - DG Joint Research Centre, Institute for Environment and Sustainability (2006)
19. Pinty, B., Lavergne, T., Dickinson, R., Widlowski, J., Gobron, N., Verstraete, M.: Simplifying the interaction of land surfaces with radiation for relating remote sensing products to climate models. J. Geophys. Res (2006)
20. Pinty, B., Lavergne, T., Kaminski, T., Aussedat, O., Giering, R., Gobron, N., Taberner, M., Verstraete, M.M., Voßbeck, M., Widlowski, J.L.: Partitioning the solar radiant fluxes in forest canopies in the presence of snow. J. Geophys. Res. p. in press (2008)
21. Pinty, B., Lavergne, T., Voßbeck, M., Kaminski, T., Aussedat, O., Giering, R., Gobron, N., Taberner, M., Verstraete, M.M., Widlowski, J.L.: Retrieving surface parameters for climate models from MODIS-MISR albedo products. J. Geophys. Res. **112** (2007)
22. Pock, T., Pock, M., Bischof, H.: Algorithmic differentiation: Application to variational problems in computer vision. IEEE Transactions on Pattern Analysis and Machine Intelligence **29**(7), 1180–1193 (2007)
23. Pryce, J.D., Reid, J.K.: AD01, a Fortran 90 code for automatic differentiation. Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England (1998)
24. Roe, P.L.: Approximate Riemann solvers, parameter vectors, and difference schemes. J. Comput. Phys. **135**(2), 250–258 (1997)
25. Tadjouddine, M., Forth, S.A., Pryce, J.D.: AD tools and prospects for optimal AD in CFD flux Jacobian calculations. In: G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (eds.) Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science, chap. 30, pp. 255–261. Springer, New York, NY (2002)