INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team tropics

Transformations et Outils Informatiques pour le Calcul Scientifique

Sophia Antipolis

THEME 1A

2003

Table of contents

1. Team **1**
2. Overall Objectives **1**
3. Scientific Foundations **2**
3.1. Automatic Differentiation 2
3.2. Static Analyses and Transformation of programs 4
3.3. Automatic Differentiation and Computational Fluid Dynamics 6
4. Application Domains **7**
4.1. Panorama 7
4.2. Multidisciplinary optimization 7
4.3. Inverse problems 7
4.4. Linearization 8
4.5. Mesh adaption 8
5. Software **8**
5.1. TAPENADE 8
6. New Results **9**
6.1. New Static Analyses for AD 9
6.2. Automatic estimation of the validity domain of derivatives 11
6.3. Common sub-expressions in derivatives 12
6.4. Multi-directional differentiation 12
6.5. Extensions and new functionalities in tapenade 13
6.6. Adjoint-based optimal control 14
6.7. SQP-One-Shot optimization 15
6.8. Multilevel optimization and reduced models 16
6.9. Multidisciplinary optimization 16
6.10. Mesh adaptation 16
9. Dissemination **16**
9.1. Links with Industry, Contracts 16
9.2. Conferences and workshops 17
10. Bibliography **18**

1. Team

Head of project team
Laurent Hascoët

Vice-head of project team
Valérie Pascual

Administrative assistant
Nathalie Balax-Bellesso

Staff members
Alain Dervieux
Rose-Marie Greborio [until may 31]

Ph. D. students
François Courty
Mauricio Araya-Polo

Partner scientists
Bruno Koobus [University of Montpellier 2]
Mariano Vazquez [GridSystems, Mayorca, Spain]

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 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

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

Participants: Mauricio Araya-Polo, Rose-Marie Greborio, Laurent Hascoët, Valérie Pascual.

Glossary

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:

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

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], inverse problems, or adjoint problems [31], 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 [33]. The computation time required for the gradient is only a small multiple of the run time of P, multiplied by the number of outputs m, which is usually small for optimization. It is independent from the number of parameters n.

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 [29] 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 [24] 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

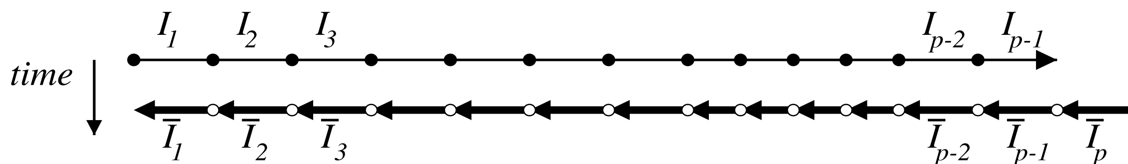

Figure 1. The "Store-All" tactic

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 **p** of the program is illustrated on figure 2. During the forward sweep, no value is stored while in **p**. Later, when the backward sweep needs values from **p**, 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 **p**. This snapshot is shown on figure 2 by slightly bigger black and white dots.

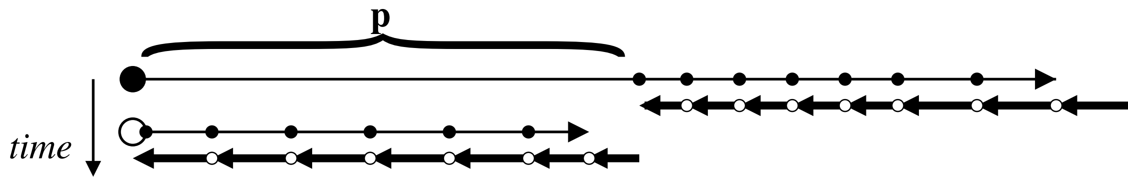

*Figure 2. Checkpointing **p** with the "Store-All" tactic*

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

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

Participants: Mauricio Araya-Polo, Rose-Marie Greborio, Laurent Hascoët, Valérie Pascual.

Glossary

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 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 [22] 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 [25][35][23]. 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 [26], 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* [27] 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 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 status of some individual element of an array. Static and dynamic *array region analyses* [42][28] 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

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

Participants: François Courty, Alain Dervieux, Laurent Hascoët, Bruno Koobus, Mariano Vazquez.

Glossary

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 [31]), 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 [30] 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. Can we avoid this using the fact that the model is steady? Actually this is possible, as shown in [32], 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. We will detail some of them in the next sections:

- first-order linearization of complex systems,
- reduced models for simulation of complex systems around a given state,
- mesh adaption with gradients or adjoints,
- sensitivity analysis,
- propagation of truncation errors,
- equation solving with the Newton method,
- inverse problems, such as data assimilation or parameter estimation,
- optimum shape design, under constraints, and more generally any algorithm based on local linearization

These applications require an AD tool that differentiates programs written in classical imperative languages, FORTRAN77, FORTRAN9X, 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 [41].

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 series, 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, Rose-Marie Greborio, 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/.

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.

- **In-Out:** Each procedure has a number of arguments, that may be inputs, outputs, or both. Compilers use this to remove useless arguments. TAPENADE uses In-Out 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. In-Out analysis makes the differentiated program safer and less costly in memory space.
- **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 FORTRAN9X. 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. The first prototype of a TAPENADE for C is planned for next year.

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 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

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.

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. New Static Analyses for AD

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

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

In addition to the *activity, in-out*, and *TBR* analyses described in section 5.1, we are investigating other analyses that could improve further the reverse code produced by TAPENADE. In particular, experiments by Paul Cusdin, at Queen's University in Belfast, showed that TAPENADE could take advantage of detecting instructions in the forward sweep which are not needed by the reverse sweep, and only contribute to compute the original program's output. Since this output is generally not wanted, these instructions can be considered dead code,

```
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
Original call graph                       Differentiated call graph
  adj                                       adj_dv
    sub2                                      maxx_dv
    sub1                                      sub1_dv
    maxx                                      sub2_dv

C                                          x(1) = y * u + t
      SUBROUTINE ADJ(u, z, t)              CALL MAXX_DV(z, zd, t, td, z)
      REAL t, u, z                         u = 0.0
      REAL x(14), y                        CALL SUB1_DV(u, ud, x(i), xd(1
      COMMON /cc/ x, y                     DO nd=1,nbdirs
      INTEGER i, MAXX                        td(nd) = td(nd) + z * xd(nd,
      REAL v                               ENDDO
      EXTERNAL MAXX                        t = t + x(1) * z + 3 * v
C                                          y = 0.0
      i = 5                                i = 6
      x(1) = y * u + t                     CALL SUB2_DV(u, ud, x(3), xd(1
      z = MAXX(z, t)                       DO nd=1,nbdirs
      u = 0.0                                td(nd) = td(nd) + z * xd(nd,
      CALL SUB1(u, x(i), z, v)             ENDDO
      t = t + x(1) * z + 3 * v             t = t + x(1) * z + 3 * u
      y = 0.0                              DO nd=1,nbdirs
      i = 6                                  zd(nd) = 0.0
      CALL SUB2(u, x(3), z, v)             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

Document: Done (0.11 secs)
```

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

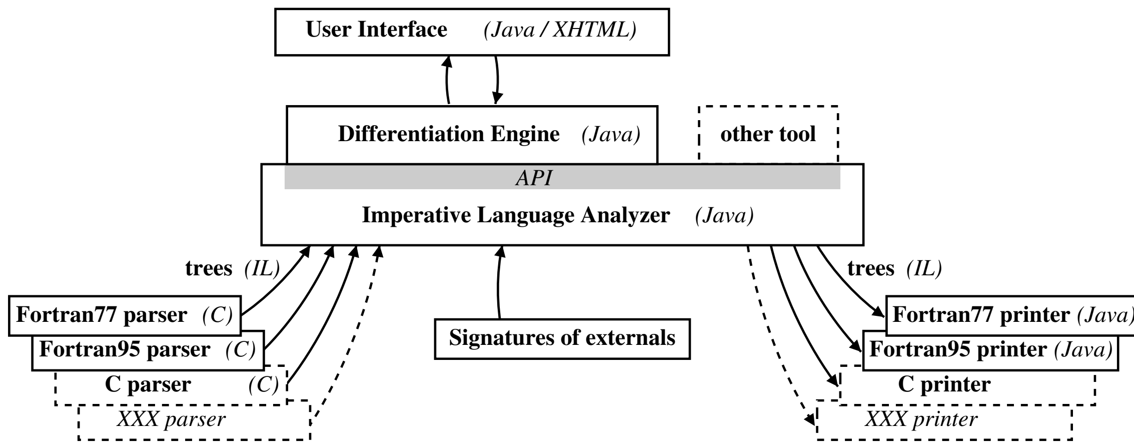

Figure 4. Overall Architecture of TAPENADE

and therefore removed. Experiments made by Cusdin with TAPENADE on one particular application showed a potential speed up of 40%.

This year, we began to study the static analysis that will detect this sort of dead code. It is an extension of the classical liveness analysis. Indeed, a classical liveness analysis can in theory do the job. However, differentiated programs are more than twice as large as their original program, and the analysis would take time. More importantly, we consider it a poor strategy because this would not take into account specificities of reverse differentiated code. We believe that a better strategy is to define a specific liveness analysis that detects dead code in the reverse program by simply looking at the original program. We are currently specifying the data-flow equations for this analysis.

It is interesting to notice that these data-flow equations involve information from the *TBR* and *activity* analysis. In fact these analyses are linked. This is also promising because simplified reverse code often does not use all of the original procedure arguments. Since the snapshot which is taken to checkpoint a procedure (*cf* figure 2) is a subset of the procedure's arguments, this new analysis will lead to smaller snapshots, and thus will improve memory consumption as well as run time.

More generally, if we call `P` a procedure and $\overline{\mathtt{P}}$ its differentiated procedure in the reverse mode, we can show that the In (resp. Out) sets of the variables needed (resp. overwritten) by `P` are such that:

$$In(\overline{\mathtt{P}}) \subset \mathtt{In(P)} \quad \text{and} \quad \mathtt{Out}(\overline{\mathtt{P}}) \subset \mathtt{Out(P)}$$

This comes from the above dead code detection, as well as from the stack mechanism that is used to store intermediate values. This can help us define even smaller snapshots when checkpointing procedure `P`. We started specification of the static analyses that yield these sets. This is part of the PhD work of Mauricio Araya-Polo.

6.2. Automatic estimation of the validity domain of derivatives

Key words: *discontinuities*, *validity of derivatives*.

Participants: Mauricio Araya-Polo, Laurent Hascoët.

This is the principal topic of the PhD research of Mauricio Araya-Polo, which started this year. 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 they can warn the end-user when such a discontinuity is coming close.

More precisely, for any particular run of a differentiated program, we want to estimate which tests came too close to their breaking point, and we want to summarize this information in terms of the program’s inputs (resp. outputs). Thus, along with the returned derivative, we could give a neighborhood of the inputs (resp. outputs) inside which no discontinuity occurs and the derivative is valid.

This probably requires a special mode of AD, close to the reverse mode. A first theoretical analysis shows this new mode may have a high computational cost, unless tradeoffs can be found. We shall experiment such tradeoffs in the years to come, between execution cost and precision of the returned neighborhood of validity.

6.3. Common sub-expressions in derivatives

Key words: *common sub-expression elimination*, *reverse mode of AD*.

Participant: Laurent Hascoët.

Naive differentiation of large expressions often give birth to even larger expressions, containing many copies of parts of the original expression. Consider the product $a.b.c.d$. Its derivatives are $a.b.c$, $a.b.d$, $a.c.d$ and $b.c.d$, which share a number of common sub-expressions. Detection of common sub-expression is a classical activity in optimizing compilers. Unfortunately its complexity is NP. When AD is concerned, many previous works [34] have considered the so-called *computation graph* of a differentiated program, and looked for the optimal order of atomic operations to minimize the computational cost. Again this is NP-hard, and in our opinion, does not take into account the way these derivative expressions are built.

Our goal is not to improve the original expression. We can freely consider that it is sort of optimal, or at least the user is satisfied with it. What we want to minimize is the complexity of the new operations *introduced* by differentiation. Put that way, the problem can be solved cheaper. Moreover, in real situations, it is not necessary to find the really best sub-expressions splitting, and a good approximate solution is probably just as good. We propose a heuristic whose complexity is about n^2, where n is the size of the original expression. We have no theoretical figures yet about the cost of the differentiated expressions, but experiments give a 10% speedup on industrial codes, and far more than that on specific examples with large expressions.

We implemented this algorithm inside TAPENADE. Currently, it is available only for the reverse mode, but should be extended to the other modes soon.

6.4. Multi-directional differentiation

Key words: *tangent mode of AD*, *Jacobian matrix*, *loop fusion*.

Participants: Laurent Hascoët, Bertrand Iooss [CEA Cadarache, France].

Last year we have been developing the multi-directional tangent mode, known as the *tangent vector* mode, in TAPENADE. This year we experimented this mode on an industrial code from the CEA in Cadarache. The code is a part of a larger code named MARGARET, and simulates the behavior of bars of nuclear fuel. The in-hose optimization algorithm requires a Jacobian matrix. Three paths were available to get this Jacobian: divided differences, tangent differentiation, and tangent vector differentiation.

Divided differences evaluates

$$\frac{\mathtt{P}(X + \varepsilon.dX) - \mathtt{P}(X)}{\varepsilon}$$

for a “well chosen” ε. If we forget about the poor precision of this approximation, it return us a Jacobian with M rows at the cost of $M+1$ executions of P, and therefore the time needed is

$$T_{dd} = (M+1)T(\mathtt{P}).$$

Tangent differentiation returns one row of the Jacobian matrix, analytically. However, the run time of the tangent program $\mathtt{P}'$ is generally more than twice the run time of P, i.e. $T(\mathtt{P}') = T(\mathtt{P}) + n.T(\mathtt{P})$, where n is a small factor. Therefore the time needed is

$$T_{td} = M.T(\mathtt{P}) + M.n.T(\mathtt{P}).$$

Tangent Vector differentiation is equivalent to running many times the tangent program at the same point for different directions. The part that computes the original function is done only once, and therefore the time needed is

$$T_{tv} = T(\mathtt{P}) + M.n.T(\mathtt{P}).$$

The experiments on MARGARET gave the following figures: $T_{dd} = 25s$, $T_{td} = 40s$ for $M = 34$. This gives us an estimation of $n = 0.65$ for the ratio between the computation time of the derivatives compared to the computation time of the original function. Besides, this shows that tangent differentiation is, unfortunately, slower than divided differences. Yet, it returns more precise derivatives! From this estimation, we would expect that $T_{tv} = 16.5$. The measured result is not as good, giving $T_{tv} = 19.8$, probably due to loop overhead. In any case $T_{tv} < T_{dd}$ which means that AD does better than divided differences! Further improvements of the vector mode could yield even better results, and we plan to study them in the years to come.

6.5. Extensions and new functionalities in tapenade

Key words: *aliasing*, *fortran95*, *tapenade*.

Participant: Mauricio Araya-Polo, Rose-Marie Greborio, Laurent Hascoët, Valérie Pascual.

TAPENADE now accepts the modular structure of FORTRAN9X. In particular `module`, `interface`, `use` statements are taken into account and differentiated correctly. The internal representation has been extended to accept modular files, and is now closer to accepting classes. This is an important step towards differentiation of object-oriented programs. Nested symbol tables can now handle `public` and `private` declarations, and multiple inheritance due for example to the `use` declaration.

Some features of FORTRAN had been disregarded in the alpha version of TAPENADE. Some real applications have required that we treat them correctly. For example function names can be passed as arguments, and some codes use this extensively. TAPENADE now handles this and differentiates this correctly.

The problem of aliasing has been treated. Aliasing occurs when two apparently different memory references sometimes refer to the same memory location. This occurs for instance when a procedure is called with the same actual argument given to two different formal arguments. This is forbidden by most language standards, but compilers don’t check it. As could be expected then, this “technique” is widely used in real programs. TAPENADE now detects potential aliasing at each procedure call, and prints a warning message. Aliasing can occur in a single assignment. The following table shows an example where the two variables present in an assignment are *certainly* the same, *certainly* different, or *possibly* the same. In the third case, TAPENADE inserts a temporary variable, otherwise the reverse program would be wrong.

The Store All strategy of TAPENADE requires that intermediate variables be stored on a stack, and retrieved from this stack. The size of this stack is not known statically. Therefore it is implemented in C to perform dynamic allocation. Surprisingly enough, most compilers generate inefficient code when FORTRAN calls C. To

original instruction	reverse mode differentiation
a = 3.2*b + 1.5	$\overline{b} = \overline{b} + 3.2*\overline{a}$ $\overline{a} = 0.0$
a = 3.2*a + 1.5	$\overline{a} = 3.2*\overline{a}$
a(i) = 3.2*a(j) + 1.5	$\overline{tmp} = 3.2*\overline{a}(i)$ $\overline{a}(i) = 0.0$ $\overline{a}(j) = \overline{a}(j) + \overline{tmp}$

Figure 5.

cope with that, we modified TAPENADE's stack mechanism: we introduce a stack buffer in FORTRAN, which is then flushed to the C stack when full. The actual calls from FORTRAN to C are thus reduced to a minimum.

The above improvements, plus many bug corrections, are available in the new version 2.0.7 of TAPENADE.

6.6. Adjoint-based optimal control

Key words: *optimum design*, *optimal control*, *gradient*, *adjoint model*.

Participants: François 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, the optimization of a dozen of real parameters will not produce an optimal shape for an aircraft, because an accurate description of a shape requires hundreds of parameters. The type of parameter can be a function defined on a surface or a volume. Therefore the number of 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 scales systems
- efficient preconditioners for this optimization.

This year, we concluded the work done in the previous years about differentiation of loops with independent iterations (*II*-loops). We published an article about differentiation of *II*-loops in an international journal [3], and we presented an application to a full-size example in [10]. Figure 6 shows the principal result for this application, which is the gradient of the sonic boom under a plane, with respect to the shape of the skin of a supersonic aircraft. Darker colors indicate places where modifying the shape has a maximal impact on the reduction of the sonic boom on the ground.

The method was demonstrated, together with TAPENADE inside the European project AEROSHAPE, that reached its final phase in 2003. Globally, this improvement to the reverse mode of AD led us to develop a set of new algorithms, especially in program static analysis and in optimization methods for steady-state flow solvers.

We provided an efficient demonstrator, ALYA, for the French program Comité d'Orientation Supersonique. ALYA has been the support or experimental platform of a set of necessary innovation in disciplines that are complementary to AD in the context of application of adjoints to optimal control.

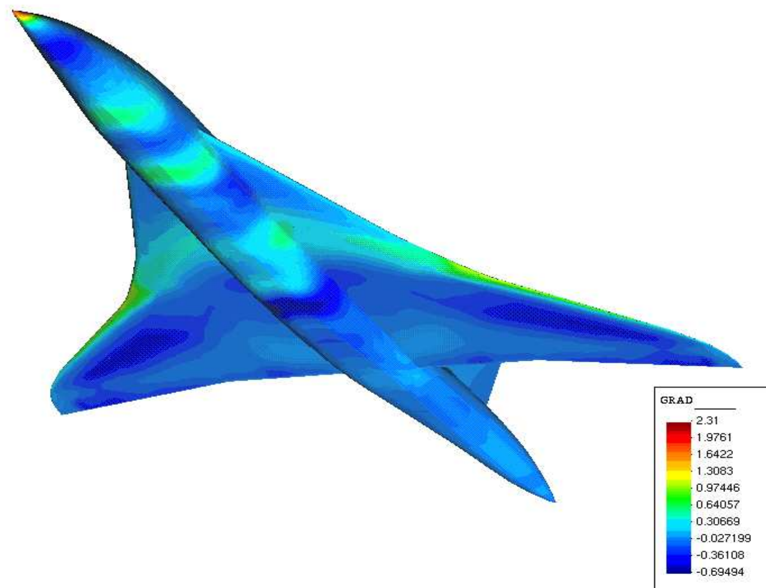

Figure 6. Gradient of the “Sonic Boom” cost functional on the skin

6.7. SQP-One-Shot optimization

Key words: *optimization*, *Sequential Quadratic Programming*, *One-shot*.

Participants: François 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 [40].

SQP methods are sophisticated ones 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 [37], or algorithms for the simultaneous solution of the KKT optimality equations were proposed [43]. 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 then 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.

This new method has been presented in the main meetings of the European Project “AEROSHAPE” and is the center of an article submitted in an international journal.

6.8. Multilevel optimization and reduced models

Key words: *optimization*, *multilevel*, *gradient*, *reduced models*.

Participants: François Courty, Alain Dervieux, Bruno Koobus, Mariano Vázquez.

As stated already, the optimal control problems of interest for an adjoint approach, the (large) number of parameter is derived from the discretization of a functional parametrization. Not only we have to take into account the number of parameters, but only we should take some benefit from the information we can get concerning the functional parametrization. This can in particular exploited for designing an efficient functional preconditioner. One word about "functional preconditioners". In contrast to algebraic ones, they 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. An article on this research has been submitted for publication. The multilevel basis is also applied to the derivation of reduced models that can be introduced in sophisticated optimizers [18].

6.9. Multidisciplinary optimization

Key words: *optimization*, *gradient*, *fluid*, *structure*.

Participants: Alain Dervieux, Bruno Koobus, Mariano Vázquez, Bijan Mohammadi, Charbel Farhat [University of Colorado, Boulder].

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. Article submitted for publication.

6.10. Mesh adaptation

Key words: *optimization*, *mesh adaptation*, *adjoint*.

Participants: Alain Dervieux, François Courty.

The innovative derivation of the adjoint and the resolution of the related optimum problem can be used in a slightly different context of the shape design, the mesh adaptation. This will be possible if we can map the mesh adaptation problem into a differentiable optimal control problem. We have introduced to do this 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. An article on this work has been submitted in a journal.
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. An article on this work has been submitted in a journal.

9. Dissemination

9.1. Links with Industry, Contracts

The TAPENADE web server has received connections from more than one hundred users this year. A few of them have then started to use TAPENADE on a regular basis. This year, two license contracts have been

signed between INRIA and industrial partners, concerning the use of TAPENADE: one with Rolls-Royce and the University of Oxford, on the “HYDRA” code, the other with Airbus-UK and Cranfield University, on the “FLITE3D” code. Also the French Commissariat à l’Energie Atomique (CEA) has successfully used TAPENADE in vector mode on their “MARGARET” code.

The European project “Aeroshape”, coordinated by V. Selmin, Alenia (I), and finished this year, was dedicated to the elaboration of methods for the shape design of aircrafts, to the building of demonstrators, and to comparison between them on test cases. In the Aeroshape consortium, the Tropics Team has contributed by proposing AD tools, by working for their demonstration, from differentiation to complete gradient assembly, and proposing the new one-shot-SQP algorithm. See in particular the two book chapters [19], [16].

The “Plan d’Etudes Amont” of Dassault-Aviation involved several important actions for the optimal control of flows. The Tropics team has contributed to a new class of reduced models relying on hierarchical basis and first-order derivatives. A second subject concentrated on the use of these bases in preconditioners.

The team was leader of a project “Optimisation d’un avion souple” supported by the Comité d’Orientation Supersonique of the French ministry of Research. Our partner is the university of Montpellier. During this project we have developed the 3D prototype ALYA, introduced the new multilevel preconditioner, applied the new methods to shape design and to the coupled fluid-structure shape design. Three articles reporting on this work are submitted for publication.

TROPICS is leader of a new 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 partner is the university of Montpellier and the Gamma project in Rocquencourt.

9.2. Conferences and workshops

- Laurent Hascoët presented TAPENADE at several seminars and conferences:
  on January 16th at the “Journées INRIA-industrie” in Rocquencourt, France;
  on April 2nd at the “ERCOFTAC Introductory Course to Design Optimization” in Munich, Germany;
  on May 15th, invited by Paul Hovland’s team at Argonne National Lab., Illinois;
  on June 5th at the Automatic Differentiation workshop in Cranfield University, UK;
- Laurent Hascoët presented the team’s results on the sonic boom reduction problem at several conferences and workshops:
  on February 11th at the AD workshop of the AMAM’03 conference in Nice, France;
  on May 19th at the AD minisymposium of the ICCSA’03 conference in Montreal, Canada;
  on June 6th at the Automatic Differentiation workshop in Cranfield University, UK;
  on September 17th at the “Deutsche Mathematiker-Vereinigung” in Rostock, Germany;
- TAPENADE was used as an example AD tool in several “hands-on” sessions and tutorials on Automatic Differentiation organized by Uwe Naumann, and in a course on adjoints given by Olivier Pironneau in Japan.
- Alain Dervieux presented our works on adjoint-based optimum design and adjoint-based mesh adaptation:
  on February 12th at the optimum design minisymposium of the AMAM’03 conference in Nice, France;
  on March 26th in an invited seminar in Ecole Normale Supérieure in Cachan, France;
  on April 23rd during an invited stay at University of Prague, Tchequia;
  on July 9th in an invited conference at the “colloque optimisation de forme” in Marseille-Luminy, France;
  on November 27th in an invited conference at the “Espace Jacques-Louis Lions”, in Saint-Cloud, France;

10. 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.* series 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 », number 5, volume 18, 2003, pages 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.* series 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, pages 299-304, 2001.

[7] L. HASCOËT. *A method for automatic placement of communications in SPMD parallelisation.* in « Parallel Computing Journal », number 27, 2001, pages 1655-1664.

[8] L. HASCOËT. *The Data-Dependence Graph of Adjoint Programs.* research report, number 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 », number 7, volume 32, 1997, pages 136-144, Proceedings of 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[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 », LNCS 2668, Springer, V.KUMAR ET AL., editor, pages 85-94, 2003.

Articles in referred journals and book chapters

[11] A. DERVIEUX, B. KOOBUS, E. SCHALL, R. LARDAT, C. FARHAT. *Application of unsteady fluid-structure methods to problems in aeronautics and space.* in « Notes in Num. Fluid Dynamics », 2003, pages 57-70.

[12] A. DERVIEUX, D. LESERVOISIER, P.-L. GEORGE, Y. COUDIÈRE. *About theoretical and practical impact of mesh adaptations on approximation of functions and of solution of PDE.* in « Int. J. Numer. Meth. Fluids », number 43, 2003, pages 507-516, invited conference at ECCOMAS-Swansea.

[13] L. HASCOËT, R.-M. GREBORIO, V. PASCUAL. *Computing Adjoints by Automatic Differentiation with TAPENADE.* in « Ecole INRIA-CEA-EDF "Problèmes non-linéaires appliqués" », 2003, to appear.

[14] L. HASCOËT, U. NAUMANN, V. PASCUAL. *TBR Analysis in Reverse Mode Automatic Differentiation.* in « Future Generation Computer Systems », 2003, to appear.

[15] E. SCHALL, D. LESERVOISIER, A. DERVIEUX, B. KOOBUS. *Mesh adaptation as a tool for certified computational aerodynamics.* in « Int. J. Numer. Meth. Fluids », 2003.

Publications in Conferences and Workshops

[16] F. COURTY, A. DERVIEUX. *A SQP-like one shot algorithm for Optimal Shape Design.* in « Aeroshape book », Springer, Notes in Num. Fluid Mechanics, SELMIN, V. ET AL., editor, 2003, to appear.

[17] A. DERVIEUX, C. FARHAT, R. CARPENTIER, B. KOOBUS, M. VAZQUEZ, E. SCHALL. *Numerical models for computing unsteady fast flows and their interaction with structures.* in « West-East High Speed Flow Fields, Marseilles, France », CIMNE, 2003.

[18] A. DERVIEUX, M. VAZQUEZ. *Grid computing in genetic algorithms: a POD-based scheme for aerodynamic optimization in supersonic flows.* in « International Congress on Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems », CIMNE, 2003.

[19] L. HASCOËT, V. PASCUAL, A. DERVIEUX. *Automatic Differentiation with TAPENADE.* in « Aeroshape book », Springer, Notes in Num. Fluid Mechanics, SELMIN, V. ET AL., editor, 2003, to appear.

Internal Reports

[20] M. VAZQUEZ, A. DERVIEUX, B. KOOBUS. *Aeroelastic coupling in sonic boom optimisation of a supersonic aircraft.* research report, number RR-4865, INRIA, 2003, http://www.inria.fr/rrrt/rr-4865.html.

[21] M. VAZQUEZ, B. KOOBUS, A. DERVIEUX, C. FARHAT. *Spatial discretization issues for the energy conservation in compressible flow problems on moving grids.* research report, number RR-4742, INRIA, 2003, http://www.inria.fr/rrrt/rr-4742.html.

Bibliography in notes

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

[23] I. ATTALI, V. PASCUAL, C. ROUDET. *A language and an integrated environment for program transformations.* research report, number 3313, INRIA, 1997, http://www.inria.fr/RRRT/RR-3313.html.

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

[25] 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 », 1986, pages 49-89, http://www.inria.fr/rrrt/rr-0416.html, Also, Information Processing Society of Japan, Technical Memorandum

PL-86-6. Also INRIA research report # 416.

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

[27] P. COUSOT. *Abstract Interpretation.* in « ACM Computing Surveys », number 1, volume 28, 1996, pages 324-328.

[28] B. CREUSILLET, F. IRIGOIN. *Interprocedural Array Region Analyses.* in « International Journal of Parallel Programming », number 6, volume 24, 1996, pages 513–546.

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

[30] J. GILBERT. *Automatic differentiation and iterative processes.* in « Optimization Methods and Software », volume 1, 1992, pages 13–21.

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

[32] A. GRIEWANK, C. FAURE. *Reduced Gradients and Hessians from Fixed Point Iteration for State Equations.* in « Numerical Algorithms », volume 30(2), 2002, pages 113–139.

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

[34] A. GRIEWANK, U. NAUMANN. *Accumulating Jacobians by Vertex, Edge, or Face Elimination.* in « proceedings of CARI'02 6eme Colloque Africain sur la Recherche en Informatique », 2002.

[35] L. HASCOËT. *Transformations automatiques de specifications semantiques: application: Un verificateur 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, number MCS-P687-0997, Argonne National Laboratory, 1997.

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

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

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

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

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

[42] 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.

[43] 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.