# *The Data-Dependence Graph of Adjoint Programs*

Laurent Hascoët

**N° 4167**

Avril 2001

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4167--FR+ENG

# The Data-Dependence Graph of Adjoint Programs

Laurent Hascoët

Thème 1 — Réseaux et systèmes
Projet Tropics

Rapport de recherche n° 4167 — Avril 2001 — 17 pages

**Abstract:** Automatic Differentiation is a technique that permits generation of *adjoint programs*, which compute gradients. In scientific computation, these gradients are a fundamental tool for optimization or data assimilation. Computation of a gradient is relatively expensive, and should therefore be optimized whenever possible. The study of these program optimizations is most often based on the data-dependence graph. Under precise assumptions, we prove that the adjoint program's data-dependence graph is isomorphic to a sub-graph of the original data-dependence graph. The proof relies on a refined definition of the notion of data-dependence, and on a formal definition of adjoint programs in terms of products of local Jacobian matrices. This theorem can be used to transpose optimizations of the original program, to the adjoint program. It can also justify specific transformations on the adjoint. We give some examples of such applications.

**Key-words:** Automatic Differentiation, Adjoint, Gradients Computation, Optimization, Data-Dependence, Parallellization

# Le graphe de dépendance des données des programmes adjoints

**Résumé :** La Différentiation Automatique est une technique de génération de *programmes adjoints*, qui calculent des gradients. En calcul scientifique, ces gradients sont une information essentielle pour l'optimisation ou l'assimilation de données. Le calcul d'un gradient est une opération relativement chère, qu'il est souhaitable d'optimiser. L'un des outils fondamentaux pour ce type d'optimisation de programmes est le graphe de dépendances de données. Sous des hypothèses que nous précisons, nous montrons que le graphe de dépendance des données correspondant au programme adjoint est isomorphe à un sous-graphe du graphe de dépendance des données du programme initial. La démonstration s'appuie sur une définition raffinée de la notion de dépendance de données, et sur la définition formelle des programmes adjoints en termes de produits de matrices jacobiennes élémentaires. Ce théorème permet de transposer pour le programme adjoint certaines transformations possibles sur le programme initial, mais aussi d'appliquer des optimisations spécifiques. Nous en donnons quelques exemples pour des optimisations courantes.

**Mots-clés :** Différentiation Automatique, Adjoints, Calcul des Gradients, Optimisation, Dépendance des Données, Parallélisation

## Contents

## 1 Introduction

In Scientific computing, the need for evaluating some mathematical derivatives of functions is commonplace. One such derivative, the *gradient*, is of particular interest. For example, gradients are used for optimization [5], optimal design [8] [3] [10], data assimilation [9], or inverse problems.

When these functions are implemented as programs, the technique of Automatic Differentiation (A.D.) generates a new program that computes the specified derivatives. In particular for gradients, the so-called *reverse* mode of A.D. [2] creates *adjoint programs* (presented in section 2), that evaluate the gradient in a computationnaly very effective manner. However, adjoint programs use a lot of memory space. Furthermore, they are not easily read by humans. They often need optimizations (not only for memory), and this is more safely done in a formal framework, using data-dependence graphs (*ddg*, presented in section 3).

To this end, we need to relate properties of the ddg of adjoint programs, to properties of the original program. This paper gives, in section 4, a proof that the ddg of the adjoint program is in some sense isomorphic to a subset of the ddg of the original program. This allows us to transpose most ddg-related properties of the program to its adjoint, as shown in section 5. In particular, this applies to vectorization, parallelization, or other transformations that reschedule the program's instructions [4]. This can also reduce dramatically the memory consumed by the adjoint approach.

## 2 Adjoint Programs

Consider a program $P$, with instructions $I_i, i \in [1..n]$, that implements a mathematical function $F : \mathbb{R}^n \to \mathbb{R}^m$. The adjoint program $\overline{P}$ is a program (generated by A.D.) that computes the gradient of $F$.

Let us explain how $\overline{P}$ is built. Assume that each instruction $I_i$ implements a locally differentiable, vectorial, elementary function $f_i$. Each particular execution of $P$ evaluates a long sequence $X_k, k \in [1..l]$ of instructions from $P$, determined by the control structures of $P$. This means there is some control function $c$, from $[1..l]$ to $[1..n]$, that gives $X_k$ the $k^{th}$ instruction executed, by $X_k = I_{c(k)}$. For this particular execution, function $F$ is thus the following composition of elementary functions $f_i$, with $V$ the vector of the $n$ inputs, and $W$ the vector of the $m$ outputs:

$$W = F(V) = f_{c(l)}(f_{c(l-1)}(\dots f_{c(2)}(f_{c(1)}(V))\dots))$$

Given now a (transposed) weighting $\overline{W}^T$ of the $m$ outputs, the transposed gradient $\overline{V}^T$ of $F$ is defined as the left product of $\overline{W}^T$ by $F'$, jacobian matrix of $F$. Using the chain rule, it can be written as:

$$\overline{V}^T = \overline{W}^T.f'_{c(l)}.f'_{c(l-1)}.\dots.f'_{c(2)}.f'_{c(1)}$$

Since $\overline{W}$ is a vector and not a matrix, this product is best executed from the left. Equivalently, it is often written transposed as:

$$\overline{V} = {f'_{c(1)}}^T.{f'_{c(2)}}^T.\dots.{f'_{c(l-1)}}^T.{f'_{c(l)}}^T.\overline{W}$$

The adjoint program $\overline{P}$ computes this product. To this end, it mainly consists of $n$ small sequences of "adjoint instructions" $\overline{I_i}$, each sequence implementing the vectorial assignment of so-called "adjoint variables":

$$\overline{V} := f_i'^T \times \overline{V}$$

These $\overline{I_i}$ are surrounded by a control structure ensuring that they are executed in the correct order, i.e. $\overline{I_{c(l)}}$ first, then $\overline{I_{c(l-1)}}$ $\ldots$, which is in fact the inverse or the original execution order. We shall not go into further detail about this adjoint control structure, which can be achieved in many ways. We shall simply assume that for each particular execution order of $P$, the adjoint $\overline{P}$ executes adjoint instructions in the reverse order.

Last but not least, notice that the coefficients of the $f'_i$ matrices may involve variables, whose values must be taken from the execution of the original instruction $I_i$. Therefore, to actually compute the gradient of $F$, one must first execute completely the original program $P$, and only then one executes $\overline{P}$, which uses some values from execution of $P$. Since imperative programs often overwrite their variables, these values must be saved during execution of $P$, and progressively restored during execution of $\overline{P}$. This is the main drawback of adjoint programs, because the memory required grows linearly with the execution time, and can become unmanageably large.

To illustrate this, figure 1 shows the generated adjoint program of a small routine called `normalize`, which implements the normalization of a 2-D vector. The original program contained only instructions $I_{[1..4]}$, and returned outputs `x`, `y`, `r` from inputs `x` and `y`. The generated program is composed of the original program followed by the *adjoint instructions* $\overline{I_{[4..1]}}$. It takes as additional inputs the *adjoint variables* $\overline{\texttt{x}}$, $\overline{\texttt{y}}$, $\overline{\texttt{r}}$ (e.g. (0,0,1) if only output `r` is of interest), and returns the gradient $(\overline{\texttt{x}}, \overline{\texttt{y}})$. To detail further the mechanism of generation, we show on the right the elementary jacobian matrices of each $I_{[1..4]}$, and the corresponding transposed jacobian product implemented by each $\overline{I_{[4..1]}}$. Notice the *save* and *restore* operations, necessary because of the over-writings of `r`, `x`, and `y`.

## 3 Data-Dependence Graphs

The data-dependence graph ("ddg") [1] [7] is a graph that defines a partial order between the (3-address code) operations performed by a program. When a reordering of the program's operations or instructions does not inverse the ddg arrows, then the semantic of the program is preserved. The ddg is a *static* information: it relates *textual* operations from the program. When the

$$\begin{array}{ll}
\overline{\texttt{normalize}}\texttt{(x,}\overline{\texttt{x}}\texttt{,y,}\overline{\texttt{y}}\texttt{,r,}\overline{\texttt{r}}\texttt{)} & \\
\quad \texttt{r := x*x + y*y} & \rbrack\ I_1 : f'_1/_{(\mathtt{r,x,y})} = \begin{pmatrix}0&2x&2y\\0&1&0\\0&0&1\end{pmatrix}\\
\qquad \texttt{r} \longrightarrow \textit{save} & \\
\quad \texttt{r := sqrt(r)} & \rbrack\ I_2 : f'_2/_{(\mathtt{r})} \quad = (1/(2*\mathtt{sqrt}(\mathtt{r})))\\
\qquad \texttt{x} \longrightarrow \textit{save} & \\
\quad \texttt{x := x/r} & \rbrack\ I_3 : f'_3/_{(\mathtt{x,r})} \ = \begin{pmatrix}1/\mathtt{r}&-\mathtt{x}/(\mathtt{r}*\mathtt{r})\\0&1\end{pmatrix}\\
\qquad \texttt{y} \longrightarrow \textit{save} & \\
\quad \texttt{y := y/r} & \rbrack\ I_4 : f'_4/_{(\mathtt{y,r})} \ = \begin{pmatrix}1/\mathtt{r}&-\mathtt{y}/(\mathtt{r}*\mathtt{r})\\0&1\end{pmatrix}\\
\qquad \texttt{y} \longleftarrow \textit{restore} & \\
\quad \overline{\texttt{r}}\texttt{ := }\overline{\texttt{r}}\texttt{ - y*}\overline{\texttt{y}}\texttt{/(r*r)} & \\
\quad \overline{\texttt{y}}\texttt{ := }\overline{\texttt{y}}\texttt{/r} & \rbrack\ \overline{I_4} : \begin{pmatrix}\overline{\mathtt{y}}\\\overline{\mathtt{r}}\end{pmatrix} := \begin{pmatrix}1/\mathtt{r}&0\\-\mathtt{y}/(\mathtt{r}*\mathtt{r})&1\end{pmatrix}\begin{pmatrix}\overline{\mathtt{y}}\\\overline{\mathtt{r}}\end{pmatrix}\\
\qquad \texttt{x} \longleftarrow \textit{restore} & \\
\quad \overline{\texttt{r}}\texttt{ := }\overline{\texttt{r}}\texttt{ - x*}\overline{\texttt{x}}\texttt{/(r*r)} & \\
\quad \overline{\texttt{x}}\texttt{ := }\overline{\texttt{x}}\texttt{/r} & \rbrack\ \overline{I_3} : \begin{pmatrix}\overline{\mathtt{x}}\\\overline{\mathtt{r}}\end{pmatrix} := \begin{pmatrix}1/\mathtt{r}&0\\-\mathtt{x}/(\mathtt{r}*\mathtt{r})&1\end{pmatrix}\begin{pmatrix}\overline{\mathtt{x}}\\\overline{\mathtt{r}}\end{pmatrix}\\
\qquad \texttt{r} \longleftarrow \textit{restore} & \\
\quad \overline{\texttt{r}}\texttt{ := }\overline{\texttt{r}}\texttt{/(2*sqrt(r))} & \rbrack\ \overline{I_2} : (\,\overline{\mathtt{r}}\,) := (\,1/(2*\mathtt{sqrt}(\mathtt{r}))\,)(\,\overline{\mathtt{r}}\,)\\
 & \\
\quad \overline{\texttt{x}}\texttt{ := }\overline{\texttt{x}}\texttt{ + 2*x*}\overline{\texttt{r}} & \\
\quad \overline{\texttt{y}}\texttt{ := }\overline{\texttt{y}}\texttt{ + 2*y*}\overline{\texttt{r}} & \rbrack\ \overline{I_1} : \begin{pmatrix}\overline{\mathtt{r}}\\\overline{\mathtt{x}}\\\overline{\mathtt{y}}\end{pmatrix} := \begin{pmatrix}0&0&0\\2\mathtt{x}&1&0\\2\mathtt{y}&0&1\end{pmatrix}\begin{pmatrix}\overline{\mathtt{r}}\\\overline{\mathtt{x}}\\\overline{\mathtt{y}}\end{pmatrix}\\
\quad \overline{\texttt{r}}\texttt{ := 0} &
\end{array}$$

Figure 1: *gradient program by A.D.*

operation is enclosed in a control structure, such as a loop, it represents all its run-time executions. Classically, the reason for a data-dependence between two operations is the utilisation of program variables. There is a dependence from operation $o_1$ to $o_2$, if these three conditions hold:

1. in some real execution, $o_1$ may be done before $o_2$ (*notation:* $o_1 \prec o_2$).
2. in some real execution, $o_1$ and $o_2$ access the same memory location,
3. either $o_1$ or $o_2$ perform a write into this memory location, while the other operation writes or reads.

For many uses, considering each operation separately is a much too small level of granularity. In the remaining of this paper, we shall consider a larger granularity, the “nodes”, which contain one instruction or a sequence of always consecutive instructions. By “always consecutive”, we mean consecutive in any execution of the program, and we call these nodes *connex*. We have a natural projection of the ddg on these nodes. Data-dependences between two operations in the same node, at the same run-time iteration of enclosing loops, are not projected. We say that a projected dependence is *motivated* by a variable, when one of its original dependences relates two accesses to this variable. A projected data-dependence may very well be motivated by many variables. Notice also that, once dependences are projected, the ordering information *inside* a given node is lost. Therefore, further reordering inside a given node is forbidden.

Nodes must not be too large: each node must represent one (vectorial) mathematical function. Therefore a single node must not write a value into a variable, and then use this variable. In other words, only values that exist *before* the node execution may be used by the node. We call these nodes *valid*.

Because of their importance in adjoint programs (*cf* figure1), we shall give a special role to a very particular kind of operations: increments on a variable. Thus, we define $E(N, v)$, the *effect* of a connex and valid node $N$ on a variable or memory location $v$, by:

- $E(N, v) = \textcircled{\textbf{n}}$ when variable $v$ does not occur at all inside $N$.
- $E(N, v) = \textcircled{\textbf{r}}$ when $v$ is only read and not overwritten in $N$.
- $E(N, v) = \textcircled{\textbf{i}}$ when $v$ is read only once. Its value is incremented, and is reassigned to $v$. There are no other occurrences of $v$ in $N$.
- $E(N, v) = \textcircled{\textbf{w}}$ otherwise.

We now introduce our main refinement to ddg. Similarly to what happens between *reads*, we notice that there is no data-dependence between two *increments* of a given variable, because of associativity and commutativity of the sum. However, this is true only if increments are *atomic*. If this is not the case, two increment operations done in parallel may create a race condition. In the following, we assume increments are atomic. This can be achieved at low level, using semaphores, or at high level, using *reduction* declarations. Also, atomicity is granted when the program is run sequentially.

We can now give an equivalent definition of data-dependence between nodes. There is a data-dependence from node $N_1$ to $N_2$, motivated by variable $v$, iff $N_1 \prec N_2$ and $\mathcal{D}(E(N_1, v), E(N_2, v))$, with relation $\mathcal{D}$ defined by the table on the right.

|   | ⓦ | ⓡ | ⓘ | ⓝ |
|---|---|---|---|---|
| ⓦ | ● | ● | ● |   |
| ⓡ | ● |   | ● |   |
| ⓘ | ● | ● |   |   |
| ⓝ |   |   |   |   |

Figure 2 illustrates the above definitions, and presents the ddg of program $\overline{\texttt{normalize}}$ shown on figure 1. We define the nodes as the four instructions $I_{[1..4]}$, plus the four adjoint composite instructions $\overline{I_{[4..1]}}$. One can easily check that these nodes are all connex and valid. The arrows show the data-dependences between them. The dependences between instructions $I_{[1..4]}$ are motivated by variables `x`, `y`, and `r`, while dependences between the $\overline{I_{[4..1]}}$ are motivated by $\overline{\texttt{x}}$, $\overline{\texttt{y}}$, and $\overline{\texttt{r}}$. The refinement introduced above, about *increments*, leads to setting no dependence between the successive increments of $\overline{\texttt{r}}$ in the adjoint part. Also recall from section 2 that the values of `x`, `y`, and `r` used in the adjoint nodes are provided by the *save* and *restore* mechanism, not represented here for readability.

## 4 The Adjoint Data-Dependence Graph

**General Context:**
We consider $P$, a (fragment of) program or subprogram, composed of instructions $I_i, i \in [1..n]$. These instructions are distributed into so-called nodes $N_j, j \in [1..m]$ that contain one or many always consecutive $I_i$. We call $d : [1..n] \longrightarrow [1..m]$, the function that defines this distribution, such that $I_i \in N_{d(i)}$. We suppose nodes $N_j$ are all connex and valid. We call $\mathcal{G}$ the projection of the data-dependence graph of $P$ onto the nodes $N_j$. We call $\overline{P}$ the adjoint of $P$, with instructions $\overline{I_i}, i \in [1..n]$ as defined in section 2. We saw that each $\overline{I_i}$ may indeed consist of several assignments, but we consider it as a single composite instruction that we will never split or rearrange. We distribute adjoint instructions $\overline{I_i}$ into adjoint nodes $\overline{N_j}, j \in [1..m]$, by means of the same distribution function $d$. Therefore there is a canonical bijection between the $N_j$ and the $\overline{N_j}$. Finally, we call $\overline{\mathcal{G}}$ the projection of the data-dependence graph of $\overline{P}$ onto the nodes $\overline{N_j}$.

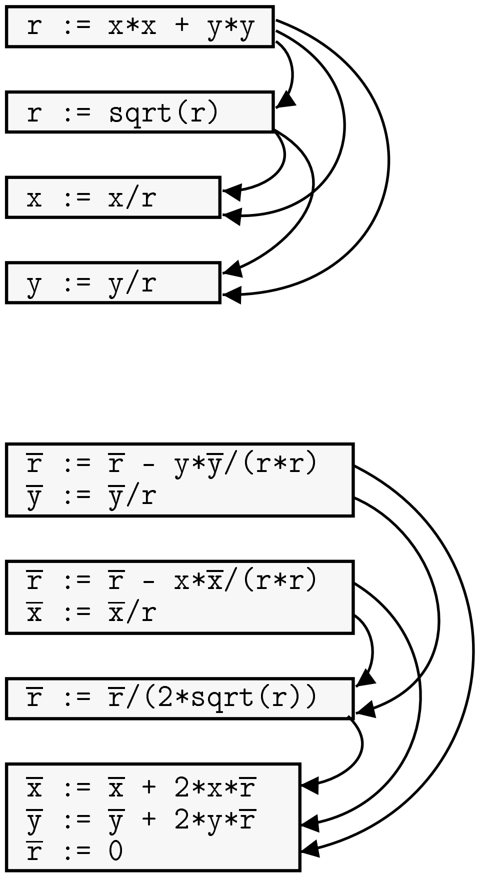
r := x*x + y*y
r := sqrt(r)
x := x/r
y := y/r
$\overline{r}$ := $\overline{r}$ - y*$\overline{y}$/(r*r)
$\overline{y}$ := $\overline{y}$/r
$\overline{r}$ := $\overline{r}$ - x*$\overline{x}$/(r*r)
$\overline{x}$ := $\overline{x}$/r
$\overline{r}$ := $\overline{r}$/(2*sqrt(r))
$\overline{x}$ := $\overline{x}$ + 2*x*$\overline{r}$
$\overline{y}$ := $\overline{y}$ + 2*y*$\overline{r}$
$\overline{r}$ := 0

Figure 2: *data-dependence graph of program from figure 1*

**Lemma 1** ***(validity of adjoint nodes)*** *In the above "general context", the adjoint nodes $\overline{N_j}$ are connex and valid.*

**Proof:**
For each $j \in [1..m]$, we know $N_j$ is connex, i.e. for every execution of $P$, all instructions in $N_j$ are executed in sequence. By construction of the adjoint program, every execution of $\overline{P}$ executes instructions $\overline{I_i}$ in the reverse order of some execution of the $I_i$ by $P$. Therefore every execution of $\overline{P}$ executes instructions of $\overline{N_j}$ in sequence, and thus $\overline{N_j}$ is connex.

Let us now show that each $\overline{N_j}$ is valid, i.e. contains no variable written and then used, inside the same execution of $\overline{N_j}$. Consider the list $v_k, k \in [1..q]$ of all variables occurring in $P$. We can suppose with no loss of generality that the $v_k$ are reordered so that the first variable written during an execution of $N_j$ is $v_1$, the next one $v_2$, and so on, and the variables that are only read or not used come at the end. Since $N_j$ is valid, no variable is written and then used, inside the same execution of $N_j$. Therefore the value assigned to some variable $v_k$ (if any) does not depend on the $v_i, i < k$. The Jacobian matrix of $N_j$ is thus upper triangular:

$$J(N_j) \;=\; \begin{pmatrix} \bullet & \bullet & \bullet & \cdots & \bullet \\ & \bullet & \bullet & \cdots & \bullet \\ & & \bullet & \cdots & \bullet \\ & & & \ddots & \vdots \\ & & & & \bullet \end{pmatrix} \tag{1}$$

By definition (*cf* section 2) the adjoint of $N_j$ is a sequence of instructions that implements the product of the transposed Jacobian matrix $J(N_j)^T$ by the vector of adjoint variables $\overline{v_k}, k \in [1..q]$, and assigns the result into this same vector:

$$\begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} := \begin{pmatrix} \bullet & & & & \\ \bullet & \bullet & & & \\ \bullet & \bullet & \bullet & & \\ \vdots & \vdots & \vdots & \ddots & \\ \bullet & \bullet & \bullet & \cdots & \bullet \end{pmatrix} \times \begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} \tag{2}$$

We can rewrite the above with an upper triangular matrix by just reversing the order of the adjoint variables:

$$\begin{pmatrix} \overline{v_q} \\ \vdots \\ \overline{v_3} \\ \overline{v_2} \\ \overline{v_1} \end{pmatrix} := \begin{pmatrix} \bullet & \cdots & \bullet & \bullet & \bullet \\ & \ddots & \vdots & \vdots & \vdots \\ & & \bullet & \bullet & \bullet \\ & & & \bullet & \bullet \\ & & & & \bullet \end{pmatrix} \times \begin{pmatrix} \overline{v_q} \\ \vdots \\ \overline{v_3} \\ \overline{v_2} \\ \overline{v_1} \end{pmatrix} \tag{3}$$

and this is implemented by a *valid* adjoint node $\overline{N_j}$, by writing the instruction that assigns $\overline{v_q}$ first, then the instruction for $\overline{v_{q-1}}$, and so on until $\overline{v_1}$. □

We want to relate the effect $E(N,v)$ of a node $N$ of $\mathcal{G}$, on some variable $v$, with the effect $E(\overline{N},\overline{v})$ of the corresponding node of $\overline{\mathcal{G}}$ on $\overline{v}$, adjoint of $v$.

**Lemma 2** ***(effect of adjoint nodes)*** *In the above "general context", the effect on $\overline{v}$ of the adjoint $\overline{N}$ of a given data-dependence node $N$ is related to the effect of $N$ on $v$ in the following manner:*

- $E(N,v) = \textcircled{\textbf{n}} \implies E(\overline{N},\overline{v}) = \textcircled{\textbf{n}}$
- $E(N,v) = \textcircled{\textbf{r}} \implies E(\overline{N},\overline{v}) \in \{\textcircled{\textbf{i}},\textcircled{\textbf{n}}\}$
- $E(N,v) = \textcircled{\textbf{i}} \implies E(\overline{N},\overline{v}) \in \{\textcircled{\textbf{r}},\textcircled{\textbf{n}}\}$
- $E(N,v) = \textcircled{\textbf{w}} \implies E(\overline{N},\overline{v}) \in \{\textcircled{\textbf{w}},\textcircled{\textbf{r}},\textcircled{\textbf{i}},\textcircled{\textbf{n}}\}$

**Proof:**
Consider any variable $v_k$. Since $N$ is valid, the effect of $N$ on $v_k$ is defined, and belongs to $\{\textcircled{\textbf{w}},\textcircled{\textbf{r}},\textcircled{\textbf{i}},\textcircled{\textbf{n}}\}$. Let us focus on the $k$-th line and column of the Jacobian matrix $J(N)$ of equation 1. If $E(N,v_k) = \textcircled{\textbf{n}}$, then these line and column are all 0, except the diagonal element, which is 1. If $E(N,v_k) = \textcircled{\textbf{r}}$, then the $k$-th line is all 0, with 1 on the diagonal. If $E(N,v_k) = \textcircled{\textbf{i}}$, then the $k$-th column is all 0, with 1 on the diagonal. And if $E(N,v_k) = \textcircled{\textbf{w}}$, the $k$-th line and column can be anything. We summarize this as:

| $E(N,v_k)$ : | $\textcircled{\textbf{n}}$ | $\textcircled{\textbf{i}}$ | $\textcircled{\textbf{r}}$ | $\textcircled{\textbf{w}}$ |
|---|---|---|---|---|
| $J(N)$ : | $\begin{pmatrix} \ddots & 0 & \\ & 1 & 0 \\ & & \ddots \end{pmatrix}$ | $\begin{pmatrix} \ddots & 0 & \\ & 1 & \bullet \\ & & \ddots \end{pmatrix}$ | $\begin{pmatrix} \ddots & \bullet & \\ & 1 & 0 \\ & & \ddots \end{pmatrix}$ | $\begin{pmatrix} \ddots & \bullet & \\ & \bullet & \bullet \\ & & \ddots \end{pmatrix}$ |

where the $\bullet$ elements can very well be 0 or 1, as will be shown below on some degenerate cases. As shown above, $\overline{N}$ implements equation 3. We observe that the matrix in equation 3 (call it $\overline{J(N)}$), is the symmetric of $J(N)$ with respect to the second diagonal. We have thus:

| $E(N, v_k)$ : | ⓝ | ⓘ | ⓡ | ⓦ |
|---|---|---|---|---|
| $\overline{J(N)}$ : | $\begin{pmatrix}\ddots & 0 & \\ & 1 & 0 \\ & & \ddots\end{pmatrix}$ | $\begin{pmatrix}\ddots & \bullet & \\ & 1 & 0 \\ & & \ddots\end{pmatrix}$ | $\begin{pmatrix}\ddots & 0 & \\ & 1 & \bullet \\ & & \ddots\end{pmatrix}$ | $\begin{pmatrix}\ddots & \bullet & \\ & \bullet & \bullet \\ & & \ddots\end{pmatrix}$ |

which implies the following:

- when $E(N, v_k) =$ ⓝ, $\overline{N}$ simply does not use nor modify $\overline{v_k}$. Therefore $\overline{v_k}$ does not occur in $\overline{N}$ and $E(\overline{N}, \overline{v_k}) =$ ⓝ.

- when $E(N, v_k) =$ ⓡ, $\overline{N}$ does not read $\overline{v_k}$, except in one instruction which adds some value into $\overline{v_k}$. In the special case where the $\bullet$ values are all 0, $\overline{v_k}$ is just unmodified. Therefore $E(\overline{N}, \overline{v_k}) \in \{$ⓘ,ⓝ$\}$.

- when $E(N, v_k) =$ ⓘ, $\overline{N}$ may read $\overline{v_k}$ several times, to compute values assigned to other adjoint variables, and then $\overline{v_k}$ itself is just unmodified. In the special case where the $\bullet$ values are all 0, $\overline{v_k}$ is not used at all. Therefore $E(\overline{N}, \overline{v_k}) \in \{$ⓡ,ⓝ$\}$.

- when $E(N, v_k) =$ ⓦ, $\overline{N}$ may read $\overline{v_k}$ several times, and then overwrite it with some value, and then not use it any more. Therefore $E(\overline{N}, \overline{v_k})$ may be ⓦ. However, since any of the $\bullet$ elements may be 0, and the diagonal $\bullet$ may be 1, $E(\overline{N}, \overline{v_k})$ may degenerate to ⓡ, ⓘ, or even ⓝ. Therefore $E(\overline{N}, \overline{v_k}) \in \{$ⓦ,ⓡ,ⓘ,ⓝ$\}$. $\square$

Here are two examples of degenerate cases. Suppose $N$ is instruction `y:=floor(x)`, where `floor` returns the integer part of a real number. The derivative of `floor`, when defined, is

$$\frac{\partial \texttt{floor(x)}}{\partial \texttt{x}} = 0$$

Node $N$ is valid, and $E(N,\mathtt{x}) = \textcircled{r}$. With respect to vector $(\mathtt{y},\mathtt{x})$, $J(N)$ is equal to $\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$. Therefore $\overline{N}$ implements $\begin{pmatrix} \overline{\mathtt{x}} \\ \overline{\mathtt{y}} \end{pmatrix} := \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \times \begin{pmatrix} \overline{\mathtt{x}} \\ \overline{\mathtt{y}} \end{pmatrix}$, thus $\overline{N}$ contains only instruction $\overline{\mathtt{y}}$`:=0`, and $E(\overline{N},\overline{\mathtt{x}}) = \textcircled{n}$.

As a second example, suppose $N$ contains the two successive instructions `y:=2*x; x:=x+floor(x)`. Node $N$ is valid, and $E(N,\mathtt{x}) = \textcircled{w}$. On vector $(\mathtt{y},\mathtt{x})$, $J(N)$ is $\begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix}$, and $\overline{N}$ implements $\begin{pmatrix} \overline{\mathtt{x}} \\ \overline{\mathtt{y}} \end{pmatrix} := \begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix} \times \begin{pmatrix} \overline{\mathtt{x}} \\ \overline{\mathtt{y}} \end{pmatrix}$, thus $\overline{N}$ contains instruction $\overline{\mathtt{x}}$`:=`$\overline{\mathtt{x}}$ `+2*`$\overline{\mathtt{y}}$ followed by $\overline{\mathtt{y}}$`:=0`, and $E(\overline{N},\overline{\mathtt{x}}) = \textcircled{i}$.

From lemma 2, we are now able to go from the effect of adjoint nodes back to the effect of the original nodes.

**Lemma 3** ***(effect of original nodes)*** *In the above "general context", the effect of a node $N$ on a variable $v$ can be deduced from the effect of its adjoint $\overline{N}$ on $\overline{v}$ in the following manner:*

- $E(\overline{N},\overline{v}) = \textcircled{n} \implies E(N,v) \in \{\textcircled{w},\textcircled{r},\textcircled{i},\textcircled{n}\}$
- $E(\overline{N},\overline{v}) = \textcircled{r} \implies E(N,v) \in \{\textcircled{w},\textcircled{i}\}$
- $E(\overline{N},\overline{v}) = \textcircled{i} \implies E(N,v) \in \{\textcircled{w},\textcircled{r}\}$
- $E(\overline{N},\overline{v}) = \textcircled{w} \implies E(N,v) = \textcircled{w}$

**Proof:**
Since $N$ and $\overline{N}$ are valid, their effect on variables $v$ and $\overline{v}$ is defined, and must be one of $\textcircled{w}$, $\textcircled{r}$, $\textcircled{i}$, or $\textcircled{n}$. Therefore, it suffices to explore all possible cases in lemma 2. $\square$

We now use our refined notion of data-dependence, as defined in section 3. We insist this supposes that incrementation operations ($\textcircled{i}$) are *atomic*. With this hypothesis, which can be enforced in various ways, we prove that the data-dependence graph of the adjoint program is isomorphic to a subgraph of the original data-dependence graph. This will allow us to transpose many properties of the original program to its adjoint.

**Theorem 1** ***(adjoint data-dependences)*** *In the above "general context", if $\overline{\mathcal{G}}$ has an arrow from node $\overline{N_a}$ to node $\overline{N_b}$, then $\mathcal{G}$ has an arrow from node $N_b$ to node $N_a$. Moreover, if the arrow from $\overline{N_a}$ to $\overline{N_b}$ is motivated by a variable $w$, then the arrow from $N_b$ to $N_a$ is motivated by a variable $v$ such that $\overline{v} = w$.*

**Proof:**
Consider an arrow in $\overline{\mathcal{G}}$, going from node $\overline{N_a}$ to node $\overline{N_b}$. By definition of data-dependence, this implies that $\overline{N_a} \prec \overline{N_b}$, which in turn implies by construction of the adjoint program that $N_b \prec N_a$. By definition, the data-dependence is motivated by (at least) one variable $w$. This means that for any such variable $w$, the effects of $\overline{N_a}$ and $\overline{N_b}$ on $w$ are such that $\mathcal{D}(E(\overline{N_a}, w), E(\overline{N_b}, w))$, $\mathcal{D}$ defined in section 3. This implies that variable $w$ is assigned, somewhere in $\overline{N_a}$, $\overline{N_b}$, or both. Since the adjoint nodes $\overline{N_j}$ only assign adjoint variables, $w$ is necessarily the adjoint $\overline{v}$ of some variable $v$ in $P$. Considering each case where $\mathcal{D}(E(\overline{N_a}, \overline{v}), E(\overline{N_b}, \overline{v}))$, we check that $\mathcal{D}(E(N_b, v), E(N_a, v))$. For example, suppose $E(\overline{N_a}, \overline{v}) = ⓦ$ and $E(\overline{N_b}, \overline{v}) = ⓘ$. By lemma 2, we get $E(N_a, v) \in \{ⓘ,ⓦ\}$ and $E(N_b, v) \in \{ⓡ,ⓦ\}$, and in the four resulting cases, we can check that $\mathcal{D}(E(N_b, v), E(N_a, v))$. Together with $N_b \prec N_a$, this shows that there is an arrow in $\mathcal{G}$, motivated by $v$, that goes from $N_b$ to $N_a$. $\square$

## 5 Applications

Consider figure 1 again. In the original program, there is no data-dependence between instructions $I_3$ and $I_4$. The theorem then tells us there is no data-dependence between instructions $\overline{I_4}$ and $\overline{I_3}$.

Consider now a vectorial instruction, such as

```
A(1:n) := B(0:n-1)*B(2:n+1) + c
```

There are no loop-carried dependences in the loop implicitly represented here, and therefore no loop-carried dependences in the adjoint loop. As far as dependences are concerned, the adjoint is itself vectorial. Notice that the read of `c` is spread among the "iterations", and this results in a `SUM` reduction in the adjoint program:

```
B̅(0:n-1) := B̅(0:n-1) + B(2:n+1)*A̅(1:n)
B̅(2:n+1) := B̅(2:n+1) + B(0:n-1)*A̅(1:n)
c̅ := c̅ + SUM(A̅(1:n))
A̅(1:n) := 0
```

On the other hand, the adjoint of instruction:

```
        A(1:n) := A(0:n-1)*A(2:n+1)
```

would not be immediately vectorial, because of the loop-carried data-dependence in the implicit loop, from the reads to the writes of `A`.

The theorem also applies to parallel loops. If a loop is parallel, there are no loop-carried dependences. Therefore, provided increments are kept atomic, the adjoint loop is parallel too. In a language such as OpenMP, we can force this atomicity using a `CRITICAL` section, or alternatively with clauses that declare the incremented adjoint variables as `REDUCTION` variables.

The theorem also has an interesting application in sequential programs. Suppose a loop has data-independent iterations, i.e. there are no loop-carried dependences. Suppose also that the loop is immediately followed by its adjoint loop. Then the theorem implies that there are no loop-carried dependences in the adjoint loop. The two loops, original and adjoint, can thus be merged into a single one. The advantage is that the *save* and *restore* operations are now interleaved, and this requires far less storage space than doing all the *save*'s first. This partly solves the main drawback of the adjoint program technique. This application is described in more detail in [6].

## 6 Conclusion

Adjoint programs are very complex and heavy, and therefore their generation by tools is welcome. But execution cost is such that further optimization is often necessary. Nowadays, this optimization is mostly done by hand, which is very unsafe. The theorem proved here is a foundation of a framework for automatic correct optimization of adjoint programs. In this framework, already existing optimizations can be checked from the data dependence viewpoint. This may also suggest new optimizations.

We proved that the data-dependence graph of an adjoint program is isomorphic to a sub-graph of the original data-dependence graph. This means there are even fewer data-dependences in the adjoint program than in the original. Parallelization, vectorization, and more generally all rescheduling of program operations rely on data-dependences. The present theorem allows us to transpose most of these transformations, from the original program to its adjoint. We believe this sort of synergy between A.D. and compilation theory improves our understanding of A.D.

The application of this theorem to the reduction of the *save* memory size is promising. We are currently testing it on a large size application in fluid dynamics. An extension of the theorem to message-passing, distributed memory parallelism also appears an interesting research direction.

## References

[1] R.Sethi A.Aho and J.Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] A.Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000.

[3] J.M.Malé B.Mohammadi and N.Rostaing-Schmidt. *Automatic differentiation in direct and reverse modes: application to optimum shapes design in fluid mechanics*. In M.Berz, C.Bischof, G.Corliss, A.Griewank, eds., SIAM, editor, *Computational Differentiation: Techniques, Applications and Tools*, pages 309–318, 1996.

[4] H.Zima and B.Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.

[5] J.L.Lions. *Optimal control of systems governed by partial differential equations*. Springer, 1971.

[6] S.Fidanova L.Hascoët and C.Held. *Adjoining Independant Computations*. In G.Corliss, C.Faure, A.Griewank, L.Hascoët, U.Naumann, eds., SPRINGER LNCSE, editor, *Automatic Differentiation: From Simulation to Optimization*, 2001. Selected proceedings of AD2000, Nice, France.

[7] M.Wolfe and U.Banerjee. *Data Dependence and its application to Parallel Processing*. *International Journal of Parallel Processing*, 16(2):137–178, 1987.

[8] O.Pironneau. *Optimal shape design for elliptic problems*. Springer, 1982.

[9] O.Talagrand. *The use of adjoint equations in numerical modelling of the atmospheric circulation.* In A.Griewank, G.Corliss, eds., SIAM, editor, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 169–180, 1991.

[10] B.Mohammadi P.Hovland and C.Bischof. *Automatic Differentiation of Navier-Stokes computations.* Technical report, Argonne National Laboratory MCS-P687-0997, 1997.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399