INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team tropics

Transformations et Outils Informatiques pour le Calcul Scientifique

Sophia Antipolis

THEME NUM

Activity Report

2006

Table of contents

1. Team

Head of project team
Laurent Hascoët [DR INRIA, HdR]

Vice-head of project team
Valérie Pascual [CR INRIA]

Administrative assistant
Marie-Line Ramfos [TR INRIA]

Staff members
Alain Dervieux [DR INRIA, HdR]

Post-doctoral students
Hicham Tber [(Since September 1st)]

Ph. D. students
Mauricio Araya-Polo [(Till October 31st)]
Benjamin Dauvergne

Junior technical staff
Christophe Massol [(Till September 30th)]

Partner scientists
Bruno Koobus [Université de Montpellier 2]

2. Overall Objectives

2.1. Overall Objectives

The TROPICS team is at the junction of two research domains:

- **AD:** On one hand, we study software engineering techniques, to analyze and transform programs semi-automatically. In the past, we developed semi-automatic parallelization strategies aiming at SPMD parallelization. Presently, we focus on Automatic Differentiation (AD). AD transforms a program `P` that computes a function F, into a program `P'` that computes some derivatives of F, analytically. In particular, the so-called *reverse mode* of AD yields gradients. However, this reverse mode remains very delicate to use, and its implementation requires carefully crafted algorithms.
- **CFD application of AD:** On the other hand, we study the application of AD, and particularly of the adjoint method, to Computational Fluid Dynamics. This involves necessary adaptation of optimization strategies. This work applies to two real-life problems, optimal shape design and mesh adaptation.

The second aspect of our work (optimization in Scientific Computing), is thus at the same time the motivation and the application domain of the first aspect (program analysis and transformation, and computation of gradients through AD). Concerning AD, our goal is to automatically produce derivative programs that can compete with the hand-written sensitivity and adjoint programs which exist in the industry. We implement our ideas and algorithms into the tool TAPENADE, which is developed and maintained by the project. Apart from being an AD tool, TAPENADE is also a platform for other analyses and transformations of scientific programs. TAPENADE is easily available. We provide a web server, and alternatively a version can be downloaded from our web server. Practical details can be found in section 5.1.

Our present research directions are :

- Modern numerical methods for finite elements or finite differences: multigrid methods, mesh adaptation.
- Optimal shape design or optimal control in the context of fluid dynamics: for example shape optimization of the wings of a supersonic aircraft, to reduce sonic bang. In this context, we study the optimization of nonsteady processes and the use of higher-order derivatives for robust optimization.
- Automatic Differentiation : differentiate particular algorithms in a specially adapted manner, validate the derivatives, and reduce runtime and memory consumption when computing gradients ("adjoints") with the reverse mode and second-order derivatives, targetting at application to large nonstationnary simulation codes.
- Common tools for program analysis and transformation: adequate internal representation, Call Graphs, Flow Graphs, Data-Dependence Graphs.

3. Scientific Foundations

3.1. Automatic Differentiation

Keywords: *adjoint models*, *automatic differentiation*, *optimization*, *program transformation*, *scientific computing*, *simulation*.

Participants: Mauricio Araya-Polo, Benjamin Dauvergne, Laurent Hascoët, Christophe Massol, Valérie Pascual, Hicham Tber.

automatic differentiation (AD) Automatic transformation of a program, that returns a new program that computes some derivatives of the given initial program, i.e. some combination of the partial derivatives of the program's outputs with respect to its inputs.

adjoint model Mathematical manipulation of the Partial Derivative Equations that define a problem, obtaining new differential equations that define the gradient of the original problem's solution.

checkpointing General trade-off technique, used in the reverse mode of AD, that trades duplicate execution of a part of the program to save some memory space that was used to save intermediate results. Checkpointing a code fragment amounts to running this fragment without any storage of intermediate values, thus saving memory space. Later, when such an intermediate value is required, the fragment is run a second time to obtain the required values.

Automatic or Algorithmic Differentiation (AD) differentiates *programs*. An AD tool takes as input a source computer program P that, given a vector argument $X \in I\!R^n$, computes some vector function $Y = F(X) \in I\!R^m$. The AD tool generates a new source program that, given the argument X, computes some derivatives of F. In short, AD first assumes that P represents all its possible run-time sequences of instructions, and it will in fact differentiate these sequences. Therefore, the *control* of P is put aside temporarily, and AD will simply reproduce this control into the differentiated program. In other words, P is differentiated only piecewise. Experience shows that this is reasonable in most cases, and going further is still an open research problem. Then, any sequence of instructions is identified with a composition of vector functions. Thus, for a given control:

$$\begin{array}{ccl} P & \text{is} & \{I_1; I_2; \cdots I_p;\}, \\ F & = & f_p \circ f_{p-1} \circ \cdots \circ f_1, \end{array} \tag{1}$$

where each f_k is the elementary function implemented by instruction I_k. Finally, AD simply applies the chain rule to obtain derivatives of F. Let us call X_k the values of all variables after each instruction I_k, i.e. $X_0 = X$ and $X_k = f_k(X_{k-1})$. The chain rule gives the Jacobian F' of F

$$F'(X) = f'_p(X_{p-1}) \,.\, f'_{p-1}(X_{p-2}) \,.\, \cdots \,.\, f'_1(X_0) \tag{2}$$

which can be mechanically translated back into a sequence of instructions I'_k, and these sequences inserted back into the control of P, yielding program P'. This can be generalized to higher level derivatives, Taylor series, etc.

In practice, the above Jacobian $F'(X)$ is often far too expensive to compute and store. Notice for instance that equation (2) repeatedly multiplies matrices, whose size is of the order of $m \times n$. Moreover, some problems are solved using only some projections of $F'(X)$. For example, one may need only *sensitivities*, which are $F'(X).\dot{X}$ for a given direction $\dot{X}$ in the input space. Using equation (2), sensitivity is

$$F'(X).\dot{X} = f'_p(X_{p-1}) \,.\, f'_{p-1}(X_{p-2}) \,.\, \cdots \,.\, f'_1(X_0) \,.\, \dot{X}, \tag{3}$$

which is easily computed from right to left, interleaved with the original program instructions. This is the principle of the *tangent mode* of AD, which is the most straightforward, of course available in TAPENADE.

However in optimization, data assimilation [38], adjoint problems [32], or inverse problems, the appropriate derivative is the *gradient* $F'^*(X).\overline{Y}$. Using equation (2), the gradient is

$$F'^*(X).\overline{Y} = f'^*_1(X_0).f'^*_2(X_1). \cdots .f'^*_{p-1}(X_{p-2}).f'^*_p(X_{p-1}).\overline{Y}, \tag{4}$$

which is most efficiently computed from right to left, because matrix$\times$vector products are so much cheaper than matrix$\times$matrix products. This is the principle of the *reverse mode* of AD.

This turns out to make a very efficient program, at least theoretically [34]. The computation time required for the gradient is only a small multiple of the run-time of P. It is independent from the number of parameters n. In contrast, notice that computing the same gradient with the *tangent mode* would require running the tangent differentiated program n times.

We can observe that the X_k are required in the *inverse* of their computation order. If the original program *overwrites* a part of X_k, the differentiated program must restore X_k before it is used by $f'^*_{k+1}(X_k)$. There are two strategies for that:

- **Recompute All (RA):** the X_k is recomputed when needed, restarting P on input X_0 until instruction I_k. The TAF [30] tool uses this strategy. Brute-force RA strategy has a quadratic time cost with respect to the total number of run-time instructions p.
- **Store All (SA):** the X_k are restored from a stack when needed. This stack is filled during a preliminary run of P, that additionally stores variables on the stack just before they are overwritten. The ADIFOR [25] and TAPENADE tools use this strategy. Brute-force SA strategy has a linear memory cost with respect to p.

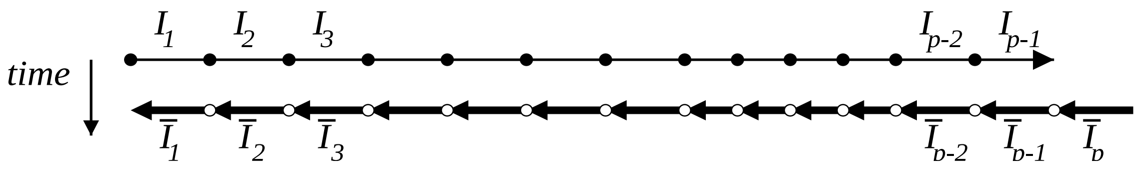


Figure 1. The "Store-All" tactic

Both RA and SA strategies need a special storage/recomputation trade-off in order to be really profitable, and this makes them become very similar. This trade-off is called *checkpointing*. Since TAPENADE uses the SA strategy, let us describe checkpointing in this context. The plain SA strategy applied to instructions I_1 to I_p builds the differentiated program sketched on figure 1, where an initial "forward sweep" runs the original program and stores intermediate values (black dots), and is followed by a "backward sweep" that computes the derivatives in the reverse order, using the stored values when necessary (white dots). Checkpointing a fragment **C** of the program is illustrated on figure 2. During the forward sweep, no value is stored while in **C**. Later, when the backward sweep needs values from **C**, the fragment is run again, this time with storage. One can see that the maximum storage space is grossly divided by 2. This also requires some extra memorization (a "snapshot"), to restore the initial context of **C**. This snapshot is shown on figure 2 by slightly bigger black and white dots.

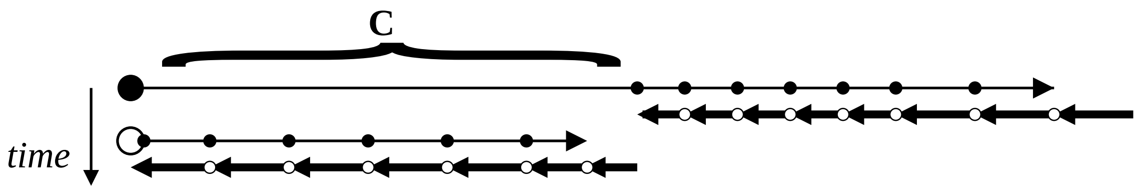


*Figure 2. Checkpointing **C** with the "Store-All" tactic*

Checkpoints can be nested. In that case, a clever choice of checkpoints can make both the memory size and the extra recomputations grow like only the logarithm of the size of the program.

3.2. Static Analyses and Transformation of programs

Keywords: *abstract interpretation*, *abstract syntax tree*, *compilation*, *control flow graph*, *data dependence graph*, *data flow analysis*, *program transformation*, *static analysis*.

Participants: Mauricio Araya-Polo, Benjamin Dauvergne, Laurent Hascoët, Christophe Massol, Valérie Pascual, Hicham Tber.

abstract syntax tree Tree representation of a computer program, that keeps only the semantically significant information and abstracts away syntactic sugar such as indentation, parentheses, or separators.

control flow graph Representation of a procedure body as a directed graph, whose nodes, known as basic blocks, contain each a list of instructions to be executed in sequence, and whose arcs represent all possible control jumps that can occur at run-time.

abstract interpretation Model that describes program static analyses as a special sort of execution, in which all branches of control switches are taken simultaneously, and where computed values are replaced by abstract values from a given *semantic domain*. Each particular analysis gives birth to a specific semantic domain.

data flow analysis Program analysis that studies how a given property of variables evolves with execution of the program. Data Flow analyses are static, therefore studying all possible run-time behaviors and making conservative approximations. A typical data-flow analysis is to detect whether a variable is initialized or not, at any location in the source program.

data dependence analysis Program analysis that studies the itinerary of values during program execution, from the place where a value is generated to the places where it is used, and finally to the place where it is overwritten. The collection of all these itineraries is often stored as a *data dependence graph*, and data flow analysis most often rely on this graph.

data dependence graph Directed graph that relates accesses to program variables, from the write access that defines a new value to the read accesses that use this value, and conversely from the read accesses to the write access that overwrites this value. Dependences express a partial order between operations, that must be preserved to preserve the program's result.

The most obvious example of a program transformation tool is certainly a compiler. Other examples are program translators, that go from one language or formalism to another, or optimizers, that transform a program to make it run better. AD is just one such transformation. These tools use sophisticated analyses [23] to improve the quality of the produced code. These tools share their technological basis. More importantly, there are common mathematical models to specify and analyze them.

An important principle is *abstraction*: the core of a compiler should not bother about syntactic details of the compiled program. In particular, it is desirable that the optimization and code generation phases be independent from the particular input programming language. This can generally be achieved through separate *front-ends*, that produce an internal language-independent representation of the program, generally an abstract syntax tree. For example, compilers like `gcc` for C and `g77` for FORTRAN77 have separate front-ends but share most of their back-end.

One can go further. As abstraction goes on, the internal representation becomes more language independent, and semantic constructs such as declarations, assignments, calls, IO operations, can be unified. Analyses can then concentrate on the semantics of a small set of constructs. We advocate an internal representation composed of three levels.

- At the top level is the *call graph*, whose nodes are the procedures. There is an arrow from node A to node B iff A possibly calls B. Recursion leads to cycles. The call graph captures the notions of visibility scope between procedures, that come from modules or classes.
- At the middle level is the control flow graph. There is one flow graph per procedure, i.e. per node in the call graph. The flow graph captures the control flow between atomic instructions. Flow control instructions are represented uniformly inside the control flow graph.
- At the lowest level are abstract syntax trees for the individual atomic instructions. Certain semantic transformations can benefit from the representation of expressions as directed acyclic graphs, sharing common sub-expressions.

To each basic block is associated a symbol table that gives access to properties of variables, constants, function names, type names, and so on. Symbol tables must be nested to implement *lexical scoping*.

Static program analyses can be defined on this internal representation, which is largely language independent. The simplest analyses on trees can be specified with inference rules [26], [35], [24]. But many analyses are more complex, and are thus better defined on graphs than on trees. This is the case for *data-flow analyses*, that look for run-time properties of variables. Since flow graphs are cyclic, these global analyses generally require an iterative resolution. *Data flow equations* is a practical formalism to describe data-flow analyses. Another formalism is described in [27], which is more precise because it can distinguish separate *instances* of instructions. However it is still based on trees, and its cost forbids application to large codes. *Abstract Interpretation* [28] is a theoretical framework to study complexity and termination of these analyses.

Data flow analyses must be carefully designed to avoid or control combinatorial explosion. The classical solution is to choose a hierarchical model. In this model, information, or at least a computationally expensive part of it, is synthesized. Specifically, it is computed bottom up, starting on the lowest (and smallest) levels of the program representation and then recursively combined at the upper (and larger) levels. Consequently, this synthesized information must be made independent of the context (i.e., the rest of the program). When the synthesized information is built, it is used in a final pass, essentially top down and context dependent, that propagates information from the "extremities" of the program (its beginning or end) to each particular subroutine, basic block, or instruction.

Even then, data flow analyses are limited, because they are static and thus have very little knowledge of actual run-time values. Most of them are *undecidable*; that is, there always exists a particular program for which the result of the analysis is uncertain. This is a strong, yet very theoretical limitation. More concretely, there are always cases where one cannot decide statically that, for example, two variables are equal. This is even more frequent with two pointers or two array accesses. Therefore, in order to obtain safe results, conservative *over-approximations* of the computed information are generated. For instance, such approximations are made when analyzing the activity or the TBR ("To Be Restored") status of some individual element of an array. Static and dynamic *array region analyses* [41], [29] provide very good approximations. Otherwise, we make a coarse approximation such as considering all array cells equivalent.

When studying program *transformations*, one often wants to move instructions around without changing the results of the program. The fundamental tool for this is the *data dependence graph*. This graph defines an order between *run-time* instructions such that if this order is preserved by instructions rescheduling, then the output of the program is not altered. Data dependence graph is the basis for automatic parallelization. It is also useful in AD. *Data dependence analysis* is the static data-flow analysis that builds the data dependence graph.

3.3. Automatic Differentiation and Computational Fluid Dynamics

Keywords: *adjoint methods*, *adjoint state*, *computational fluid dynamics*, *gradient*, *linearization*, *optimization*.

Participants: Alain Dervieux, Laurent Hascoët, Bruno Koobus.

linearization The mathematical equations of Fluid Dynamics are Partial Derivative Equations, that are discretized and then solved by a computer program. Linearization of these equations, or alternatively linearization of the computer program, gives a modelization of the behavior of the flow when small perturbations are applied. This is useful when the perturbations are effectively small, like in acoustics, or when one wants the sensitivity of the system with respect to one parameter, like in optimization.

adjoint state Consider a system of Partial Derivative Equations that define some characteristics of a system with respect to some input parameters. Consider one particular scalar characteristic. Its sensitivity, (or gradient) with respect to the input parameters can be defined as the solution of "adjoint" equations, deduced from the original equations through linearization and transposition. The solution of the adjoint equations is known as the adjoint state.

Computational Fluid Dynamics is now able to make reliable simulations of very complex systems. For example it is now possible to simulate completely the 3D air flow around a plane that captures the physical phenomena of shocks and turbulence. The next step in CFD appears to be optimization. Optimization is one degree higher in complexity, because it repeatedly simulates, evaluates directions of optimization and applies optimization steps, until an optimum is reached.

We restrict here to gradient descent methods. One risk is obviously to fall into local minima before reaching the global minimum. We do not address this question, although we believe that more robust approaches, such as evolutionary approaches, could benefit from a coupling with gradient descent approaches. Another well-known risk is the presence of discontinuities in the optimized function. We investigate two kinds of methods to cope with discontinuities: we can devise AD algorithms that detect the presence of discontinuities, and we can design optimization algorithms that solve some of these discontinuities.

We investigate several approaches to obtain the gradient. There are actually two extreme approaches:

- One can write an *adjoint system*, then discretize it and program it by hand. The adjoint system is a new system, deduced from the original equations, and whose solution, the *adjoint state*, leads to the gradient. A hand-written adjoint is very sound mathematically, because the process starts back from the original equations. This process implies a new separate implementation phase to solve the adjoint system. During this manual phase, mathematical knowledge of the problem can be translated into many hand-coded refinements. But this may take an enormous engineering time. Except for special strategies (see [32]), this approach does not produce an exact gradient of the discrete functional, and this can be a problem if using optimization methods based on descent directions.

- A program that computes the gradient can be built by pure Automatic Differentiation in the reverse mode (*cf* 3.1). It is in fact the adjoint of the discrete functional computed by the software, which is piecewise differentiable. It produces exact derivatives almost everywhere. Theoretical results [31] guarantee convergence of these derivatives when the functional converges. This strategy gives reliable descent directions to the optimization kernel, although the descent step may be tiny, due to discontinuities. Most importantly, AD adjoint is *generated* by a tool. This saves a lot of development and debug time. But this systematic approach leads to massive use of storage, requiring code transformation by hand to reduce memory usage. Mohammadi's work [36] [39] illustrates the advantages and drawbacks of this approach.

The drawback of AD is the amount of storage required. If the model is steady, can we use this important property to reduce this amount of storage needed? Actually this is possible, as shown in [33], where computation of the adjoint state uses the iterated states in the direct order. Alternatively, most researchers (see for example [36]) use only the fully converged state to compute the adjoint. This is usually implemented by a hand modification of the code generated by AD. But this is delicate and error-prone. The TROPICS team investigate hybrid methods that combine these two extreme approaches.

4. Application Domains

4.1. Panorama

Automatic Differentiation of programs gives sensitivities or gradients, that are useful for many types of applications:

- optimum shape design under constraints, multidisciplinary optimization, and more generally any algorithm based on local linearization,
- inverse problems, such as parameter estimation and in particular variational data assimilation in climate sciences (meteorology, oceanography)
- first-order linearization of complex systems, or higher-order simulations, yielding reduced models for simulation of complex systems around a given state,
- mesh adaptation and mesh optimization with gradients or adjoints,
- equation solving with the Newton method,
- sensitivity analysis, propagation of truncation errors.

We will detail some of them in the next sections. These applications require an AD tool that differentiates programs written in classical imperative languages, FORTRAN77, FORTRAN95, C, or C++. We also consider our AD tool TAPENADE as a platform to implement other program analyses and transformations. TAPENADE does the tedious job of building the internal representation of the program, and then provides an API to build new tools on top of this representation. One application of TAPENADE is therefore to build prototypes of new program analyses.

4.2. Multidisciplinary optimization

A CFD program computes the flow around a shape, starting from a number of inputs that define the shape and other parameters. From this flow, it computes an optimization criterion, such as the lift of an aircraft. To optimize the criterion by a gradient descent, one needs the gradient of the output criterion with respect to all the inputs, and possibly additional gradients when there are constraints. The reverse mode of AD is a promising way to compute these gradients.

4.3. Inverse problems

Inverse problems aim at estimating the value of hidden parameters from other measurable values, that depend on the hidden parameters through a system of equations. For example, the hidden parameter might be the shape of the ocean floor, and the measurable values the altitude and speed of the surface. Another example is *data assimilation* in weather forecasting. The initial state of the simulation conditions the quality of the weather prediction. But this initial state is largely unknown. Only some measures at arbitrary places and times are available. The initial state is found by solving a least squares problem between the measures and a guessed initial state which itself must verify the equations of meteorology. This rapidly boils down to solving an adjoint problem, which can be done though AD [40].

4.4. Linearization

Simulating a complex system often requires solving a system of Partial Differential Equations. This is sometimes too expensive, in particular in the context of real time. When one wants to simulate the reaction of this complex system to small perturbations around a fixed set of parameters, there is a very efficient approximate solution: just suppose that the system is linear in a small neighborhood of the current set of parameters. The reaction of the system is thus approximated by a simple product of the variation of the parameters with the Jacobian matrix of the system. This Jacobian matrix can be obtained by AD. This is especially cheap when the Jacobian matrix is sparse. The simulation can be improved further by introducing higher-order derivatives, such as Taylor expansions, which can also be computed through AD. The result is often called a *reduced model*.

4.5. Mesh adaptation

It has been noticed that some approximation errors can be expressed by an adjoint state. Mesh adaptation can benefit from this. The classical optimization step can give an optimization direction not only for the control parameters, but also for the approximation parameters, and in particular the mesh geometry. The ultimate goal is to obtain optimal control parameters up to a precision prescribed in advance.

5. Software

5.1. Tapenade

Participants: Laurent Hascoët [contact], Mauricio Araya-Polo, Nicolas Chleq [Development Engineer], Benjamin Dauvergne, Christophe Massol, Valérie Pascual, Hicham Tber.

TAPENADE is the Automatic Differentiation tool developed by the TROPICS team. TAPENADE progressively implements the results of our research about models and static analyses for AD. From this standpoint, TAPENADE is a research tool. Our objective is also to promote the use of AD in the scientific computation world, including the industry. Therefore the team constantly maintains TAPENADE to meet the demands of our industrial users. TAPENADE can be simply used as a web server, available at the URL
http://tapenade.inria.fr:8080/tapenade/index.jsp
It can also be downloaded and installed from our FTP server
ftp://ftp-sop.inria.fr/tropics/tapenade/README.html
A documentation is available on our web page
http://www-sop.inria.fr/tropics/
and as an INRIA technical report (RT-0300)
http://hal.inria.fr/inria-00069880

TAPENADE differentiates computer programs according to the model described in section 3.1. It supports three modes of differentiation:

- the *tangent* mode that computes a directional derivative $F'(X).\dot{X}$,
- the *vector tangent* mode that computes $F'(X).\dot{X}_n$ for many directions X_n simultaneously, and can therefore compute Jacobians, and
- the *reverse* mode that computes the gradient $F'^*(X).\overline{Y}$.

An obvious fourth mode could be the *vector reverse* mode, which is not yet implemented. Many other modes exist in the other AD tools in the world, that compute for example higher degree derivatives or Taylor expansions. For the time being, we restrict ourselves to first-order derivatives and we put our efforts on the reverse mode. But as we said before, we also view TAPENADE as a platform to build new program transformations, in particular new differentiations.

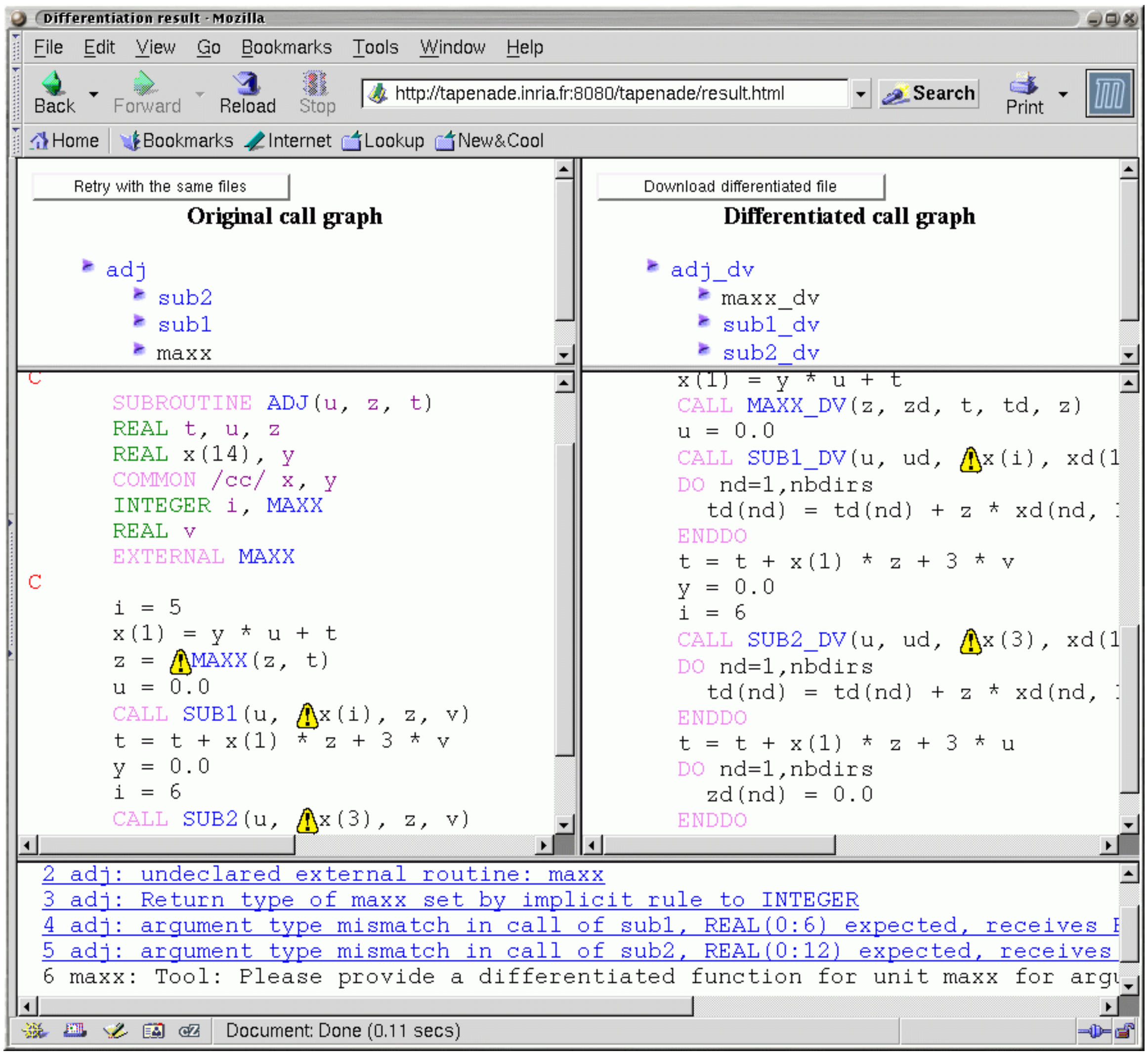
Like any program transformation tool, TAPENADE needs sophisticated static analyses in order to produce an efficient output. Concerning AD, the following analyses are a must, and TAPENADE now performs them all:

- **Pointers destinations:** For any static program transformation, and in particular differentiation, it is essential to have a precise knowledge of the possible destinations of each pointer at each code line. Otherwise one must make conservative assumptions that will lead to less efficient code. Our static pointer analysis finds precise information about pointer destinations, taking into account memory allocation and deallocation operations.
- **Activity:** The end-user has the opportunity to specify which of the output variables must be differentiated (called the dependent variables), and with respect to which of the input variables (called the independent variables). Activity analysis propagates the dependent, backward through the program, to detect all intermediate variables that possibly influence the dependent. Conversely, activity analysis also propagates the independent, forward through the program, to find all intermediate variables that possibly depend on the independent. Only the intermediate variables that both depend on the independent and influence the dependent are called *active*, and will receive an associated derivative variable. Activity analysis makes the differentiated program smaller and faster.
- **Adjoint Liveness and Read-Write:** Programs produced by the reverse mode of AD show a very particular structure, due to their mechanism to restore intermediate values of the original program in the *reverse* order. This has deep consequences on the liveness and Read-Write status of variables, that we can exploit to take away unnecessary instructions and memory usage from the reverse (adjoint) program. This makes the adjoint program smaller and faster by factors that can go up to 40%.
- **TBR:** The reverse mode of AD, with the Store-All strategy, stores all intermediate variables just before they are overwritten. However this is often unnecessary, because derivatives of some expressions (e.g. linear expressions) only use the derivatives of their arguments and not the original arguments themselves. In other words, the local Jacobian matrix of an instruction may not need all the intermediate variables needed by the original instruction. The *To Be Restored (TBR)* analysis finds which intermediate variables need not be stored during the forward sweep, and therefore makes the differentiated program smaller in memory.

Several other strategies are implemented in TAPENADE to improve the differentiated code. For example, a data-dependence analysis allows TAPENADE to move instructions around safely, gathering instructions to reduce cache misses. Also, long expressions are split in a specific way, to minimize duplicate sub-expressions in the derivative expressions.

The input languages of TAPENADE are FORTRAN77 and FORTRAN95. Thanks to the language-independent internal representation of programs, as shown on figure 4, we could relatively easily extend TAPENADE to C, at least for simple programs. More work is still required to obtain a completely reliable differentiation of C.

There are two user interfaces for TAPENADE. One is a simple command that can be called from a shell or from a Makefile. The other is interactive, using JAVA SWING components and HTML pages. This interface allows one to use TAPENADE from WINDOWS as well as LINUX. The input interface lets one specify interactively the routine to differentiate, its independent inputs and dependent outputs. The output interface, shown on figure 3, displays the differentiated programs, with HTML links that implement source-code correspondence, as well as correspondence between error messages and locations in the source.


Differentiation result - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://tapenade.inria.fr:8080/tapenade/result.html
Search Print
Home Bookmarks Internet Lookup New&Cool
Retry with the same files
Download differentiated file
Document: Done (0.11 secs)

Original call graph

```
adj
  sub2
  sub1
  maxx
```

Differentiated call graph

```
adj_dv
  maxx_dv
  sub1_dv
  sub2_dv
```

```
C
      SUBROUTINE ADJ(u, z, t)
      REAL t, u, z
      REAL x(14), y
      COMMON /cc/ x, y
      INTEGER i, MAXX
      REAL v
      EXTERNAL MAXX
C

      i = 5
      x(1) = y * u + t
      z = ⚠MAXX(z, t)
      u = 0.0
      CALL SUB1(u, ⚠x(i), z, v)
      t = t + x(1) * z + 3 * v
      y = 0.0
      i = 6
      CALL SUB2(u, ⚠x(3), z, v)
```

```
x(1) = y * u + t
CALL MAXX_DV(z, zd, t, td, z)
u = 0.0
CALL SUB1_DV(u, ud, ⚠x(i), xd(1
DO nd=1,nbdirs
  td(nd) = td(nd) + z * xd(nd,
ENDDO
t = t + x(1) * z + 3 * v
y = 0.0
i = 6
CALL SUB2_DV(u, ud, ⚠x(3), xd(1
DO nd=1,nbdirs
  td(nd) = td(nd) + z * xd(nd,
ENDDO
t = t + x(1) * z + 3 * u
DO nd=1,nbdirs
  zd(nd) = 0.0
ENDDO
```

```
2 adj: undeclared external routine: maxx
3 adj: Return type of maxx set by implicit rule to INTEGER
4 adj: argument type mismatch in call of sub1, REAL(0:6) expected, receives
5 adj: argument type mismatch in call of sub2, REAL(0:12) expected, receives
6 maxx: Tool: Please provide a differentiated function for unit maxx for argu
```

Figure 3. TAPENADE *output interface, with source-code-error correspondence*

TAPENADE is now available for LINUX, SUN, and WINDOWS-XP platforms.

Figure 4 shows the architecture of TAPENADE. It is implemented mostly in JAVA, apart from the front-ends which are separated and can be written in their own languages.

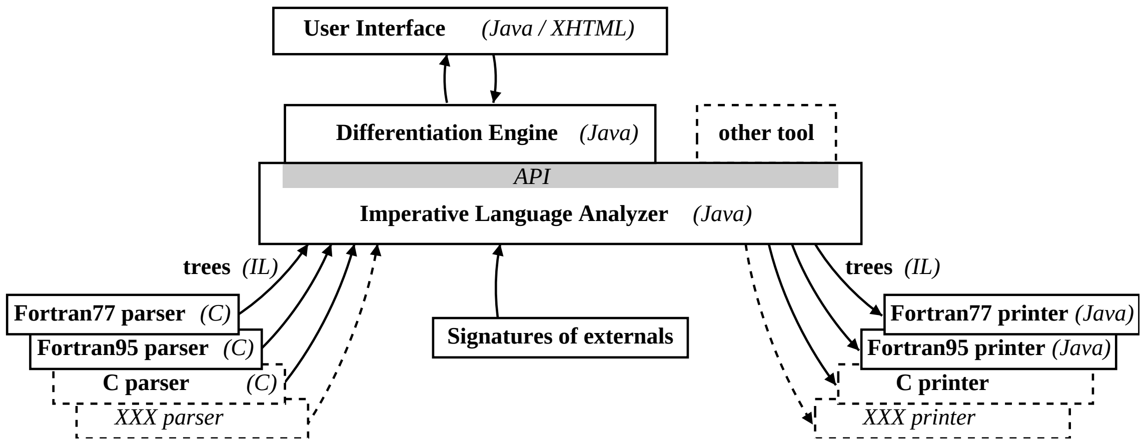


Figure 4. Overall Architecture of TAPENADE

Notice the clear separation between the general-purpose program analyses, based on a general representation, and the differentiation engine itself. Other tools can be built on top of the Imperative Language Analyzer platform.

The end-user can specify properties of external or black-box routines. This is essential for real industrial applications that use many libraries. The source of these libraries is generally hidden. However AD needs some information about these black-box routines in order to produce efficient code. TAPENADE lets the user specify this information in a separate signature file. Specifically for the reverse mode of AD, TAPENADE lets the user specify finely which procedure calls must be checkpointed or not, to improve the overal performances of the differentiated program.

6. New Results

6.1. Program Analyses to improve Checkpointing in Reverse AD

Keywords: *checkpointing*, *data-flow analyses*, *reverse mode of AD*, *snapshots*, *static analyses*.

Participants: Mauricio Araya-Polo, Benjamin Dauvergne, Laurent Hascoët.

As described in 3.1, the reverse mode of AD is a very appealing approach to compute gradients, but it suffers from the need to restore intermediate values in the reverse of their original computation order. In our approach, this question is basically solved through storage, whose cost can be very high. We use checkpointing to mitigate this cost, trading extra recomputations for memory space. An efficient storage and checkpointing strategy is really the key to a wide usage of reverse AD in the industry. This is why we continue our modelization and experimentation efforts to find the best possible strategies.

This year we formalized the “snapshot”. When a fragment of code is “checkpointed”, it is executed twice: the snapshot is the set of variables which must be saved and restored to ensure that the second execution is equivalent to the first. There exist several formulae that give the snapshot of a given checkpoint, all based on common sense. However we observe that further improvement of these formulae is delicate, and common sense is not enough to justify the improvements. In particular, reducing a snapshot can lead to an increase of storage in other parts of the code, and the overall memory consumption may turn out to be worse.

We study this question starting from a formal description of the interactions between all memory storage involved in checkpointing, and from it we derive a system of set equations that we solve formally, using Maple. After resolution, it turns out that there exist a continuum of optimal snapshots, none of which is strictly better (smaller) than the others. We are able to characterize these good solutions, and then to propose heuristics to select the most appropriate. One solution appears to give the best results in most applications, and it is now the default strategy in TAPENADE. We presented these results at the ICCS 2006 conference in Reading UK [17].

These results on snapshots are an additional contribution to the more general problem of selecting the best possible placement of checkpoints in a given code. This central problem is addressed from two different angles in the PhD research of Benjamin Dauvergne and Mauricio Araya.

Benjamin Dauvergne focuses on finding indicators that can be used to place checkpoints, i.e. find the nested code fragments that will give best performances if checkpointed. These indicators can result from a profiling run of the original program or of the differentiated program. One of this year's results is the development of a model that correctly predicts the memory and time behavior of a given checkpoints placement. Benjamin Dauvergne presented his preliminary results at this spring euro-AD workshop in Oxford, UK.

At the same time, Mauricio Araya experimented the new functionality of TAPENADE that allows the user to select manually which procedure calls must be checkpointed or not. The results are quite interesting: on one large code, an improved placement of checkpoints gained a factor 10 in execution time. On the average, 20% gains are commonplace. In turn, studying why one placement gives better results suggests heuristics that can be applied systematically. Mauricio Araya presented this new functionality of TAPENADE together with the experiments on real codes at the ECCOMAS conference in Egmond aan Zee, The Netherlands [21].

6.2. Target language extensions in TAPENADE

Keywords: *Automatic Differentiation*, *C*, *Fortran90*, *Tapenade*, *data-flow analysis*, *pointer analysis*, *static analysis*.

Participants: Nicolas Chleq, Laurent Hascoët, Christophe Massol, Valérie Pascual.

Following the evolution of programming practices among our end-users, we continuously adapt TAPENADE to new languages and additional programming constructs. This year's results concern the languages Fortran90 and C. About programming constructs, this year's progress concern pointers, dynamic allocation, array ("vectorial") notation, and modular constructs.

The bulk of new developments related to Fortran90 is now complete. TAPENADE now completely handles the modular constructs of Fortran90, namely modules with public and private components, interfaces, renaming, overloading, and optional or default arguments of procedures. Like last year, these developments were driven by the large Fortran90 applications that we are working on, such as the oceanography code OPA 9.0 (*cf* 6.3). Inside TAPENADE, these constructs do not depend on the particular target language: our objective is that these developments will be reused when TAPENADE handles other modular languages such as C++.

TAPENADE also handles the array notation of Fortran90, also known as "vectorial" notation. This year's developments have dealt with the arbitrary combinations of `WHERE`, `SUM`, and `MASK` constructs, for which the differentiated code is far from intuitive. In particular we found an interesting duality between `WHERE` and `MASK` through reverse AD: the best adjoint code for a `WHERE` statement will often need `SUM`'s with `MASK`'s, and vice-versa.

Christophe Massol has finished the development of the pointer static data-flow analysis in TAPENADE, including memory allocation and deallocation primitives. He also has adapted all subsequent analyses in TAPENADE, so that they give correct results on programs with pointers. Finally, the tangent differentiation module was also adapted to pointers, introducing the notion of a differentiated pointer variable, required when the pointed target variables are themselves differentiated.

Nicolas Chleq has developped a C front-end and back-end for TAPENADE. This development takes advantage of the existing architecture of TAPENADE, which is mostly language-independent in the middle-end: without any extension to the inside analysis and differentiation parts, we could differentiate a small C program. Yet more testing and validation is needed before a TAPENADE for C can be released.

6.3. Differentiation of large real applications

Keywords: *Agronomy*, *Automatic Differentiation*, *Oceanography*, *Tapenade*, *Variational Data Assimilation*.

Participants: Samuel Buis [INRA, Avignon], Benjamin Dauvergne, Bruno Ferron [IFREMER, Brest], Laurent Hascoët, Hicham Tber, Arthur Vidard [LMC/IMAG, Grenoble].

We studied application of Automatic Differentiation to several very large scientific computation codes.

Bruno Ferron of IFREMER Brest gave us the latest Fortran90 version of the ocean simulation code OPA, version 9.0, 80,000 lines long, developed mainly by the LOCEAN lab. at Paris 6 university. We obtained a validated adjoint code for one test configuration of OPA, named GYRE. We were invited to present the results at the Data Assimilation meeting in Toulouse [20]. This configuration simulates the behavior of a large rectangular basin of salt water, under the influence of the wind and of an initial vertical distribution of temperature, during 20 days. With TAPENADE, we computed the gradient of the heat flux across the northern boundary, with respect to the temperature field 20 days earlier.

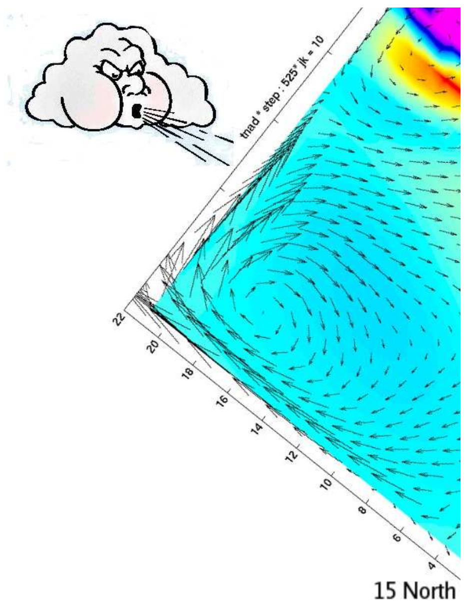


Figure 5. Influence of the temperature field at the depth of 300 metres on the heat flux across the northern boundary after 20 days

Figure 5 shows the computed gradient, which was validated automatically by comparison with divided differences, and validated as well by oceanographers who recognized on it classical shapes known as the Rossby and Kelvin waves. Computing this gradient takes only about 7 times as long as the simulation itself.

Since september 2006, Post-Doc student Hicham Tber has started differentiation of OPA on a much larger configuration of OPA, named NEMO, that simulates the North Atlantic basin on a longer period of time and with more realistic physics.

Colleagues from INRA in Avignon are starting to use TAPENADE on large agriculture simulation codes. We collaborate with them to understand their specific needs and to correct the errors they find in TAPENADE. They use AD in two main directions. One is sensitivity analysis for simulation codes such as STICS and ISBA, that simulate growing plants on a one-year like period. The other is the inverse problem of estimating ground parameters from satellite images, using simulation codes such as SAIL or MULTISAIL [37].

6.4. Optimal control

Keywords: *adjoint model*, *gradient*, *optimal control*, *optimum design*.

Participants: Frédéric Alauzet [Projet Gamma, INRIA-Rocquencourt], Francois Beux [Scuola Normale Superiore di Pisa, Italy], Alain Dervieux, Laurent Hascoët, Bruno Koobus, Massimiliano Martinelli [Universita di Pisa, Italy], Youssef Mesri [Université de Nice], Mariano Vázquez [Universitat de Girona, Spain].

In industry research groups, simulation is well mastered and the next frontier is optimization. This problem is very difficult, because the typical number of optimization parameters is very high, particularly in CFD optimal control. In an industrial context, the optimization of a dozen of scalar parameters will not produce an optimal shape for an aircraft, because an accurate description of a shape requires hundreds of parameters. The optimization parameter can be a function defined on a surface or a volume. In the discrete case, the number of parameters depends on the discretization chosen, and is a priori large. Therefore, optimization requires an enormous computing power. We discuss in 3.1 why the reverse mode of AD is an elegant way to obtain the adjoints that optimization uses. The reverse mode, and the subsequent adjoint state, are in fact the best way to get the gradients when the number of parameters is large.

In the European project HISAC on supersonic aircrafts (6.5), the state equation cannot be solved accurately without a strong anisotropic mesh adaptation. Therefore, we have to design a new algorithm for the simultaneous solution of shape optimisation and mesh adaptation [22].

Beside specific AD problems, practical application to control problems requires that we consider the following issues, discussed this year in [13],[18],[16]:

- efficient computation of a large scale adjoint system
- efficient optimization algorithms for large scales systems
- efficient preconditioners for this optimization.

6.5. Mesh adaptation

Keywords: *adjoint*, *mesh adaptation*, *optimization*.

Participants: Frederic Alauzet, Alain Dervieux, Bruno Koobus, Adrien Loseille [Projet Gamma, INRIA-Rocquencourt], Youssef Mesri.

This subject is addressed jointly by INRIA teams GAMMA (Rocquencourt), TROPICS, and SMASH. In this collaboration, GAMMA brings mesh and approximation expertise, TROPICS contributes AD and adjoint methods, SMASH works on approximation and CFD applications.

The resolution of the optimum problem using the innovative approach of an AD-generated adjoint, can be used in a slightly different context than optimal shape design namely, mesh adaptation. This will be possible if we can map the mesh adaptation problem into a differentiable optimal control problem. To this end, we have introduced a new methodology that consists in setting the mesh adaptation problem under the form of a purely functional one: the mesh is reduced to a continuous property of the computational domain, the continuous metric, and we minimize a continuous model of the error resulting from that metric. Then the problem of searching an adapted mesh is transformed in the research of an optimal metric.

In the case of mesh interpolation minimization, the optimum is given by a close formula and gives access to a complete theory demonstrating that second order accuracy can be obtained on discontinuous field approximation. In the case of adaptation for Partial Differential Equations, an Optimal Control is obtained. Together with project-team GAMMA (Frédéric Alauzet, Adrien Loseille), TROPICS contributes this research to the HISAC IP European project, which involves 30 other partners in aeronautics [22].

7. Dissemination

7.1. Links with Industry, Contracts

- Mike Giles from Oxford University, and his student Devendra Ghate continue using TAPENADE to build second-order derivatives programs in relation with Rolls-Royce turbine developments.
- TROPICS participates in the European project HISAC, which started at the end of last year.
- TROPICS participates in the project EVA-Flo: “Evaluation et Validation Automatique pour le calcul FLOttant”, which is an ANR project accepted this year, and whose main contractor in ENS Lyon (Nathalie Revol).
- TROPICS participates in the project LEFE, “Les Enveloppes Fluides et l’Environnement”, which is a CNRS API project accepted this year.
- TROPICS participates in the project NODESIM, “Non-Deterministic Simulation for CFD-based design methodologies”, which is an European STREP project which was accepted this year.
- TROPICS has submitted an INRIA-internal proposal for “équipes associées” with our partners in RWTH Aachen, whose results will be known shortly.
- We are aware of TAPENADE regular use by research groups in Argonne National Lab. (USA), Cranfield university (UK), Oxford university (UK), RWTH Aachen (Germany), and Humboldt university Berlin (Germany).

7.2. Conferences and workshops

- Alain Dervieux presented his research on March 14 when receiving the Dassault prize of the French Academy of Sciences.
- Laurent Hascoët presented AD and TAPENADE at the International Conference on High Performance Scientific Computing HPSC 2006 in Hanoi, Vietnam.
- Alain Dervieux gave a presentation at the CANUM 2006 conference, where he co-organized a minisymposium on Optimum Design together with J. Sokolowsky.
- Alain Dervieux and Bruno Koobus visited the Barcelona Computer Center, and gave two presentations, respectively on Optimal Shape Design and on the VMS turbulence model.
- Laurent Hascoët gave an invited talk to present results on reverse AD of the OPA 9.0 oceanography code at the “Colloque National sur l’Assimilation de Données” in Toulouse.
- Laurent Hascoët presented the team’s results on the Data-Flow Equations of Checkpointing at ICCS 2006 in Reading, UK.
- Laurent Hascoët is one of the organizers of the euro-AD workshops. This year, one edition took place in Oxford, UK, and one in Aachen, Germany. Benjamin Dauvergne presented some preliminary results at the Oxford edition in June.
- Laurent Hascoët gave a lecture on Automatic Differentiation at the CEA-EDF-INRIA “école d’été” on numerical analysis in June.
- Mauricio Araya gave a presentation and Alain Dervieux was co-author of another at ECCOMAS 2006 in Egmond aan Zee, The Netherlands.
- Laurent Hascoët presented the team’s research to colleagues at INRA Avignon, with the medium-term goal of introducing AD into the popular INRA codes.
- The TROPICS team co-organizes the French-Indian workshop “Numerical Simulation, Control and Design for Aeronautical and Space Applications” at INRIA Sophia-Antipolis from November 29th through December 1st 2006. Alain Dervieux and Laurent Hascoët will both make presentations.

- Mauricio Araya defended his PhD thesis on “Approaches to Assess Validity of Derivatives and to Improve Efficiency in Automatic Differentiation of Programs”, on November 24th in Sophia-Antipolis.

8. Bibliography

Major publications by the team in recent years

[1] F. COURTY. *Optimisation Différentiable en Mécanique des Fluides Numérique*, Ph. D. Thesis, Université Paris-sud, 2003.

[2] F. COURTY, A. DERVIEUX, B. KOOBUS, L. HASCOËT. *Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation*, in "Optimization Methods and Software", vol. 18, n^{o} 5, 2003, p. 615-627.

[3] A. DERVIEUX, L. HASCOËT, M. VÁZQUEZ, B. KOOBUS. *Optimization loops for shape and error control*, in "Recent Trends in Aerospace Design and Optimization", Tata-McGraw Hill, New Delhi, 2005, p. 363-373.

[4] A. GRIEWANK. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, Frontiers in Applied Mathematics, 2000.

[5] L. HASCOËT, M. ARAYA-POLO. *The Adjoint Data-Flow Analyses: Formalization, Properties, and Applications*, in "Automatic Differentiation: Applications, Theory, and Tools", H. M. BÜCKER, G. CORLISS, P. HOVLAND, U. NAUMANN, B. NORRIS (editors). , Lecture Notes in Computational Science and Engineering, Springer, 2005.

[6] L. HASCOËT, S. FIDANOVA, C. HELD. *Adjoining Independent Computations*, in "Automatic Differentiation of Algorithms: From Simulation to Optimization, New York, NY", G. CORLISS, C. FAURE, A. GRIEWANK, L. HASCOËT, U. NAUMANN (editors). , Computer and Information Science, chap. 35, Springer, 2001, p. 299-304.

[7] L. HASCOËT. *Analyses statiques et transformations de programmes: de la parallélisation à la différentiation*, Habilitation, Université de Nice Sophia-Antipolis, 2005.

[8] L. HASCOËT, U. NAUMANN, V. PASCUAL. *“To Be Recorded” Analysis in Reverse-Mode Automatic Differentiation*, in "Future Generation Computer Systems", vol. 21, n^{o} 8, 2004.

[9] L. HASCOËT, V. PASCUAL. *TAPENADE 2.1 user’s guide*, Technical report, n^{o} 300, INRIA, 2004, http://hal.inria.fr/inria-00069880.

[10] M. VÁZQUEZ, A. DERVIEUX, B. KOOBUS. *Multilevel optimization of a supersonic aircraft*, in "Finite Elements in Analysis and Design", vol. 40, 2004, p. 2101-2124.

Year Publications

Doctoral dissertations and Habilitation theses

[11] M. ARAYA-POLO. *Approaches to assess Validity of Derivatives and to Improve Efficiency in Automatic Differentiation of Programs*, PhD, Université de Nice Sophia-Antipolis, 2006.

Articles in refereed journals and book chapters

[12] P.-H. COURNÈDE, B. KOOBUS, A. DERVIEUX. *Positivity statements for a mixed-element-volume scheme on fixed and moving grids*, in "Revue européenne de mécanique numérique", vol. 15, n^{o} 7-8, 2006, p. 767-799.

[13] F. COURTY, A. DERVIEUX. *Multilevel functional Preconditioning for shape optimisation*, in "International Journal of CFD", vol. 20, n^{o} 7, 2006, p. 481-490.

[14] F. COURTY, D. LESERVOISIER, P.-L. GEORGE, A. DERVIEUX. *Continuous metrics and mesh optimization*, in "Applied Numerical Mathematics", vol. 56, 2006, p. 117-145.

[15] A. DERVIEUX, F. COURTY, T. ROY, M. VÁZQUEZ, B. KOOBUS. *Optimization loops for shape and error control*, in "Verification and validation methods for challenging multiphysics problems", B. BUGEDA, ET AL. (editors). , Extended version in INRIA Research Report 5413, CIMNE, Barcelona, 2006.

[16] M. VÁZQUEZ, A. DERVIEUX, B. KOOBUS. *A methodology for the shape optimization of flexible wings*, in "Engineering Computations", vol. 23, n^{o} 4, 2006, p. 344-367.

Publications in Conferences and Workshops

[17] B. DAUVERGNE, L. HASCOËT. *The Data-Flow Equations of Checkpointing in reverse Automatic Differentiation*, in "International Conference on Computational Science, ICCS 2006, Reading, UK", 2006.

[18] A. DERVIEUX, Y. MESRI, F. COURTY, L. HASCOËT, B. KOOBUS, M. VÁZQUEZ. *Calculs de sensibilité par différentiation pour l'Aérodynamique*, in "proceedings of CANUM06, ESAIM journal", to appear, 2006.

[19] M. FAGAN, L. HASCOËT, J. UTKE. *Data Representation Alternatives in Semantically Augmented Numerical Models*, in "6th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2006, Philadelphia, PA, USA", 2006.

[20] B. FERRON, L. HASCOËT. *Capacités actuelles de la Différentiation Automatique: l'adjoint d'OPA par TAPENADE*, in "Colloque National sur l'Assimilation de Données, Toulouse, France", 2006.

[21] L. HASCOËT, M. ARAYA-POLO. *Enabling User-driven checkpointing strategies in Reverse-mode Automatic Differentiation*, in "Proceedings of the ECCOMAS CFD 2006 conference, Egmond aan Zee, The Netherlands", also INRIA Research Report #5930, 2006, https://hal.inria.fr/inria-00079223.

Internal Reports

[22] F. ALAUZET, Y. MESRI, A. DERVIEUX. *Sonic Boom reduction by mesh adapted optimal control*, Research Report, n^{o} European project HISAC 18th Month 2.3, 2006.

References in notes

[23] A. AHO, R. SETHI, J. ULLMAN. *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[24] I. ATTALI, V. PASCUAL, C. ROUDET. *A language and an integrated environment for program transformations*, research report, n^{o} 3313, INRIA, 1997, http://hal.inria.fr/inria-00073376.

[25] A. CARLE, M. FAGAN. *ADIFOR 3.0 overview*, Technical report, n^{o} CAAM-TR-00-02, Rice University, 2000.

[26] D. CLÉMENT, J. DESPEYROUX, L. HASCOËT, G. KAHN. *Natural semantics on the computer*, in "K. Fuchi and M. Nivat, editors, Proceedings, France-Japan AI and CS Symposium, ICOT", Also, Information Processing Society of Japan, Technical Memorandum PL-86-6. Also INRIA research report # 416, 1986, p. 49-89, http://hal.inria.fr/inria-00076140.

[27] J.-F. COLLARD. *Reasoning about program transformations*, Springer, 2002.

[28] P. COUSOT. *Abstract Interpretation*, in "ACM Computing Surveys", vol. 28, n^{o} 1, 1996, p. 324-328.

[29] B. CREUSILLET, F. IRIGOIN. *Interprocedural Array Region Analyses*, in "International Journal of Parallel Programming", vol. 24, n^{o} 6, 1996, p. 513–546.

[30] R. GIERING. *Tangent linear and Adjoint Model Compiler , Users manual 1.2*, 1997, http://www.autodiff.com/tamc.

[31] J. GILBERT. *Automatic differentiation and iterative processes*, in "Optimization Methods and Software", vol. 1, 1992, p. 13–21.

[32] M.-B. GILES. *Adjoint methods for aeronautical design*, in "Proceedings of the ECCOMAS CFD Conference", 2001.

[33] A. GRIEWANK, C. FAURE. *Reduced Gradients and Hessians from Fixed Point Iteration for State Equations*, in "Numerical Algorithms", vol. 30(2), 2002, p. 113–139.

[34] A. GRIEWANK. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, SIAM, Frontiers in Applied Mathematics, 2000.

[35] L. HASCOËT. *Transformations automatiques de spécifications sémantiques: application: Un vérificateur de types incremental*, Ph. D. Thesis, Université de Nice Sophia-Antipolis, 1987.

[36] P. HOVLAND, B. MOHAMMADI, C. BISCHOF. *Automatic Differentiation of Navier-Stokes computations*, Technical report, n^{o} MCS-P687-0997, Argonne National Laboratory, 1997.

[37] C. LAUVERNET, F. BARET, L. HASCOËT, F.-X. LEDIMET. *Improved estimates of vegetation biophysical variables from MERIS TOA images by using spatial and temporal constraints*, in "proceedings of the 9th International symposium on Physical measurements and signatures in remote sensing, ISPMSRS 2005", 2005.

[38] F. LEDIMET, O. TALAGRAND. *Variational algorithms for analysis and assimilation of meteorological observations: theoretical aspects*, in "Tellus", vol. 38A, 1986, p. 97-110.

[39] B. MOHAMMADI. *Practical application to fluid flows of automatic differentiation for design problems*, in "Von Karman Lecture Series", 1997.

[40] N. ROSTAING. *Différentiation Automatique: application à un problème d'optimisation en météorologie*, Ph. D. Thesis, université de Nice Sophia-Antipolis, 1993.

[41] R. RUGINA, M. RINARD. *Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions*, in "Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation", ACM, 2000.