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

2004

# Table of contents

# 1. Team

**Head of project team**
Laurent Hascoët

**Vice-head of project team**
Valérie Pascual

**Administrative assistant**
Nathalie Balax-Bellesso

**Staff members**
Alain Dervieux

**Ph. D. students**
Mauricio Araya-Polo
Benjamin Dauvergne [Since september 2004]
Tristan Roy [Till may 2004]

**Junior technical staff**
Christophe Massol [Since october 2004]

**Partner scientists**
Bruno Koobus [University of Montpellier 2]
Mariano Vazquez [GridSystems, Mayorca, Spain]
Stephen Wornom [From july to september 2004]

# 2. 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 requires time and care.
- **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 adaption.

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 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 adaption.

- Optimal shape design, in the context of fluid dynamics: for example shape optimization of the wings of a supersonic aircraft, to reduce sonic bang. Also, new optimization tactics combining interior point, SQP or one-shot algorithms.
- Automatic Differentiation : reduce runtime and memory consumption when computing gradients (“adjoints”) or Jacobian matrices, differentiate parallel programs, differentiate particular algorithms in a specially adapted manner, validate the derivatives.
- 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.

**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, that returns 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 [41], adjoint problems [35], 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×vector products are so much cheaper than matrix×matrix products. This is the principle of the *reverse mode* of AD.

This turns out to make a very efficient program, at least theoretically [37]. 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 [33] 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 [28] and TAPENADE tools use this strategy. Brute-force SA strategy has a linear memory cost with respect to $p$.

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

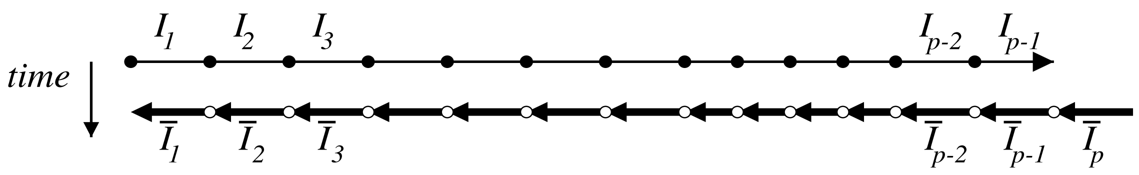


*Figure 1. The “Store-All” tactic*

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.

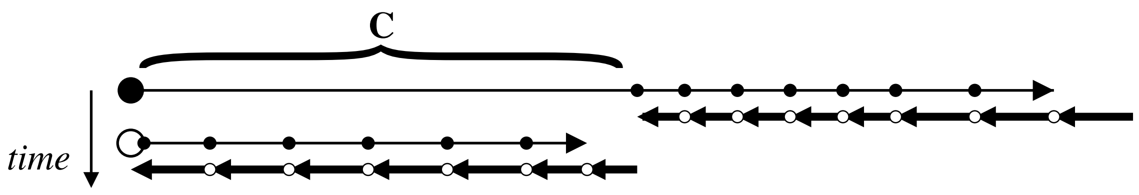


*Figure 2. Checkpointing **C** with the "Store-All" tactic*

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.

**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 conservaive 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 [26] 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 a 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 [29][38][27]. 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 [30], 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* [31] 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* [45][32] 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:** Tristan Roy, Alain Dervieux, Laurent Hascoët, Bruno Koobus, Mariano Vazquez, Stephen Wornom.

**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 [35]), 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 [34] 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 [39] [42] 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 [36], where computation of the adjoint state uses the iterated states in the direct order. Alternatively, most researchers (see for example [39]) 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 data assimilation or parameter estimation,
- first-order linearization of complex systems, or higher-order simulations, yielding reduced models for simulation of complex systems around a given state,
- mesh adaption 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 [44].

## 4.4. Linearization

To simulate 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 parameter. 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 adaption

It has been noticed that some approximation errors can be expressed by an adjoint state. Mesh adaption 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 [correspondant], Mauricio Araya-Polo, Benjamin Dauvergne, Christophe Massol, Valérie Pascual.

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, and therefore in 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. A documentation is available on our web page http://www-sop.inria.fr/tropics/ and as an INRIA technical report (RT-0300) http://www.inria.fr/rrrt/rt-0300.html

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 $F'^*(X).\overline{Y}$.

A 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. This could be done in cooperation with other teams.

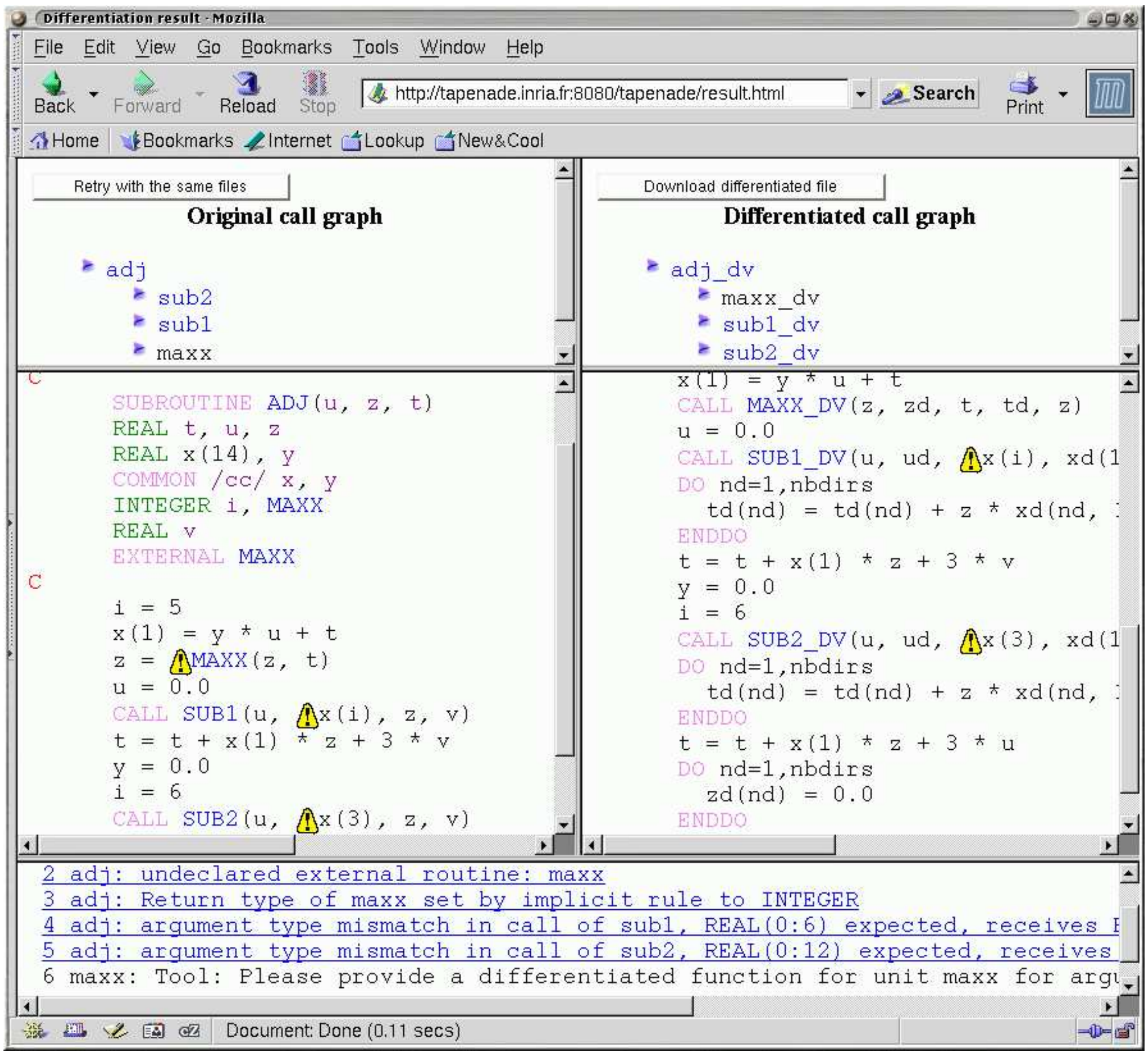
Like any program transformation tool, TAPENADE needs sophisticated static analyses in order to produce an efficient output. Concerning AD, the following analyses are a must, and TAPENADE now performs them all:

- **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.
- **Read-Write:** Each procedure has a number of arguments, that may be inputs, outputs, or both. Compilers use this to remove useless arguments. TAPENADE uses Read-Write information more specifically to detect aliasing, which is very harmful in AD, and to reduce the size of snapshots needed by checkpointing in the reverse mode. Read-Write analysis makes the differentiated program safer and less costly in memory space.
- **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 today are FORTRAN77 and FORTRAN95. Notice however that the internal representation of programs is language-independent, as shown on figure 4, so that extension to other languages should be easier. Development of the prototype of a TAPENADE for C started in december.

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. The interactive interface displays the differentiated programs, with HTML links that implement source-code correspondence, as well as correspondence between error messages and locations in the source. This is shown on figure 3.


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 R
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, or 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.

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.

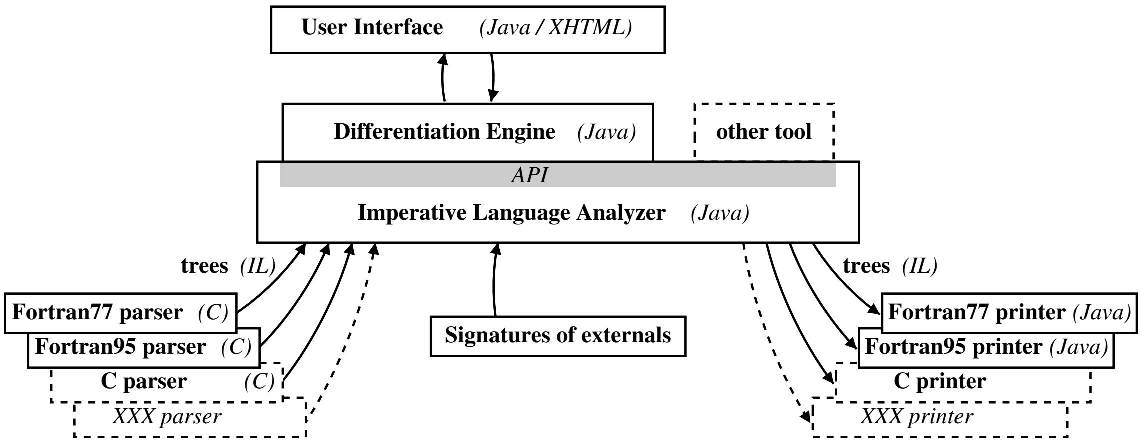


*Figure 4. Overall Architecture of* TAPENADE

The end-user can also 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.

# 6. New Results

## 6.1. Static Analyses for reverse AD

**Keywords:** *checkpointing*, *dead code*, *reverse mode of AD*, *snapshots*, *static analyses*.

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

In addition to the *activity*, and *read-write* analyses described in section 5.1, we focus on specific analyses that take advantage of adjoint programs' special structure summarized for example on figure 2. Our goal is to improve the speed and memory usage of these adjoint programs. Indeed, classical data-flow analyses performed by compilers could do the job only partly, because those cannot detect not use the mirror structure of adjoint programs i.e. a forward sweep followed by a backward sweep, with matching control decisions. Moreover, specific analyses will run faster because they operate on the original code rather than on the larger and more complex differentiated code.

This year we formalized the notions of variables that are:

- needed by the differentials of instructions that are upstream in the original program (*TBR*),
- needed by the adjoint of the instructions that are downstream in the original program (*adjoint liveness*)
- needed to re-execute a checkpointed piece of the original program, and to this end the variables that are overwritten by the adjoint of a piece of the original program (*adjoint write analysis*),

and expressed them together in a common formalization. We explicited the data-flow equations that specify the activity, adjoint liveness, adjoint write and adjoint read analyses, in the form of simple set equations. This allowed us to study their relationship and find an optimal order which is: study activity first, then study adjoint liveness, then finally compute the adjoint read, TBR, and adjoint write sets. This is part of the PhD work of Mauricio Araya-Polo.

During his student internship in spring, Benjamin Dauvergne implemented these analyses into TAPENADE and experimented on our set of validation applications, exhibiting speedups ranging from 0 to 30%. In the beginning of his PhD work, Benjamin Dauvergne investigated the application of the adjoint write analysis to build smaller snapshots for checkpointing.

The data-flow equations that specify the activity, adjoint liveness, adjoint write and adjoint read analyses still do not take checkpointing fully into account. We are currently developing a new set of data-flow equations, this time with checkpointing. This way we expect to be able to capture some hand manipulations made by experts on adjoint codes. For example, expert developers often take advantage of a series of successive checkpoints, included into a common parent checkpoint level. It is often possible to share some arrays that appear in many of the snapshots, so as to store them only once. We believe we are able to automate this improvement directly into an AD tool.

This work was presented in the Eccomas 2004 conference in Jyvaskyla, Finland, and at the AD2004 conference in Chicago, Illinois. It will also be published in the book of selected presentations from this conference.

## 6.2. Automatic estimation of the validity domain of derivatives

**Keywords:** *discontinuities*, *validity of derivatives*.

**Participants:** Mauricio Araya-Polo, Laurent Hascoët.

This is the principal topic of the PhD research of Mauricio Araya-Polo. The program generated by AD always returns derivatives, even if the mathematical function computed by the program is discontinuous or non differentiable at the current point. Discontinuities can also be introduced by the translation from the math equations to the computer program. In fact, most computer programs are only *piecewise* differentiable. When the current program execution comes very “close” to a discontinuity, then the derivatives may be invalid. Following these derivatives, for example in an optimization process, may go across a discontinuity and thus can actually degrade the solution.

Automatic resolution of this discontinuity problem is probably out of reach. However this problem hampers the confidence that users can put into AD. We think it is desirable to design differentiation modes which, if they don’t solve the discontinuity problem, at least can warn the end-user when such a discontinuity is coming close. Therefore we want to design a differentiation mode which estimates the size of the neighborhood around the current values, inside which the flow of control doesn’t change.

This year, Mauricio Araya-Polo studied different approaches from the point of view of complexity. It appears that the most reasonable aproach is a directional estimation, i.e. an estimation of the intersection of the validity neighborhood along the current direction of tangent differentiation. A more complete estimation can be achieved by repeating the analysis for each direction in the Cartesian basis of the input space.

We made an experimental implementation of this approach. The resulting code is not significantly more expensive than the standard tangent mode. On some examples taken from our validation set of applications, it actually detected discontinuitiesthat are very close to places where our end-users found optimization difficult. Therefore we feel confident that this analysis can be a valuable help to users.

## 6.3. The Recompute-or-Store alternative

**Keywords:** *recompute-all*, *reverse mode of AD*, *storage-recomputation tradeoff*, *store-all*.

**Participants:** Laurent Hascoët, Benjamin Dauvergne.

In the reverse mode of Automatic Differentiation, values must be made available to the computation of the derivatives in the reverse of their original computation order. This can be achieved either by storing intermediate values as they are computed, or by recomputing them when needed. This work is an effort to study the relative merits of these two strategies, known as *store-all*, implemented in TAPENADE and ADIFOR, and *recompute-all*, implemented in TAF. Neither strategy is optimal in terms of execution time and memory space. The optimal is probably a combination of these two strategies.

This year, we designed a framework which, for each intermediate value, offers the choice between storing or recomputing. We call it the Recompute-Or-Store Alternative (ROSA). This framework is based on the data dependence graph of the considered program. We investigated how this framework captures the previous strategies, and we proposed heuristics to find an efficient combination of recomputing and storing. We study how the ROSA framework blends with the classical checkpointing strategy which trades execution time for memory space at a coarser grain level in the program. Experiments are under way in TAPENADE

This ROSA framework was presented at the AD2004 conference.

## 6.4. Extensions and new functionalities in tapenade

**Keywords:** *aliasing*, *fortran95*, *tapenade*.

**Participants:** Valérie Pascual, Benjamin Dauvergne, Laurent Hascoët.

The work on the extension of TAPENADE to FORTRAN95 continued. The number of new constructs in FORTRAN95 is a constant challenge to find unifying representations of data and control structures of imperative programs.

This year, we extended the internal data flow analyses of TAPENADE to finely take into account records ("derived types"). Since different components of a data structure often have different behaviors, a variable of a structured type must be considered as a set of individual variables, on which the analyses may find different results. For example "activity" analysis can find out that only some components are active and thus need a derivative. Thus the derivative structured type can hold fewer components.

Benjamin Dauvergne studied in detail the common behavior of TAPENADE's data flow analyses. The common part of these analyses' strategy was embodied in a new class "DataFlowAnalyzer", from which every particular analysis is derived. The general strategy for analyses consists of iterative sweeps on the Call Graph, because of recursivity, and at the procedure level it also consists of iterative sweeps on the Flow Graph, because of control loops. An improved implementation of these sweeps was made, in which cycles in the graphs are detected and treated in a specific manner to avoid redundant analyses. As a result, the time spent on analyses during differentiation has decreased by a factor 10 on large applications.

The above improvements, plus many bug corrections, are available in the latest version 2.1 of TAPENADE.

## 6.5. Optimal control

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

**Participants:** Francois Courty, Bruno Koobus, Alain Dervieux, Laurent Hascoët, Mariano Vázquez, Bijan Mohammadi.

Now that simulation is well mastered by research groups in industry, optimization is naturally the next frontier. Optimization problems in aerodynamics are still very difficult, because they require a enormous computing power. To meet these needs, our investigations in AD are focused on the reverse mode, which is an elegant way to obtain the adjoints that optimization uses.

The reverse mode, and the subsequent adjoint state, are the best way to get the gradients when the number of parameters is large. This corresponds to what happens in industry. For example, optimizing only a dozen shape parameters will not produce an optimal shape for an aircraft, because an accurate description of a shape requires hundreds of parameters. Some shape parameters can be functions defined on a surface or a volume. Therefore the number of scalar parameters depends on the discretization chosen, and is a priori large.

Therefore, practical application of AD to control problems requires that we consider the following issues:

- efficient computation of a large scale adjoint system

- efficient optimization algorithms for large scale systems
- efficient preconditioners for this optimization.

This year is a reflexion and redaction year for this subject, see [11] [19] [24] [16].

New application on ship sail optimization have been also completed, published [14] and was the theme of a PhD thesis (Marco Michieli de Vitturi) in the partner laboratory in Pisa.

## 6.6. SQP-One-Shot optimization

**Keywords:** *One-shot*, *Sequential Quadratic Programming*, *optimization*.

**Participants:** Francois Courty, Alain Dervieux.

The class of methods that applies best to optimal control with a state equation is the Sequential Quadratic Programming. We refer for example to the monography of Nocedal and Wright [43].

SQP methods are sophisticated methods combining a lot of useful heuristics. They enjoy robustness properties due for example to Trust Region heuristics relying on powerful theory (Wolfe criteria for gradient convergence), and due to quasi Newton formulas such as BFGS. However, they are not well adapted to large scale systems such as those handled in Optimal Control loops with adjoints. Indeed, the standard SQP methods involve at each main iteration to solve several linearized state systems. This difficult point has been identified by many researchers in optimal shape design and the result is that SQP methods have been not always applied, but instead, either less modern but less complex algorithms like gradient algorithm were applied [40], or algorithms for the simultaneous solution of the KKT optimality equations were proposed [46]. The latter class of algorithm is in fact an important key for large scale optimization. However, existing one-shot algorithm are deprived of the many robustness heuristics that are involved in SQP modern algorithms. We have derived a family of one-shot-SQP algorithm devoted to the robust application of the one-shot principle:

- they solve progressively the three equations of optimality, yielding a good complexity for obtaining the final result,
- they involve some important features of SQP allowing for a quasi-black box resolution of a new problem.

Results have been described and published in [18] [19] [24].

## 6.7. Multilevel optimization and reduced models

**Keywords:** *gradient*, *multilevel*, *optimization*, *reduced models*.

**Participants:** Francois Courty, Alain Dervieux, Bruno Koobus, Mariano Vázquez.

As stated in 6.5, the very large numebr of parameters of interest for optimal control problems using the adjoint approach comes from the discretization of a functional parametrization. Not only do we have to take into account the number of parameters, but we should take some benefit from the information we can get about the functional parametrization. This can help designing an efficient functional preconditioner. In contrast to algebraic ones, “functional preconditioners” are not derived from the operator to precondition by some algebraic transformation. They are derived from an analysis of the functional context at the origin of the discrete problem. The functional preconditioner we have built for shape design application is an additive multilevel preconditioner. The multilevel basis was also applied in cooperation with project-team Smash to the derivation of reduced models that can be introduced in sophisticated optimizers. [16][19] [24].

## 6.8. Multidisciplinary optimization

**Keywords:** *fluid*, *gradient*, *optimization*, *structure*.

**Participants:** Alain Dervieux, Bruno Koobus, Mariano Vázquez, Bijan Mohammadi, Charbel Farhat [University of Colorado, Boulder], Francois Beux [Scuola Normale Superiore di Pisa], Marco Michieli de Vitturi [Scuola Normale Superiore di Pisa].

The optimization of a complex product as an airplane needs to be done by taking in account simultaneously as much as different physical effects as possible. It is useless to determine the optimal aerodynamic shape of an aircraft if the fact the the airflow will act on the structure and deform this shape is not anticipated. Many of the physics to take into account are described by complex and computer intensive models. The team have proposed a new coupling algorithm for the aerodynamic shape optimization of an aircraft geometry deformed by the wind, [19] [24].

## 6.9. Mesh adaptation

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

**Participants:** Alain Dervieux, Francois Courty, Tristan Roy.

The team continued his reflexion on the use of smart mesh adaptation to increase the convergence order of a series of adapted computation [15]. The innovative derivation of the adjoint and the resolution of the related optimum problem can be used in a slightly different context than 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 continuous property. 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 rather complete theory demonstrating that second order accuracy can be obtained on discontinuous field approximation [12].

In the case of adaptation for Partial Differential Equations, an Optimal Control is obtained. It involves a state equation and the optimality is expressed in terms of an adjoint state that can be derived by AD. In our first prototypes, the one-shot-SQP algorithm has been applied successfully, [18][19] [24]. This will be the focus of a cooperation with project-team GAMMA at INRIA-Rocquencourt (Paul-Louis George, Frédéric Alauzet) for the INRIA contribution to the HISAC IP European project in assocoation with 30 other partners in aeronautics. A thesis will start next year in cooperation with the SMASH project-team on 3D anisotropic mesh adaption.

# 7. Dissemination

## 7.1. Links with Industry, Contracts

Again this year, the number of connections to the TAPENADE web server has increased to more than one hundred. The most regular users of TAPENADE have subscribed to our “tapenade-users” mailing list, now registering 29 users.

Our collaboration with Mike Giles (Oxford University) and Rolls-Royce continues, with a regular use of TAPENADE by Mike Giles’ team of researchers on the Rolls-Royce HYDRA code.

INRIA supports the industrial development of TAPENADE by funding our engineer Christophe Massol. His main task is to develop an efficient pointer analysis and differentiation of programs with pointers and dynamic memory allocation. In addition, his first work was to port TAPENADE to WINDOWS-XP platforms.

TROPICS is leader of a project “Optimisation de forme et adaptation de maillage pour le bang supersonique” supported by the Comité d’Orientation Supersonique of the French ministry of Research. Our partners are the university of Montpellier and the Gamma project in Rocquencourt.

TROPICS contributed to the european project HISAC, as the main contributor for one package. The HISAC project was accepted in october.

We established links with EDF, former users of the previous AD tool ODYSSEE. We presented the new developments in TAPENADE during a meeting in november, showing that the important limitations of ODYSSEE have been lifted and now make it possible to differentiate very large industrial code in a reasonable time. We shall give a course on Automatic Differentiation during the EDF-CEA-EADS-INRIA “école d’été” on uncertainties in the summer of 2005.

## 7.2. Conferences and workshops

- Alain Dervieux gave a lecture in june at the PROMUVAL’04 short course in Barcelona, Spain.
- The team presented three papers in july, at the AD2004 conference on Automatic Differentiation in Chicago Illinois. Laurent Hascoët presented the work with Mauricio Araya-Polo on “The adjoint Data-Flow analyses: formalization, properties, and applications”, and his work on “The Recompute-Or-Store Alternative in reverse Automatic Differentiation”. Valérie Pascual presented her work on “Extension of Tapenade towards Fortran9x”.
- Laurent Hascoët presented the team’s results in july, at the ECCOMAS’04 conference in Jyvaskyla, Finland. One presentation was on the “Data Flow Algorithms in the Tapenade tool for Automatic Differentiation”, and another presentation focused more on TAPENADE, entitled: “Tapenade: a tool for Automatic Differentiation of programs”.
- Laurent Hascoët was on the PhD jury of Vincent Fischer, of ENSIEG Grenoble, France, who used Automatic Differentiation for optimization applications in electrical engineering.
- Alain Dervieux made a presentation in november at the PROMUVAL’04 conference in Athens, Greece.
- Laurent Hascoët attended the november AD workshop in Hatfield UK, and was an organizer of the special session on FORTRAN95.
- Laurent Hascoët gave an afternoon lecture and demonstration on Automatic Differentiation, during the CARI’04 African conference on computer science in Hammamet, Tunisia, in november.

# 8. Bibliography

### Major publications by the team in recent years

[1] G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann. *Automatic Differentiation of Algorithms, from Simulation to Optimization*, LNCSE, Springer, 2001.

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

[3] 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.

[4] A. Dervieux, F. Courty, M. Vázquez, B. Koobus. *Additive multilevel optimization and its application to sonic boom reduction*, in "Proceedings of Conference JP60, Jyvaskyla, 12-15 juin 2002", 2003.

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

[6] L. HASCOET, S. FIDANOVA, C. HELD. *Adjoining Independent Computations*, in "Automatic Differentiation of Algorithms, from Simulation to Optimization", Springer, LNCSE, 2001, p. 299-304.

[7] L. HASCOËT. *A method for automatic placement of communications in SPMD parallelisation*, in "Parallel Computing Journal", n° 27, 2001, p. 1655-1664.

[8] L. HASCOËT. *The Data-Dependence Graph of Adjoint Programs*, research report, n° 4167, INRIA, 2001, http://www.inria.fr/rrrt/rr-4167.html.

[9] L. HASCOËT. *Automatic Placement of Communications in Mesh-Partitioning Parallelization*, in "ACM SIGPLAN Notices", Proceedings of 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, vol. 32, n° 7, 1997, p. 136-144.

[10] L. HASCOËT, M. VAZQUEZ, A. DERVIEUX. *Automatic Differentiation for Optimum Design, applied to Sonic Boom reduction*, in "Proceedings of the International Conference on Computational Science and its Applications, ICCSA'03, Montreal, Canada", V.KUMAR ET AL. (editor)., LNCS 2668, Springer, 2003, p. 85-94.

## Articles in referred journals and book chapters

[11] F. COURTY, A. DERVIEUX. *A SQP-like one-shot algorithm for optimal shape design*, V. S. ED. (editor)., to appear, Springer, 2004.

[12] F. COURTY, D. LESERVOISIER, P.-L. GEORGE, A. DERVIEUX. *Continuous metrics and mesh optimization*, in "Applied Numerical Mathematics", to appear, 2004.

[13] L. HASCOËT, U. NAUMANN, V. PASCUAL. *"To be recorded" analysis in reverse-mode automatic differentiation*, in "Future Generation Computer Systems", to appear, 2004.

[14] M. MICHIELI DE VITTURI, F. BEUX, G. LOMBARDI, A. DERVIEUX. *Optimum shape design for turbulent viscous flows around complete configurations of 2D flying sails*, in "Journal of Computational Methods in Sciences and Engineering", vol. 4, n° 1-2, 2004, p. 43-55.

[15] E. SCHALL, D. LESERVOISIER, A. DERVIEUX, B. KOOBUS. *Mesh adaptation as a tool for certified computational aerodynamics*, in "International Journal for Numerical Methods in Fluids", vol. 45, 2004, p. 179-196.

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

## Publications in Conferences and Workshops

[17] M. ARAYA-POLO, L. HASCOËT. *Data Flow Algorithms in the Tapenade tool for Automatic Differentiation*, in "Proceedings of 4th European Congress on Computational Methods, ECCOMAS'2004, Jyvaskyla, Finland", 2004.

[18] F. COURTY, T. ROY, A. DERVIEUX. *Continuous Optimal Control approach and mesh adaptation*, in

"Proceedings of 4th European Congress on Computational Methods, ECCOMAS'2004, Jyvaskyla, Finland, Jyvaskyla, july 24-28", 2004.

[19] A. DERVIEUX, F. COURTY, T. ROY, M. VÁZQUEZ, B. KOOBUS. *Optimization loops for shape and error control*, in "PROMUVAL Short Course on Multidisciplinary Modelling, Simulation and Validation in Aeronautics, Barcelona, june 28-29", CIMNE, 2004.

[20] L. HASCOËT, M. ARAYA-POLO. *The adjoint Data-Flow analyses: formalization, properties, and applications*, in "Proceedings the AD 2004 conference, Chicago, Illinois", July 2004.

[21] L. HASCOËT. *The Recompute-Or-Store Alternative in reverse Automatic Differentiation*, in "Proceedings the AD 2004 conference, Chicago, Illinois", July 2004.

[22] L. HASCOËT. TAPENADE*: a tool for Automatic Differentiation of programs*, in "Proceedings of 4th European Congress on Computational Methods, ECCOMAS'2004, Jyvaskyla, Finland", 2004.

[23] V. PASCUAL, L. HASCOËT. *Extension of Tapenade towards Fortran9x*, in "Proceedings the AD 2004 conference, Chicago, Illinois", July 2004.

## Internal Reports

[24] A. DERVIEUX, F. COURTY, T. ROY, M. VÁZQUEZ, B. KOOBUS. *Optimization loops for shape and error control: extended lecture notes*, Research report, n° 5413, INRIA, 2004, http://www.inria.fr/rrrt/rr-5413.html.

[25] L. HASCOËT, V. PASCUAL. TAPENADE *2.1 user's guide*, Technical report, n° 300, INRIA, 2004, http://www.inria.fr/rrrt/rt-0300.html.

## Bibliography in notes

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

[27] I. ATTALI, V. PASCUAL, C. ROUDET. *A language and an integrated environment for program transformations*, research report, n° 3313, INRIA, 1997, http://www.inria.fr/rrrt/rr-3313.html.

[28] A. CARLE, M. FAGAN. *ADIFOR 3.0 overview*, Technical report, n° CAAM-TR-00-02, Rice University, 2000.

[29] 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://www.inria.fr/rrrt/rr-0416.html.

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

[31] P. COUSOT. *Abstract Interpretation*, in "ACM Computing Surveys", vol. 28, n° 1, 1996, p. 324-328.

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

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

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

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

[36] 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.

[37] A. GRIEWANK. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, SIAM, 2000.

[38] L. HASCOËT. *Transformations automatiques de specifications semantiques: application: Un verificateur de types incremental*, Ph. D. Thesis, Université de Nice Sophia-Antipolis, 1987.

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

[40] A. JAMESON. *Aerodynamic design via control theory*, Report, n° 1824 MAE, Princeton University, New Jersey, 1988.

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

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

[43] J. NOCEDAL, S. WRIGHT. *Numerical Optimization*, Springer Series in Operations Research, 1999.

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

[45] 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.

[46] S. TA'ASAN, G. KURUVILA, M. SALAS. *Aerodynamic design and optimization in one shot*, in "30th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, AIAA Paper 91-0025", 1992.