Generating adjoints of industrial codes with limited memory

Christèle Faure
INRIA Sophia Antipolis, 2004 route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France

Mai 2000

Abstract. The challenge is to generate an adjoint code of `Thyc-3D` by means of the automatic differentiator `Odyssée`. This adjoint code must compute the gradient using at most 3 times more core memory than the original function. The other constraint is to do as few manual interventions as possible. The original code contains nearly 500 sub-programs (70000 lines, 1000 parameters) and calls black-box functions whose codes are not available. Those functions are only available in the form of a compiled library. We discuss various storage/retrieval and precomputation options. The results suggest further improvements and interactive features of automatic differentiation tools.

Keywords: Automatic differentiation, adjoint codes, memory limited, thermalhydraulic, two-phase flow, condenser, steam generator.

1. Introduction

Automatic Differentiation (Griewank and Corliss, 1991; Berz et al., 1996; Griewank, 2000) is a set of techniques that are aimed at differentiating functions that are given as programs for their evaluation. The differentiated code computes the same values as the original function plus some derivative information. 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. The *reverse mode* of automatic differentiation (AD) that can be viewed as a method for generating adjoint codes automatically is considered here. The theoretical complexities of the reverse mode AD are studied in (Faure, 2000), this paper focuses on practical aspects of automatic adjoining.

`Thyc-3D` is an industrial 3D thermalhydraulic in bundles code developed at EDF-DER and associated to a 1D mockup named `Thyc-1D`. `Thyc-1D` is a one dimensional thermalhydraulic module for two-phase flow modeling. It consists of: three conservation equations for the two-phase mixture (mass, momentum and energy), one conservation equation for the vapor mass and one conservation equation for the relative liquid-vapor velocity between the two phases. `Thyc-3D` allows the study of various flows in bundles such as tube heat transfer exchangers and rob bundles like reactor cores. We do not give details on these softwares, but a complete description can be found in (Briere et al., 1993; Jouhanique and Rascle, 1995). We have first worked on the mockup to evaluate the feasibility of automatic adjoining using `Odyssée` (Faure and Papegay, 1998) and have compared the various adjoining strategies. In (Duval et al., 1996) we show that the ratio of the execution time of the adjoint code of `Thyc-1D` with respect to the original code is 7.1 which is acceptable. The ratio 9 in terms of memory requirement is acceptable on the 1D code, but not on the 3D code. Faure and Duval (Faure and Duval, 1998) show that the use of the optimal checkpointing technique makes the generated adjoint code more flexible. For example, if the chosen ratio in terms of memory is 3 (instead of 9), the ratio in terms of execution time is 7.5 (compared to 7.1). Using optimal loop checkpointing, the memory requirement is divided by 3 whereas the execution time is slightly augmented.

The use of the automatic differentiation tool `Odyssée` to compute some gradients of `Thyc-3D` is presented. The challenge is to differentiate the program with as few manual interventions as possible, in such a way that the resulting code needs only a moderate amount of core memory. Using automatic differentiation we are sure of the accuracy of the computed derivatives, only execution time and memory requirement are to be studied. For this study[1], the end user (namely EDF-DER) imposed no constraint on the execution time, but limited the adjoint core memory to at most 3 times the original memory need.

Section 2 presents briefly the reverse mode from a theoretical point of view. The strategical choices used for automatic generation of the adjoint codes of `Thyc-3D` are explained in Section 3. The execution time and memory requirements of the generated adjoint codes are analysed in Section 4.

2. Reverse mode problems

The evaluation of derivatives in the opposite order to the computation of the original variables makes the reverse mode much more complicated than the direct mode. The straight-forward solution is to record every intermediate computation (named *trajectory*) during a first run of the original program and during a second phase to retrieve all these values before computing the elementary Jacobian matrices in the opposite order of the original execution. This basic strategy is optimal in terms of theoretical execution time but cannot be applied on many

[1] This study was initiated and financed in part by EDF-DER.

real world programs because of the large amount of storage required. A trade-off between storing the whole trajectory but never repeating even parts of the original calculation and storing parts of the trajectory but partially recomputing the original function is to be applied. When the differentiation of a multi-level program is concerned, a variety of strategies can be applied for generating the adjoint code. In (Faure, 2000) these strategies are described and compared in terms of time and memory complexities. `Odyssée` and `Tamc` (Giering, 1997) are the only AD tool that allow for automatic adjoining of industrial size programs. They are both based on source transformation techniques which makes compile time optimizations of the adjoint code possible. The operator overloading technique used within `Adolc` (Griewank et al., 1996) makes compile time optimizations impossible. In this paper, we consider AD by source transformation. Any adjoining strategy is the combination of two sub-strategies: a first one for reverting the original execution described in Section 2.1 and a second one for computing and storing the trajectory described in Section 2.2. The AD-tool `Odyssée` implements the strategies used for this study, it is shortly described in Section 2.3.

2.1. Reverting the original execution

Reverse mode AD requires the reversion of the original execution to compute the adjoint variables. This task is often difficult on a real world program as: general programs use branches and loops in such a way that they are difficult to reverse at compile time. AD is a transformation performed at compile time, if the flow graph formed by the computed values and their data dependencies is completely static (i.e. can be extracted at compile time) there is no problem in reverting it. To the extend that branches and loops can only be detected at runtime, the only possible way to revert a general flow is to have an interpreter of the flow graph.

Figure 1.4 shows two possible ways of computing the derivative variables in reverse order with respect to the computation of original variables. We demonstrate some strategies on a really sample example that computes $|x^2 + 4x + 12k^2|$. We denote `xccl, yccl, zccl` the adjoint variables associated with the original variables `x, y, z`. The original code `A` (see Figure 1.1) is split into three basic blocks of statements `A1`, `A2` and A3 (see Figure 1.2), each of them is associated with an adjoint basic block respectively denoted by `A1`', `A2`' and `A3`' (see Figure 1.3).

Two methods for reverting the execution of a program are described. The first method named “Syntax reversal” reproduces the control values to revert the execution syntactically: each original control statement is

```
y = x**2 + 4*x + 12*k**2          A1
if (y>0) then                     if (y>0) then
   z = y                             A2
else                              else
   z = -y                            A3
end if                            end if
```

Figure 1.1. Original code `A`

```
y = x**2 + 4*x + 12*k**2     z = y            z = -y
```

Figure 1.2. Original codes of blocks `A1`, `A2` and `A3`

```
xccl += (2*x+4)*yccl     yccl += zccl       yccl += - zccl
yccl  = 0.               zccl  = 0.         yccl  = 0.
```

Figure 1.3. Adjoint blocks `A1`', `A2`' and `A3`' of blocks `A1`, `A2` and `A3`

```
                      push (1,stack)
A1                    A1

if (y>0) then         if (y>0) then
                         push (2,stack)
   A2                    A2
else                  else
                         push (3,stack)
   A3                    A3
end if                end if

                      do block=3, 1, -1
                         block = pop (stack)
if (y>0) then            case block of
   A2'                        1 => A1'
else                          2 => A2'
   A3'                        3 => A3'
end if                   end case
A1'                   end do
```

Figure 1.4. Adjoint codes of `A`: Syntax and Flow reversal

```
y = x**2 + 4*x + 12*y**2
```

Figure 2.1. Original code B_1

reproduced. In code example `A`, the value of the test `y>0` does not change and the same test is performed. If `y` was modified, the value of the test should be recorded and retrieved.
The second one named “Flow reversal” is based on the introduction of an interpreter that reverts the original exexcution path at runtime. In this case, basic block indexes are recorded in the control stack are retrieved by the interpreter.

The control values are not of the same nature in both strategies: in the first method only originally computed values are recorded, whereas in the second method new values w.r.t the original computation are required. On the sample code `A` three indexes are recorded if the flow reversal algorithm is used, but none is if the syntax reversal algorithm is applied. Moreover, one supplementary test w.r.t the original execution is necessary in the first case and three tests plus one loop are necessary for the second one. The number of required control values using the flow reversal method is at least the same as using the syntax reversal. While the “Flow reversal” algorithm can be applied on any source code, the “Syntax reversal” algorithm cannot be used when the execution cannot be foreseen at compile time.

2.2. Computing the adjoint values

On the sample example from Figure 1.1, no original variable must be recorded to compute the adjoint variables because none of them is overwritten.
Let consider the block `B1` Figure 2.1 which is the block `A1` where the variable `k` is replaced by `y`. The original value of `y` will be required to compute the adjoint variables of `B1` as shown in Figure 2.2. Let call `B` the original code `A` where the block `A1` is replaced by `B1`. The adjoint code `B`’ of `B` is the same as `A`’ except that `A1` is replaced by the recording part of `B1` and `A1`’ is replaced by the retrieving part of `B1` (see Figure 2.2). The values required for the computation of adjoint values are recorded in a stack. This stack can be very large if all the required values are stored.

AD-tools by source transformation like `Tamc` and `Odyssée` implement a trade-off between execution time and memory requirement called *checkpointing*. The checkpointing strategy tends to replace some storage by the corresponding recomputation: if a value v is not recorded, then it has to be recomputed. In general, one checkpoint c_i records some state

```
                                   y = pop (stack)
push (y,stack)                     xccl = xccl + (2*x+4)*yccl
y = x**2 + 4*x + 12*y**2           yccl =         24*y  *yccl
```

Figure 2.2. Adjoint code `B1`': Recording and retrieving part.

of the computation at time t_i and is used to compute the adjoint of the computation from t_{i+1} to t_i as the initial point of the forward execution from t_i to t_{i+1}.
Two levels of checkpoints are implemented in `Tamc` and `Odyssée`: one consists in storing the context of call of sub-programs to be able to call the corresponding derivatives, the second consists in storing the context of call of loop's bodies (Faure, 2000). In the next two sections, we describe briefly both techniques which are necessary to differentiate a real world program.

The sub-program checkpointing strategy is a structural checkpointing described in (Volin and Ostrovskii, 1985): i.e. the place where the checkpoints are set depends on the structure of the original program. In practical terms, each generated units is composed of two parts: the *recording part* computes and saves the trajectory, and the *retrieving part* restores the trajectory and computes the derivatives. As a result, a sub-program that appears at level l in the call tree will be recomputed l times. The extra-cost in recomputation of the original sub-programs is proportional to the depth of the call tree (in the worst case).

The optimal checkpointing strategy can be theoretically described as follows. Consider any straight line program P whose execution can be divided into n blocks of instructions with roughly the same execution time t, and the same size of context of call m (where the context of call of a block is the set of input variables necessary to run this block). If one can accommodate C such checkpoints, it is possible to compute their optimal distribution all along the execution of the sequence of blocks that minimizes the recomputation of those blocks. This optimal schedule has been described in different papers (Griewank, 1992; Grimm et al., 1996). This strategy is implemented in AD-tools on some patterns of code such as loops with fixed number of iterations. In this case, the body of the loop is a block and the sequence of instructions is already syntactically split into blocks. If all the iterations of the loop are considered of the same execution time and context size, the theory applies. The hypothesis that the execution time of the body is constant is a bit unrealistic, but even though one may use the maximum execution time instead.

Tamc and Odyssée automatically combine both levels of checkpointing on adjoint codes. A general package called treeverse described in (Griewank and Walther, 1997) has been developed to optimally place checkpoints on any adjoint code.

2.3. The automatic differentiation tool Odyssée

Odyssée is an automatic differentiation tool developed at INRIA that differentiates Fortran-77 units. It is able to "differentiate a program", with respect to the *input* variables (arguments and/or common variables) of the head-unit chosen by the user. 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, that computes at arbitrary points the derivative of the function evaluated in the original program.

Odyssée differentiates a program as a whole, detecting *active variables*, namely those whose value depend on the variables with respect to which the function is differentiated. Both the direct and reverse modes of differentiation are implemented. In direct mode, Odyssée generates a tangent code that computes $J \cdot v$ whereas in reverse mode it generates an adjoint (or cotangent) code that computes $J^T \cdot v^\star$ where J is the Jacobian matrix and v, $v^\star$ are some input vectors. The former is appropriate for computing directional derivatives, the latter is suitable for the efficient computation of gradients.

Odyssée implements the two methods for reverting the original execution described in Section 2.1. The default value is the Syntax reversal method but the user can ask for the Flow reversal method to be applied. The two checkpointing levels are implemented: the default strategy is the sub-program checkpointing but the user can ask for loop checkpointing on specific loops.

3. Adjoint code of Thyc-3D

The source code of Thyc-3D (Briere et al., 1993; Jouhanique and Rascle, 1995) contains nearly 500 sub-programs that makes it 70000 line large without considering the source of the libraries where the black-box functions are defined. Thyc-3D is an evolutionary code: it consists of a main loop that iteratively computes the state vector x of the system at time tmax from some initial values. Let consider that: from the value of x at time t-1 and the value of a second vector y, the sub-program Solv computes x at time t. This program can be described by the following loop:

```
Do t=1, tmax
   x = Solv (x,y)
End do
```

If the standard reverse mode is applied on such a loop, the value of the state `x` must be recorded for each time-step. This requires a memory space of $(\texttt{tmax} + 1) * s_1$ where s_1 is the size of `x`. If this loop is checkpointed and if at most N checkpoints are used, the memory requirement is $N * s_2$ where s_2 is the sum of the sizes of `x` and `y`. This will cost a number of recomputations of `Solv` that depends on N and `tmax`.

We were not sure to be able to run the standard adjoint code under the constraint of an extra-cost in core memory of only 2 times the original cost. We chose to apply the two possible levels of checkpoints (sub-program and loop) to `Thyc-3D`. Section 3.1 shows how to generate the two adjoint codes. We chose the flow reversal algorithm because some `goto` instructions appear in the source code.

Section 3.2 describes the memory components of the adjoint codes whereas Section 3.3 describes the implementation choices due to the user constraint in core memory requirement.

3.1. Automatic generation of the adjoint codes

In `Thyc-3D`, some thermal-hydraulic libraries whose source code is not available are used. `Odyssée` handles black-box sub-programs: this means that a coherent derivative call is associated to each black-box call. The corresponding derivative codes must be written by hand.

`Odyssée` makes an inter-procedural analysis of the program, which results in a dependency graph between the input/output variables of each unit. When the sub-program source is available, these dependencies are computed by the system. If the source code is not available, the system associates a default value or reads the corresponding user dependency definition. This information for each sub-program is enough for `Odyssée` to make a consistent analysis of the overall program and to propagate the active variables from the head-unit to all the other units. Thus the active inputs of each sub-program are known and `Odyssée` can differentiate them independently.

To generate the two adjoint codes of `Thyc-3D` using `Odyssée`, the following commands are run:

load *thyc3d* loads all the Fortran units defined in file *thyc3d.*

loadbasis *thermo* loads the dependency definitions of the black box units from the thermal-hydraulic libraries in file *thermo.*

diff -cl -goto -vars *parcor* **-h** *princp* differentiates the program from head unit *princp* in adjoint mode `-cl` with respect to active variable *parcor*. The default sub-program checkpointing and the flow reversal algorithms `-goto` are applied.

diff -cl -goto -opt *solv* **-vars** *parcor* **-h** *princp* differentiates the program in adjoint mode using the same strategies as in the previous command. The optimal checkpointing algorithm `-opt` is applied to the loop that calls the subprogram *solv*.

After the execution of these `Odyssée`-commands, the two adjoint codes are generated: we denote by `Std` the adjoint code that implements subprogram checkpointing and `Ckp` the adjoint code that combines both levels of checkpointing. We also generate the tangent code of `Thyc-3D` using the differentiation command: **diff -tl -vars** *parcor* **-h** *princp*.

Two classes of sub-program have not been generated: the derivative of black-box sub-programs and the trajectory/checkpoint management sub-programs. The derivatives of the black-box sub-programs have to be hand-coded, using user known derivatives or finite differences approximations. These derivatives must be really precise not to obscure the accuracy of AD derivatives. This is quite difficult and often requires the use of multi-step finite differences as described in (Duval et al., 1996). Several implementations of the trajectory/checkpoint management libraries can be chosen. The efficiency of the adjoint runtimes deeply depends on these choices as we show in Section 4.

3.2. Memory components

In this section, we divide into separable components the memory requirement of the adjoint code when compared to the original code. The total memory amount is the sum of: *(1)* the memory space necessary for the original variables, *(2)* the extra space necessary to record the derivative variables, the space required to record the trajectory *(3)* and the checkpoints *(4)*.

The first component due to the original variables is the same in the original and the adjoint codes: 14 Mega bytes.
The second component is at most the same as the original space (first component) if one gradient is computed: indeed each original variable is associated to at most one derivative variable of the same size.
The third component is the trajectory. It consists of the recorded values required to compute the elementary partial derivatives, but also those necessary to reverse the flow graph. The size of this component is nearly impossible to predict at compile time, but can be computed from an

enhanced original code.
The fourth component contains the checkpoints of the main loop.

3.3. Memory management

Each of the memory component can be managed independently: internally/externally, dynamically/statically, globally/locally...
The management modes used for this study are:

static core core memory using statically allocated space,

dynamic core core memory using dynamically allocated space,

file unformatted files.

The derivative variables are implemented using the same mode as for the corresponding original variable (in `Thyc-3D` static core). This choice is supposed to be the right one for the original variable, then AD-systems just transpose this choice to the corresponding derivatives.
The trajectory management commands deal with scalar values and are spread all over the code: before each original statement some storage statement are executed, and before each derivative set of instructions the values are retrieved. Using a static core mode is difficult because the trajectory size is difficult to predict and using a file to record scalar values is quite expensive in time. Writing or reading a value on resp. from the disk is time consuming: it is a lot more expensive than an elementary operation like a product. As a consequence, the chosen trajectory management mode is dynamic core memory.
The checkpoints are recorded within direct access files: one file is associated to each checkpoint. This mode is chosen because the core memory is limited. Moreover, each checkpoint requires a large amount of memory location. Under this hypothesis, the use of files does not slow down the generated code.

The trajectory management commands are implemented using a C library. It uses a list of buffers of bytes: when the current buffer is full, it is added to the list and a new current buffer is allocated. We avoid the copy of the arrays within the buffers by using C pointers. This library is general and is distributed with `Odyssée`.
The two checkpoint management commands (record/retrieve) are specific to `Thyc-3D`. They have been written in Fortran and use unformatted Fortran files.
For the purpose of this study, both libraries have been modified to compute profiling information on the trajectory or the checkpoints.

4. Validation of the adjoint code

The sensitivity of the vapor title `tit`(2,2,30) with respect to all the components of the relative velocity between phases named `parcor` is computed. We compare the results obtained by the tangent and adjoint codes generated by `Odyssée` with the derivatives approximated by finite differences. For the latter, we have chosen the step-increment which gives the largest number of correct significant digits (this requires trying many step-increments). The runtimes use double precision.

Table I presents results obtained for five non zero components of the gradient $\partial \text{tit}(2,2,30)/\partial \text{parcor}(i,j,k)$. The values obtained by the two adjoint codes coincide exactly. From the fourth partial derivative presented in line 4, one can see that the values obtained using the tangent code and the cotangent code coincide to at least 11 significant digits, whereas with divided differences only 4 digits are correct. The relative values of the components of the gradient are quite different, which makes precise finite difference approximations rather tedious. For the first two components whose values are really small, only the magnitude (D-17, D-22) and the sign of the values obtained by the tangent and adjoint codes agree. For the three other components whose values are of size D-06, D-01, D-05, the values coincide to at least 7 digits (resp. 7, 11, 10).

Table I. Non zero partial derivatives $\partial \text{tit}(2,2,30)/\partial \text{parcor}(i,j,k)$.

i	j	k	`Std` or `Ckp`	tangent code	divided differences
1	1	4	-1.4645246970897D-17	-1.1340902358399D-17	
2	1	1	-6.4270625949449D-22	-4.9769392807577D-22	
5	1	4	1.7930113258011D-06	1.7930113627136D-06	
11	1	1	-1.6563046881561D-01	-1.6563046881616D-01	-0.16562916599905
14	1	4	-9.4198958637361D-05	-9.4198958638484D-05	

In the latter, we compare the adjoint codes to the original code in terms of execution time and memory requirements. The simulation time of `Thyc-3D` is 600 seconds. We have run the codes for two time steps: 0.5 seconds (1200 main loop iterations) and 0.05 seconds (12000 main loop iterations). Both adjoint codes `Std` and `Ckp` could be run for 1200 iterations **I** without running out of core memory whereas only `Ckp` could be run for 12000 iterations. The number of checkpoints **C** is chosen to be 300 to obtain a fast runtime on the target platform but can be easily modified.

Table II shows the execution **Time** and **Core** memory size of `Thyc-3D` and the ratios of the adjoint codes with respect to the original execution of `Thyc-3D`. Using the sub-program level checkpointing does not diminish enough the core memory size to fit the constraint given for this study. Indeed the ratio in core memory of the adjoint code `Std` with respect to the original code is 70.8. If the main loop is checkpointed `Ckp`, the ratio is less than the authorised maximum ratio: 2.6 compared to 3. Surprisingly, this core memory gain leads to an execution time gain. The core memory of `Ckp` does not depend on the number of checkpoints chosen, neither on the number of main loop iterations. This makes this code really flexible when different test cases are to be run.

In this study, we have considered execution time and external memory to be cheap resources whereas the core memory was an expensive resource. The execution time ratio 50 is large compared to the theoretical cost, but not that bad compared to finite differences or tangent code. In this study, the size of the gradient is 1000 which makes the gradient cost ratio of 1001 using finite differences and 2600 using the tangent code. Even though the aim of this study was to minimize the core memory requirement, execution time ratios of more than 50 is really difficult to understand. In several (Duval et al., 1996; Malé et al., 1996; Faure and Duval, 1998) other studies where the execution time is to be minimized (without restrictions on core memory), the theoretical ratio could be obtained. The theoretical ratio between the execution times of the adjoint code and the original code is 5 if and only if the original code is a straight line code and the memory hierarchy is assumed to be flat. If the depth of the program is d and if the sub-program checkpointing strategy is chosen, the execution time complexity ratio is $5 + d$ (Faure, 2000). The depth of `Thyc-1D` is 10 which leads to a ratio of 15 between adjoint and original execution times. This theoretical result does not take into account the execution time due to trajectory management. We have profiled the cotangent code to check the coherency between the theoretical and practical results.

5. Analysis of the results

In the latter we investigate the relation between execution time and memory requirement.

Table III and Table IV present the execution time and memory requirements of the three possible executions. The execution **Time** values are expressed in seconds, the memory **Size** component is expressed in

Table II. Execution time and core memory requirement

	$\bar{P}$		Core (Mega bytes)		Time (s)	
	I	C	Total	Ratio	Total	Ratio
`Ckp`	1200	300	36.9	2.6	1030	51.5
`Ckp`	12000	300	36.9	2.6	10213	54.3
`Std`	1200	-	992.0	70.8	1057	52.8

Table III. Memory components

	$\bar{P}$		Size (Mega bytes)				Moves		
	I	C	Total	Run.	Traj.	Check.	Total	Traj.	Check.
`Ckp`	1200	300	276.9	33	3.9	0.8*C	1252.1	1222.1	0.1*I
`Ckp`	12000	300	276.9	33	3.9	0.8*C	12191.0	12161.0	0.1*I
`Std`	1200	-	992.0	33	959.0	-	1345.2	1345.2	-

Mega bytes and the **Moves** component is a non-dimensional value.
Table III presents the memory requirement and memory moves. The **Total** size decomposes in: **Run**time, **Traj**ectory and **Check**point sizes. The **Total** moves decomposes in: **Traj**ectory moves and **Check**point size. The total sizes and moves are the same for `Std` or `Ckp` with I=1200 and C=1200: for the size as well as the number of moves, the increase in `Check` component counterbalance the reduction in `Traj`.
The comparison of `Ckp 1200` and `Ckp 12000` shows that the change of step size in the simulation only affects the number of `Moves`: if I is multiplied by M, the number of moves is also multiplied by M.

Table IV shows the **Total**, **Traj**ectory and **Check**point management executions times. It also shows the execution time due to the supplementary computations of **Solv** and to the derivatives of the **Tab**ulated functions. The **Traj.** plus **Check.** management takes around 50% of the total execution time, the extra computations of **Solv.** is 3% and **Tab.** is 9% of total execution time. As a result, the computation of the original and adjoint variables is around 40% of the total execution time and is nearly 20 times the original computation (instead of 15 in theory).

The two adjoint codes have been run for I=1200, but only the `Ckp` code has been run for I=12000. The characteristics of `Std` for I=12000 can be extrapolated from Tables III and IV . Moreover, the characteristics of the adjoint code of `Thyc-3D` that applies **No** checkpointing at

Table IV. Execution time components

	$\bar{P}$		Time (s)						
	I	C	Total	Tab.	Base.	Recomp.	Solv.	Traj.	Check.
`Ckp`	1200	300	979	96	102	275	28	478	51
`Ckp`	12000	300	10213	963	1102	2844	313	4508	483
`Std`	1200	-	1057	96	101	298	-	562	-

all are also extrapolated. The execution time cost due to the trajectory management is neglected to allow for the comparison of the adjoining methods whatever management is implemented. Table V presents the adjoint code sizes as well as the **Theoretical** execution times. To extrapolate the characteristics, we consider the execution with 12000 iterations to be 10 times more consuming than the execution with 1200 iterations. This is verified by the execution time of `Ckp` and is also verified by the trajectory size using the `Std` or `No` checkpointing methods. The first part of this table summarizes Tables III and IV , and presents theoretical execution time of the adjoint codes that could be run. The second part shows the characteristics one can extrapolate from the first part on the adjoint codes without running them. The trajectory size of the `No` checkpointing code is the number of trajectory moves times the size of the objects moved. 8 bytes is the size of the real used in this application and we consider that most part of the trajectory values are of type real. To extrapolate the theoretical execution time, we sum the components presented in Table IV except for the `Traj.` and `Check.` components. The `Ckp` method execution time is the sum of the `Tab.`, `Base.`, `Recomp.` and `Solv.` components. The `Std` and `No` checkpointing methods do not involve any `Solv.` recomputation and the `No` checkpointing method does not involve `Recomp.`.

The combination of the two checkpoint levels on the adjoint code drastically diminishes the memory requirement (factor of 388) without increasing the execution time that much (factor of 3) compared to no use of checkpoint level.

6. Conclusion and future work

The initial objectives of: running the automatically generated adjoint code of `Thyc-3D` without manual optimization, and limiting the core memory requirement have been achieved. No optimization by hand have been performed on the generated code. Moreover the increase in core

Table V. Extrapolations of adjoint code characteristics

	$\bar{P}$		Size (Mega bytes)		Theoretical time (s)	
	I	C	Total	Ratio	Total	Ratio
`Ckp`	1200	300	277	19	501	25
`Ckp`	12000	300	277	19	5223	28
`Std`	1200	-	992	71	495	25
`Std`	12000	-	9623	687	5045	27
`No`	1200	-	10794	771	197	10
`No`	12000	-	107649	7688	2065	11

memory is bounded by 2.6 times the original requirement. The execution time overhead 52.8 times the original execution time is analyzed component by component to determine which of them can be reduced. The use of expensive divided differences to get precise approximation of the derivatives of the tabulated function is inherent to the original code and cannot be optimized.

The dynamic trajectory management costs a large percentage of the total execution time. Some bufferized static/dynamic trajectory management is currently studied to gain some trajectory management execution time. One other to reduce the trajectory management execution time, is to reduce the number of trajectory moves. The current version of `Odyssée` records all the modified intermediate values as well as all basic block indexes.
It does not analyze the source code to detect the variables to be memorized or not. For example, the memorization of the trajectory has been proven suboptimal on `Thyc-1D` (Faure and Charpentier, 1999). Some (static) trajectory analysis to detect which value are really to be recorded is being studied. A new version of `Odyssée` using this refined analysis is under development and gives a gain of 2/3 on the number of trajectory moves for `Thyc-3D`.

The last component of the execution time that could be reduced is the recomputation of the original function. If no sub-program checkpointing is applied, there is no need for recomputation of the original function, but the trajectory is a lot larger. For `Thyc-3D`, the trajectory would be: 9870 Mega bytes for 1200 iterations, and ten times more for 12000 iterations. This would not increase in execution time because trajectory management execution time is mainly proportional to the number of trajectory moves which is the same in both cases. But this would certainly exhaust the memory and the generated adjoint would not run on the target platform.

In future version of AD-tools, the user should be given a way to define the management of each component of the memory space. In this way, he would control the memory required by its code but also the execution time as shown above. The choice between core and external memory is really dependent on the machines and must be left in the user's control. On some machines like Crays, writing on the disk is really cheap whereas in general it is expensive. Let us consider an application for which the core memory is exhausted. The trade-off between storage of the trajectory and recomputation of these intermediate values must be examined: Moore's Law indicates that the computational power is doubled every 2 or 3 years, whereas the transmission to the disk is not. Under this hypothesis, the replacement of storage by computation seems preferable. We are also working on some way to enable the user to define which variable is really to be recorded. From these chosen variables, the system should be able to introduce automatically the necessary recomputation.

As a conclusion, we recommend the use of AD tools to adjoint developers. Even if the result has to be modified, the gain in debugging is really important. After this automatic phase, the restructuring of the storage (if necessary) is easy to do by hand. At least it is a lot easier than testing the derivative statements one by one in an hand coded adjoint. One can easily optimize the storage by adding a semi-automatic post-processing step. This has been done for generating the adjoint code of Meso-NH (Charpentier and Ghémirès, 2000).

References

Berz, M., C. Bischof, G. Corliss, and A. Griewank (eds.): 1996, *Computational Differentiation: Applications, Techniques, and Tools*. Philadelphia: SIAM.

Briere, E., F. David, P. Rascle, and P.-J. Bonamy: 1993, 'THYC V3.0 : Modélisation et Méthodes Numériques'. Rapport HT-13/92056B, HT-33/92.11B, EDF/DER.

Charpentier, I. and M. Ghémirès: 2000, 'Efficient adjoint derivatives: Application to the atmospheric model Meso-NH'. *Optimization Methods and Software* **13**(1), 35–63.

Duval, C., P. Erhard, C. Faure, and J. Gilbert: 1996, 'Application of the Automatic Differentiation Tool Odyssée to a system of thermohydraulic equations.'. In: J.-A. Désidéri, P. Le Tallec, E. Oñate, J. Périaux, and E. Stein (eds.): *Proc. of ECCOMAS'96*, Vol. Numerical Methods in Engineering'96. pp. 795–802.

Faure, C.: 2000, 'Adjoining strategies for multi-layered programs'. *Optimisation Methods and Software.* To appear.

Faure, C. and I. Charpentier: 1999, 'Comparing automatically generated and hand coded adjoints'. Rapport de recherche 3811, INRIA.

Faure, C. and C. Duval: 1998, 'Automatic Differentiation for Sensitivity Analysis. A test case.'. In: K. Chan, S. Tarantola, and F. Campolongo (eds.): *Proceedings of Second International Symposium on Sensitivity Analysis of Model Output (SAMO'98)*, Vol. 17758.

Faure, C. and Y. Papegay: 1998, 'Odyssée User's Guide. Version 1.7'. Rapport technique 0224, INRIA.

Giering, R.: 1997, 'Tangent linear and Adjoint Model Compiler , Users manual'. Unpublished, available from `http://puddle.mit.edu/~ralf/tamc`.

Griewank, A.: 1992, 'Achieving logarithmic growth of temporal and spatial complexity in Reverse Automatic Differentiation'. *Optimization Methods and Software* **1**, 35–54.

Griewank, A.: 2000, *Principles and Techniques of Algorithmic Differentiation.* SIAM.

Griewank, A. and G. Corliss (eds.): 1991, *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* Philadelphia: SIAM.

Griewank, A., D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther: 1996, 'ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++'. *ACM TOMS* **22**, 131–167.

Griewank, A. and A. Walther: 1997, 'Treeverse: An implementation of the checkpointing for the reverse or ajoint mode of differentiation'. Technical report, TU Dresden.

Grimm, J., L. Pottier, and N. Rostaing-Schmidt: 1996, 'Optimal time and minimum space-time product for reversing a certain class of programs'. In: M. Berz, C. Bischof, G. Corliss, and A. Griewank (eds.): *Computational Differentiation: Applications, Techniques, and Tools.* pp. 95–106.

Jouhanique, T. and P. Rascle: 1995, 'A fifth equation to model the relative velocity in the 3-D thermal-hydraulic code THYC.'. In: R. Block and F. Feiner (eds.): *Proceedings of the sixth Nuclear Reactor Thermal-hydraulic (NURETH) Congress.*

Malé, J.-M., N. Rostaing-Schmidt, and N. Marco: 1996, 'Automatic Differentiation: an Application to Optimum Shape Design in Aeronautics'. In: J.-A. Désidéri, C. Hirsch, P. Le Tallec, E. Oñate, M. Pandolfi, J. Périaux, and E. Stein (eds.): *Minisymposia of ECCOMAS 96.*

Volin, Y. and G. Ostrovskii: 1985, 'Automatic Computation of derivatives with the use of the multilevel differentiating technique - 1. Algorithmic basis'. *Computers & Mathematics with Applications* **11**(11), 1099–1114.