Copyright information to be inserted by the Publishers

ADJOINING STRATEGIES FOR MULTI-LAYERED PROGRAMS

CHRISTÈLE FAURE

INRIA Sophia Antipolis, 2004 route des Lucioles,
BP 93, F-06902 Sophia-Antipolis Cedex, France.
Christele.Faure@sophia.inria.fr,Christele.Faure@polyspace.com

Some papers present the rules to apply to a straight line program to differentiate it in reverse mode, as well as theoretical complexity measures. This paper recalls these rules, but also summarises the different possible strategies for the differentiation of a multi-level program in reverse mode. We focus on the reverse mode, because the computation of derivatives in reverse order (w.r.t. the computation of original variables) makes the problem much more complicated. A lot of strategies can be applied to generate an adjoint code: these strategies are applied within hand-coded discrete adjoints or within automatically generated adjoints. But they are not necessarily known by both communities, this is why we describe several of them in this paper. Until now, the comparison of these strategies on a code is difficult because no complexity measure has been associated to them. This paper presents complexity measures of the adjoint codes generated using these strategies in terms of execution time and memory requirement. These complexities are based on some elementary measures such as execution time and memory requirement of their sub-programs. For sake of simplicity, in a first phase we neglect the memory management within time costs. In a second phase, we explain how the extra cost first neglected can be accommodated for in the complexity measures.

KEY WORDS: Automatic Differentiation, Computational Differentiation, Complexity, Adjoint code, Reverse mode, Forward mode

1 MOTIVATION

Automatic Differentiation [2, 13] covers a set of techniques for computing derivatives at arbitrary points. Currently, two modes of Automatic Differentiation are studied: the direct (or forward) mode that computes the derivatives and the original values simultaneously, and the reverse (or backward) mode that computes first the original values and second the derivatives in reverse order. The reverse mode is especially efficient for computing gradients of scalar-valued functions because its cost is independent of the number of inputs. Two classes of automatic differentiation tools exist: those working by code generation and those working by operator overloading. `Odyssée`, `Adifor`, `GRESS`, and `TAMC` belong to the first class of automatic differentiation tools based on code generation, whereas `ADOL-C` belongs to the second one.

Several papers [1, 10, 16, 18] present the rules to apply to a straight line program to differentiate it in reverse mode, as well as theoretical complexity measures. This paper recalls these rules, but also summarises the different possible strategies for the differentiation of a multi-level program in direct or reverse mode. The strategy preferred in direct mode is straightforward and can be directly extended from the straight line case. For the reverse mode, the computation of derivatives in reverse order (w.r.t. the computation of original variables) makes the problem much more complicated. We show that many strategies [17] can be applied between storing recursively the variables and recomputing them all from the initial point. These strategies sketched in [12] are applied within hand-coded discrete adjoints or within automatically generated adjoints.

In this paper, we summarise the main approaches used to make the reverse mode of automatic differentiation applicable on real world codes. Some techniques are implemented in automatic differentiation tools (`TAMC`, `Odyssée`), but others are only used to write discrete adjoint codes by hand. They are not necessarily known in both communities. Some of them have been described in papers previously published but they where not associated with complexity measures and this is why they are really difficult to compare. We describe all the strategies in a uniform way and associate to each of them complexity measures for execution time and memory requirement. Looking at these measures they compare easily, even though it is often still difficult to know which of them to choose for adjoining a given original code.

Section 2 describes the basics of automatic differentiation. Section 3 is dedicated to a formal description of program executions associated with elementary complexity indicators and a graphical representation of a program we are using in the rest of the paper. Section 4 describes different possible strategies applicable on a real code and shows their associated complexity. These measures are theoretical in the sense that they only take into account components of the complexity which depend only on the source code and are independent from the platform on which the adjoint code is run. Section 5 gives some complementary information on the strategies depending on the machine: number of sub-programs generated, execution time due to calls to sub-programs and memory management. In Section 6, we project future research directions for making the use of adjoint code easier and generated codes more efficient.

2 PRINCIPLES OF AUTOMATIC DIFFERENTIATION

Automatic (or computational) differentiation is based on two observations. First, any statement run on a computer can be seen as an elementary function using simple operations and a program is a composition of those elementary functions. Second, a program can be differentiated as a composition of functions using the chain rule.

Automatic Differentiation can generate two kinds of derivative codes: a code for computing the product of the Jacobian matrix by one (or more) direction(s), which is called the tangent code, or a code for computing the product of the transposed

```
Z = X * Y**2
```

(a) code A

$$\begin{array}{ccc} \mathbb{R}^2 & \to & \mathbb{R} \\ (x,y) & \to & x*y^2 \end{array}$$

(b) f

$$\begin{array}{ccc} \mathbb{R}^3 & \to & \mathbb{R}^3 \\ (x,y,z) & \to & (x,y,x*y^2) \end{array}$$

(c) $\mathcal{F}$

FIGURE 1: An assignment A seen as an elementary function $\mathcal{F}$

$$\begin{pmatrix} dX \\ dY \\ dZ \end{pmatrix}_o := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Y^2 & 2XY & 0 \end{pmatrix} \begin{pmatrix} dX \\ dY \\ dZ \end{pmatrix}_i$$

(a)

$$\begin{array}{lcl} dX_o & := & dX_i \\ dY_o & := & dY_i \\ dZ_o & := & Y^2 dX_i + 2XY dY_i \end{array}$$

(b)

FIGURE 2: Product of the Jacobian matrix of A by the direction $(dX, dY, dZ)_i$

Jacobian matrix with one (or several) dual directions, which is called the cotangent code or "adjoint code". The terminology adjoint code is used in communities where the derivative code is written by hand, while it is called cotangent code in the automatic differentiation community.

In the next sections, we explain how the chain rule is used to generate tangent or cotangent codes. First, we show how to deal with a single statement and afterwards with a straight line program.

2.1 Differentiation of a single assignment

The differentiation of one assignment can be deduced using the idea that any statement can be seen as an elementary function.

For example, the assignment A shown in Fig. 1(a) can be seen as the mathematical elementary function f shown in Fig. 1(b). The mathematical input and output domains of f are respectively $\mathbb{R}^2$ and $\mathbb{R}$.
In order to built up the composition of the elementary functions corresponding to a sequence of statements, one has to extend the input and output domains. The definition of $\mathcal{F}$ shows in Fig. 1(c) the extension of f to $\mathbb{R}^3 \to \mathbb{R}^3$. This leads to consider all the variables involved in the computation as potential input and output.

Using the definition $\mathcal{F}$ of the code A, the product of the Jacobian matrix of A by some direction $(dX, dY, dZ)_i$ can be easily built. Fig. 2 shows two equivalent representations of this computation using mathematical notation: Fig. 2(a) shows the matrix computation ,and Fig. 2(b) shows the equivalent scalar computation, where the subscripts $_i$ and $_o$ indicate input and output, respectively.

The product of the transposed Jacobian matrix of $\mathcal{F}$ by the direction in the dual space $(dX^\star, dY^\star, dZ^\star)_i$ can also be easily built. Fig. 3 shows two equivalent representations of this computation in the same manner as Fig. 2.

$$\begin{pmatrix} dX^\star \\ dY^\star \\ dZ^\star \end{pmatrix}_o := \begin{pmatrix} 1 & 0 & Y^2 \\ 0 & 1 & 2XY \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} dX^\star \\ dY^\star \\ dZ^\star \end{pmatrix}_i \qquad \begin{aligned} dX^\star_o &:= dX^\star_i + Y^2 dZ^\star_i \\ dY^\star_o &:= dY^\star_i + 2XY dZ^\star_i \\ dZ^\star_o &:= 0. \end{aligned}$$

(a) (b)

FIGURE 3: Product of the transposed Jacobian matrix of A by $(dX^\star, dY^\star, dZ^\star)_i$

```
dZ = Y**2 * dX + 2*X*Y*dY
```

(a) $J_{\mathcal{F}} * d$

```
dX = dX + Y**2*dZ
dY = dY + 2*X*Y*dZ
dZ = 0.
```

(b) $J^\top_{\mathcal{F}} * d^\star$

FIGURE 4: Derivatives of the assignment A

From the two scalar computations shown in Fig. 2(b) and Fig. 3(b), one can simply get the corresponding tangent and cotangent straight line codes. We introduce the subscripts $_i$ and $_o$ because a mathematical variable can be set but can never be modified in place. In contrast, for computers, variables are memory locations and can therefore be modified. In order to optimise the code in terms of the number of intermediate variables, the variables X_i and X_o are identified and denoted by X. Therefore, the statement $dX_o = dX_i$ is transformed into $dX = dX$ and discarded. Fig 4(a) and Fig. 4(b) show respectively the optimised tangent code and the optimised cotangent code of A.

The only values of the original code necessary for the computation of the cotangent code are the elements of the Jacobian matrix. One can choose either to get these partial derivatives directly or to recompute them from the values of the original variables. Under the second strategy, the partial derivatives must be evaluated on correct input, so in the context of a real program, the computed state reached by the original code must be reproduced before the evaluation of the Jacobian.

2.2 Differentiation of a straight line code without re-assignment

We said in the previous sections that one may consider any statement of a straight line program as an elementary function and a straight line program as a composition of functions. In this section, we show how to use the chain rule to generate the tangent and cotangent codes of a straight line program.

First we recall that, using the chain rule, if $\mathcal{F}$ is the composition of n elementary functions $\mathcal{F} = \mathcal{F}_n \circ \cdots \circ \mathcal{F}_2 \circ \mathcal{F}_1$, its Jacobian matrix J is the product of the n elementary Jacobian matrices $J = J_n \cdots J_2 J_1$, where J_i denotes $\mathrm{J}(\mathcal{F}'_i)$ the Jacobian matrix of $\mathcal{F}_i$ computed at the point $(\mathcal{F}_{i-1} \circ \ldots \circ \mathcal{F}_1)(x)$. The transposed Jacobian

```
Z = X * Y**2
W = Z**2 * Y
```

(a) G

$$\begin{array}{lcl} \mathbb{R}^4 & \to & \mathbb{R}^4 \\ (x,y,z,w) & \to & (x,y,x*y^2,w) \end{array}$$

(b) $\mathcal{G}_1$

$$\begin{array}{lcl} \mathbb{R}^4 & \to & \mathbb{R}^4 \\ (x,y,z,w) & \to & (x,y,z,y*z^2) \end{array}$$

(c) $\mathcal{G}_2$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ Y^2 & 2XY & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(d) $J_{\mathcal{G}_1}$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & Z^2 & 2YZ & 0 \end{pmatrix}$$

(e) $J_{\mathcal{G}_2}$

FIGURE 5: Applying the first principle on G

matrix of $\mathcal{F}$ denoted by $J^\top$ is the product in reverse order of the n transposed elementary Jacobian matrices: $J^\top = J_1^\top J_2^\top \cdots J_n^\top$.

Fig. 5(a) shows a toy straight line program G of two statements without re-assignment of variable. The mathematical function $\mathcal{G}$ implemented in G can be seen as the composition of the two elementary functions $\mathcal{G}_1$ and $\mathcal{G}_2$ (see Fig 5(b) and Fig. 5(c)) $G = \mathcal{G}_2 \circ \mathcal{G}_1$. As shown in the previous section, it is easy to get the two elementary Jacobian matrices $J_{\mathcal{G}_1}$ and $J_{\mathcal{G}_2}$ (see Fig 5(d) and Fig. 5(e)) of $\mathcal{G}_1$ and $\mathcal{G}_2$.

We denote by $J = J_{\mathcal{G}_2} J_{\mathcal{G}_1}$ the Jacobian matrix of $\mathcal{G}$ and d ($d^\star$) the direction (respectively dual direction). The generation of tangent and cotangent codes of G_1 and G_2 can be performed as described in the previous section and lead to the code shown in Fig. 6.

Now, we want to built up the product of the two Jacobian-vector products using the chain rule. From the chain rule, we know that the composition of functions leads to the product of matrices. We know also that we want to generate a code that computes only Jacobian vector products, so we generate a code that splits $d_o := J_2 J_1 d_i$ into two Jacobian vector products $d_1 := J_1 d_i$ and then $d_o := J_2 d_1$.

The last problem is to generate a code that computes J_1 and J_2 at the correct points i_0 and i_1, where i_0 is the state before the call to $\mathcal{G}_1$ and $i_1 = \mathcal{G}_1(i_0)$. In this section, we show how to deal with straight line codes without re-assignment, so i_0 is not modified by the computation of G, and i_1 is computed by G_1 but not modified by g_2. Therefore, one can compute i_0, i_1 before the computation of the adjoint derivatives. The computation of $i_2 = \mathcal{G}_1(i_1)$ is not necessary but cannot lead to errors, so one can add after the straight line program G the new straight

```
dZ = Y**2 * dX + 2*X*Y * dY
```

(a) $d_o := J_{\mathcal{G}_1} d_i$

```
dW = Z**2 * dY + 2*Y*Z * dZ
```

(b) $d_o := J_{\mathcal{G}_2} d_i$

```
dX = dX + Y**2 * dZ
dY = dY + 2*X*Y * dZ
dZ = 0.
```

(c) $d_o^\star := J_{\mathcal{G}_1}^\top d_i^\star$

```
dX = dX + Z**2 * dW
dY = dY + Z**2 * dW
dZ = dZ + 2*Y*Z * dW
dW = 0.
```

(d) $d_o^\star := J_{\mathcal{G}_2}^\top d_i^\star$

FIGURE 6: Derivatives of G_1, G_2

```
Z = X * Y**2
W = Z**2 * Y

dZ = Y**2 * dX + 2*X*Y * dY
dW = Z**2 * dY + 2*Y*Z * dZ
```

(a) $J_{\mathcal{G}} * d$

```
Z = X * Y**2
W = Z**2 * Y

dX = dX + Z**2 * dW
dY = dY + Z**2 * dW
dZ = dZ + 2*Y*Z * dW
dW = 0.

dX = dX + Y**2 * dZ
dY = dY + 2*X*Y * dZ
dZ = 0.
```

(b) $J_{\mathcal{G}}^\top * d^\star$

FIGURE 7: Derivatives of G

```
Y = X * Y**2
Y = Y**3 * X**2
```

(a) $\mathcal{H}$

$$\begin{array}{ccc} \mathbb{R}^2 & \to & \mathbb{R}^2 \\ (x,y) & \to & (x, x*y^2) \end{array}$$

(b) $\mathcal{H}_1$

$$\begin{array}{ccc} \mathbb{R}^2 & \to & \mathbb{R}^2 \\ (x,y) & \to & (x, y^3*x^2) \end{array}$$

(c) $\mathcal{H}_2$

$$\begin{pmatrix} 1 & 0 \\ Y^2 & 2XY \end{pmatrix}$$

(d) $J_{\mathcal{H}_1}$

$$\begin{pmatrix} 1 & 0 \\ 2XY^3 & 3X^2Y^2 \end{pmatrix}$$

(e) $J_{\mathcal{H}_2}$

```
dY = Y**2 * dX + 2*X*Y * dY
Y = X*Y**2

dY = 2XY**3 * dX + 3Y**2X**2 * dY
Y = Y**3*X**2
```

(f) $J_{\mathcal{H}} * d$

FIGURE 8: Applying the first principle on $\mathcal{H}$

line program $G' = d_1 := J_1 d_i; d_o := J_2 d_1$ as shown in Fig. 7.

2.3 Differentiation of a straight line code with re-assignment

In this section, we show how to differentiate a straight line code with some re-assignment of variables.

If we name $J_{\mathcal{H}}$ the Jacobian matrix of $\mathcal{H}$, $J_{\mathcal{H}} * d$ and $J_{\mathcal{H}}^{\top} * d^{\star}$ are computed using the chain rule in the same way as for the previous example (see Fig. 8). But in this example, the variable `Y` is assigned twice and that makes a big difference in the way the resulting code is written.

We name (X_0, Y_0) the initial value of the point `X` and `Y`, and (X_1, Y_1) its value after the first assignment $(X_1, Y_1) = (X_0, X_0 * Y_0^2)$. In this example, $J_{\mathcal{H}_1}$ must be evaluated on the point (X_0, Y_0), and $J_{\mathcal{H}_2}$ must be evaluated on (X_1, Y_1).

In the tangent code, we insert the statement that computes (X_1, Y_1) before the computation of $J_{\mathcal{H}_2} * d$ but after the computation of $J_{\mathcal{H}_1} * d$. In order to get a general method for getting the tangent code, we choose to insert the original statement after the derivative statement. The tangent code of $\mathcal{H}$ using this rule

is shown in Fig. 8(f). For this example, the last statement `Y=Y**3 * X**2` is not used for the computation of $J_{\mathcal{H}} * d$.

In the cotangent code, the first Jacobian matrix to be evaluated is $J_{\mathcal{H}_2}^{\top} * d^{\star}$, and the second is $J_{\mathcal{H}_1}^{\top} * d^{\star}$. The code to be written must evaluate $J_{\mathcal{H}_2}^{\top} * d^{\star}$ on $(X, Y) = (X_1, Y_1)$ and $J_{\mathcal{H}_1}^{\top} * d^{\star}$ on $(X, Y) = (X_0, Y_0)$. For this example, the statements `Y=Y**3*X**2` as well as the storage associated to it are unnecessary for the evaluation of $J_{\mathcal{H}}^{\top} * d^{\star}$.

The necessary information can be provided in at least three ways: the adjoint code can store the components of the Jacobian matrix, store the values of the variables which are modified, or recompute the values of the modified variables. The three resulting adjoint codes are presented in Fig. 9.

In the sample example, one can choose to store the value of `Y` before the first assignment to the intermediate variable `S` and to restore this value before the computation of $J_{\mathcal{H}_1}^{\top} * d^{\star}$. In order to get a general method for getting the cotangent code, we choose to store any computed variable in the code of $\mathcal{H}$ and to restore it before each Jacobian matrix evaluation. The cotangent code of $\mathcal{H}$ using this store/restore rule is shown in Fig. 9(a). For this example, the three statements `Y=Y**3 * X**2`, and the `S1=Y` and `Y=S1` are not used for the computation of $J_{\mathcal{H}}^{\top} * d^{\star}$.

One can alternatively store the values of the non zero partial derivatives $\frac{\partial Y}{\partial X}$ and $\frac{\partial Y}{\partial Y}$, within `dYdX1` and `dYdY1` before the execution of the first statement, and `dYdX2,dYdY2` before the execution of the second statement. From these values, the partial derivatives can be accessed easily to compute the Jacobian matrix vector product as shown in Fig. 9(b).

The other method to evaluate $J_{\mathcal{H}_2}^{\top} * d^{\star}$ and $J_{\mathcal{H}_1}^{\top} * d^{\star}$ at the correct points is to recompute the correct values of `X` and `Y` from their original value (X_0, Y_0) instead of storing/restoring the intermediate results. In order to show the general method for getting the cotangent code, we choose to save the values of the point before the computation of $\mathcal{H}$ even if some components are not modified. The cotangent code of $\mathcal{H}$ using this recomputation rule is shown in Fig. 9(c).

On this toy example, we have shown that getting an efficient tangent code is simple even though it is error prone, to write it by hand. However we have also shown clearly the difficulties of getting an efficient cotangent code even on this small example. In this case, using general rules without optimisation leads to a lot of useless computations and/or extra memory cost. One cannot rely on the compiler to verify that the values of a variable `X` stored are not necessary because they have not been modified since the previous store statement. As far as possible, the optimisations of the adjoint code should be performed at differentiation time.

2.4 Conclusion

In the previous sections, the basic ideas of automatic differentiation have been recalled. We have shown how to differentiate straight line programs, but real world codes typically include many complex statements `do loops, branches, sub-program calls ...` One can convince oneself that even for complex statements, the tangent code is easy to derive from the previous sections. The cotangent

```
S0 = Y
Y = X*Y**2
S1 = Y
Y = Y**3*X**2

Y = S1
dX = dX + 2XY**3 * dY
dY =   3Y**2X**2 * dY

Y = S0
dX = dX + Y**2 * dY
dY =       2YX * dY
```

(a) $J_{\mathcal{H}}^{\top} * d^{\star}$

```
dYdX1 = Y**2
dYdY1 = 2*X*Y
Y = X*Y**2

dYdX2 = 2XY**3
dYdY2 = 3Y**2X**2
Y = Y**3*X**2

dX = dX + dYdX2 * dY
dY =      dYdY2 * dY

dX = dX + dYdX1 * dY
dY =      dYdY1 * dY
```

(b) $J_{\mathcal{H}}^{\top} * d^{\star}$

```
YY = Y
XX = X

Y = X*Y**2
Y = Y**3*X**2

Y = YY
X = XX
Y = X*Y**2
dX = dX + 2XY**3 * dY
dY =   3Y**2X**2 * dY

Y = YY
X = XX
dX = dX + Y**2 * dY
dY =       2YX * dY
```

(c) $J_{\mathcal{H}}^{\top} * d^{\star}$

FIGURE 9: Cotangent codes of $\mathcal{H}$

mode seems clearly much more difficult to apply on complex statements. In the rest of this paper, we therefore only consider the reverse mode of AD.

As shown above on straight-line codes, within an adjoint code, various strategies can be applied to evaluate the partial derivatives at correct values of the original variables. The three strategies presented above are: the non-zero partial derivatives are stored (shown in Fig. 9(b)), the variables modified are stored (shown in Fig. 9(a)), and the values of the variables are recomputed from the input values (shown in Fig. 9(c)). The values stored within this phase form the trajectory.

Treating a real code instead of a straight line program, a new problem arises: how to transpose the original flow graph to combine the adjoint variables correctly. This problem is solved by storing a trace of the execution path and to traverse it backward. The corresponding values belong to what we call the trajectory. Different strategies for doing this can be implemented, as we describe in Section 5.2.

In the next sections, we describe the existing strategies in terms of method and complexity measure.

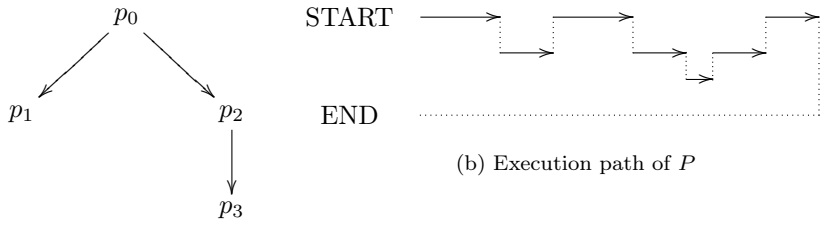
3 DESCRIPTION OF A PROGRAM EXECUTION

Let us look at the differentiation of a program that decomposes into several sub-programs. We choose not to give a definition of a sub-program: it can be either a separate unit in the original source code, a sub-tree consisting of some separate units, or a sequence of statements. One can think of a program as a straight line code where instances of common sequences of statements are represented by sub-program calls. In order to describe clearly the different possible combinations of storage and recomputation, we need program execution descriptions. As we want also to evaluate the complexity of the generated code, elementary complexity measures are required. In this section, we introduce the terminology used throughout this paper.

3.1 Representations of a program execution

In this paper, we use two descriptions of a program execution: one is the **call tree** and the second is the **execution path**. Both representations are known at run time, which means that several call tree/execution path pairs may be associated with a program depending on its inputs. In the call tree, each node represents an instance of a sub-program of the source, and each arrow links a sub-program with a sub-program it actually calls. We take into account instances of sub-programs: if a sub-program is called twice in the program we replicate the corresponding node. The execution path shows the different pieces of code really used during one execution of a program.

For example, Fig. 10(a) shows the call tree of one execution E of a program P made of four sub-programs $p_0, \cdots p_3$. To read this call tree, one must start from p_0 and see that it calls first p_1 followed by p_2 and that p_2 calls p_3. Fig. 10(b) shows the execution path of E. To read the execution path, one must follow the arrow



(a) Call tree of P

(b) Execution path of P

FIGURE 10: Descriptions of one execution of P

that goes from the label $START$ to the label END. The dotted line stands for calling and returning from a sub-program.

In the call tree shown Fig. 10(a), one can see that each sub-program is represented by its name whereas in the execution path (shown in Fig. 10(b)) it is split into line segments representing pieces of code. For example, the code of p_0 is split into three parts, one before the call to p_1, one between the call to p_1 and p_2, and one after the call to p_2. The code of p_2 is also split into two pieces due to the call to p_3.

3.2 Complexity of a program execution

Let P be a program execution and $\{p_i | i = 1, N\}$ be the set of sub-programs it calls. For each node p_i in P, we define three elementary measures denoted by: *(i)* d_i the depth of p_i, *(ii)* t_i the execution time of p_i without its descendants in the call tree, *(iii)* and m_i the number of memory locations necessary to run p_i without its descendants in the call tree. From those elementary measures $\{(d_i, m_i, t_i) | i = 1, N\}$, the complexity of P are computed.

We define the sequential sum of an elementary measure e over P to be:

$$\mathrm{SSum}\{e_i\} = \mathrm{ssum}(p_0, e_0) = e_0 + \sum_{j \in \mathrm{Sons}(p_0)} \mathrm{ssum}(p_j, e_j)$$

and the parallel sum to be:

$$\mathrm{MSum}\{e_i\} = \mathrm{msum}(p_0, e_0) = e_0 + \max_{j \in \mathrm{Sons}(p_0)} \mathrm{msum}(p_j, e_j)$$

where $\mathrm{Sons}(p_i)$ is the set of sons of p_i in the call tree of P. The execution time is the sum over all the sub-programs p_i in P of the elementary execution times t_i. The memory requirement of P is the maximum sum over all the branches of P (in the call tree) of the elementary memory requirements m_i.

$$M = \mathrm{MSum}\{m_i\} \quad (1)$$
$$T = \mathrm{SSum}\{t_i\}. \quad (2)$$

In our example shown in Fig. 10, the depth of all the sub-programs $\{p_i, i = 0, 3\}$ of P are respectively $d_0 = 0, d_1 = 1, d_2 = 1$ and $d_3 = 2$. The memory requirement M and execution time T of the execution P are:

$$M = m_0 + \max(m_1, m_2 + m_3) \quad (3)$$
$$T = t_0 + t_1 + t_2 + t_3. \quad (4)$$

3.3 Components of the derivative code

In the previous section, we have shown various strategies for the generation of the cotangent code of a straight line code. In order to describe those strategies on a whole program in a coherent manner, we choose to introduce some concepts with the corresponding notations.

First, we separate the two different components of the cotangent code of a straight line code. Fig. 11 shows the two components of the cotangent code of $\mathcal{H}$ (see source code in 8(a)) using two strategies; $\mathcal{H}_1^s$ and $\mathcal{H}_1'^r$ are the two components of the cotangent code of $\mathcal{H}$ using the **all storage** strategy, $\mathcal{H}_2^s$ and $\mathcal{H}_2'^r$ are the two components of the cotangent code using the **all recomputation** strategy. One can see from this toy example, that using the **all storage** strategy leads to the storage of 3 scalar values and only one execution of each original statement whereas the **all recomputation** strategy leads to the storage of only 2 values but `n-i` executions of statement of index `i` in the straight line code of length `n`. These components can be optimised by noticing first that the final value of `Y` is never used and its computation can be discarded. Using the **all storage** strategy the `store (3,Y)` can be also discarded.

The forward component computes the original function and stores the trajectory, and the backward component restores this trajectory and computes the derivatives. Those two components can be written using several storage/recomputation strategies, the only constraint is that if the context of execution of the original function is restored, then the execution of a forward component followed by the execution of the backward component of a function computes the cotangent values of this function. Those components must share only a common space where the values of the variables are stored in (`store`) and retrieved from (`retrieve`). These basic commands can be implemented using arrays, stacks, files, or any other data structure.

Now we generalise this notion of component to a sub-program in order to explain the different strategies that can be applied at the call tree level to get the cotangent code of a program. We decide to treat a sub-program as a whole (not to split it) which means that each sub-program is associated with one forward component and one reverse component. If p_i is the name of a sub-program, we denote by p_i^s the forward component of its cotangent code and by $p_i'^r$ the reverse component. The sub-program p_i^s computes the same output as p_i and stores the trajectory (modified variables, partial derivatives ...), and the sub-program $p_i'^r$ restores this information and computes the gradient. We denote by m_i' the memory necessary for the storage of the values of the derivative variables. The adjoint code computes one gradient,

```
store (1,Y)
Y = X*Y**2
store (2,Y)
Y = Y**3*X**2
store (3,Y)
Y = X*Y
```

(a) $\mathcal{H}_1^s$

```
retrieve (3,Y)
dX = dX + Y * dY
dY =      X * dY

retrieve (2,Y)
dX = dX + 2XY**3 * dY
dY =   3Y**2X**2 * dY

retrieve (1,Y)
dX = dX + Y**2 * dY
dY =      2YX * dY
```

(b) ${\mathcal{H}_1'}^r$

```
store (1,Y)
store (2,X)
Y = X*Y**2
Y = Y**3*X**2
Y = X*Y
```

(c) $\mathcal{H}_2^s$

```
retrieve (1,Y)
retrieve (2,X)
Y  = X*Y**2
Y  = Y**3*X**2
dX = dX + Y * dY
dY =      X * dY

retrieve (1,Y)
Y  = X*Y**2
dX = dX + 2XY**3 * dY
dY =    3Y**2X**2 * dY

retrieve (1,Y)
dX = dX + Y**2 * dY
dY =      2YX * dY
```

(d) ${\mathcal{H}_2'}^r$

FIGURE 11: $\mathcal{H}^s$ and ${\mathcal{H}'}^r$

each variable in p_i is associated with at most one derivative variable of the same size, this means that:

$$\forall i \quad m_i + m'_i \quad \leq \quad 2 * m_i. \tag{5}$$

We denote by t_i^s the execution time of p_i^s and by $t_i'^r$ the execution time of $p_i'^r$. We consider the execution time due to the storage or the retrieval of the information to be null. Under this hypotheses, we get that ${t_i}^s = t_i$ and ${t_i'}^r = t'_i$. Moreover, the theoretical result that the ratio [1, 16] between the operations count of one gradient and the original function and the execution time of the function is lower than 5 (5 stands for the case when only divisions are computed) applies. The operations count translates into execution time if no optimization is performed at compile time. This theoretical ratio appears to be the worst case ratio for execution time. It is then possible to deduce that:

$$\forall i \quad {t_i}^s + {t_i'}^r \quad \leq \quad 5 * t_i. \tag{6}$$

The cotangent code of the code example P should execute: the reverse parts in reverse order ${p'_3}^r; {p'_2}^r; {p'_1}^r; {p'_0}^r$ and the direct part ${p_0}^s; {p_1}^s; {p_2}^s; {p_3}^s$ in direct order. The only constraint on the sequence of execution of the forward parts w.r.t. reverse part is that each forward part ${p_i}^s$ must be run before the corresponding reverse part ${p_i'}^r$. In the following section we show different strategies to generate the cotangent code and give the corresponding complexity measures.

4 THEORETICAL COMPLEXITIES

In this section, we compute the complexity in execution time and memory requirement of the cotangent code of one execution of a program P. In order to extrapolate the complexities of one execution path of P to all execution paths of a program P, one has to consider all the possible executions of P.

The results described in this section are only correct under the hypothesis that *(1)* the same strategy is applied to all the sub-programs of the program, *(2)* the memory management does not cost any execution time, and *(3)* a call to a sub-program does not cost any time either. In Section 5.3, we take these components of the complexity into account to give more realistic measures.

4.1 Extreme strategies

The first idea for getting a cotangent code from a program is to consider this program globally and to apply the **all storage** strategy or the **all recomputation** strategy. We name those strategies "extreme" because they do not do any compromise between storage and recomputation.

Using the **all recomputation** strategy, the computation of the point at which to run each sequence $p_i^s; {p_i}'^r$ is performed by running again the path from p_0 to the call to p_i in the original execution path.

We name $P'r$ the adjoint code generated from P using the **all recomputation** strategy. The call tree and execution path of $P'r$ are shown in Fig 12(a) and Fig. 12(b), respectively. In Fig. 12(b), the icon $\bullet$ means storage of the context of the sub-program, and $\circ$ means the retrieval of this context. The symbol $\Rightarrow$ means execution of the function with storage, and $\Leftarrow$ means retrieve of the stored trajectory and computation of the derivatives.

Using the **all recomputation** strategy, the storage of basic information is performed recursively on each branch, and the execution of ${p_i}'^r$ follows directly the one of p_i^s. We denote by l_i the memory amount required to store the trajectory of p_i whatever basic information is chosen: all the instances of all the variables directly modified by p_i, or all the partial derivatives necessary to compute the intermediate adjoint variables of p_i.

If we denote by in_0 the size of the input context of p_0, the amount of storage necessary for the computation of the derivative is:

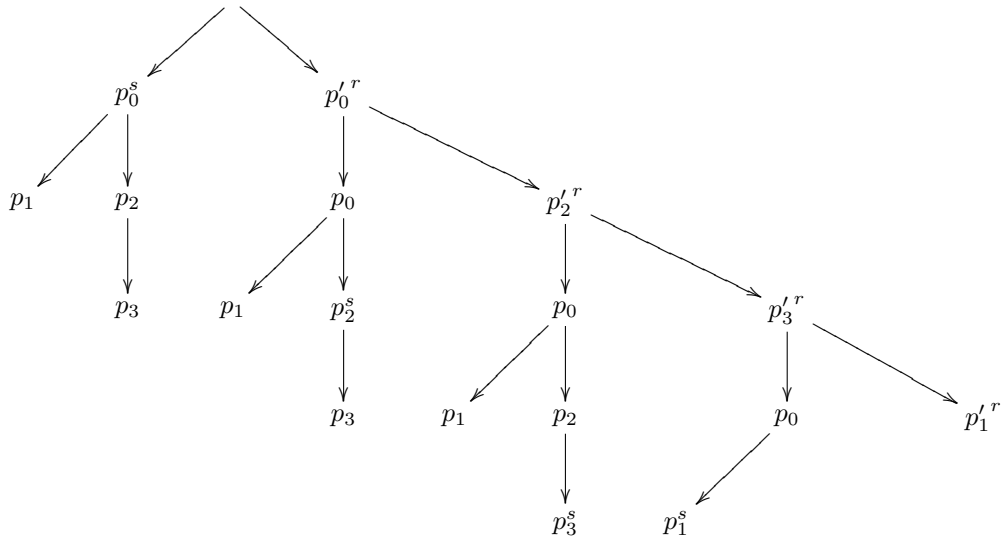
$$\begin{array}{rcl} M' & = & \mathrm{MSum}\{m_i + l_i + m'_i\} \\ & \leq & 2 * M + \mathrm{MSum}\{l_i\} \end{array} .$$

Even if the memory required is minimal, it is easy to figure out that the growth in terms of execution time is quadratic in the number of sub-programs run in E. The exact cost in execution time is difficult to formulate because in this strategy the recomputation of each sub-program p_i is split. For example p_0 is split into three parts: the last part is never recalled, the second part is recalled two times and the first part is recalled three times. We choose to over-approximate the execution time of one part of the sub-program p_i by the total execution time t_i of p_i. Under this approximation, one recomputation of a sub-program p_i is necessary to generate a correct context for each sub-program p_j executed after p_i in E which is $N - i$ times in total. However p_i is also recomputed when its ancestors in the call-tree are recomputed, which gives a number of times equal to its depth d_i. In the example, the extra cost in terms of recomputation of the sub-program p_0 is $3 * t_0$.

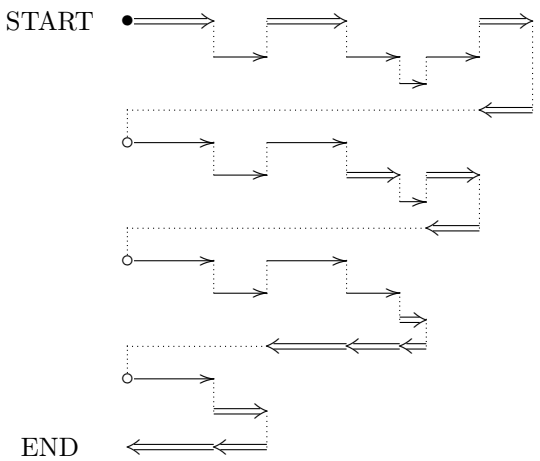
$$\begin{array}{rcl} T' & \leq & \mathrm{SSum}\{{t_i}^s + {t'_i}^r + (N - i + d_i) * t_i\} \\ & \leq & 5 * T + \mathrm{SSum}\{(N - i + d_i) * t_i\}. \end{array} .$$

Using the **all storage** strategy, the execution of the cotangent code $P's$ of P is made of the execution of the forward component of $P's$ followed by the reverse component of $P's$. This means first storing the trajectory all along the execution path of the forward component recursively on the call-tree and second restoring those values all along the execution path of the reverse component of $P's$. This strategy is used in meteorology to write adjoint codes by hand but has not been implemented in AD-tools because of the large memory required to store the original trajectory without optimisations.

The execution path of $P's$ shown in Fig. 13(b) consists of two parts: the first one is the same path as E where each call to p_i is replaced by a call to p_i^s, the second one is the reverse path of E where each call to p_i is replaced by a call to ${p_i}'^r$.

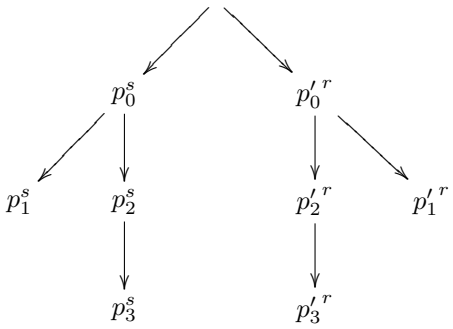


(a) Call tree of $P'r$

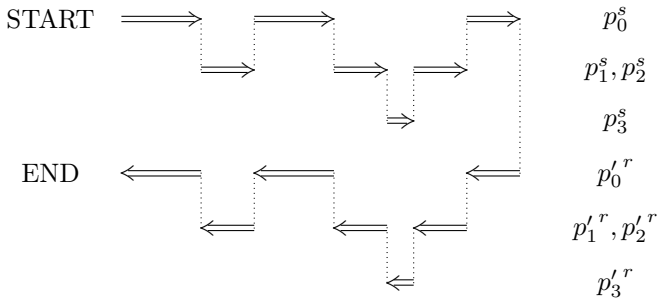


(b) Execution path of $P'r$

FIGURE 12: **all recomputation** strategy



(a) Call tree of $P's$



(b) Execution path of $P's$

FIGURE 13: **all storage** strategy

The execution time of $P's$ the cotangent code corresponding to the execution E of P is then:

$$\begin{aligned} T' &= \mathrm{SSum}\{{t_i}^s + {t'_i}^r\} \\ &\leq 5 * T \end{aligned}$$

The global memory G necessary to store recursively all the modified variables through the execution path E is the sum over all the run sub-programs of the memory necessary for one sub-program: $G = \mathrm{SSum}\{l_i\}$.

As one can understand from the execution path of $P's$, the memory necessary for the computation of the function followed by the computation of the derivative is:

$$\begin{aligned} M' &= \max(\mathrm{MSum}\{m_i\}, \mathrm{MSum}\{m'_i\}) + \mathrm{SSum}\{l_i\} \\ &\leq M + \mathrm{SSum}\{l_i\}. \end{aligned}$$

This is the maximum memory necessary for the computation of each component of P' because the memory necessary for the computation of the forward component can be released at the end of its computation. But $\forall i\ m'_i \leq m_i$, the maximum between $\mathrm{MSum}\{m_i\}$ and $\max_{b \in B} \sum_{i \in b} m'_i$ is M, the memory required by the computation of the original program P. One must notice that in general $G \gg M$.

4.2 Intermediate strategies

The extreme strategies consider applying the same mode to all the nodes in the call-tree. Many intermediate strategies that combine **all storage** on some nodes and **all recomputation** on the others can be thought of. In the following, we describe some of these intermediate strategies that replace storage by recomputation using checkpoints. As a result, the execution time is diminished at the cost of a (controlled) memory requirement augmentation. One checkpoint c_i stores some state 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 original point of the forward execution from t_i to t_{i+1}.

Two applications of this idea have been implemented in order to generate code that can be run even for real world original code; the **optimal checkpointing** strategy the **in-checkpointing** strategy and **in-out-checkpointing** strategy. In this section, we describe those three strategies, but it is clear that many other combinations of the two extreme strategies can be applied depending on the original code.

The so-called **optimal checkpointing** strategy is based on theoretical results on the reversion of a sequence of statements. If one considers any execution of P as a straight line, and if one can divide this execution into n blocks of statements of the same execution time t, and the same size of context of call m (where the context of call to a block is the set of input variables necessary to run this block). If N checkpoints are available, it is possible to compute their optimal repartition all along the execution of the sequence of blocks so as to minimise the recomputation of those blocks. This optimal schedule has been described in different papers [11, 15]. This strategy has been implemented in AD-tools on some patterns of code such as a loop

with a fixed number of iterations. Therefore, the sequence of statements is syntactically split into blocks which is the body of the loop of (nearly) the same execution time and context size. In practice the execution time and memory requirement of the body of a loop are not constant because control values (for branches or loops) are different. In general, the optimal schedule applied on a loop is no longer truly optimal. In `TAMC` [9] and `Odyssée` [8], the user can ask the system to apply this strategy using some differentiation options. A general package called `treeverse` described in [14] has also been developed to optimally place checkpoints on any adjoint code.

The second strategy called **in-checkpointing** strategy has been implemented in `TAMC` and `Odyssée`. This is a structural checkpointing [20] as the places where the checkpoints are set depends on the structure of the call tree but is not based on any optimal trade-off between storage and recomputation. Using this strategy, the checkpoints are used to store once the input context of each sub-program, in order to run the sequence $p_i^s; {p'_i}^r$ at the correct point. Thus, instead of recomputing the original point from scratch, the generated code restores the original context of each sub-program to get the correct computation of the derivative.

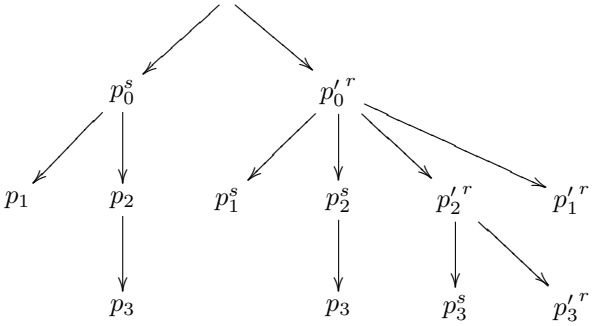
This is the strategy used within `Odyssée` as well as in `TAMC` because it is easy to automate and gives a good compromise between storage and recomputation. If the program is made of one sub-program there is no difference between the **all storage** strategy and the **in-checkpointing** strategy, but those codes are quite different on a general call tree.

Fig 14(a) and Fig. 14(b) show the call tree and execution path of the derivative, where $\bullet$ means storage of the context of the next sub-program, and $\circ$ means retrieval of this context.

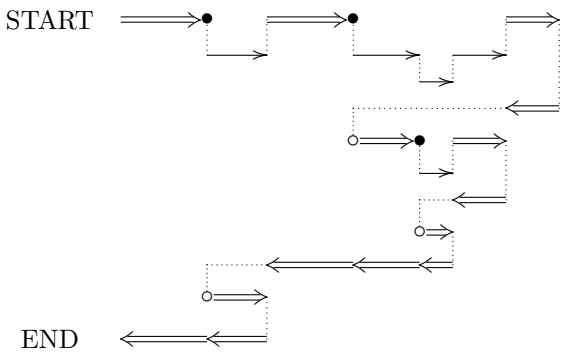
We denote by in_i the size of the input context of p_i. Using this strategy, the execution time can be computed by adding for each sub-program in E the execution of the forward and reverse components plus an extra cost due to the recomputation of the original sub-program. The number of recomputations of each sub-program p_i is exactly the depth of the sub-program in the call tree.

$$\begin{array}{rcl} T'_1 & = & \mathrm{SSum}\{t_i^s + {t'_i}^r\} + \mathrm{SSum}\{d_i * t_i\} \\ & \leq & 5 * T + \mathrm{SSum}\{d_i * t_i\} \\ M'_1 & = & \mathrm{MSum}\{m_i + m'_i + in_i + l_i\} \\ & \leq & 2 * M + \mathrm{MSum}\{in_i + l_i\}. \end{array}$$

The main drawback of the **in-checkpointing** strategy is the d_i recomputations of all sub-program p_i where d_i is the depth of p_i in the call tree. The **in-out-checkpointing** strategy is derived from the **in-checkpointing** strategy and stores the state not only before each sub-program call but also after each sub-program call. In this way, each sub-program p_i is called only once, but it costs the storage of one in-checkpoint (input context) before each call to a sub-program (from depth 0 to depth d-1) as well as one out-checkpoint (output context) after each call to a sub-program (from depth 1 to depth d). One must notice that the in-checkpoints and out-checkpoints can be treated in a last-in first-out way, but can only be managed separately. The second difference in the management of both types of checkpoints,

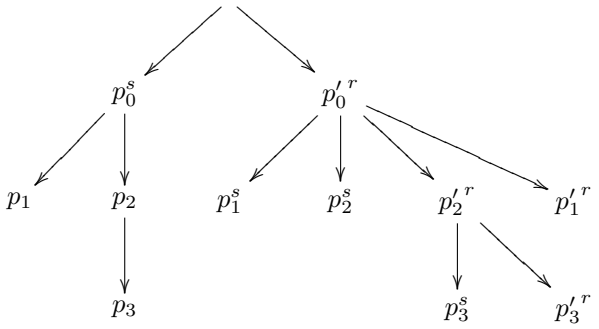


(a) Call tree of P_1'

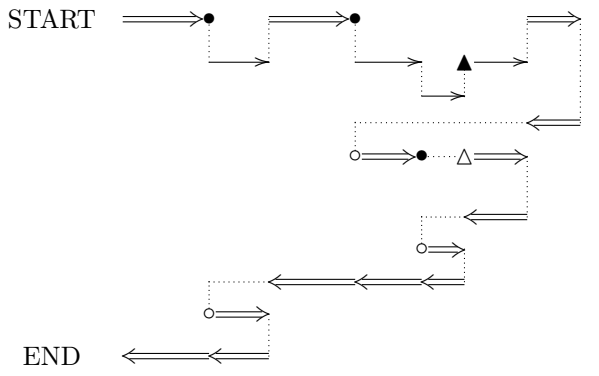


(b) Execution path of P_1'

FIGURE 14: **in-checkpointing** strategy



(a) Call tree of P_2'



(b) Execution path of P_2'

FIGURE 15: **in-out-checkpointing** strategy

is that the in-checkpoints are stored and restored all along the execution of the derivative code, whereas the out-checkpoints are all stored at first and then restored during the direct parts of the derivative.

Fig 15(a) and Fig. 15(b) show the call tree and execution path of the derivative of P using the **in-out-checkpointing** strategy where $\bullet$ means storage of the in-checkpoint (context before call to a sub-program) and $\circ$ means the retrieval of this checkpoint, $\blacktriangle$ means storage of the out-checkpoint (context after call to a sub-program) and $\triangle$ means the retrieval of this checkpoint. As we said before the checkpoints are not dealt with in a last-in first-out way if the two types of checkpoints are mixed. In our example in- and out- checkpoints are treated in the following order: $\bullet^1; \bullet^2; \blacktriangle^3; \circ^2; \bullet^3; \triangle^3; \circ^3; \circ^1$ where the upper script ^{i} means associated to sub-program i. It is clear that if they are treated separately the nice last-in first-out way appears: $\bullet^1; \bullet^2; \circ^2; \bullet^3; \circ^3; \circ^1$, and $\blacktriangle^3; \triangle^3$.

If we denote by out_i the size of the out-checkpoint of p_i, the complexity measures

Temporal complexity	Spatial complexity
$T_{base} = (1 + c_t) * T$	$M_{base} = (1 + c_m) * M$

FIGURE 16: Base temporal and spatial complexities

Strategy	Execution time	Memory requirement
all recomputation	$\text{SSum}\{(N - i + d_i) * t_i\}$	$\text{MSum}\{l_i\}$
all storage	0	$\text{SSum}\{l_i\}$
in-checkpointing	$\text{SSum}\{d_i * t_i\}$	$\text{MSum}\{in_i + l_i\}$
in-out-checkpointing	$\text{SSum}\{t_i\}$	$\text{MSum}\{in_i + l_i\} + \text{SSum}\{out_i\}$

FIGURE 17: Summary of complexity measures

associated with the second strategy of checkpointing at the sub-program level are:

$$\begin{array}{rcl} T'_2 & = & \text{SSum}\{t_i^s + t_i'^r\} + \text{SSum}\{t_i\} \\ & \leq & 5 * T + T \\ M'_2 & = & \text{MSum}\{m_i + m'_i + in_i + l_i\} + \text{SSum}\{out_i\} \\ & \leq & 2 * M + \text{MSum}\{in_i + l_i\} + \text{SSum}\{out_i\} \end{array}$$

4.3 Conclusion

Fig 16 and Fig. 17 summarise the complexity results for each strategy presented in the previous sections. In order to help the comparisons between all of these strategies, we show these complexity measures as a sum of two components: a base complexity presented in Fig. 16 plus an extra cost shown in Fig. 17. We denote by c_t ($c_t \leq 5$) the actual ratio in execution time of the adjoint statements with respect to the original function and by c_m ($c_m \leq 1$) the ratio in memory requirement of the derivative variables with respect to the original variables. Those coefficients characterise the derivative code and do not depend on the strategy: they depend only on the sequence of statements run in the original function and on the variable with respect to which those statements are differentiated. Thus, we define an optimal cotangent code to be a derivative code whose execution time is $(c_t+1)*T$ and whose memory requirement is $(c_m+1)*M$ where T is the execution time of the original code, and M is the memory required for the computation of the original variables. None of the strategies described above lead to the generation of a cotangent code optimal both in execution time and memory requirements.

To compare the complexities of the different strategies, one has only to compare the extra costs presented in Fig. 17.

The temporal complexities shown in the first column of Fig. 17 can be easily compared because they are given in terms of execution time of the original function t_i. As for the memory requirement, the complexities shown above are difficult to compare because they are evaluated in terms of some elementary complexity

measures l_i, in_i, out_i. When the basic information to be stored in the trajectory is chosen, these elementary parameters depend only on the original code p_i.

If one wants to extrapolate the measures presented above to the computation of N gradients within the same code, the only modifications appeal to the base complexities: the coefficients c_t and c_m are $c_t \leq 5 * N$ and $c_m \leq N$.

5 PRACTICAL IMPLICATIONS

In this section, we describe some practical implications of the strategy choice which were not described in the previous section. We first study what the components $p_i^s, p_i'^r$ of the adjoint codes are in practice: their semantics are clear enough, but their implementation has to be clarified. The only functionality implemented independently from the chosen strategy is for p_i^s to store the trajectory and execute the same statements as p_i, and for $p_i'^r$ to restore the trajectory and execute the corresponding adjoint statements. The implementation of these components depends on the chosen strategy as described in Section 5.1. The second point studied is the way the trajectory is defined: we know from the previous section that the trajectory depends on the strategy, but some more explanation is necessary to go from the theory to the practice as shown in Section 5.2. The third point described in Section 5.3 is the extra cost in execution time due to memory management and calls to sub-programs. In the previous section, we only take into account computational operations, but we know that in practice the trajectory management demands execution time. In some cases, the call to sub-programs can be of importance, so we quantify it in this section.

5.1 Components of the derivative code

In the previous sections, we have denoted by p_i^s and $p_i'^r$ the direct and reverse parts of a sub-program p_i. The actual definitions of these sub-programs are significantly different depending on the strategy considered: the sub-programs called are different.

Using the **all storage** strategy, any sub-program p_i^s calls only sub-programs of the base type: p_j^s for each of its children p_j. In the same manner, $p_i'^r$ calls $p_j'^r$ for all its children.

Using the **all recomputation** strategy, any sub-program p_i^s calls only sub-program of the base type: p_j for each of its children p_j. For each of its children p_j in the call-tree, the second component $p_i'^r$ calls: *(i)* a sub-program r_0^{in} that restores the input context of p_0, *(ii)* a copy of p_0 in which p_j is replaced by p_j^s and *(iii)* $p_j'^r$.

Using the first sub-program checkpointing strategy, any sub-program p_i^s calls to sub-programs for each of its children p_j: *(i)* s_j^{in} that stores the input context of p_j and *(ii)* p_j. For each of its children p_j in the call-tree, the second component $p_i'^r$ calls three forms of the original sub-program p_j: r_j^{in}, p_j^s, and $p_j'^r$.

Using the second sub-program checkpointing strategy, p_0^s calls s_j^{in} for its direct children, p_j and s_j^{ou} for all its children. For each p_i such that $i > 0$, p_i^s calls r_j^{in}

and r_j^{out}. For each of its children p_j in the call-tree, the second component $p_i'^r$ calls three forms of the original sub-program p_j: r_j^{in}, p_j^s, and $p_j'^r$.

Finally, any adjoint code is the composition of different forms of each sub-program p_i involved in the original execution: $p_i, p_i^s, p_i'^r, r_i^{in}, s_i^{in}, r_i^{out}, s_i^{out}$. The strategies which are described in this paper show some methods for composing these forms, but many other compositions can be invented. If one considers the call tree of a program, any adjoint code is perfectly defined by the flag attached to each sub-program in the tree. Any form of a sub-program p_i can be implemented within a separate sub-program. However, for a chosen adjoining strategy, a gathering of some forms of a sub-program is possible. For example, using a **in-checkpointing** strategy, the two forms $p_i^s, p_i'^r$ are always executed together and can be glued in the same sub-program.

5.2 Components of the trajectory

Theoretically, the elementary parameters of the complexity in memory requirements in_i and out_i depend only on the original code p_i, while l_i depends on the original code and on the basic information stored. In practice, it is difficult to reach the theoretical values and store the minimal amount of information. As shown in [19], detecting automatically which components of an array are computed through some complex statement is impossible. In this way, the theoretical complexities in memory requirements are only minimal bounds, and the actual complexity implemented can be much more expensive.

We have said before that l_i depends on the original code and on the basic information stored. The basic information stored by any code p_i^s to enable the backward computation of the elementary Jacobian matrices is either *(i)* all the instances of all the variables directly modified by p_i or *(ii)* all the partial derivatives necessary to compute the intermediate adjoint variables of p_i. If the choice is to store original values before modification and after read (or before read and after modification), one can restrict to those which are involved in non zero partial derivatives. The drawback of this is that the implementation of p_i^s depends on the activity of variables. As we said before, the trajectory contains the values necessary to compute the elementary Jacobian matrices but also some values necessary to traverse the computational graph backward. These control values can be of many different types depending on the strategy chosen to traverse the graph backward. If one decides to traverse the code using an interpreter of the trace, the indexes of the basic blocks must be stored. If the choice is that the control part of the control statements must be reproduced the information stored includes: the tests of the branches, the bounds of the loops, etc

Depending on the strategy, the input and output contexts of sizes in_i and out_i, respectively, can be implemented in different manner. For example, using some checkpointing at the sub-program level, the minimal input context to be stored is not the whole input context of p_i but only the set of variables (or memory locations) which are modified by p_i. In this, we profit from the fact that the modification of the variables performed after the end of p_i have been un-done by the corresponding

adjoint code. Such optimisations must be studied for each intermediate strategy.

Writing adjoint codes by hand or generating them automatically, the same problem arises: the basic information is stored (or retrieved) before each executable statement. In practice, this means that the computer on which the code is run spends much of its time accessing the trajectory values instead of computing the derivatives. This can be avoided by grouping the store (or retrieve) instructions for scalar values within one store instruction for a buffer of values. This is equivalent to forming local functions by gathering original statements and differentiating these functions globally. In the first approach, some analysis of the derivative code can be performed to choose where to set the store statement, whereas the second approach analyses the original function to obtain optimal sub-functions.

The implementation of the trajectory can be of several forms: files, stacks, arrays etc. The final choice must be left to the user because the efficiency of the resulting code using one or the other of these implementation depends deeply on the platform on which the code is run.

5.3 Practical temporal complexity

The temporal complexity presented in Fig. 17 do not take into account the extra cost due to the trajectory management, as well as that due to calls to sub-programs. On some large adjoint codes, the memory management can consume most of the execution time (1/3 in [7]). In this section, we describe these extra costs in a naive manner. We do not describe any abstract machine, but we formalise the behaviour of the execution time.

The approximation we use in this paper for the cost of a call of p_i is proportional to the number in_i of its input variables and out_i the number of its output variables. Let us denote by $io_i < in_i + out_i$ the number of input or output variables and by t_{io} the elementary cost due to the management of one input or output variable (move from the memory to a register or the stack). The extra cost due to calls to sub-programs within an execution E of a program P is:

$$T_{io} \quad = t_{io} * \mathrm{SSum}\{io_i\}.$$

Let io'_i be the number of input or output derivative variables of the adjoint program. We know that $io'_i \leq io_i$. The extra cost in execution time due to the calls to all the couples $p_i^s; {p_i}'^r$ involved in the adjoint code is:

$$\begin{array}{lcl} T'_{io} & = & t_{io} * \mathrm{SSum}\{2 * io_i + io'_i\} \\ T'_{io} & = & 2 * T_{io} + t_{io} * \mathrm{SSum}\{io'_i\} \\ T'_{io} & \leq & 3 * T_{io}. \end{array}$$

The extra cost in memory moves is implementation and machine dependent, but it is roughly proportional to the number of memory moves. We consider that storing or retrieving one scalar value has the same elementary cost. This is not exactly true, but formalising this cost is not the purpose of this paper. An elementary cost is associated with each of these implementations, but this cost depends on the

machine, on the network etc. Let us denote by mm_i the number of memory moves used in an original sub-program p_i and by t_{mm} the elementary cost of one memory move. Then the execution time due to memory moves is:

$$T_{mm} = t_{mm} * \mathrm{SSum}\{mm_i\}.$$

Let us denote by mm'_i the added derivative memory moves due to the derivative variables in its derivative sub-program. The cost due to memory moves in the adjoint code without taking into account the storage of the trajectory is :

$$\begin{array}{rcl} T'_{mm} & = & t_{mm} * \mathrm{SSum}\{2 * mm_i + mm'_i\} \\ T'_{mm} & = & 2 * mm + t_{mm} * \mathrm{SSum}\{mm'_i\} \\ T'_{mm} & \leq & 3 * T_{mm}. \end{array}$$

We denote by c_{io} and c_{mm}, respectively, the actual ratios in terms of number of input or output variables and of memory moves between the adjoint code and the corresponding original code. These coefficients only depend on the code. The only general bounds on them are $0 \leq c_{io} \leq 1$ and $0 \leq c_{mm} \leq 1$. Fig. 18 presents the component of the practical temporal complexity independent from the strategy named T_{base}, due to calls to sub-programs as well as scalar memory moves.

Fig. 19 presents a summary of the extra costs due to calls to sub-programs as well as scalar moves for each strategy presented in this paper. We denote by α the extra computational time due to the calls of the original sub-programs. If $call_i$ denotes the number of extra calls of p_i, the contribution of p_i to α is $call_i * io_i$ and $\alpha = \mathrm{SSum}\{call_i * io_i\}$. The number of extra calls to the original sub-program p_i deduced from Fig. 17 is presented in the first column of Fig. 19. The extra cost in scalar moves is due to the storage of the trajectory, but also to the recomputation of the original sub-programs and can be also deduced from Fig. 17. This cost can be described by the formula $\beta * t_{mm} + \gamma * t_{traj}$, where β is the total number of scalar moves due to the recomputation of the original function, γ is the total number of trajectory moves, and t_{mm} is the elementary time necessary for storing or retrieving a scalar trajectory value. This elementary time t_{traj} is certainly greater than t_{mm} because any the store/retrieve commands can be implemented using: files, stacks, arrays etc. The contribution of sub-program to β p_i is $call_i * mm_i$ and $\beta = \mathrm{SSum}\{call_i * mm_i\}$. The value of γ is deduced from the second column of Fig. 17 by changing the operator max to the operator $\sum$ to compute the total number of store and retrieve. The extra practical temporal complexity is then $\alpha * t_{io} + \beta * t_{mm} + \gamma * t_{traj}$, where $t_{traj} \gg t_{io}$ and $t_{traj} \gg t_{mm}$. For each strategy, Fig. 19 presents general formulas for parameters α, β, and γ of the extra practical temporal complexity. In Fig. 20 one can find formulas for the sub-program contributions that depend on the strategy. For a given strategy, the total practical temporal complexity can be obtained from the measures presented in Fig. 18 and Fig. 19 by adding the base complexity PT_{base} to the sum of the three components which depend on the strategy.

$$\begin{array}{lcl} PT_{base} & = & 2*(T_{io}+T_{mm})+c_{io}*T_{io}+c_{mm}*T_{mm} \\ c_{io} & = & \mathrm{SSum}\{io'_i\}/\mathrm{SSum}\{io_i\} \\ c_{mm} & = & \mathrm{SSum}\{mm'_i\}/\mathrm{SSum}\{mm_i\} \end{array}$$

FIGURE 18: Base practical temporal complexity

$$\begin{array}{|l|} \hline \alpha*t_{io}+\beta*t_{mm}+\gamma*t_{traj} \\ \hline \alpha = \mathrm{SSum}\{call_i*io_i\} \\ \beta = \mathrm{SSum}\{call_i*mm_i\} \\ \gamma = 2*\mathrm{SSum}\{l_i+\gamma_i\} \\ \hline \end{array}$$

FIGURE 19: Extra practical temporal complexity

Strategy	$call_i$	γ_i
all recomputation	$N-i+d_i$	0
all storage	0	0
in-checkpointing	d_i	in_i
in-out-checkpointing	1	in_i+out_i

FIGURE 20: Strategy dependent sub-program contributions

5.4 Conclusion

The elementary execution times t_{mm} and t_{io} are in general a lot smaller than t_{traj}, the time due to the management of a move from/to the memory, some disk, etc. Moreover, until now there is no attempt to manage the memory moves in a way different from the original code. For the original or derivative variables, the management is defined by the original code. This leads to a key research direction: how to diminish the extra cost $\gamma * t_{traj}$. This can be done either by diminishing the factor γ or the elementary time t_{traj}.

To diminish the factor γ, one can refine code analysis to detect whether a memory location is to be stored. This leads to problem when the location is a component of an array, as code analysis is not powerful enough to detect the constraint on any indirection on an array as shown in [19]. We are working on this subject and we obtain some good preliminary results by mixing specific compilation techniques and activity propagation. To diminish γ, one can also think of replacing storage by recomputation, but then the factor α is increased. One idea is to store values difficult to compute, and recompute elemental values. This leads to a mix of the strategies presented above and gives some new trade-offs between execution time due to memory management and recomputation of the function.

Even if the factor γ is smaller, the problem of moving efficiently (small t_{traj}) a large number of values from/into the memory or a file (or anything else) still arises. This problem must be studied in a general way: how is it possible to store/retrieve

a large number of values at a minimal cost. It is pretty certain that spreading the storage command of scalar values all over executable statements is not the optimal method. Scalar values must certainly be gathered to buffers which can be transfered efficiently. Writing a package which offers static storage, dynamic storage, and buffered storage seems a problem in itself. New AD-tools have to leave the user to choose the best storage implementation depending on his machine and network and they should profit greatly from such a package. We consider that the trajectory itself or the input or output contexts are stored and restored using the same strategy. This is not necessary. Depending on the code, each component can be dealt with using different strategies.

6 CONCLUSION AND FUTURE WORK

From the complexity measures presented in Fig 17 and Fig. 18, one can understand that the choice of the strategy cannot be automatic but must be left to the user. In addition, the actual properties of the adjoint code are deeply related to the original source, and also to the computing environment. On some computers storing the trajectory on the disk is expensive as on others it is cheap. It seems difficult to get an adjoint code optimal for all platforms. To choose the best strategy (or mixed strategy), the elementary measures $t_i, m_i, l_i, in_i, out_i$ and t_{traj} must be known. This is perhaps the kind of information that should be provided by profilers. If the AD-tool is able to compute the complexity measures for the chosen strategy it certainly helps.

We have described the main approaches used to make the reverse mode of automatic differentiation or the hand coding of discrete adjoints applicable on real world codes. Some of those techniques are implemented in automatic differentiation tools (`TAMC`, `Odyssée`), but some are used only to write discrete adjoint codes by hand. It is important to automate all of the possible strategies, but this is a challenge: for example the **all storage** strategy can only be used for hand written discrete adjoints because it requires a great deal of optimisation in order to be applied to a real code. A semi-automatic application of `Odyssée` to generate a discrete adjoint using the **all storage** strategy for a large code `Meso-nh` has been successful. It has required a two step optimisation of the generated adjoint code as presented in [4, 5]: minimisation of the trajectory and replacement of storage by recomputation. Part of the step of minimisation of the trajectory can be automate using a specific static analysis named TBS (to be saved) analysis. As for the second step, it is used when hand coding adjoint, and consists in storing only the state vector: minimal set of variables computed at one temporal or spatial iteration that enables the execution of the next iteration. In this case, instead of storing all the modified variables at all steps of the loop, only the most important variables are stored, and the other "intermediate" variables are recomputed. This is closely related to the notion of a "living variable" already known by people doing code analysis for optimised compilation.

Some other directions for making the reverse mode automatically applicable on real

world code are being studied. For example, the hierarchical approach [3] and the cross country elimination (in [6] `pp 47-51`) can be used to mix direct and reverse modes. These techniques diminish the memory cost as well as the execution time. The use of parallelism for adjoint code generation is also investigated (in [6] `pp 23-29`) as well as parallel checkpointing.

The formal descriptions of adjoining strategies and their associated complexity measures are given in terms of sub-programs associated to their elementary complexity measures. We name "compilation units" the pieces of code that can be defined independently for a given language: subroutine, function, program, and block-data for Fortran. The results presented in this paper are correct for any type of sub-program: separate units in the original source code, sub-trees consisting of some separate units, or sequences of statements. Until now, in AD-tools, the smallest piece of code which can be differentiated by the user is a compilation unit, and the same adjoining strategy is applied to all the compilation units. Future AD-tools can take a better profit of all the possible strategies by generalising their notion of differentiation unit and adjoining strategy. The notion of differentiation unit can be generalised to the so-called sub-program notion and a different strategy can be applied to each sub-program. In this way, automatic differentiation could be as flexible as hand coding and could (perhaps sometimes automatically) adapt to the original code.

Until now, it is clear that hand written adjoint codes are more efficient than automatically generated ones. The introduction in future AD tools of new mixed strategies partially solves this problem, but something remains the same: hand coded adjoints are derived from original source codes written with this intend, whereas automatically generated adjoint codes are derived from any original code. Perhaps, we should attempt to characterize a "good" program for differentiation in reverse mode. In this context "good" means that the original code should lead to an efficient adjoint code. Moreover, a problem such as the aliasing problem does not arise when hand coding adjoint codes. For example, if one replaces `Y` by `Z` in the adjoint code of $\mathcal{G}_1$ shown in Fig. 6, a wrong answer is computed. Moreover, if one replaces `Y` by `X` in the adjoint code of $\mathcal{G}_2$ shown in Fig. 6, one gets a non valid code: the result depends on the compiler if `dX` and `dY` are arguments of the sub-program. In fact aliasing by call to a sub-program leads to great trouble using the reverse mode, because the status (read, written) of the adjoint variables is the transposed status of the corresponding original variable. This problem of aliasing should be studied because it happens within codes. More generally, one direction of research can be the definition of a norm for writing original models for which automatically generated adjoint codes are efficient.

REFERENCES

1. W. Baur and V. Strassen, *The complexity of partial derivatives*, Theoretical Comp. Sci. **22** (1983), 317–330.
2. M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank (eds.), *Computational differentiation: Applications, techniques, and tools*, SIAM, Philadelphia, 1996.

3. C. H. Bischof and M. R. Haghighat, *Hierarchical approaches to automatic differentiation*, Computational Differentiation: Applications, Techniques, and Tools (M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds.), SIAM, 1996, pp. 83–94.
4. I. Charpentier, *Génération de codes adjoints : Traitement de la trajectoire du modèle direct.*, Rapport de recherche 3405, INRIA, April 1998.
5. I. Charpentier and M. Ghemires, *Génération automatique de codes adjoints : Stratégies d'utilisation pour le logiciel Odyssée. Application au code météorologique Meso-NH*, Rapport de recherche 3251, INRIA, September 1997.
6. C. Faure (ed.), *Automatic Differentiation for Adjoint Code Generation, Proceedings of imacs'aca, Automatic Differentiation session*, Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France), November 1998, Rapport de recherche 3555, http://www.inria.fr/RRRT/RR-3555.html.
7. ———, *Le Gradient de THYC3D par Odyssée*, Rapport de recherche 3519, INRIA, October 1998.
8. C. Faure and Y. Papegay, *Odyssée User's Guide. Version 1.7*, Rapport technique 0224, INRIA, September 1998.
9. R. Giering, *Tangent linear and Adjoint Model Compiler , Users manual*, 1997, Unpublished, available from `http://puddle.mit.edu/~ralf/tamc`.
10. J.C. Gilbert, G. Le Vey, and J. Masse, *La Différentiation Automatique de fonctions représentées par des programmes*, Rapport de recherche 1557, INRIA, November 1991.
11. A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software **1** (1992), 35–54.
12. ———, *Principles and Techniques of Algorithmic Differentiation*, SIAM, 2000.
13. A. Griewank and G.F. Corliss (eds.), *Automatic differentiation of algorithms: Theory, implementation, and applications*, SIAM, Philadelphia, 1991.
14. A. Griewank and A. Walther, *Treeverse: An implementation of the checkpointing for the reverse or ajoint mode of differentiation*, Tech. report, TU Dresden, 1997.
15. J. Grimm, L. Pottier, and N. Rostaing-Schmidt, *Optimal time and minimum space-time product for reversing a certain class of programs*, Computational Differentiation: Applications, Techniques, and Tools (M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds.), SIAM, 1996, pp. 95–106.
16. J. Morgenstern, *How to compute fast a function and all its derivatives, a variation on the theorem of Baur-Strassen*, Sigact News **16** (1985), 60–62.
17. G.M. Ostrovskii, Yu.M. Volin, and W.W. Borisov, *Uber die berechnung von ableitungen*, Wiss. Z. Tech. Hochschule fur Chimie **13** (1971), 382–384.
18. B. Speelpening, *Compiling fast partial derivatives of functions given by algorithms*, Ph.D. thesis, University of Illinois, Urbana-Champaign, 1980.
19. M. Tadjouddine, C. Faure, and F. Eyssette, *Sparse Jacobian Computation in Automatic Differentiation by Static Program Analysis*, Static Analysis (G. Levi, ed.), Lecture Notes in Computer Science, vol. 1503, Springer-Verlag, September 1998, pp. 311–326.
20. Yu.M. Volin and G.M. Ostrovskii, *Automatic computation of derivatives with the use of the multilevel differentiating technique - 1. algorithmic basis*, Computers & Mathematics with Applications **11** (1985), no. 11, 1099–1114.