Application of the automatic differentiation tool `Odyssée` to a system of thermohydraulic equations [1]

C. Duval [2] and **P. Erhard** [2] and **Ch. Faure** [3] and **J.Ch. Gilbert** [4]

Abstract. The applicability of automatic differentiation on a set of partial differential equations governing thermohydraulic phenomena in heat exchangers is examined. More specifically, the challenge is to differentiate `Thyc-1D`, a 1-D mockup of the 3-D code `Thyc` implementing these equations, by means of the automatic differentiator `Odyssée` with as few manual interventions as possible. The program to differentiate contains 23 subroutines, including linear solvers and black-box functions, whose code is not available.

1 Introduction

Automatic differentiation is a set of techniques that are aimed at differentiating functions from a program that computes its values at arbitrary points (see for example [1, 5, 8]). The method differs from finite differences in that the value of the derivatives are computed exactly (up to rounding errors) in a much more efficient way. This is particularly true when gradients (set of partial derivatives) are to be computed. In this case, the *reverse mode* of automatic differentiation can be viewed as a method for generating adjoint codes automatically.

The aim of this work is to differentiate a one-dimensional mockup `Thyc-1D` of the 3-D code `Thyc` from EDF. This code implements a set of five partial differential equations governing thermohydraulic phenomena in heat exchangers. `Thyc-1D` contains 23 subroutines, including linear solvers and black-box functions, whose code is not available. This is achieved by improving and using the automatic differentiator `Odyssée` [9], developed at INRIA. The challenge is to differentiate the program with as few manual interventions as possible, in such a way that the resulting code will be efficient with respect to the CPU time, sufficiently modest on its memory needs, and will provide accurate derivatives.

[1] This work was initiated and financed in part by EDF-DER.
[2] EDF-DER, Département Physique des Réacteurs, 1, avenue du Général de Gaulle, F-92141 Clamart Cedex, France.
[3] INRIA Sophia Antipolis, 2004 route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France.
[4] INRIA Rocquencourt, BP 105, F-78153 Le Chesnay Cedex, France.

2 Description of the thermohydraulic application

More concretely, we aim at computing the sensitivity of a partial differential equation system answer to a set of input parameters. In fact, even if an industrial computer code is validated on a large experimental data basis, it could provide bad results particularly when it is used far from the limits of its applicability domain. Sensitivity evaluation provides information on influent parameters according to this calculation response. These parameters could be boundary or initial conditions, closure laws, . . . An uncertainty may be evaluated on calculation results by using system answers and their derivatives given by the derivated computer code. In our study, we aim at knowing closure laws influence in order to perform the most sensitive ones.

The chosen model is a mock-up of an industrial computer code for thermohydraulics in bundles, `Thyc-1D` developed at EDF/Direction des Etudes et Recherches. This code is used to study a single or two-phase flow in reactor cores, steam generators and condensers. The differentiated system consists of: three conservation equations for the mixture (mass, momentum and energy), one conservation equation for the vapor mass and a last one for relative momentum between liquid and vapor phases. The system is shown below.

$$\frac{\partial \rho}{\partial t} + \operatorname{div} \rho u = 0,$$

$$\frac{\partial \rho u}{\partial t} + \operatorname{div}(\rho u^2 + \rho c(1-c)u_r^2)) = -\nabla p + \rho g - M_w,$$

$$\frac{\partial \rho h}{\partial t} + \operatorname{div}(\rho u h + \rho \mathcal{L} c(1-c)u_r) = \frac{\partial p}{\partial t} + u\nabla p + q,$$

$$\frac{\partial \rho c}{\partial t} + \operatorname{div}(\rho c(u + (1-c)u_r)) = \gamma,$$

$$\begin{aligned}\frac{\partial u_r}{\partial t} &+ (u_r\nabla)u + (((1-2c)u_r + u)\nabla)u_r \\ &= (\frac{1}{\rho_l} - \frac{1}{\rho_v})\nabla p - \beta_1 \rho u - \beta_2 u_r + \frac{\gamma}{\rho}(1-2c)u_r.\end{aligned}$$

The main variables of the previous system are: enthalpy h, pressure p, mass flow rate of the mixture ρu, vapor quality c, and relative velocity between phases u_r.

Many laws close this system: friction pressure loss tensor (M_w), volumetric power (q), interfacial drag coefficient (beta terms), interfacial mass transfer term (γ), ... This last closure law is not well known and system answers sensitivities to this parameters are very important.

Having the derivated code of this model is interesting to study one of the many variables sensitivity `ur`, the relative velocity between vapor and liquid phases according to two parameters included in the interfacial drag coefficient (`cd` and `qsi`) and the following ones: relaxation time `tau` included in the previously mentioned interfacial mass exchange closure law, and power exchanged with the fluid q.

The studied flow is a liquid-vapor mixture. Entering in a heating tube at its lower part, the fluid gets out with a high vapor ratio at its top.

3 The automatic differentiation tool `Odyssée`

`Odyssée` is an automatic differentiation tool developed at INRIA. It is able to "differentiate a program", with respect to the *input* variables (arguments and/or common variables) of the head-unit. For `Odyssée`, a *program* is a set of Fortran-77 units (functions or subroutines), whose call-graph forms a tree. The root of the tree is called the *head-unit*. The result of the differentiation of a program is a new program, which can compute at arbitrary points the derivative of the function evaluated in the original program.

`Odyssée` can differentiate a program as a whole, detecting *active variables*, those whose value depends on the variables with respect to which the function is differentiated. Both the direct and reverse modes of differentiation are implemented. The first one is appropriate for computing directional derivatives, the second is adapted to the efficient computation of gradients. In `Odyssée`, the adjoint code generated by the reverse mode has the same structure as the original code.

3.1 Analysis of the code

The process of differentiation consists of four phases:

1. preparation of the code,
2. interprocedural analysis of the program,
3. differentiation of the algebraic expressions,
4. simplification of the code.

During the first phase, the system modifies the original units and puts them in a suitable form for differentiation. It rewrites the nondifferentiable operators `abs`, `min` into `if`-statements, which are tractable by `Odyssée`. The `function` statements are also in-lined into the code. The third transformation applied in this phase comes from theoretical reasons: if the code is formed of binary expressions, then the complexity of the calculation of the function and its derivative is less than four or five times the one of the function alone (this depends on the differentiation mode, see Section 3.2). We have implemented a process that splits expressions into equivalent binary expressions by introducing intermediate variables (see [3]).

During the second phase, `Odyssée` makes an interprocedural analysis of the program, which results in a dependency graph between the input/output variables of each unit. These dependencies are stored in a data basis, which is filled by the differentiator for the available subroutines. For the subroutines whose code is not supplied (black-box subroutines of `Thyc-1D` or user-supplied derivatives), the user has to fill first the data basis, by declaring what are the arguments of the subroutine and which are the input or/and output variables. This information is enough for `Odyssée` to make a consistent analysis of the overall program. At that point, the system is able to propagate the active variables from the head-unit to all the other units. Thus the active inputs of each subroutine are known and `Odyssée` can differentiate them independently.

In the third phase, the unit by unit differentiation of the program is done, according to the following two rules. First, each subroutine of the program is differentiated with respect to its input variables, and not with respect to the input variables of the head-unit. Secondly, the differentiation is *maximal* in the sense that a subroutine is differentiated with respect to the maximum set of input variables appearing in every `call` statement. This allows `Odyssée` to generate only one subroutine for each unit of the original program. The methods used to differentiate a subroutine are described in the next section.

The fourth phase aims at simplifying the resulting code. The algebraic expressions are simplified in the same way as in Computer Algebra Systems except that the arguments of sums and products cannot be reordered modulo associativity and commutativity. The dead code is suppressed, which is very important in the reverse mode of differentiation, because the system generates to many `saving` instructions.

3.2 Two modes of differentiation

Automatic differentiation is based on two principles. The first one is that a program can be seen as a composition of functions, the second one is the chain rule.

In its simplest form a program is formed of a finite sequence of assignment statements. At each of these statements, one can associate a function leaving unchanged all the variables of the program except the one modified by the statement. Then, the whole program can be viewed a composition of these functions.

Now, suppose that there are K statements in the program. Then the function f that it computes is the composition of K element functions: $f = f_K \circ \cdots \circ f_2 \circ f_1$. Denote by J_k the Jacobian matrix of f_k computed at $(f_{k-1} \circ f_{k-2} \circ \ldots \circ f_1)(x)$. From the chain rule, the Jacobian matrix of f at x is

$$J = J_K \cdots J_2 J_1,$$

and the transposed Jacobian matrix of f is of course

$$J^\top = J_1^\top J_2^\top \cdots J_K^\top.$$

The *direct mode* of differentiation consists in calculating the directional derivative Ju with the first formula, while the *reverse mode* of differentiation consists in computing $J^\top v$ with the second formula. If f is defined from $\mathbb{R}^n$ to $\mathbb{R}$, one sees that the reverse mode can compute the n partial derivatives forming the gradient of f by a sequence of K matrix-vector products (take $v = 1$ above).

Given a program P (an example is the subroutine `sample` given in Figure 1) and some *input* or *independent* variables, `Odyssée` can generate two kinds of codes: the tangent code TP and the cotangent code T^*P. The tangent code computes the product of the Jacobian matrix by a vector u (direct mode) and the cotangent code computes the product of the transposed Jacobian matrix by a vector v (reverse mode). It is important to note that in the reverse mode, it is necessary either to store or to recalculate the intermediate values ($f_1(x)$, $f_2(f_1(x))$, ... in the example above) to be able to compute the transposed Jacobian matrix.

```
SUBROUTINE sample (x,y,z)
PARAMETER (n = 3)
DIMENSION x(n), y(n)

CALL norme2 (x,xx)
CALL norme2 (y,yy)
IF (xx.le.yy) THEN
   z = xx-yy
ELSE
   z = yy-xx
END IF
RETURN
END

SUBROUTINE norme2 (x,y)
PARAMETER (n = 3)
DIMENSION x(n)

y = 0.
DO i =1, n
   y = y + x(i)**2
END DO
RETURN
END
```

Figure 1. `Sample` code

In the tangent code TP generated by `Odyssée` (see Figure 2 for the tangent code `sampletl` associated with the subroutine `sample` of Figure 1), a new variable `vtl` (with suffix `tl`) is introduced for each variable `v` in the original code. It represents its directional derivative. The tangent code is obtained by inserting one assignment statement before each assignment statement of the original code. Its goal is to compute the directional derivative of the modified variable. The control structure of the code is preserved.

In the case of the subroutine `sampletl`, the directional derivatives of the input variables `x` and `y` are supposed given in `xtl` and `ytl`, and the directional derivative of the output

```
SUBROUTINE sampletl (x, y, z, xtl, ytl, ztl)
PARAMETER (n = 3)
DIMENSION x(n), y(n), xtl(n), ytl(n)

CALL norme2tl (x, xx, xtl, xxtl)
CALL norme2tl (y, yy, ytl, yytl)
IF (xx.LE.yy) THEN
  ztl = -yytl+xxtl
  z = -yy+xx
ELSE
  ztl = yytl-xxtl
  z = yy-xx
END IF
RETURN
END

SUBROUTINE norme2tl (x, y, xtl, ytl)
PARAMETER (n = 3)
DIMENSION x(n), xtl(n)

ytl = 0.
y = 0.
DO i = 1, n
   ytl = ytl+2*xtl(i)*x(i)
   y = y+x(i)**2
END DO
RETURN
END
```

Figure 2. Tangent code of `sample` generated by `Odyssée`, for the input variables `x` and `y`

variable `z` is placed in `ztl`. After execution, one has

$$\texttt{ztl} := \frac{\partial \texttt{z}}{\partial \texttt{x}}\texttt{xtl} + \frac{\partial \texttt{z}}{\partial \texttt{y}}\texttt{ytl}.$$

Each subroutine of the cotangent code T^*P generated by `Odyssée` is formed of two parts (see Figure 3 for the cotangent code `samplead` associated with the subroutine `sample` of Figure 1). In the first part, the original code is executed with appropriate savings. In the second part, the adjoint variables `vad` (with suffix `ad`) associated with the original variables `v` are computed by transposition of tangent code, as explained above. As a result, the execution flow of the second part is reversed with respect to the one of the original code. For example, the `do`-loop `DO i=1,n` of the subroutine `norme2` becomes `DO i=n,1,-1` in `norme2ad`.

In the case of the subroutine `samplead`, the adjoint variables `xad` and `yad` are computed from the adjoint variables `xad`, `yad`, `zad`, and from the value of the original variables `x`, `y`, and `z`. After execution, one has

$$\begin{aligned}\texttt{xad} &:= \texttt{xad} + \frac{\partial \texttt{z}}{\partial \texttt{x}}\texttt{zad}\\ \texttt{yad} &:= \texttt{yad} + \frac{\partial \texttt{z}}{\partial \texttt{y}}\texttt{zad}\\ \texttt{zad} &:= 0.\end{aligned}$$

```
SUBROUTINE samplead (x, y, z, xad, yad, zad)
PARAMETER (n = 3)
DIMENSION x(n), y(n), xad(n)
LOGICAL save

xxad = 0.
yyad = 0.

CALL norme2 (x, xx)
CALL norme2 (y, yy)
save = xx.LE.yy
IF (save) THEN
  z = xx-yy
ELSE
  z = yy-xx
END IF

IF (save) THEN
  xxad = xxad+zad
  yyad = yyad-zad
  zad = 0.
ELSE
  xxad = xxad-zad
  yyad = yyad+zad
  zad = 0.
END IF
CALL norme2ad (y, yy, yad, yyad)
CALL norme2ad (x, xx, xad, xxad)
RETURN
END

SUBROUTINE norme2ad (x, y, xad, yad)
PARAMETER (n = 3)
DIMENSION x(n), xad(n)

DO i = n, 1, -1
xad(i) = xad(i)+yad*(2*x(i))
END DO
RETURN
END
```

Figure 3. Cotangent (or adjoint) code of `sample` generated by `Odyssée`, for the input variables `x` and `y`

3.3 Limitations and future work

The current limitations of `Odyssée` are the following:

– `common` blocs with different partitions and `equivalence` statements are not accepted,
– an argument of a subroutine cannot be a subroutine name,
– only subroutines can be differentiated (not the `function`s or `program`s),
– the `goto`s are not accepted in reverse mode.

`Odyssée` generates the derivatives of all the variables of the code that depend on the input variables. In the present version, `Odyssée` does not take advantage of a selection of output variables to reduce the execution time.

4 Using `Odyssée` on `Thyc-1D`

As described in Section 2, we study how the variables `cd`, `qsi`, `tau`, and `puisvol` $= q$ influence `ur` $= u_r$. Therefore the program `Thyc-1D` must be differentiated with respect to these four input variables. More precisely, we want to know how the maximum (in time and space) of `ur` is influenced by these variables. Since this maximum occurs at the time step 306 and the node 23 in space, it is the program that computes `urmax` $=$ `ur(306,23)` that is differentiated.

4.1 Treatment of black-box subroutines and linear solvers

Two difficulties have been encountered in differentiating `Thyc-1D`. First, the program has black-box subroutines, whose code is not available. Secondly, `Thyc-1D` contains linear solvers implementing iterative algorithms (Jacobi's and Gauss-Seidel's methods) with stopping tests, so that the number of iterations is not known beforehand and depends on the value given to the independent variables.

The derivatives of the black-box subroutines have been hand-coded, using finite differences to approximate the derivatives.

For the linear solvers, we have chosen to supply the associated subroutines to `Odyssée` instead of letting it generate the code. This simplifies the code generation (differentiating iterative processes is a rather sophisticated operation, see [4]) and makes the resulting code very efficient.

To be more specific, suppose that the original subroutine solves the linear system in y

$$Ay = b,$$

where A is a nonsingular matrix and b is a vector. Then in direct mode, the associated subroutine has to solve $Ay = b$ to find y and the linear system

$$A\dot{y} = \dot{b} - \dot{A}y$$

to find $\dot{y}$. Here $\dot{v}$ denotes the directional derivatives of the variable v. In reverse mode, the adjoint variables $\bar{A}$, $\bar{b}$, and $\bar{y}$ are obtained by

$$\begin{cases} A^\top z = \bar{y} \quad \text{(to solve in } z\text{)} \\ \bar{b} := \bar{b} + z \\ \bar{A} := \bar{A} - zy^\top \\ \bar{y} := 0. \end{cases}$$

More details can be found in [2].

4.2 Description of user-differentiated subroutines

The subroutines that must not be processed by `Odyssée` (here, the black-box subroutines and the linear solvers) have to be described in a file that is read by the system before it analyses the program. This description specifies what are the dummy arguments and whether they are input or output variables.

An appropriate language is used for this task. For example, we have described the black-box function `tbalps` by:

```
setdummys tbalps (p1 p2 p3 p4 p5 tbalps)
setinout tbalps (p2 p3) () (tbalps) ()
```

and the linear solver `tdma` by:

```
setdummys tdma (c a d b x imax)
setinout tdma (c a d b x) () (c a d b x) ()
```

This information is enough for `Odyssée` to make a consistent analysis of the overall program.

4.3 Towards derivatives

As described in Section 3, `Odyssée` can derive a set of subroutines, whose call graph forms a tree, the root of which is called the head-unit. To give this structure to the set of units to derive in `Thyc-1D`, we had to build the head-unit, called `princp_sub`, and had to specify the input variables of this subroutine.

Then the global analysis of the set of units starting with `princp_sub` was performed by `Odyssée`, which was able to differentiate the overall program either in direct mode to generate the tangent code `princp_subtl` or in reverse mode to generate the cotangent code `princp_subad`. The hand-derivated subroutines (for the black-box functions and linear solvers) fit the required format, so that all the subroutines could be compiled and linked together.

Then, we had to write the main programs `princptl` and `princpad` that compute the four partial derivatives of `urmax` with respect to the four input variables, either in direct mode (by calling `princp_subtl`) or in reverse mode (by calling `princp_subad`).

In direct mode, `princp_subtl` has to be called four times to get the four derivatives. The part of the code computing and printing the derivatives with respect to `cd` and `qsi` is:

```
  cdtl = 1.
  qsitl = 0.
  tautl = 0.
  puisvoltl = 0.
  call princp_subtl (npas)
  print *, "durmaxdcd = ", urmaxtl

  cdtl = 0.
  qsitl = 1.
  tautl = 0.
  puisvoltl = 0.
  call princp_subtl (npas)
  print *, "durmaxdqsi = ", urmaxtl
  ...
```

In reverse mode, `princp_subad` has to be called only once to get the four derivatives. The part of the code computing and printing the four derivatives is:

```
  cdad = 0.
  qsiad = 0.
  tauad = 0.
  puisvolad = 0.
  urmaxad = 1.
  call princp_subad (npas)
  print *, "durmaxdcd = ", cdad
  print *, "durmaxdqsi = ", qsiad
  print *, "durmaxdtau = ", tauad
  print *, "durmaxdpuisvol = ", puisvolad
```

5 Numerical results

We have compared the results obtained by the tangent and cotangent codes generated by `Odyssée` with the derivatives approximated by finite differences. For the latter, we have chosen the step-size giving the largest number of correct significant digits (this requires trying many step-sizes). Comparisons have been made in single and double precision.

Here, we present results obtained for the derivatives of all the 25 space components of `ur`, not only the 23*rd* giving the maximum of `ur`, with respect to the four inputs. This requires running `princp_subtl` four times (in direct mode) and `princp_subad` twenty five times (in reverse mode).

5.1 Comparison between tangent and cotangent codes

The differentiation of the black-box subroutines by finite differences can be done in different ways. In order to make the tangent and cotangent codes more similar, we have chosen to compute in both modes the Jacobian matrix of the function realized by each of these subroutines. In direct mode there is a cheaper way of doing, which consists in computing only the directional derivative of the function.

Figures 4 and 6 show the values of the 25 components of the derivatives of `ur` with respect to `cd` and `qsi`, as computed by the tangent code. We do not show those computed by the cotangent code because they are similar. Table 1 shows the values of the derivatives of `urmax` $=$ `ur(306,23)` computed with the two modes of differentiation. We see that these derivatives have between 8 and 12 significant digits in common.

5.2 Comparisons with finite differences

The step-size for the finite differences has been chosen to make the derivatives as precise as possible. This *optimal* step-size has been obtained in the following way. The approximate derivatives of `urmax` has been computed for several step-sizes and the one that gives the greatest number of unvariant significant digits has been chosen. For example, for the derivative with respect to `cd`, we have tried ten different step-sizes. The results are given in Table 2. In this case, we see that the optimal step-size is $5\,10^{-6}$ and that the correct derivative is likely to be $-2.6877707\ldots$.

In Table 3, we show the approximate derivatives obtained by finite differences and those computed by `Odyssée`. In

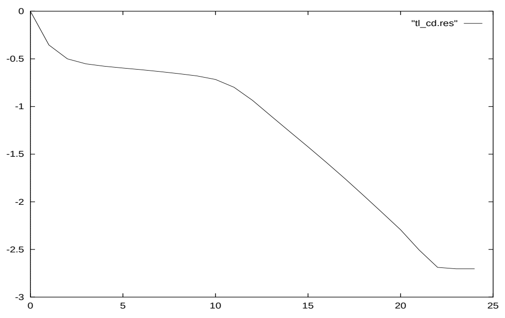

Figure 4. Derivatives in direct mode with respect to `cd`

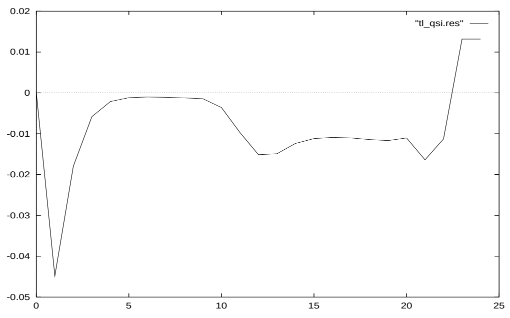

Figure 6. Derivatives in direct mode with respect to `qsi`

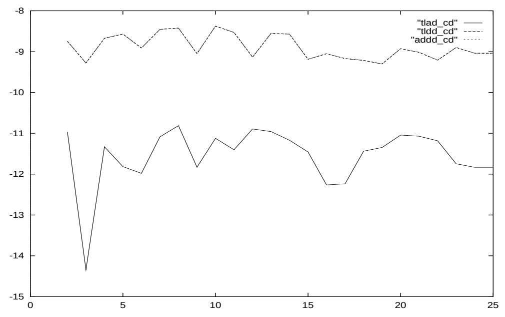

Figure 5. Comparison of derivatives with respect to `cd`

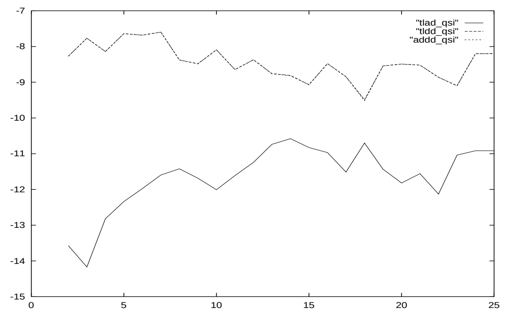

Figure 7. Comparison of derivatives with respect to `qsi`

	Tangent code
`cd`	$-2.6877707476674D+00$
`qsi`	$-1.1283833581385D-02$
`tau`	$2.3953473434534D-01$
`puisvol`	$8.4189731568450D-09$

	Cotangent code
`cd`	$-2.6877707476626D+00$
`qsi`	$-1.1283833581488D-02$
`tau`	$2.3953473304208D-01$
`puisvol`	$8.4189731614290D-09$

Table 1. Derivatives of `ur(306,23)` by `Odyssée` in double precision

Step-size	Approximate derivative
$5.E-11$	$-2.687743361207140E+00$
$5.E-10$	$-2.687798428269161E+00$
$5.E-09$	$-2.687771161191677E+00$
$5.E-08$	$-2.687771023524022E+00$
$5.E-07$	$-2.687770758846852E+00$
$5.E-06$	$-2.687770744280727E+00$
$5.E-05$	$-2.687770756817365E+00$
$5.E-04$	$-2.687771680981221E+00$
$5.E-03$	$-2.687864088674541E+00$
$5.E-02$	$-2.697140333223182E+00$

Table 2. Trial step-sizes for the derivative of `urmax` with respect to `cd`.

the latter case, we only display the most significant digits in common to the tangent and cotangent derivatives. The last column shows the maximum number of correct significant digits that `Odyssée` can yield with respect to "optimal" finite differences.

In Figures 5 and 7, we have compared the values of the derivatives computed by finite differences with optimal step-size, with those computed by `Odyssée`. The curves show the relative differences between two methods (the vertical axis has a logarithmic scale): one compares the tangent code to finite differences (dashed line), another compares the cotangent code to finite differences (dotted line) and the last one compares the tangent and cotangent codes (plain line). The lowest curve indicates the couple of methods that most agree. In each case, it is formed of the tangent and cotangent codes of `Odyssée`.

The same tests have been performed in single precision. The derivatives of `urmax` obtained by `Odyssée` are given in Table 4.

As in double precision, we have chosen the "optimal" step-size for computing approximate derivatives by finite differences and have compared them with the one obtained by `Odyssée`. The results are shown in Table 5. To give an interpretation to these results, let us compare them to those in

	Finite differences	Odyssée	Max gain
cd	$-2.6877707D+00$	$-2.68777074766D+00$	$+4$
qsi	$-1.1283833D-02$	$-1.1283833581\ D-02$	$+3$
tau	$2.395347\ D-01$	$2.3953473\ D-01$	$+1$
pui.	$8.4189731D-09$	$8.4189731\ D-09$	$+0$

Table 3. Comparison of derivatives in double precision

	Tangent code	Cotangent code
cd	$-2.68763E+00$	$-2.68763E+00$
qsi	$-1.12781E-02$	$-1.12782E-02$
tau	$2.39515E-01$	$2.39520E-01$
puisvol	$8.41967E-09$	$8.41967E-09$

Table 4. Derivatives of `ur(306,23)` by `Odyssée` in single precision

Table 3. If we trust the first significant digits obtained by finite differences in double precision, we see that with the optimal step-size, only the last digit may be erroneous in the second column of Table 5. We also see that the digits in common to the direct and reverse mode of `Odyssée` are not necessary correct: up to 2 digits can be erroneous. This implies that the number in the column 'Max gain' in Table 3 are not the actual number of digits that `Odyssée` can yield with respect to finite differences. It is an upper bound for this number. On the other hand, always based on the digits obtained in double precision, we can give the number of digits that `Odyssée` yields in single precision, with respect to finite differences. They are given in the last column of Table 5.

	Finite differences	Odyssée	Actual gain
cd	$-2.68E+00$	$-2.68763E+00$	$+1$
qsi	$-1.19E-02$	$-1.1278\ E-02$	$+1$
tau	$2.\ \ E-01$	$2.395\ E-01$	$+3$
pui.	$8.3\ E-09$	$8.41967E-09$	$+2$

Table 5. Comparisons of derivatives in single precision

5.3 Efficiency in memory-space and CPU time

In order to compare the efficiency of the codes generated by `Odyssée`, we have generated the derivatives of `Thyc-1D` with respect to 1 input variable (`cd`), 2 input variables (`cd`, `qsi`), 3 input variables (`cd`, `qsi`, `tau`), and 4 input variables (`cd`, `qsi`, `tau`, `puisvol`). Table 6 shows the results: the CPU column gives the CPU time in seconds, the `text` column gives the text length in bytes, and the `bss` column gives the size of the initialized data in bytes.

	Tangent code			Cotangent code		
	CPU	`text`	`bss`	CPU	`text`	`bss`
0	5.72	270336	105608	5.93	270336	105608
1	14.08	417792	266680	27.61	573440	2540712
2	26.91	417792	266712	27.30	573440	2540912
3	38.25	417792	267592	27.20	581632	2540544
4	55.21	417792	267616	27.31	581632	2540608

Table 6. Execution time and memory space

Let $\mathcal{T}(P)$ be the execution time of the original code, $\mathcal{T}(TP)$ be the execution time of the tangent code, $\mathcal{T}(T^*P)$ be the execution time of the cotangent code, and n be the number of input variables. Extrapolating the results in Table 6 leads to the following ratios:

$$\frac{\mathcal{T}(TP)}{\mathcal{T}(P)} \approx 2.3\,n$$

and

$$\frac{\mathcal{T}(T^*P)}{\mathcal{T}(P)} \approx 4.6.$$

These numbers are in accordance with the theoretical bounds, which are $4n$ for the first ratio and 5 for the second (see for example [7, 6, 5]).

In terms of the size of the code, the tangent code takes 35% more memory space whereas the cotangent code takes about twice that of the original code. As for the size of the data space, it is approximately doubled in the tangent code and multiplied by 25 in the cotangent code. One must notice that the only memory management strategy implemented in the current version of `Odyssée` saves all the modified variables. Other strategies are being studied for the next version of `Odyssée`.

6 Conclusion

These tests show that, in double precision, 7 or 8 correct significant digits can be obtained by "optimal" finite differences. With respect to this, the derivatives obtained by `Odyssée` are likely to have between 1 and 3 more correct significant digits. We believe that the accuracy of the derivatives obtained by `Odyssée` have been limited by the need to differentiate by finite differences frequently used black-box functions.

As for the CPU time efficiency, it fits the theory: the evaluation of the tangent code takes about 2.3 times the time to run the original code, and the evaluation of the cotangent code takes around 4.6 times the time to evaluate the original function. The size of the memory used by the tangent code is twice as large as the memory used by the original code, while the cotangent code used about 25 times more memory than the original code. It is likely that this last number could be reduced by using more sophisticated strategies for determining the information to save during the computation of the original code. This optimization will become necessary when automatic differentiation of the industrial code `Thyc` will be undertaken, which is next stage of the EDF-INRIA collaboration.

REFERENCES

[1] G. Corliss, A. Griewank (Editors). *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, number 53 in Proceedings in Applied Mathematics. SIAM, Philadelphia, (1991).

[2] F. Eyssette, Ch. Faure, J.Ch. Gilbert, N. Rostaing-Schmidt (1996). Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant. Rapport de Recherche INRIA n^o 2795 ou Rapport EDF/DER, HT-13/96/001/A.

[3] C. Faure (1996). Splitting of Algebraic Expressions for Automatic Differentiation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation : Techniques, Applications, and Tools*. SIAM, Philadelphia, Penn., 1996. (To appear).

[4] J.Ch. Gilbert (1992). Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1, 13–21.

[5] J.Ch. Gilbert, G. Le Vey, J. Masse (1991). La différentiation automatique de fonctions représentées par des programmes. Rapport de Recherche n^o 1557, INRIA, BP 105, F-78153 Le Chesnay, France.

[6] A. Griewank (1989). On automatic differentiation. In M. Iri, K. Tanabe (Editors), *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, Dordrecht.

[7] J. Morgenstern (1985). How to compute fast a function and all its derivatives, a variation on the theorem of Baur-Strassen. *SIGACT News*, 16, 60–62.

[8] N. Rostaing-Schmidt (1993). *Différentiation automatique : application à un problème d'optimisation en météorologie*. Thèse, Université de Nice-Sophia Antipolis, France.

[9] N. Rostaing, S. Dalmas, A. Galligo (1993). Automatic differentiation in Odyssée. *Tellus*, 45, 558–568.