Différentiation symbolique orientée objets

Laurent Cogné, Bruno Arnaldi

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Différentiation symbolique orientée objets

Laurent Cogné et Bruno Arnaldi

N° 2601

Juin 1995

PROGRAMME 4

ISSN 0249-6399

INRIA
RENNES

Différentiation symbolique orientée objets

Laurent Cogné* et Bruno Arnaldi**

Programme 4 — Robotique, image et vision
Projet Siames

Rapport de recherche n° 2601 — Juin 1995 — 28 pages

Résumé : Ce rapport présente un noyau orienté-objets de différentiation symbolique, destiné à être utilisé dans un système d'animation et de simulation. Différents travaux sur la différentiation sont présentés et discutés. À partir de cette étude nous mettons en lumière les besoins d'un tel noyau pour l'application visée. Les concepts de base du noyau sont présentés et deux aspects particuliers sont développés : la gestion des composants des expressions et l'utilisation du "double polymorphisme" dans la simplification de ces expressions. L'efficacité du noyau, due à l'utilisation d'opérateurs vectoriels et matriciels, est illustrée par un exemple concret de mécanisme. Avant de conclure nous présentons les possibilités d'extension du noyau en y ajoutant un opérateur.

Mots-clé : Différentiation symbolique, double polymorphisme, animation, simulation.

(Abstract: pto)

*. Laurent.Cogne@irisa.fr
** Bruno.Arnaldi@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Object oriented symbolic differentiation

Abstract: This paper presents an object-oriented symbolic kernel dedicated to differentiation and its utilization in a system for both animation and simulation. Previous works in the differentiation area are presented and discussed. From this study we derive the needs of such a kernel. The kernel's basic concepts are presented from the user interface side, as well as from its internal structure. The advantages of an object-oriented langage for the organization and the processing of data structures are shown by the exposition of two specials aspects : the management of the expression's components and the use of "double polymorphism" in the simplifications of those expressions. The efficiency of the kernel, due to the use of vectorial an matricial operators, is shown on a concrete example of mechanism. We then present kernel's extension possibilities by adding an operator in the kernel.

Key-words: Symbolic differentiation, double polymorphism, animation, simulation

Table des matières

Table des figures

1 Introduction

L'utilisation des lois de la physique dans des systèmes d'animation ou de synthèse d'images est maintenant reconnu comme étant un des moyens les plus sûrs d'obtenir des résultats réalistes tout en déchargeant l'animateur d'un travail fastidieux. De plus, la qualité des résultats numériques de ces systèmes (dit générateurs) permet de dépasser le cadre de l'animation pour travailler dans celui de la simulation. Dans le domaine plus précis de la simulation de systèmes mécaniques composés de corps rigides poly-articulés, les lois physiques utilisées sont celles de la dynamique [3, 8].

Le but de ce rapport est de montrer que les paradigmes de la conception orientée-objets [4, 17] peuvent être utilisés pour une mise en œuvre efficace d'un noyau de différentiation symbolique. Ces recherches sont conduites dans le cadre du développement du logiciel NMecam. Ce système d'animation et de simulation, développé à l'Irisa, se base sur le principe des travaux virtuels et les équations de Lagrange [1, 7]. Associé à une méthode de pénalisation ce formalisme permet de construire le système algébro-différentiel constitué par l'ensemble d'équations (1) :

$$f_i = \mathcal{Q}_i - \frac{d}{dt}(\frac{\partial \mathcal{C}}{\partial \dot{q}_i}) + \frac{\partial \mathcal{C}}{\partial q_i} + k\frac{\partial {f_h}^2}{\partial q_i} + k\frac{\partial {g_l}^2}{\partial \dot{q}_i} \tag{1}$$

Où les q_i sont les paramètres du mécanisme, $\mathcal{C}$ son énergie cinétique, f_h (resp. g_l) les contraintes holonomes (resp. non holonomes) et k une constante de pénalisation. L'algorithme de résolution employé (Newton-Raphson) pour résoudre ce système nécessite le calcul de la matrice jacobienne $\mathcal{J}$ de terme générique :

$$\mathcal{J}_{ij} = \frac{\partial f_i}{\partial q_j} \tag{2}$$

Les équations (1) et (2) montrent bien l'importance de la différentiation. Dans notre système, celle-ci est effectuée de manière symbolique.

Nous ne nous attarderons pas sur les avantages du calcul symbolique par rapport au calcul numérique (et plus particulièrement ici la différentiation) pour la construction et la résolution des équations du mouvement. L'expérience nous a montré que des systèmes algébriques généraux tels que Macsyma et Maple, s'ils sont précieux lors de prototypages, sont difficilement utilisables comme noyau d'un système d'animation et de simulation. Cette conclusion nous a conduit à concevoir un noyau de différentiation symbolique dédié.

Afin de mettre en évidence les avantages de l'utilisation d'un formalisme orienté-objets dans la mise en œuvre d'un tel noyau, ce rapport est organisé de la façon suivante : tout d'abord nous présentons un aperçu des travaux menés dans le cadre de la différentiation en insistant sur la façon dont sont utilisées les particularités des langages d'implémentation choisis. De cette étude nous déduisons les besoins de notre noyau dont nous présentons les concepts de base. Parmi les différents problèmes rencontrés nous en présentons deux des plus significatifs : la gestion des nœuds dans le graphe des expressions construites, et l'utilisation du "double polymorphisme" lors de la mise en œuvre de simplifications dans la

construction de ces expressions. Nous montrons ensuite les avantages de la représentation choisie en matière de complexité spatiale. Avant de conclure nous mettons en évidence les possibilités d'extension du noyau sur un exemple présentant l'ajout d'un opérateur.

2 La différentiation symbolique

Dans cette partie, nous présentons un certain nombre de travaux dans le domaine de la différentiation, leurs intérêts et leurs manques par rapport à nos besoins.

Dans [18], L.B. Rall présente le calcul symbolique de dérivées comme une translation du code représentant le calcul de la fonction, grâce à un dictionnaire de règles de transformation (cf figure 1). Cette approche implique la construction explicite des expressions dérivées et

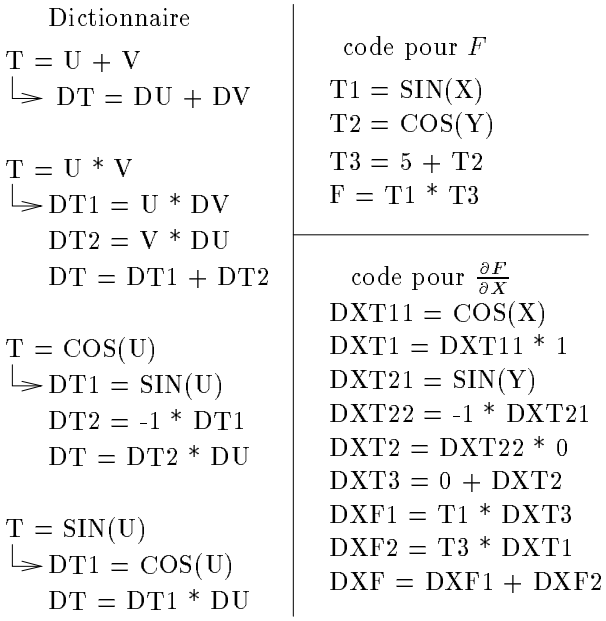

FIG. 1 - *Différentiation symbolique par translation de code*

peut parfois poser des problèmes de taille des expressions générées. Elle permet cependant la réentrance du processus, le code produit pouvant être à nouveau dérivé.

Souvent présenté comme une méthode concurrente, le calcul automatique de dérivée se base en général sur l'équation (3). Dans la mesure où quelque soit $f(x_i, \ldots, x_n)$ on est capable de calculer les $\frac{\partial f}{\partial x_i}$ en fonction des x_i, cette méthode permet de calculer dans une même passe la valeur de la fonction et celle de sa dérivée (figure 2).

$$\frac{\partial f(x_1,\ldots,x_n)}{\partial y} = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \cdot \frac{\partial x_i}{\partial y} \tag{3}$$

$$\frac{\partial y}{\partial z} = \begin{cases} 1 & \text{si } y = z \\ 0 & \text{sinon} \end{cases} \quad \forall y, z \text{ variables libres}$$

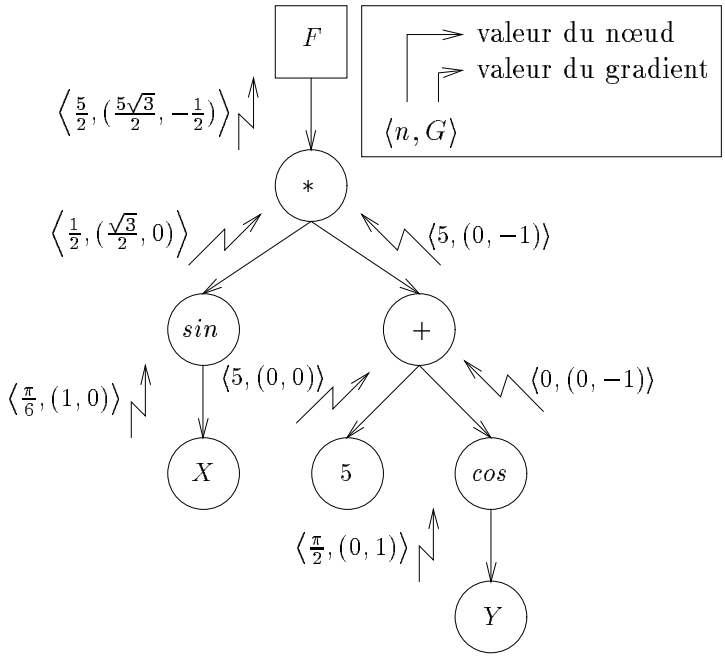

FIG. 2 - *Différentiation automatique de F par rapport à X et Y, avec $X = \frac{\pi}{6}$ et $Y = \frac{\pi}{2}$*

Cette méthode est d'un coût mémoire moindre quand il suffit de calculer des dérivées du premier ordre mais cet avantage se perd dès que l'on veut atteindre les ordres supérieurs, la construction d'information sur le graphe de calcul étant alors nécessaire.

En fait, la différence majeure entre les méthodes de dérivation se situe dans le sens dans lequel le graphe de calcul de la fonction à dériver est traversé, plus que dans le choix des techniques symboliques ou automatiques. On peut trouver dans l'article de A. Griewank [11] une bonne synthèse de ces méthodes, dites directes et inverses. Afin de gérer au mieux la complexité des expressions construites par dérivation, il est important de disposer de ces deux méthodes [20].

Les outils nous permettant de calculer la dérivée d'une fonction sont de deux types : les systèmes algébriques généraux tels que Maple, Macsyma, Mathematica, et les différentiateurs

de code du type JAKEF, PADRE2 ou Adol-C[1]. Les premiers, parfaits pour le prototypage, sont difficilement intégrables dans une application, constituent une "boîte noire" sur laquelle on ne possède aucun contrôle et ne proposent à l'heure actuelle que la différentiation directe. Les seconds, plus spécialisés, disposent en général des méthodes directes et inverses de dérivation. Ils utilisent souvent des techniques de précompilation (JAKEF, PADRE2) mais aussi la surcharge (Adol-C). L'inconvénient de ces différentiateurs est que, ne générant pas de code explicite pouvant être à nouveau dérivé (réentrance), nous ne pouvons les utiliser pour la construction des équations du mouvement *et* pour celle de leur jacobienne. D'autre part, leur utilisation ne nous permettrait pas de manipuler le graphe des dérivées construites. Il est quand même intéressant d'étudier comment les différientiateurs basés sur la surcharge utilisent celle-ci.

La surcharge permet de mettre en place les mécanismes de différentiations en minimisant les transformations à effectuer par l'utilisateur sur son code. Utilisée par L. B. Rall en PASCAL-SC [19], elle permet l'écriture d'une expression utilisant un type particulier de variables. Ces variables contiennent en plus de leur valeur scalaire, la valeur de leur gradient. Les opérateurs arithmétiques élémentaires sont surchargés pour prendre en compte ce nouveau type et effectuer le calcul automatique du gradient lors de l'évaluation de l'expression. M. E. Jerrel [14] reprend ce principe en ajoutant une matrice permettant le calcul de l'hessian. Sa mise en œuvre, du fait des plus grandes possibilitées du C++ en matière de surcharge, permet l'utilisation d'un plus grand nombre d'opérateurs. Ces travaux montrent bien l'intérêt et les possibilités de la surcharge. Cependant, mal adaptés à la création dynamique d'expressions dérivables et à la gestion de grosses équations, ils sont d'une utilisation limitée.

Adol-C [12] par contre, constitue un véritable différentiateur. Il est basé sur un principe différent : l'évaluation d'une expression utilisant les opérateurs surchargés n'entraîne pas le calcul immédiat de la valeur de ses dérivées, mais construit un graphe la représentant. Il est alors possible d'obtenir le calcul de dérivées de plus hauts degrés et de disposer d'un ensemble de méthodes pour les effectuer.

Parallèlement à ces travaux, nous pouvons mentionner l'utilisation de la différentiation automatique par M. Gleicher et A. Witkin [10]. Ils proposent un ensemble de classes permettant de donner des fonctionnalités mathématiques à des objets géométriques ou à des mécanismes sous contraintes, ceci permettant leur manipulation interactive. Cette approche, très intéressante dans le cadre d'un système d'animation, met bien à profit le formalisme orienté objet. Cependant, la simulation de systèmes mécaniques demande des résultats numériques précis (même au dépend des temps de calcul) et nous impose une représentation détaillée des expressions mathématiques utilisées pour la résolution de ces mécanismes.

Notre démarche, pour l'écriture de notre noyau, sera d'effectuer la construction symbolique explicite d'une fonction et de ses dérivées (comme dans un système algébrique). Cette fonction sera décrite par un code qui, grâce à l'utilisation de la surcharge permise par le C++, restera très proche de celui qui aurait été "naturellement" écrit pour la simple évaluation de cette fonction en C.

1. pour une synthèse, voir [15].

3 Concepts de base

Dans cette partie nous présentons tout d'abord l'aspect utilisateur du noyau avant de discuter des choix de représentation interne de celui-ci.

3.1 Interface du noyau

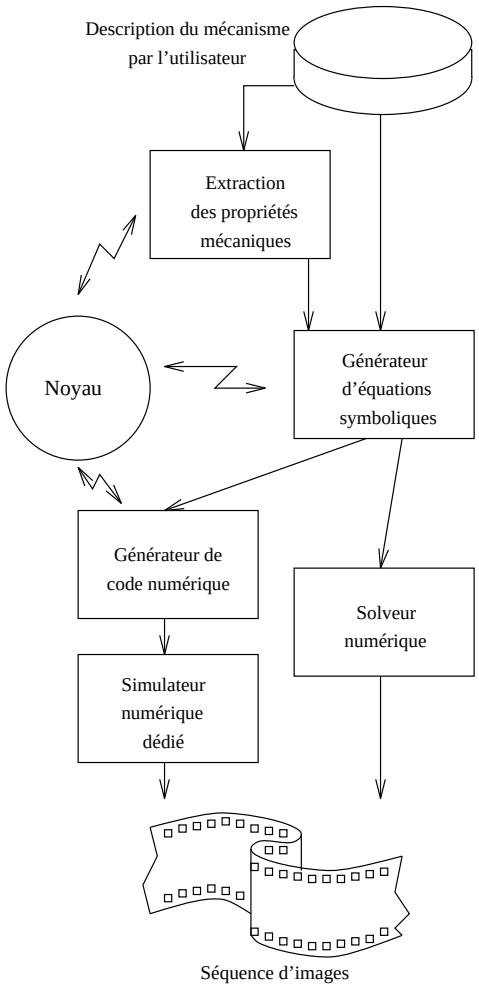

FIG. 3 - *Synopsis simplifié de NMecam*

Le noyau est conçu comme une boîte à outils autour de laquelle l'utilisateur construit son application. La figure 3 montre l'exemple d'une telle intégration à partir du synopsis

simplifié de NMecam. La connection avec les applications est faite par programmation, dans le cas présent avec le C++ [9], lui même utilisé pour la mise en œuvre du noyau.

La tâche principale de NMecam est la construction des équations du mouvement telles qu'elles sont décrites dans l'équation (1). Cette méthode n'est pas unique et il existe d'autres formalismes permettant de construire ces équations, chacun ayant ses avantages et inconvénients [8].

```
// degrés de libertés :
VarSet q;
// énergie cinétique et travaux donnés:
scalar C, W;
// sommes des contraintes holonomes et non-holomes:
scalar fh, gl;
// constante de penalisation :
double k;
// vecteur des équations du mouvement :
vector motion;
// matrice jacobienne :
matrix jac;

...
// calcul de l'énergie cinétique, des travaux,
// des contraintes holonomes et non-holonomes
// relativement à une description du mécanisme
...

// calcul des équations du mouvement :
motion = gradient(W, q) + dotted_gradient(W, q)
              + k * gradient(fh, q) + k * dotted_gradient(gl, q);
              - ddt(dotted_gradient(C, q)) + gradient(C, q)

// calcul de la jacobienne :
jac = jacobian(motion, q)

// calcul des valeurs numériques pour
// les équations du mouvement et la jacobienne :
numericMatrix valJac = jac->value();
numericVector valMotion = motion->value();
```

FIG. 4 - *Un exemple de code "utilisateur"*

La possibilité de passer de l'un à l'autre de ces formalismes est d'autant plus aisée que leurs codages restent simples et proches de leurs expressions mathématiques. La figure 4 présente la partie du code de NMecam assurant la construction des équations du mouvement.

Cet exemple montre bien comment le formalisme choisi (ici les travaux virtuels associés aux équations de Lagrange) peut être transcrit en un code qui lui est similaire. Cet aspect est particulièrement agréable lorsqu'il s'agit d'effectuer une maintenance du code vis à vis de modifications dans le formalisme mathématique.

Nous pouvons également noter que ce code utilise une forme vectorielle de l'équation (1). En effet, un des principaux buts était l'intégration d'opérateurs vectoriels et matriciels dans le noyau. L'intérêt de tels opérateurs est plus amplement discuté dans la section 6.

L'utilisateur dispose d'une classe (VarSet) lui permettant de gérer un ensemble de variables, les opérations de dérivation se faisant par rapport à des ensembles de ce type. Le noyau supporte trois types essentiels : les scalaires, les vecteurs et les matrices. Un scalaire peut être initialisé par une constante réelle, par un élément d'un ensemble de variables, ou par un autre scalaire. Dans un premier temps, nous pouvons considérer les vecteurs (resp. matrices) comme des tableaux de scalaires à une (resp. deux) dimension(s). Les opérations arithmétiques $(+, -, *, /)$ ainsi qu'un ensemble de fonctions mathématiques $(sin, cos, sqrt, \ldots)$ sont surchargées pour permettre l'écriture "naturelle" d'expressions scalaires. Un ensemble d'opérateurs de dérivation est fourni $(gradient, hessian, jacobian, \ldots)$ pour effectuer la construction symbolique des dérivées des expressions scalaires.

3.2 Structure interne

Le principe de base du noyau est de construire l'arbre de calcul des expressions programmées par l'utilisateur. L'arbre représenté dans la figure 5 est l'exemple d'une telle génération pour le fragment de code `x + 3 * y - z`. Cet arbre construit, nous devons être capable de

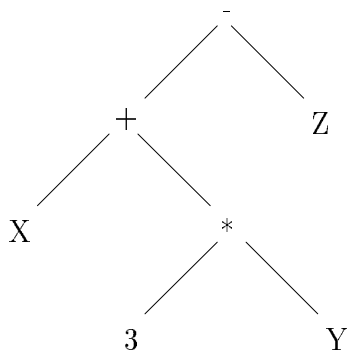

FIG. 5 - *Arbre de calcul pour l'expression* $x + 3 * y - z$

le traverser afin de pouvoir effectuer divers traitements (dérivation, évaluation, génération de code, ...). Pour la majorité de ces traitements, le travail à faire au niveau d'un nœud dépend de la nature de celui-ci. Le choix a donc été fait de représenter chaque nœud par

l'instance d'une classe représentant son type (constante, variable, addition, ...). D'autre part chaque nœud présente des caractéristiques communes: il doit pouvoir fournir la valeur numérique et l'expression symbolique de la dérivée du sous-arbre dont il est la racine ou encore servir d'opérande à n'importe quel opérateur. Ceci impose l'existence d'une classe de base, commune à tout élément du graphe, constituant l'interface minimale que doit posséder un nœud et permettant à un opérateur de référencer ses opérandes sans avoir à connaître la nature exacte de celles-ci (polymorphisme par héritage).

Si chaque nœud a un comportement différent suivant son type, il suit globalement un principe général commun à tous. Prenons par exemple le calcul de la dérivée par la méthode basée sur l'équation (3). Si les expressions des $\frac{\partial f}{\partial q_i}$ dépendent de la nature de f, leurs compositions pour former la dérivée de f est toujours la même. Le code effectuant cette partie du traitement peut donc être centralisé dans une classe de base, commune à tout opérateur. De même nous pouvons créer des classes regroupant les caractéristiques des opérateurs suivant leur arités. Ceci nous conduit au graphe d'héritage (simplifié) présenté dans la figure 6. L'utilisateur, grâce aux opérateurs surchargés pour le type scalar (dont la nature exacte est

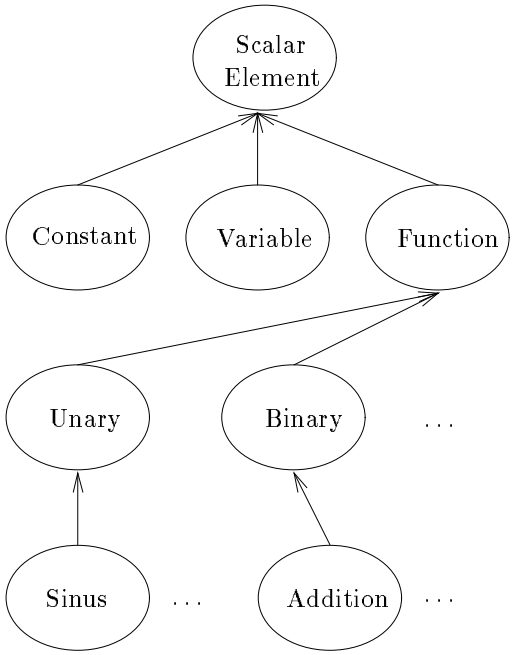

FIG. 6 - *Graphe d'héritage*

précisée plus loin), n'a pas besoin de connaître le détail de ce graphe [2].

2. voir par exemple l'opérateur "−" fig.11

Cette représentation permet de découper les algorithmes à appliquer sur le graphe en petites unités comme l'illustre le pseudo-code C++ de la figure 7. Le code pris en charge

```
scalar Function::ddq(Variable q) {
    scalar result = Constant(0);
    for (i = 1; i <=  number_of_operand; i++)
        result += operand(i).ddq(q) * ddop(i);
    return result;
}

scalar Binary::ddop(int i) {
    scalar result;
    assert(1 <= i <= 2);
    if (i == 1)
        result = ddop1();
    else
        result = ddop2();
    return result;
}

scalar Multiplication::ddop1() {
    return operand(2);
}

scalar Multiplication::ddop2() {
    return operand(1);
}
```

FIG. 7 - *Code de dérivation pour la classe* Multiplication

par une classe représentant un élément effectif du graphe d'expression est alors très réduit ce qui rend la création de nouveaux opérateurs particulièrement simple.

La taille des équations engendrées pour la représentation d'un mécanisme limite de maniere dramatique la taille des mécanismes pouvant être simulés. Pour réduire ce problème, deux démarches essentielles sont mises en œuvre : le partage de sous-expressions et la simplification. Ces aspects sont traités dans les parties suivantes.

4 Gestion des nœuds

La construction des équations du mouvement associées à un système mécanique engendre la duplication d'un nombre non négligeable d'éléments de ces équations. Pour éviter la

duplication effective de ces éléments, les équations ne sont pas représentées par un arbre mais par un graphe permettant le partage d'expressions (figure 8).

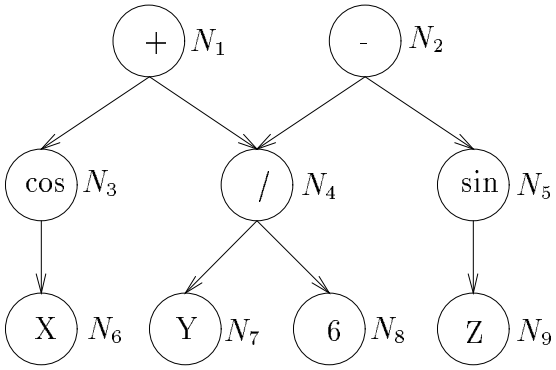

FIG. 8 - *Partage de sous-expressions*

Ceci signifie que la destruction d'un opérateur n'entraîne pas nécessairement la destruction de ces opérandes, celles-ci pouvant être référencées par d'autres opérateurs (dans l'exemple, la destruction du nœud N_1 ne doit pas entraîner la destruction de N_4 tant que N_2 est maintenu). Par contre, pour gérer au mieux l'espace mémoire disponible, il est souhaitable qu'un élément du graphe soit libéré dès qu'il n'est plus utilisé. Ce problème pourrait être résolu par l'emploi d'un *garbage collector*, des études pour l'intégration de tels outils en C++ ayant été effectuées. Cependant, le graphe représentant des expressions mathématiques, il est acyclique et orienté (DAG). Il ne peut donc se produire de références circulaires. Ce point permet d'envisager un simple mécanisme de comptage de références, plus simple à implémenter. D'autre part, il peut être souhaitable de référencer une expression pour d'autres raison qu'une liaison opérateur/opérande. Si l'on reprend l'exemple de la figure 8, on s'aperçoit que le calcul des dérivées de N_1 et N_2 nécessitent toutes deux le calcul de la dérivée de N_4. Il est donc intéressant d'adjoindre à N_4 un cache pointant sur sa dérivée, évitant ainsi de la calculer plusieurs fois. Ce cache ne devant pas entraîner le maintien de l'expression dérivée si celle-ci n'est plus utilisée comme opérande d'un nœud du graphe, il ne doit pas faire partie du comptage des références qui lui sont portées. Par contre il doit être invalidé dès que cette expression est détruite.

Ceci nous a conduit à définir les notions de références bloquantes et non-bloquantes.

Ces notions sont mises en œuvre par les deux classes génériques BRef<T> et NBRef<T>. Ces classes utilisent la notion de *smart-pointer* du C++ (surcharge de l'opérateur ->) et peuvent donc être utilisées de manière transparente comme des pointeurs [3].

3. Le type scalar utilisé dans la figure 4 est une spécialisation de la classe BRef<T> sur la classe ScalarElement.

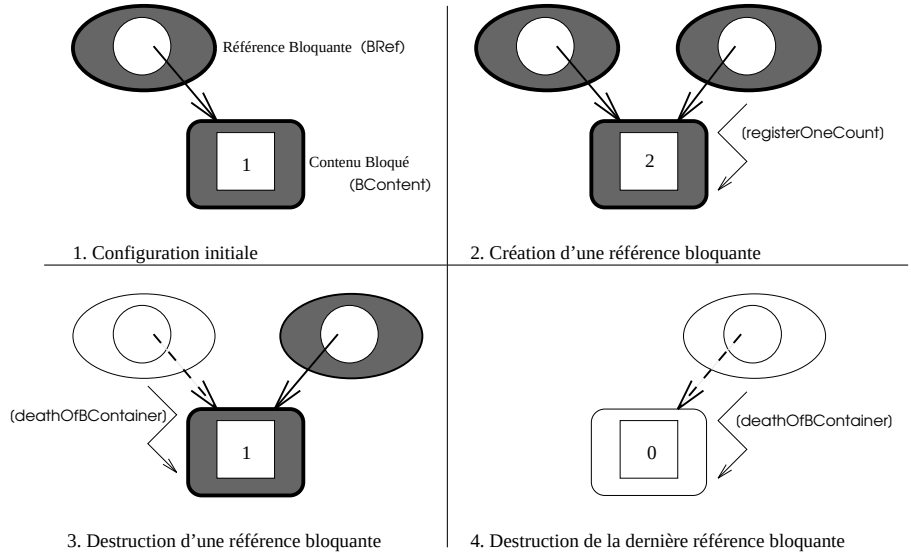

FIG. 9 - *Références bloquantes et contenus bloqués*

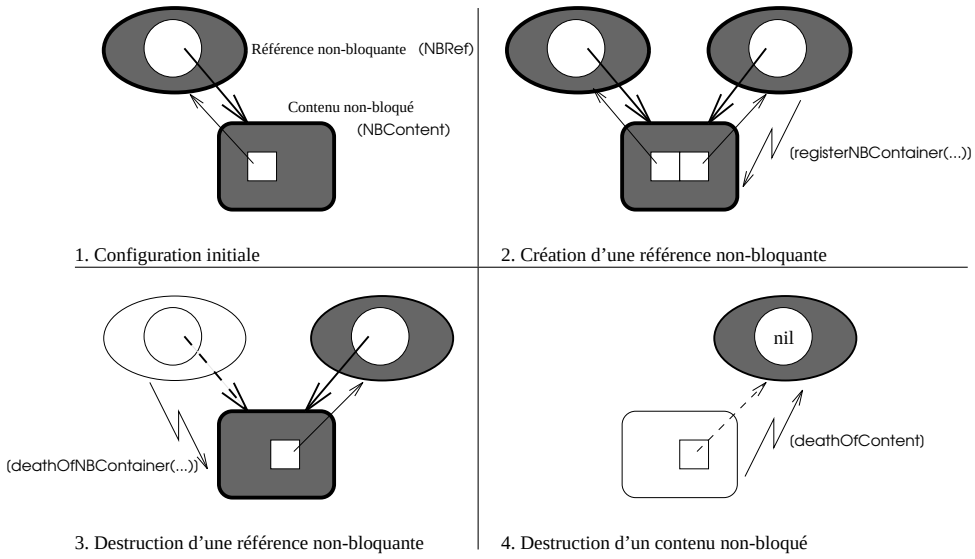

FIG. 10 - *Références non-bloquantes et contenus non-bloqués*

Une instance de la classe BRef<T> représente une référence bloquante sur un objet de type T, alors considéré comme un contenu bloqué. Un contenu bloqué doit gérer un compteur de références et mettre en œuvre deux méthodes d'incrémentation et de décrémentation (registerOneCount et deathOfBContainer). La construction d'une réference bloquante sur un contenu bloqué entraîne l'incrément du compteur (réception du message registerOneCount), la destruction de la référence son décrément (message deathOfBContainer). Quand le compteur devient nul, l'objet référencé est détruit (figure 9). Ce comportement d'un contenu bloqué est synthétisé dans la classe BContent.

Une instance de la classe NBRef<T> représente une référence non-bloquante sur un objet de type T, alors considéré comme un contenu non-bloqué. Un contenu non-bloqué doit gérer une liste de références et mettre en œuvre deux méthodes d'abonnement et de désabonnement (registerNBContainer et deathOfNBContainer). Lors de la construction d'une réference non-bloquante celle-ci est enregistrée auprès de son contenu (message registerNB-Container). Lors de sa destruction une référence non bloquante informe son contenu, celui-ci le supprimant alors de sa liste (message deathOfNBContainer). La destruction d'un contenu non-bloqué entraîne l'invalidation des références qui lui sont portées (message deathOfContent). Le comportement d'un contenu non-bloqué est synthétisé dans la classe NBContent.

Les classes BContent et NBContent sont donc utilisées comme superclasses de la classe ScalarElement et un élément du graphe est toujours référencé soit par une référence bloquante (un opérateur) soit par une référence non-bloquante (un cache). Ceci permet de déléguer la gestion des désallocations au destructeur de la classe BRef, la durée de vie de ces références (qui ne sont jamais allouées dynamiquement) étant gérée par le compilateur.

5 Simplifications

Un second aspect important dans la limitation de la taille des expressions du graphe est la simplification des expressions qu'il contient. Ces simplifications peuvent être envisagées de deux façons : soit lors de la construction de l'expression (a priori), soit dans une phase suivant la construction (a posteriori). Cette seconde classe de simplification demande la réécriture complète du DAG pour mettre les expressions sous la forme de sommes de produits. C'est une opération coûteuse qui de plus présente le risque de la perte du partage des sous-expressions communes. Les simplifications à priori ($0 + x = 0$, $1 * x = x$, $x * y + x * z = x * (y + z)$, ...), si elles ne peuvent rester que triviales, sont par contre beaucoup plus aisées d'utilisation et permettent à elles seules un élagage important du graphe. Seules ces dernières sont utilisées dans notre noyau.

5.1 Influence de l'arité des opérateurs

Une simplification dépend de deux choses : la nature de l'opérateur considéré et la nature de ses opérandes. Dans le cas d'un opérateur unaire, la mise en œuvre est très simple. Il suffit d'envoyer à l'opérande un message correspondant au type de l'opérateur, l'opérande pouvant alors simplement construire un nouveau nœud représentant cet opérateur, soit effectuer une

simplification. Ceci est mis en œuvre grâce aux méthodes virtuelles : la classe de base définit un comportement par défaut et les classes dérivées peuvent, si nécessaire, redéfinir celui-ci. La figure 11 illustre la mise en œuvre de la simplification $-(-x) = x$.

```
scalar operator-(scalar ref) {
    return ref->minus();
}

class ScalarElement {
    ...
    virtual scalar minus() {
        return scalar(new UnaryMinus(this));
    }
    ...
};

class UnaryMinus : public Unary {
    ...
    virtual scalar minus() {
        return operand();
    }
  ...
};
```

FIG. 11 - *Un exemple de simplification*

Dans le cas des opérateurs binaires (ou d'arité supérieure à deux), ce schéma ne s'applique plus aussi bien. Prenons par exemple la multiplication et une de ses simplifications associées : $x*1 = 1*x = x$. Si à l'opérateur "$*$" nous associons la méthode multiply(scalar), l'expression $x * y$ se traduit par l'appel x.multiply(y). La simplification $1 * x = x$ peut alors se gérer simplement dans la classe Constant. Par contre la simplification symétrique $x*1 = x$ demande lors de la réception par x du message multiply une reconnaissance de la constante 1 pour pouvoir effectuer la simplification. A notre connaissance seul CLOS, grâce au principe des *multi-methods* permet la spécialisation d'une méthode sur plusieurs de ses arguments [16]. Le C++ permettrait, en utilisant l'inférence de type dynamique (RTTI) et le transtypage dynamique (*dynamic cast*), d'effectuer ce type de reconnaissance dans le corps de la méthode du receveur. Si l'on ne considère qu'un ensemble réduit d'opérateurs et de simplifications, cette solution peut être satisfaisante, mais dès que cet ensemble s'agrandit, les méthodes doivent prendre en compte un grand nombre de possibilités ce qui, d'une part se traduit par l'écriture d'un vaste "switch", d'autre part entraîne une importante duplication de code dans chacune des classes voulant spécialiser l'opérateur concerné. Outre son aspect peu "orienté-objets", cette méthode induit un code difficilement maintenable.

5.2 Double polymorphisme

Dans [5] T. Budd présente une alternative déja évoquée par D. Ingalls [13] permettant de gérer le polymorphisme sur plusieurs variables. En bref, cette technique (*double polymorphism*) consiste à faire circuler un message sur tous les arguments d'une fonction, ce message étant progressivement affiné avec le type précis (dynamique) de ces arguments. Ainsi, dans l'exemple de la multiplication, x reçoit le message multiply(scalar x, scalar y) et le transmet à y sous la forme multiply(type_x x, scalar y), où type_x représente le type effectif de x.

En C++ ce schéma peut être mis en œuvre grâce à la liaison dynamique utilisée pour les méthodes virtuelles. La figure 12 présente l'exemple d'une telle mise en œuvre pour la

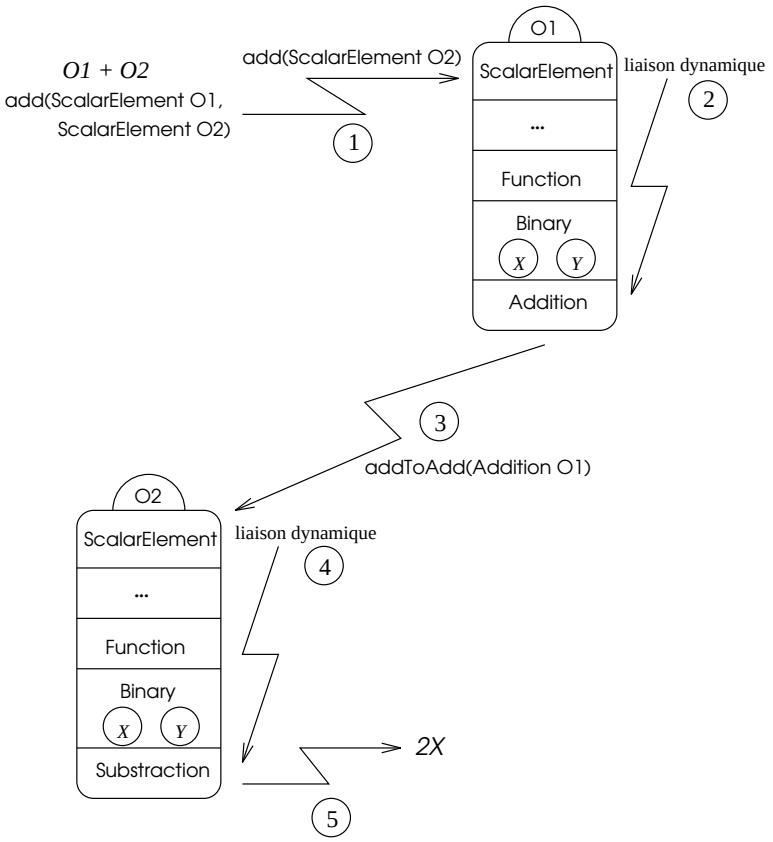

FIG. 12 - *Simplification* $(x + y) + (x - y) = 2x$

simplification $(x + y) + (x - y) = 2x$. Dans cet exemple il s'agit d'additionner deux objets O_1 et O_2 représentant respectivement les expressions $x+y$ et $x-y$. À l'opérateur d'addition

est associée la fonction binaire add, cette fonction prenant en paramètre deux scalaires. L'addition peut alors se décomposer en cinq phases :

1. La fonction add(ScalarElement O_1, ScalarElement O_2) appelle la méthode virtuelle add(ScalarElement O_2) de l'objet O_1.
2. Par liaison dynamique, la méthode add(ScalarElement O_2) de la classe Addition est sélectionnée.
3. Cette méthode appelle alors la méthode virtuelle addToAdd(Addition O_1) de l'objet O_2.
4. Le comportement spécifique de la classe Substraction est sélectionné par la liaison dynamique.
5. Le type dynamique de O_1 et O_2 est maintenant connu, il est possible d'accéder à leurs opérandes, de constater leur identité et de retourner l'expression $2x$.

Dans cet exemple, le message associé à la fonction add(ScalarElement O_1, ScalarElement O_2) aura pris les deux formes O_1.add(ScalarElement O_2) et O_2.addToAdd(Addition O_1).

Le point fort du double polymorphisme est la possibilité qu'il apporte de séparer les diverses simplifications en différentes unités représentées par différentes méthodes. L'ajout de simplification ne se traduit ainsi jamais par la modification du code des méthodes existantes mais par la mise en œuvre de nouvelles méthodes.

Il existe pourtant un défaut majeur à cette solution : le nombre de méthodes nécessaires. Si l'on considère un ensemble de N opérateurs, composé de u opérateurs unaires et $N-u$ opérateurs binaires, il est nécessaire de mettre en œuvre N classes, u fonctions unaires et $N-u$ fonctions binaires représentant ces opérateurs. Dans chacune des classes, il y aura donc N méthodes représentant la première version d'un message associé à une fonction et pour chaque fonction binaire, N méthodes représentant la seconde version. Soit dans chaque classe, $N+(N-u)N$ méthodes. Le nombre de méthodes nécessaires pour un ensemble de N opérateurs composé de u opérateurs unaires et de $N-u$ opérateurs binaires, est donc :

$$N(N+(N-u)N)$$

L'ensemble des 4 opérateurs binaires $\{+,-,*,/\}$ augmenté de l'opérateur unaire "$-$", nécessite à lui seul 125 méthodes ! De plus, il est nécessaire de représenter les éléments terminaux du graphe (constantes, variables ...). Ceci implique la présence de méthodes supplémentaires pour coder la deuxième forme des messages binaires. Si l'on note z le nombre de ces éléments terminaux, le nombre de méthodes nécessaires devient :

$$(N+z)(N+(N-u)(N+z)) \tag{4}$$

et si l'on en adjoint deux à l'ensemble d'opérateurs précédent, ce nombre passe de 125 à 231.

Sachant que la version actuelle du noyau implémente 9 opérateurs unaires, 4 opérateurs binaires et 7 types d'éléments terminaux, la situation serait bien peu gérable si ce résultat

théorique restait vérifié. Heureusement ceci n'est pas le cas pour ce qui est de la mise en œuvre des simplifications. En effet une règle de simplification ne représente qu'un cas particulier de l'application d'un opérateur et ne concerne en général qu'un nombre réduit d'éléments suceptible de s'y conformer. La technique pour exploiter cette propriété est de définir un comportement par défaut au sein de la classe de base. Ce comportement est simple : créer une nouvelle instance de la classe correspondante à l'opération demandée (voir pour exemple la méthode minus() de la classe ScalarElement, figure 11). Ce comportement par défaut étant fourni, une classe n'a besoin de le redéfinir que si elle veut le modifier (sur 20 classes concernées, seule 3 redéfinissent la méthode minus()). Grâce à ce principe, le noyau ne comporte que 72 méthodes pour la gestion des simplifications sur les opérateurs scalaires, au lieu des 1860 données en théorie par l'équation (4).

6 Vecteurs et matrices

Dans cette partie nous discutons du choix fait pour la représentation des vecteurs et des matrices créés par les opérations de dérivation et de son impact sur la taille des sytèmes mécaniques représentables.

En reconsidérant les équations (1) et (2) nous pouvons les réécrire sous la forme :

$$F = \mathcal{Q} - \tfrac{d}{dt}(\nabla_{\dot{q}}\mathcal{C}) + \nabla_q\mathcal{C} + k\nabla_q {f_h}^2 + k\nabla_{\dot{q}} {g_l}^2 \tag{5}$$

$$\mathcal{J} = \nabla_q F \tag{6}$$

Les équations du mouvement et leur jacobienne peuvent donc être calculées aussi bien sous la forme d'ensembles d'expressions scalaires que sous des formes vectorielles et matricielles. De même le gradient d'une fonction $f(x_1, \ldots, x_n)$ en fonction d'un vecteur de variables libres $Y = (y_1, \ldots, y_m)$ peut s'exprimer :

$$\nabla_Y f(x_1, \ldots, x_n) = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \cdot \nabla_Y x_i \tag{7}$$

$$\nabla_Y y_j = (\delta_{1j}, \ldots, \delta_{mj})\ \forall j = 1 \ldots m$$

où δ_{ij} est le symbole de Kronecker.

Deux stratégies sont alors possibles : considérer systématiquement un vecteur (resp. une matrice) comme un tableau à une (resp. deux) dimension(s) d'éléments scalaires et répercuter les opérations sur ces tableaux sur leurs éléments, ou bien se doter d'un ensemble de classes similaire à celui des scalaires permettant de construire des graphes d'expressions symboliques contenant des opérateurs vectoriels et matriciels. La figure 13 présente les graphes construits lors du calcul du gradient de la fonction $f(x, y, z) = x * y + z/3$ par rapport au vecteur de variables (x, y, z).

Cet exemple montre que l'utilisation d'expressions vectorielles peut bloquer des simplifications qui sont facilement détectées lors d'une décomposition scalaire, et engendrer ainsi un graphe plus important comportant des vecteurs très creux. Cependant, cet exemple n'est

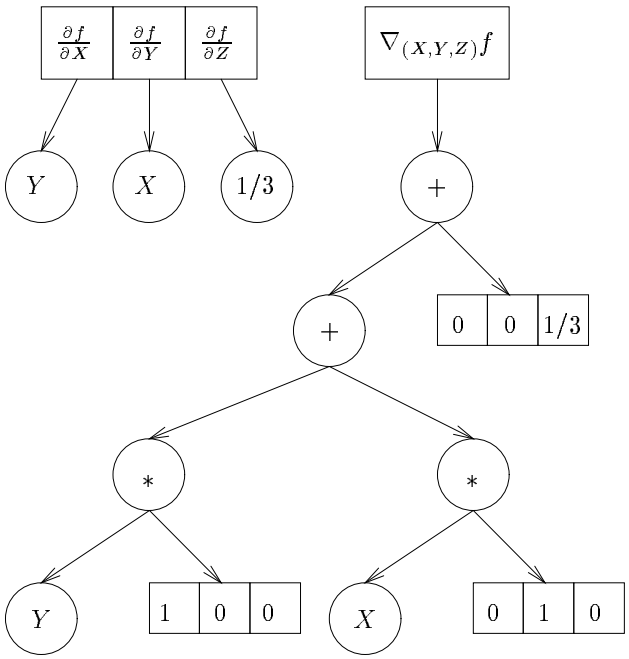

FIG. 13 - *Graphe pour le gradient de $f(x, y, z) = x * y + z/3$ par rapport à (x, y, z) avec et sans opérateurs vectoriels*

évidemment pas représentatif des graphes construits pour représenter les équations du mouvement d'un système mécanique même très simple. Dans ce cas la profondeur des graphes fait que seule une portion réduite des vecteurs construits pour le calcul d'un gradient, présents dans le bas du graphe, possède un caractère creux. L'utilisation d'expressions vectorielles permet alors de rassembler sous un même opérateur vectoriel plusieurs opérateurs scalaires et réduire ainsi la taille du graphe.

Une comparaison entre une version précédente du noyau écrite en C, n'utilisant que des opérateurs scalaires, et le noyau C++, mettant en œuvre des expressions vectorielles et matricielles, a été effectuée. Le mécanisme de test est un N-pendule en trois dimensions composé de N tiges reliées par des rotules (3 degrés de liberté en rotation chacune). En faisant varier le nombre de tiges (de 1 à 15) il est possible d'augmenter la complexité du mécanisme. Les figures 14 et 15 montrent le nombre de nœuds et l'occupation mémoire nécessaires pour la construction des équations du mouvement et de la jacobienne associées à ce N-pendule, en fonction du nombre de variables (degrés de liberté) utilisées pour le représenter. Ces résultats démontrent l'intérêt des opérateurs vectoriels et matriciels, la taille du graphe construit étant linéaire en fonction du nombre de degrés de liberté lorsqu'ils

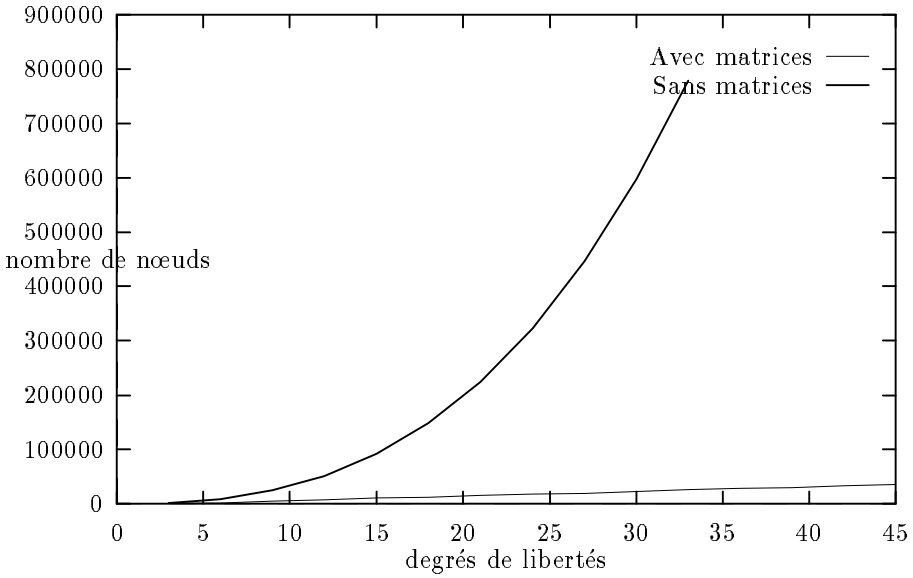

FIG. 14 - *Nombre de nœuds utilisés par les équations du mouvement et la jacobienne.*

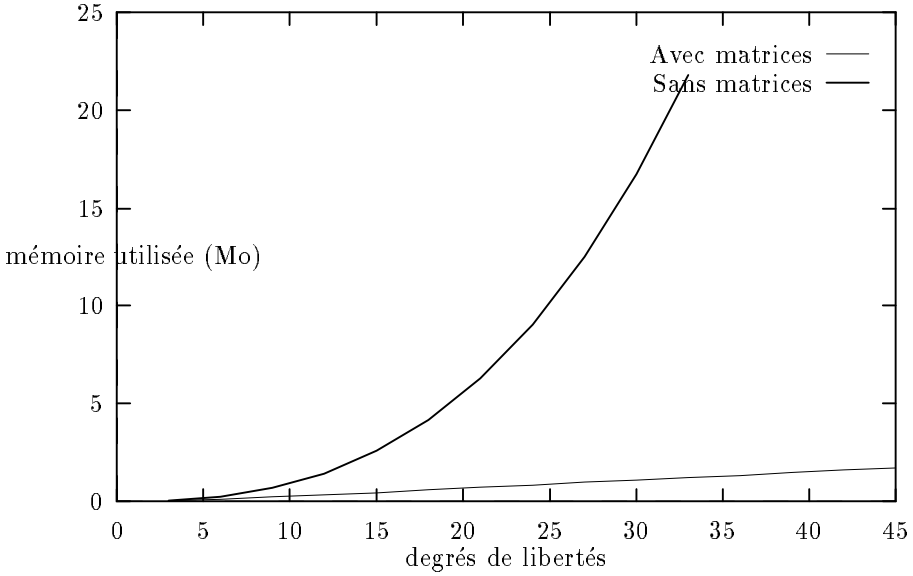

FIG. 15 - *Mémoire utilisée par les équations du mouvement et la jacobienne.*

sont utilisés alors qu'elle est quadratique dans le cas contraire. Une telle différence s'explique simplement par le fait que le nouveau noyau permet de compacter dans un seul opérateur des vecteurs de taille proportionnelle au nombre de variables du système et des matrices de taille proportionnelle au carré du nombre de ces variables. Ceci signifie que le nombre d'opérations scalaires nécessaire à l'évaluation numérique des équations et de leur jacobienne reste le même quelque soit la méthode employée. Malgré cela, la réduction de la taille du graphe reste très intéressante pour l'application d'un certain nombre d'algorithmes, en particulier dans le cadre de recherches en cours visant à produire du code parallèle associé à l'évaluation numérique de ce graphe. D'autre part, des travaux sur la différentiation effectués dans le même cadre [20] ont permis d'exhiber des stratégies de dérivation réduisant le nombre d'opérations effectives nécessaires à l'évaluation des équations du mouvement.

7 Un exemple d'évolution

Nous avons mentionné dans la partie précédente que ces travaux ont pour but le remplacement d'un noyau existant écrit en C. L'ajout d'expressions vectorielles et matricielles ne justifiait pas à lui seul une refonte complète dans un nouveau formalisme, en l'occurence l'orienté-objets. Nous nous proposons dans cette partie d'illustrer par un exemple que les avantages apportés par la réécriture du noyau concernent également la maintenance de celui-ci.

Dans le cadre de la simulation de systèmes mécaniques, il peut parfois s'avérer nécessaire d'écrire des expressions ayant des formulations différentes suivant la valeur d'un paramètre. La force de réaction du sol sur la roue d'un véhicule qui devient nulle lorsque cette roue décolle est un exemple caractéristique de ce besoin.

Ceci peut être mis en œuvre à l'aide d'un opérateur conditionnel. Le noyau ne gérant pas la notion d'expression booléenne symbolique, nous choisissons d'effectuer le codage de celle-ci comme en C par des valeurs numériques. Si nous prenons une fonction conditionnelle f telle que :

$$f(c, q_1, q_2) = \text{si } c \text{ alors } q_1 \text{ sinon } q_2$$

nous pouvons exprimer sa dérivée par rapport à une variable x sous la forme :

$$\frac{\partial f(c, q_1, q_2)}{\partial x} = \text{si } c \text{ alors } \frac{\partial q_1}{\partial x} \text{ sinon } \frac{\partial q_2}{\partial x}$$

Les classes générales du noyau s'appuyant sur le formalisme exprimé par l'équation (3), il peut être intéressant de l'utiliser pour réécrire cette dérivée :

$$\frac{\partial f(c, q_1, q_2)}{\partial x} = \frac{\partial f}{\partial c} \cdot \frac{\partial c}{\partial x} + \frac{\partial f}{\partial q_1} \cdot \frac{\partial q_1}{\partial x} + \frac{\partial f}{\partial q_2} \cdot \frac{\partial q_2}{\partial x}$$

avec :

$$\frac{\partial f}{\partial c} = 0$$

$$\frac{\partial f}{\partial q_1} = \text{si } c \text{ alors 1 sinon 0}$$
$$\frac{\partial f}{\partial q_2} = \text{si } c \text{ alors 0 sinon 1}$$

En effet cette seconde formulation permet d'utiliser la classe Binary comme super-classe du nouvel opérateur. Grâce à cet héritage, il ne reste qu'à coder au sein de la nouvelle classe

```
scalar cond(scalar c, scalar q1, scalar q2) {
    return scalar(new cond(c,q1,q2));
}

class Conditional : public Binary {
public:
    Conditional(scalar c, scalar q1, scalar q2)
        : cond(c), Binary(q1,q2) {}
    ~Conditional() {}
    double value() {
        double result;
        if (cond->value())
            result = operand(1)->value();
        else
            result = operand(2)->value();
        return result;
    }
.../...
```

```
.../...

    scalar ddop1() {
        return scalar(new Conditionnal(cond,
                                       Constant(1),
                                       Constant(0)));
    }
    scalar ddop2() {
        return scalar(new Conditionnal(cond,
                                       Constant(0),
                                       Constant(1)));
    }
private:
    scalar cond;
};
```

FIG. 16 - *Mise en œuvre d'un opérateur conditionnel*

les informations qui lui sont spécifiques : le calcul de sa valeur numérique associée et le calcul symbolique de ses dérivées partielles par rapport à ses opérandes. La figure 16 montre la mise en œuvre complète de cette classe.

8 Applications

Ce noyau de différentiation symbolique, utilisé comme composant du simulateur de mécanisme décrit précédemment, est mis à profit au sein du projet Siames dans différentes actions de recherche.

Intégrations des modèles physiques pour l'animation Ces travaux visent à spécifier un modèle intégrant les interactions entre les différents états de la matière et décrit par une représentation homogène [6]. La génération automatique des équations du mouvement

pour ces différents états (solides rigides, solides déformables et systèmes de particules) peut être effectuée dans un formalisme lagrangien. Le noyau, augmenté de quelques fonctionnalités (intégration d'expressions polynomiales, nouveaux types d'éléments terminaux, ...), est utilisé pour la validation de ce modèle.

Projet Praxitèle La contribution du projet Siames à ce projet consiste à créer un environnement de simulation permettant de simuler physiquement un train de véhicules tout en permettant à l'utilisateur de simuler ses algorithmes de contrôle [2]. Cet environnement doit permettre en particulier la simulation du flot de circulation automobile, nécessitant des modèles comportementaux de pilotes mais également des modèles de véhicules (figure 17). Les possibilités du noyau en matière de génération de code permettent la création de classes C++ représentant les différents types de véhicules modélisés. Chaque véhicule est alors une instance d'une de ces classes et peut être créé, dupliqué, paramétré et contrôlé de manière simple et indépendamment des autres. Cette représentation, encapsulant le code utile à un véhicule, facilite également la distribution des calculs sur un système réparti.

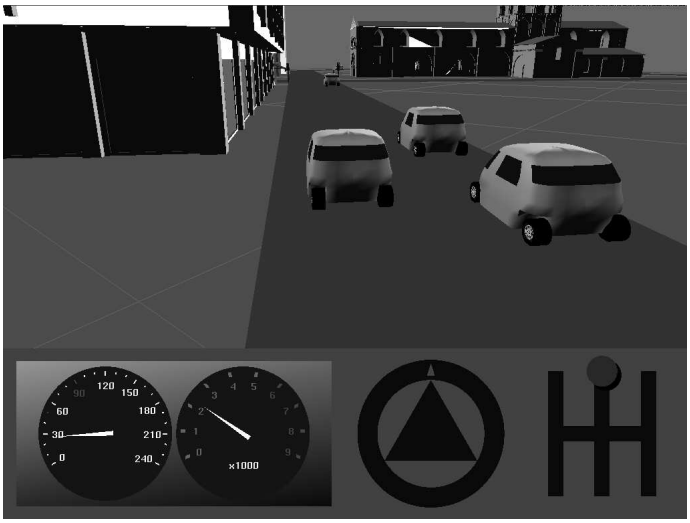

FIG. 17 - *Simulation de véhicules*

9 Conclusion

Nous avons présenté dans ce rapport la mise en œuvre d'un noyau de différentiation symbolique dans un formalisme orienté-objets. Nous avons vu comment l'utilisation du C++ permet, grâce à la surcharge, d'associer à une écriture naturelle d'expressions mathématiques la construction des structures de données nécessaires au calcul des dérivées de ces expressions.

Nous avons décrit comment, malgré le manque de *garbage collecting* intégré au langage, les classes génériques et les notions de constructeurs et de destructeurs permettent une gestion simple de la cohérence d'un graphe acyclique possédant des éléments multi-référencés.

Nous avons vu comment l'utilisation des notions d'héritage et d'appel dynamique permettent la construction d'un graphe polymorphe et la décomposition des algorithmes utilisés en composants minimaux rendant ainsi l'adjonction de nouveaux éléments particulièrement simple.

Nous avons montré que si le polymorphisme sur plusieurs variables est particulièrement utile, son utilisation dans un langage qui ne l'implémente pas directement est réservée à des cas particuliers. Cet aspect est quand même légèrement atténué par la possibilité d'utiliser un schéma méthodique de conception.

Les avantages des opérateurs symboliques vectoriels et matriciels, dont la mise en œuvre s'est trouvée facilitée par l'utilisation d'un langage orienté-objets, a été montrée sur un exemple concret.

Notre principal objectif pour l'avenir est d'utiliser le noyau présenté ici pour générer du code parallèle afin d'effectuer des simulations temps réel de mécanismes importants, par exemple des modèles réalistes de véhicules.

Références

[1] Arnaldi (B.). – *Conception du noyau d'un système d'animation de scènes tridimensionnelles intégrant les lois de la mécanique.* – Thèse de doctorat, Université de Rennes I, juillet 1988.

[2] Arnaldi (B.), Cozot (R.) et Donikian (S.). – Virtual urban environment for the simulation of an automated electrical cars platoon in the praxitele project. *In : Second Eurographics Workshop on Virtual Environments.* – Monte Carlo, Jan. 1995.

[3] Arnaldi (B.), Dumont (G.) et Hégron (G.). – Animation of physical systems from geometric, kinematic and dynamic models. *In : Modeling in Computer Graphics*, pp. 37–53. – Springer-Verlag, Avr. 1991.

[4] Booch (G.). – *Object-oriented design : with applications.* – Benjamin-Cummings, 1991.

[5] Budd (T. A.). – Generalized arithmetic in C++. *Journal of Object Oriented Programming*, vol. 3, n° 6, Fév. 1991, pp. 11–22.

[6] Cozot (R.), Arnaldi (B.) et Dumont (G.). – A unified model for physically bases animation and simulation. *In : Applied Modellisation Simulation and Optimization.* – Cancun Mexico, Juin 1995.

[7] Dumont (G.). – *Animation de scènes tridimensionnelles : la mécanique du solide comme modèle de synthèse du mouvement.* – Thèse de doctorat, Université de Rennes I, Mai 1990.

[8] Dumont (G.), Gascuel (M.-P.) et Verroust (A.). – *Animation contrôlée par la dynamique, état de l'art.* – Publication interne n° 571, Campus de Beaulieu, Rennes, IRISA, Jan. 1991.

[9] Ellis (M. A.) et Stroustrup (B.). – *The Annotated C++ reference manual.* – Addison-Wesley, 1990.

[10] Gleicher (M.) et Witkin (A.). – Snap together mathematics. *In : Advances in object-oriented graphics I*, éd. par Blake (E. H.) et Wisskirchen (P.), pp. 21–33. – Springer-Verlag, 1991.

[11] Griewank (A.). – *On automatic differentiation.* – Rapport technique, 9700 South Cass Avenue, Argonne, Illinois 60439, Argonne national laboratory, Nov. 1988.

[12] Griewank (A.), Juedes (D.), Srinivasan (J.) et Tyner (C.). – *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++.* – Rapport technique, Argonne National Laboratory, 1990.

[13] Ingalls (D. H. H.). – A simple technique for handling multiple polymorphism. *In : OOPSLA '86*, pp. 347–349.

[14] Jerrel (M. E.). – Function minimization and automatic differentiation using C++. *In : OOPSLA'89*, pp. 169–173.

[15] Juedes (D. W.). – A taxonomy of automatic differentiation tools. *In : Automatic Differentiation of Algorithms : Theory, Implementation, and Application.* – A. Griewank and G. F. Corliss, 1991.

[16] Keene (S. E.). – *Object-Oriented Programming in COMMON LISP.* – Addison-Wesley, 1989.

[17] Meyer (B.). – *Object-oriented software construction.* – Prentice-Hall international, 1988.

[18] Rall (L. B.). – *Automatic differentiation : Techniques and applications.* – Springer-Verlag, 1981, *Lecture notes in computer science*, volume 120.

[19] Rall (L. B.). – Differentiation in pascal-sc : Type gradient. *ACM Transactions On Mathematicals Software*, vol. 10, n° 2, Juin 1984, pp. 161–184.

[20] Villard (D.) et Arnaldi (B.). – Symbolic differentiation library for simulation of multibody rigid systems. *In : International IMACS symposium on symbolic computation*, pp. 130–135.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399