# "Quaterman" : une bibliotheque Maple de calculs et de manipulations sur l'algebre des quaternions

Patrick Capolsini, Stéphane Dalmas, Yves Papegay

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Rapports Techniques

N°148

***Programme 2***

*Calcul symbolique, Programmation et Génie logiciel*

# "QUATERMAN" UNE BIBLIOTHÈQUE MAPLE DE CALCULS ET DE MANIPULATIONS SUR L'ALGÈBRE DES QUATERNIONS

**Patrick Capolsini**
**Stéphane Dalmas**
**Yves Papegay**

**Janvier 1993**

Institut National
de Recherche
en Informatique
et en Automatique

2004 route des Lucioles
B.P. 93
06902 Sophia-Antipolis
France

# "QUATERMAN"
# une bibliothèque *Maple* de calculs et de manipulations sur l'algèbre des quaternions

# "QUATERMAN"
# a library for computing with quaternions in *Maple*

Patrick Capolsini, Stéphane Dalmas et Yves Papegay

Projet SAFIR

**Résumé**

L'intérêt pour les calculs dans le corps des quaternions augmente depuis les années soixantes, grâce à plusieurs applications pratiques dans des domaines variés dont la caractéristique commune est d'être de gros consommateur de calcul sur les rotations dans l'espace : mécanique (des systèmes multi-corps articulés), robotique, automatique (contrôle dans l'espace des rotations). Ceci est du aux propriétés particulières de la représentation des rotations par des quaternions (par rapport à une représentation plus classique par des matrices orthogonales). Ce rapport, après avoir rappelé les propriétés essentielles du corps des quaternions et leur lien avec la géométrie dans l'espace, décrit la réalisation d'un ensemble de procédure écrites dans le système de calcul formel *Maple*, permettant de manipuler symboliquement des expressions contenant des quaternions.

**Abstract**

The interest for computations in the field of quaternions is increasing thanks to many applications in various area whose common characteristic is the need to compute with rotations: mechanics (multibody systems), robotics, systems theory. This is due to the particular properties of the representation of rotations using quaternions. This report will briefly state the basic properties of the field of quaternion and describe a package (written for the *Maple* symbolic computation system) to symbolically compute with quaternions.

Le projet SAFIR est un projet commun à l'INRIA (unité de recherche de Sophia Antipolis), à l'université de Nice Sophia Antipolis et au CNRS (laboratoire de Mathématiques Jean Alexandre Dieudonné, URA 168 et I3S, URA 1376).

# Introduction

Nous allons décrire la réalisation d'un ensemble de procédures permettant de manipuler des quaternions dans le système de calcul formel *Maple*. Cette étude a été réalisé à la demande et grâce au financement du Centre National d'Etudes Spatiales (CNES).

Plus précisément, on désire manipuler des expressions contenant des objets représentant des quaternions, aussi bien sous une forme "numérique" (quatre coordonnées dans la base canonique) que sous forme de variable formelle ou bien encore d'un couple formé d'une expression (partie réelle) et d'une expression représentant un vecteur. Il est donc indispensable de disposer des opérations de base sur le corps des quaternions ainsi que la possibilité d'opérer toutes les simplifications souhaitées sur ces expressions.

Ce document est divisé en trois parties. La première est un résumé des propriétés fondamentales du corps des quaternions et de ses liens avec le groupe des rotations de $\mathbb{R}^3$. Cette partie posséde néanmoins une section originale décrivant toutes les identités algébriques entre quaternions (faisant intervenir la partie réelle et la norme). Le matériel de cette section est entiérement du à Bernard Mourrain et elle a été rédigé à partir d'une note qu'il a bien voulu nous fournir. La seconde partie du document décrit les fonctionnalités qu'un utilisateur peut attendre d'un système de calcul formel "général" pour calculer dans $\mathbb{H}$ et des problèmes que l'on peut rencontrer. On y discute les particularités liées au choix de *Maple* comme système d'implémentation de *Quaterman* et des choix motivés qui ont du être fait. La troisième partie est constituée du manuel de référence de *Quaterman*, formé du descriptif détaillé de toutes les fonctions disponibles dans cette bibliothèque. Ces descriptifs existent aussi dans le format naturel des pages de manuel de *Maple* afin d'être consultable "en ligne" (grâce à la fonction *help* de *Maple*). Le lecteur trouvera également à la fin de ce document un exemple non trivial d'utilisation de *Quaterman* (composition de trois rotations "génériques").

# Chapitre 1

# A quoi servent les quaternions

## 1.1 Le corps des quaternions

Les quaternions sont les éléments d'une algèbre de dimension 4 sur $\mathbb{R}$ qui est un corps non commutatif et que nous noterons $\mathbb{H}$ dans toute la suite. Le corps des quaternions est le plus gros corps qui contienne $\mathbb{R}$ au sens du théorème de Frobenius : tout corps de dimension finie sur $\mathbb{R}$ et le contenant dans son centre est isomorphe à $\mathbb{R}$, $\mathbb{C}$ ou $\mathbb{H}$. Historiquement, c'est aussi le premier exemple connu de corps non commutatif, "découvert" par le mathématicien irlandais William R. Hamilton en 1843.

Le corps des quaternions peut se construire simplement comme la sous-algèbre de l'algèbre des matrices $2 \times 2$ sur $\mathbb{C}$ engendrée par les quatre matrices Id, $e_1$, $e_2$ et $e_3$ suivantes :

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix} \quad \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix} \quad \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

On peut vérifier, de manière élémentaire, que cette sous-algèbre de dimension 4 est un corps non commutatif. Les $e_i$ vérifient les relations de commutation suivantes :

$$\frac{1}{2}(e_i e_j + e_j e_i) = \delta_{ij} I$$

Il est en effet trivial de vérifier que les $e_i$ sont de carré $-1$ :

$$e_1 e_1 = e_2 e_2 = e_3 e_3 = -\mathrm{Id}$$

et

$$\begin{array}{ccccc} e_1 e_2 & = & -e_2 e_1 & = & e_3 \\ e_2 e_3 & = & -e_3 e_2 & = & e_1 \\ e_3 e_1 & = & -e_1 e_3 & = & e_2 \end{array}$$

Par analogie avec la notation des nombres complexes sous la forme $a + ib$ on note traditionnellement un quaternion sous la forme d'une combinaison linéaire formelle $x_1 + ix_2 + jx_3 + kx_4$ où $i$, $j$ et $k$ représentent les trois matrices $e_1$, $e_2$ et $e_3$. Tout quaternion $q$ s'écrit donc sous la forme $x_1 + ix_2 + jx_3 + kx_4$ où $x_1$ est appelée la *partie réelle* du quaternion $q$. Si $x_1 = 0$ on dit que $q$ est un quaternion *pur*. Tout quaternion non réel (i.e. dont la partie pure est non nulle) engendre avec 1 un sous-corps de $\mathbb{H}$ isomorphe à $\mathbb{C}$.

### 1.1.1 Le produit

La table de multiplication de l'algèbre (donnant les produits $e_i e_j$) montre que si $q_1 = x_1 + ix_2 + jx_3 + kx_4$ et $q_2 = y_1 + iy_2 + jy_3 + ky_4$, le produit $q_1 q_2$ peut s'écrire :

$$q_1 q_2 \quad = \quad (x_1 y_1 - x_2 y_2 - x_3 y_3 - x_4 y_4)$$

$$
\begin{aligned}
&+ i(x_1y_2 + x_2y_1 + x_3y_4 - x_4y_3) \\
&+ j(x_1y_3 - x_2y_4 + x_3y_1 + x_4y_2) \\
&+ k(x_1y_4 + x_2y_3 - x_3y_2 + x_4y_1)
\end{aligned}
$$

Cette façon de calculer le produit de deux quaternions n'est d'ailleurs pas optimale en le nombre de multiplications. Seulement huit produits sont nécessaires, il s'agit des $p_i$ suivants :

$$
\begin{aligned}
p_1 &= x_1y_1 \\
p_2 &= x_4y_3 \\
p_3 &= x_2y_4 \\
p_4 &= x_3y_2 \\
p_5 &= (x_1 + x_2 + x_3 + x_4)(y_1 + y_2 + y_3 + y_4) \\
p_6 &= (x_1 - x_2 - x_3 + x_4)(y_1 - y_2 - y_3 - y_4) \\
p_7 &= (x_1 + x_2 - x_3 - x_4)(y_1 + y_2 - y_3 - y_4) \\
p_8 &= (x_1 - x_2 + x_3 - x_4)(y_1 - y_2 + y_3 - y_4)
\end{aligned}
$$

ensuite, les coefficients du produit sont les $z_i$ donnés ainsi :

$$
\begin{aligned}
z_1 &= 2p_1 - 1/4(p_5 + p_6 + p_7 + p_8) \\
z_2 &= -2p_2 + 1/4(p_5 - p_6 + p_7 - p_8) \\
z_3 &= -2p_3 + 1/4(p_5 - p_6 - p_7 + p_8) \\
z_4 &= -2p_4 + 1/4(p_5 + p_6 - p_7 - p_8)
\end{aligned}
$$

Même si ce résultat théorique ne semble pas pratiquement intéressant (du fait du nombre important d'additions nécessaires), il pourrait être cependant utile dans le cas où les $x_i$ ne sont pas des nombres, mais des expressions formelles de taille suffisament importante.

### 1.1.2 Inverse, norme et conjugaison

On vérifie aisément que l'inverse de $q$ peut s'écrire :

$$q^{-1} = \frac{x_1 - ix_2 - jx_3 - kx_4}{x_1^2 + x_2^2 + x_3^2 + x_4^2}$$

Cette formule nous permet d'introduire naturellement deux opérations classiques sur $\mathbb{H}$ : la conjugaison, notée $\overline{q}$ définie par $\overline{q} = x_1 - ix_2 - jx_3 - kx_4$ et la "norme" notée traditionnellement $N(q)$ définie par $N(q) = x_1^2 + x_2^2 + x_3^2 + x_4^2$. On peut alors écrire :

$$q^{-1} = \frac{\overline{q}}{N(q)}$$

La conjugaison est un automorphisme involutif de $\mathbb{H}$. Il est immédiat de constater que $q$ commute avec son conjugué $\overline{q}$ ($q\overline{q} = \overline{q}q$). La norme est une fonction à valeur réelle positive, et on a $N(q) = q\overline{q}$. Bien que cette opération soit traditionnellement appelée *norme*, ce n'est évidemment pas une norme d'espace vectoriel mais, en revanche, il est clair que l'application qui à un quaternion $q$ fait correspondre $\sqrt{N(q)}$ est bien une norme d'espace vectoriel : c'est la norme euclidienne de $\mathbb{R}^4$ si on identifie canoniquement $\mathbb{H}$ et $\mathbb{R}^4$). On peut vérifier que $\mathbb{H}$ muni de cette norme est une algèbre de Banach (cette norme est compatible avec le produit : on a $||q_1q_2|| = ||q_1||||q_2||$).

Le produit scalaire correspondant peut s'écrire :

$$\langle q_1, q_2 \rangle = \frac{1}{2}(\overline{q}_1q_2 + \overline{q}_2q_1)$$

On peut également remarquer pour terminer cette section, que $\mathbb{H}$ peut se construire comme une algèbre de Clifford (sur $\mathbb{R}$) de $\mathbb{R}^2$ (avec la forme quadratique de signature $(0, 2)$).

## 1.2 Les identités de quaternions

Une autre façon de construire $\mathbb{H}$ est de remarquer que tout quaternion $a + i\,b + j\,c + k\,d$ se décompose sous la forme $\lambda + j\,\mu$ avec $\lambda = a + i\,b, \mu = c - i\,d$ dans $\mathbb{C}$. Le corps gauche $\mathbb{H}$ est donc un $\mathbb{C}$-module libre à droite de rang 2 sur $\mathbb{C}$. La multiplication à gauche par $\lambda + \mu\,j$ d'un élément $x + y\,j$ nous donne

$$\begin{aligned}(\lambda + j\,\mu)\,(x + j\,y) \\ &= \lambda\,x + j\,\mu\,x + \lambda\,j\,y + j\,\mu\,j\,y \\ &= \lambda\,x - \overline{\mu}\,y + j\,(\mu\,x + \overline{\lambda}\,y)\end{aligned}$$

et la matrice de cette aplication dans la $\mathbb{C}$-base $1, j$ est donc

$$\begin{pmatrix} \lambda & -\overline{\mu} \\ \mu & \overline{\lambda} \end{pmatrix}$$

Soit $\phi$ l'application :

$$\begin{aligned}\phi : \mathbb{H} &\rightarrow \mathcal{M}_{2\times 2}(\mathbb{C}) \\ \lambda + j\,\mu &\mapsto \begin{pmatrix} \lambda & -\overline{\mu} \\ \mu & \overline{\lambda} \end{pmatrix}\end{aligned}$$

C'est un homorphisme de $\mathbb{R}$-algèbres ($\phi(q\,q') = \phi(q)\,\phi(q')$) permettant d'identifier $\mathbb{H}$ à un sous-ensemble des matrices $2 \times 2$ à coefficients complexes.

On remarque que pour tout quaternion $q = a + i\,b + j\,c + k\,d$, la trace de $\phi(q)$ est deux fois la partie réelle $R(q) = a$, et le déterminant de $\phi(q)$ est le carré de la norme $\|q\|^2 = \lambda\,\overline{\lambda} + \mu\,\overline{\mu}$.

En identifiant les quaternions à une sous-algèbre de $\mathcal{M}_{2\times 2}(\mathbb{C})$, nous voyons que toute identité verifiée par les matrices implique une identité sur les quaternions. Par exemple, l'identité de Cayley-Hamilton se traduit par

$$q^2 - \frac{1}{2}R(q)\,q + \|q\|^2 = 0$$

Nous pouvons en fait montrer aussi la réciproque :

— *Les identités à coefficients réels faisant intervenir des carrés de normes et des partie réelles vérifiées par les quaternions sont exactement celles entres matrices faisant intervenir les traces.*

Ce résultat se démontre en transformant toute identité algèbrique vérifié par des quaternions, leurs normes et leurs parties réelles en un polynôme de matrices (en utilisant les identifications précédentes), qui s'annule pour toutes les matrices de l'image de $\phi$. On montre ensuite que les quatre composantes de la matrice résultat sont nuls en utilisant le fait que $x$ et $\bar{x}$ sont algébriquement indépendant sur $\mathbb{C}$ (il n'existe pas de polynôme $p$ en deux variables à coefficients dans $\mathbb{C}$ tel que $p(x, \bar{x}) = 0$).

Il y a évidemment un grand intérêt à s'être ramené aux identités algébriques entre matrices car elles sont bien connues. Elles découlent toutes de l'identité de Cayley-Hamilton (voir [Mou92] et [Mou91]). De plus, il existe un algorithme pour décomposer explicitement toute identité en fonction de cette identité de Cayley-Hamilton. Ainsi on peut réduire à une forme canonique tout polynôme de quaternions où interviennent des parties réelles et des carrés de norme et tester formellement si un polynôme de quaternions est nul sans développer les calculs dans la base $1, i, j, k$.

En ce qui concerne les identités algébriques (pour les matrices $2 \times 2$) dont les coefficients sont simplement des nombres, on peut montrer (Drensky) qu'elles sont "engendrées" par les deux identités (écrites ici dans le cadre des quaternions) :

$$\sum_{\sigma \in \mathcal{S}_4} \epsilon(\sigma)\, q_{\sigma(1)} q_{\sigma(2)} q_{\sigma(3)} q_{\sigma(4)}$$

et

$$[[q_1, q_2]^2, q_3] = 0.$$

où $[q_1, q_2] = q_1 q_2 - q_2 q_1$. La dernière est une conséquence directe de l'identité de Cayley-Hamilton. En effet, $[q_1, q_2]^2$ est le carré d'un quaternion imaginaire, donc réel, et commute avec tout autre quaternion $q_3$.

## 1.3 Quaternions et géométrie

### 1.3.1 Représentations des rotations

Notre intérêt pour le corps des quaternions vient du fait que certaines opérations géométriques dans $\mathbb{R}^3$ s'expriment comme des opérations algébriques dans $\mathbb{H}$.

Soit $q_1$ un quaternion pur de norme 1. L'application $r$ définie par

$$r(q) = (\cos\frac{\alpha}{2} - q_1 \sin\frac{\alpha}{2})q(\cos\frac{\alpha}{2} + q_1 \sin\frac{\alpha}{2})$$

est un automorphisme intérieur de $\mathbb{H}$ qui conserve les quaternions purs et qui laisse invariant $q_1$. Soit $q_2$ un quaternion pur de norme 1 orthogonal à $q_1$ et $q_3$ le produit $q_1 q_2$ (on a $q_1 q_2 = -q_2 q_1$ puisqu'ils sont orthogonaux). $q_1, q_2, q_3$ forment alors une base des quaternions purs. On obtient aisément l'expression de $r$ dans cette base :

$$r(xq_1 + yq_2 + zq_3) = xq_1 + (y\cos\alpha + z\sin\alpha)q_2 + (z\cos\alpha - y\sin\alpha)q_3$$

Ce qui montre que $r$ représente la rotation d'angle $\alpha$ autour de $q_1$.
On peut aussi obtenir plus généralement une rotation à partir d'un quaternion $q_r$ non unitaire par la formule :

$$r(q) = \frac{1}{N(q_r)} q_r q \overline{q}_r$$

Si $q = x_1 + ix_2 + jx_3 + kx_4$ est un quaternion unitaire, on vérifie aisément que la matrice de la rotation associée peut s'écrire :

$$\begin{pmatrix} x_0^2 + x_1^2 - x_2^2 - x_3^2 & 2(x_1x_2 - x_3x_0) & 2(x_0x_2 - x_1x_3) \\ 2(x_0x_3 + x_1x_2) & x_0^2 + x_2^2 - x_1^2 - x_3^2 & 2(x_2x_3 - x_0x_1) \\ 2(x_1x_3 - x_0x_2) & 2(x_0x_1 - x_2x_3) & x_0^2 + x_3^2 - x_1^2 - x_2^2 \end{pmatrix}$$

Inversement, à partir d'une matrice de rotation $R$ il est toujours possible d'obtenir le vecteur unitaire $u$ et l'angle $\theta$ du quaternion associé par les formules classiques :

$$\begin{aligned} \mathrm{tr}(R) &= 1 + 2\cos(\theta) \\ \frac{1}{2}(R - R^t) &= \sin(\theta)\tilde{u} \end{aligned}$$

où l'on note $\sim$ l'opérateur qui à un vecteur $u$ associe la matrice (antisymétrique) $\tilde{u}$ telle que pour tout vecteur $v$, on ait $\tilde{u}v = u \wedge v$. Si les composantes de $u$ sont notées $u_1, u_2, u_3$, la matrice $\tilde{u}$ a donc la forme

$$\begin{pmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & u_1 \\ -u_2 & u_1 & 0 \end{pmatrix}$$

ce qui fournit l'un des quaternions unitaires représentant la rotation (les quaternions $q$ et $-q$ étant associés à la même rotation). Les parties scalaire $s$ et vectorielle $v$ sont alors :

$$\begin{aligned} s &= \frac{1}{2}\sqrt{1 + \mathrm{tr}(R)} \\ \tilde{v} &= \frac{1}{2\sqrt{1 + \mathrm{tr}(R)}}(R - R^t) \end{aligned}$$

Dans le cas d'une trace égale à $-1$, on a une rotation d'angle $\pi$. L'axe se calcule facilement (comme on peut s'en convaincre en regardant $R - Id$) : il s'agit d'un des vecteurs $R_1 + i$, $R_2 + j$, $R_3 + k$ (deux seulement peuvent être nuls) où $i, j, k$ sont les vecteurs de bases et les $R_i$ sont les colonnes de la matrices $R$ (images des vecteurs de bases : $R_1 = Ri, R_2 = Rj, R_3 = Rk$).

### 1.3.2 Intérêt de la représentation des rotations par quaternions

Le problème de la représentation des rotations de l'espace est présent dans toutes les applications de mécanique et de robotique. Le mouvement d'un corps rigide est en effet naturellement représenté par une translation donnant le mouvement de son centre de gravité et une rotation donnant son orientation (angles d'Euler ou de Bryant par exemple), le tout dépendant bien évidemment du temps.

Dans le cas de systèmes mécaniques constitués d'un assemblage de corps rigides, la position relative de chacun des corps est donnée par une rotation et le mouvement global du système nécessite la composition de ces rotations. La gestion de ces rotations est donc un point crucial d'autant plus que le nombre de corps est grand.

La représentation matricielle des rotations n'est plus du tout intéressante dès lors qu'il s'agit de composer plus de deux ou trois rotations. En effet, une matrice de rotation comporte neuf paramètres (liés) alors qu'un quaternion n'en présente que quatre. La composition de trois rotations matricielles requiert 54 multiplications et 36 additions alors que la multiplication de trois quaternions demande 32 multiplications et 24 additions.

Des expériences pratiques ont montré que dans le cas où l'on exprime les équations différentielles d'un mouvement angulaire à l'aide de quaternions, l'intégration numérique est plus efficace est plus précise. Les erreurs commises sont en effet du même ordre quelque soit la position de l'engin. Ce n'est pas le cas avec une représentation en angles d'Euler qui a une singularité en $\pi/2$ et dont les erreurs augmentent avec la valeur de l'angle. De plus on économise le calcul des fonctions trigonométriques qui n'interviennent pas dans une formulation directe par quaternions.

L'utilisation des quaternions dans un contexte formel présente aussi l'intérêt de pouvoir facilement exprimer des relations entre les axes de différentes rotations et de permettre ainsi des simplifications qui auraient été très complexes à réaliser dans une représentation purement matricielle. Cela demande bien entendu des outils adaptés qu'on ne trouve pas directement dans les systèmes de calcul formel, et c'est justement le but de *Quaterman* que de les fournir.

### 1.3.3 Vecteurs et quaternions

Si $X$ et $Y$ sont deux quaternions purs, $x$ et $y$ deux réels, la formule de multiplication des quaternions peut s'écrire en utilisant les opérations classiques sur les vecteurs de $\mathbb{R}^3$ :

$$(x + X)(y + Y) = xy - X.Y + xY + yX + X \wedge Y$$

où $X \wedge Y$ représente le produit vectoriel usuel et $X.Y$ le produit scalaire usuel de $\mathbb{R}^3$. Ce qui amène à l'expression de ces opérations vectorielles pour deux quaternions purs $X$ et $Y$:

$$\begin{aligned} X.Y &= -\frac{1}{2}(XY + YX) \\ X \wedge Y &= \frac{1}{2}(XY - YX) \end{aligned}$$

# Chapitre 2

# Quaternions et systèmes de calcul formel, conception de Quaterman

Nous désirons effectuer des calculs dans $\mathbb{H}$ à l'aide d'un système de calcul formel, en l'occurence *Maple*. L'ensemble des fonctionnalités nécessaires pour satisfaire un utilisateur exigeant peut se scinder assez naturellement en quatre groupes :

- les transformations algèbriques de bases. Elles utilisent les propriétés des opérations de $\mathbb{H}$ (somme, produit, multiplication par un scalaire, norme, conjugaison) pour transformer une expression contenant des quaternions.
- les calculs numériques dans $\mathbb{H}$. Si on voit $\mathbb{H}$ comme une algèbre réelle de dimension 4 on doit être capable de calculer au niveau des composantes d'un quaternion.
- les calculs vectoriels. Dans le cas où l'on voit $\mathbb{H}$ comme $\mathbb{R} \times \mathbb{R}^3$, on doit pouvoir introduire les vecteurs de $\mathbb{R}^3$ et leurs opérations usuelles (somme, produit scalaire, produit vectoriel) et exprimer les relations possibles entre deux vecteurs (colinéarité, orthogonalité).
- les liens quaternions – rotations – matrices. Il s'agit essentiellement d'avoir le moyen de passer d'une représentation des rotations à une autre.

Nous ne nous intéresserons pas au dernier groupe dans la suite de ce chapitre, puisqu'il s'agit essentiellement d'écrire les fonctions qui permettent de passer d'une représentation des rotations à une autre et qui sont des applications triviales des formules du chapitre précédent.

Les trois premiers groupes sont en revanche beaucoup plus intéressants, d'une part parce qu'ils nécessitent le choix et l'implémentation de simplifications par défaut (des transformations appliquées sur toutes les expressions contenant des quaternions) et d'autre part par les liens entre les différentes façons d'exprimer un quaternion : avec ses quatre coordonnées, sous forme d'un scalaire et d'une expression vectorielle, comme une expression purement formelle. Il est en effet relativement naturel de commencer par écrire une expression "complétement formelle" contenant des quaternions représentés par des symboles et sur lesquels on ne dispose d'aucune information, par exemple $q_1 q_2 - q_2 q_1$. L'utilisateur peut alors introduire une information supplémentaire, par exemple en exprimant certains quaternions sous forme de partie réelle et vectorielle. Si $q_1$ est un quaternion pur, l'expression précédente devient (en notant $x_i$ et $v_i$ les parties réelles et vectorielles) $2v_1 \wedge v_2$. Finalement, il peut donner des valeurs à toutes les composantes de ses quaternions et obtenir un calcul explicite des quatres composantes de son expression de départ. *Quaterman* doit donc être capable d'effectuer toutes les simplifications requises et de gérer ces différentes représentations.

## 2.1 Représentation des quaternions dans *Quaterman*

Comme nous venons de le voir, on est amené à manipuler un quaternion sous deux formes différentes, suivant l'information que l'on posséde :

- quatre composantes
- un scalaire et un vecteur

Si ces deux formes peuvent paraitre a priori mathématiquement equivalentes, elles sont cependant très différentes du point de vue algorithmique qui nous occupe. En effet, si on reprend l'exemple précédent du passage de $q_1q_2 - q_2q_1$ à $2v_1 \wedge v_2$, on aurait évidemment pu exprimer directement toutes les composantes de $q_1$ comme $x_2i + x_3j + x_4k$ et faire le calcul en exprimant $q_2$ comme $y_1 + y_2i + y_3j + y_4k$. On aurait alors obtenu l'expression $(-2x_4y_3 + 2x_3y_4)i + (2x_4y_2 - 2x_2y_4)j + (-2x_3y_2 + 2x_2y_3)k$ qui représente bien $2v_1 \wedge v_2$, mais cette façon de faire un peu "brusque" a plusieurs inconvénients qui apparaissent assez clairement dans notre exemple:

- taille du résultat et complexité des calculs : l'utilisation de la formule donnant le produit en fonction des opérations vectorielles est moins couteuse et donne un résultat plus petit,
- perte du sens géométrique : les expressions des trois composantes ont perdu un sens géométrique évident,
- les simplifications potentielles sont occultées : il est évident sur l'expression $2v_1 \wedge v_2$ que celle-ci s'annule si $v_1$ et $v_2$ sont colinéaires. C'est moins clair sur l'expression des composantes.

Tous ces arguments militent donc fortement en faveur de l'utilisation conjointe des deux représentations : quatre composantes scalaires ou une composante scalaire et une composante vectorielle.

La représentation que nous avons choisi dans *Maple* pour les quaternions est simple et permet de traiter ces deux cas de manière homogène. Un quaternion est représenté par un opérateur `Quater` à deux arguments, le premier est la partie scalaire, le second la partie vectorielle. Cette dernière peut être une liste de trois expressions qui sont alors les trois composantes non réélles du quaternion. L'utilisation d'un opérateur particulier permet un test rapide. Evidemment, l'utilisateur ne manipule pas directement cette représentation (l'opérateur `Quater`) mais dispose d'une fonction générale de création d'un quaternion (`qmake`).

## 2.2 Les transformations par défaut

Nous allons nous intéresser dans cette section aux transformations automatiques appartenant au premier groupe. Il s'agit donc de transformations qui sont faites par défaut sur toutes expressions contenant des quaternions et leurs opérations usuelles. De telles transformations sont appelées des simplifications par défaut. Voici une listes de celles qui sont appliquées par *Quaterman* (on a $a, b \in \mathbb{R}, q, q_1, q_2 \in \mathbb{H}$):

$$q + 0 \longrightarrow 0$$
$$aq + bq \longrightarrow (a + b)q$$
$$0q \longrightarrow 0$$
$$(aq_1)(bq_2) \longrightarrow (ab)(q_1q_2)$$
$$(q^{-1})^{-1} \longrightarrow q$$
$$(aq)^{-1} \longrightarrow (1/a)q^{-1}$$
$$\overline{\overline{q}} \longrightarrow q$$
$$\overline{aq} \longrightarrow a\overline{q}$$

Toutes ces règles devant être évidemment appliquées modulo l'associativité de la somme et du produit et la commutativité de la somme.

Comme on le voit, chacune de ces transformations est liée à un opérateur (l'opérateur de tête de la règle correspondante).

Pour l'implémentation de ces règles dans *Maple* deux choix s'offrent à nous :

- utiliser au mieux les opérateurs usuels déjà présents dans *Maple*

- introduire de nouveaux opérateurs auxquels on liera des procédures effectuant les simplifications requises.

La première solution n'est malheureusement pas envisageable, le comportement des opérateurs prédefinis de *Maple* étant trop spécialisés aux cas des corps commutatifs. Le premier problème auquel on se heurte est simple : on ne peut pas utiliser l'opérateur usuel `*` pour le produit de deux quaternions. Les mécanismes de simplification présents dans les systèmes de calcul formel classiques ne permettent pas de calculer simplement dans un corps non commutatif tel que $\mathbb{H}$. L'opérateur de multiplication est évidemment supposé commutatif et l'expression `q1 * q2 - q2 * q1` sera toujours considérée comme nulle. Les systèmes classiques calculent naturellement dans $\mathbb{Q}$ ou $\mathbb{R}$ i.e. les symboles représentent des rationnels ou des réels et les règles de simplification sont toutes valides dans ces corps. On peut noter que le traitement de $\mathbb{C}$ pose déjà des problèmes essentiellement à cause des fonctions multiformes comme la racine carrée...

Il n'existe pas, à notre connaissance, de système qui permette de calculer directement dans $\mathbb{H}$ bien que certains systèmes spécialisés dans des calculs de physique théorique permettent de calculer dans des algèbres non commutatives (algèbre de Clifford). Cependant, certains systèmes disposent d'un opérateur de produit non commutatif.

### 2.2.1 Les opérateurs de Maple

Une autre solution naturelle dans le contexte de *Maple* est l'utilisation de son mécanisme de *neutral operators*. Celui-ci permet la création d'opérateurs, distingués lexicalement des autres noms par leur premier caractère qui est `&` qui peuvent s'utiliser de manière infixe (comme le signe `+`) et auxquelles on peut attacher des propriétes particulières (par la fonction *define*). Il est possible de préciser que le nouvel opérateur possède les propriétés de l'opération d'un groupe, d'une application linéaire ainsi que d'associativité, de commutativité, d'élément neutre, ... Un mécanisme de déclaration de règles (la procédure *forall*) permet de définir des règles particulières de simplification.

Le mécanisme de définition d'opérateurs de *Maple* permet donc de créer rapidement et avec un minimum d'efforts de nouveaux opérateurs. Il n'est malheureusement pas suffisant pour implémenter les règles que nous désirons pour nos calculs sur les quaternions. Malgré plusieurs essais nous n'avons pu obtenir un résultat satisfaisant. Cette expérience a de plus montrée que le *forall* de *Maple* était peu fiable et engendrait des opérateurs assez lents.

On peut également regretter que les règles de précédences soient fortement non intuitives dans le cas où l'on écrit les *neutral operators* de manière infixes puisque que tous ces operateurs ont la même précédence et que l'utilisateur n'a pas la possibilité de changer cela. Cela réduit considérablement l'intérêt de cette écriture.

### 2.2.2 Nos opérateurs

L'effort requis pour implémenter de manière satisfaisante nos règles avec les mécanismes particuliers de *Maple* ayant été jugé trop important, nous avons donc choisi la solution d'écrire nos propres opérateurs en leur associant les procédures réalisant les simplifications par défaut que nous avons présentées. Dans un soucis d'homogénéité les noms choisis commencent tous par la lettre `q`. Il s'agit de `qadd`, `qmult`, `qinv`, `qnorm` et `qsmult` (pour le produit par un scalaire). Dans un premier temps nous avons décidé de ne pas implémenter les opérateurs "derivés" de soustraction et de division, pas plus qu'un opérateur de puissance. Cette simplification qui pourrait sembler un peu cavaliére s'explique par le fait que les applications envisagées n'en font pas un usage naturel.

## 2.3 Calcul et simplification

L'existence de deux niveaux de représentations pour les quaternions pose des problèmes intéressants sur les liens entre calcul et simplification.

Si une somme contient plusieurs quaternions dont les quatres composantes sont exprimés, on effectue l'opération. De même dans un produit pour ceux qui sont adjacents. Ces règles de calcul élémentaires sont appliquées lors de la simplification et correspondent assez bien à un comportement "intuitif" du système

(par analogie avec son comportement sur les autres types de données). Elles sont en pratique codées dans les procédures liées aux opérateurs `qadd` et `qmult`.

Dans le cas de quaternions dont on connait explicitement seulement les parties réelles et vectorielles, un autre mécanisme doit être mis en place. Il est en effet probable que l'utilisateur peut désirer effectuer des sommes et des produits sans que ces opérations ne soient effectuées effectivement sur les deux composantes. En effet, dans ce cas, on passerait brutalement d'une expression faisant intervenir les opérateurs `qadd`, `qmult` *etc* à une expression qui ne serait plus qu'un seul quaternion. Il est donc naturel de pouvoir s'arréter à un niveau qui reflète mieux la structure de l'expression de départ. Le passage de ce niveau à celui où l'on exprime les opérations entre quaternions en opérations vectorielles se fait donc par l'intermédiaire d'une fonction particulière de *Quaterman*, `qeval`.

## 2.4 Les opérations sur les vecteurs de $IR^3$

Voici les règles qui sont employées (dans lesquelles $a$ et $b$ sont des expressions scalaires, $v$, $v_1$ et $v_2$ des vecteurs de $\mathbb{R}^3$) :

$$v + 0 \longrightarrow v$$

$$0v \longrightarrow 0$$

$$av_1 + bv_1 \longrightarrow (a+b)v_1$$

$$a(bv) \longrightarrow (ab)v$$

$$0 \wedge v \longrightarrow 0$$

$$av_1 \wedge bv_2 \longrightarrow (ab)(v_1 \wedge v_2)$$

$$v_1 \wedge v_1 \longrightarrow 0$$

$$0.v \longrightarrow 0$$

$$(av_1).(bv_2) \longrightarrow (ab)(v_1.v_2)$$

$$(v_1 \wedge v_2).v_1 \longrightarrow 0$$

Toutes ces règles doivent être appliqués (comme d'habitude) modulo la commutativité et l'associativité de la somme et du produit scalaire, l'anticommutativité du produit vectoriel.

Comme dans le cas des opérations sur les quaternions, ces opérateurs ont été implémenté "à la main" (sans utiliser le mécanisme des *neutral operators*), et ce, pour les même raisons.

### 2.4.1 Relations entre vecteurs

Dans le cadre des applications envisagées pour *Quaterman* il est fondamental d'avoir les moyens d'exprimer les relations possibles entre deux vecteurs. Il s'agit essentiellement de la colinéarité et de l'orthogonalité. Ces relations peuvent permettrent de grandes simplifications dans les expressions représentant des produits de quaternions.

Ceci peut se faire de manière tout à fait élémentaire en utilisant l'affectation de *Maple*. Par exemple, pour exprimer que `v1` et `v2` sont colinéaires, on peut écrire dans *Maple* : `v2 := vsmult(k, v1)` (traduction de $v_2 = kv_1$). Ensuite nos règles de simplifications font tout le travail, de sorte que l'expression `vprod(v1, v2)` est bien évaluée en `[0, 0, 0]` (le vecteur nul).

Mais le mécanisme d'affectation de *Maple* permet d'avoir des expressions plus complexes qu'un simple identificateur en partie gauche du signe d'affectation. Grâce à cette possibilité, on peut exprimer de manière agréable l'orthogonalité de deux vecteurs. L'expression `vscal(v1, v2) := 0` suffit à "déclarer" que `v1` et `v2` sont orthogonaux. De même on peut exprimer qu'un vecteur `v3` est orthogonal à `v1` et `v2` par `vprod(v1, v2) := v3`, on que `v1` est de unitaire par `vnorm(v1) = 1`.

Cette façon de faire (l'utilisation de l'affectation), si elle a le mérite d'être particulièrement simple, souffre néanmoins d'un petit inconvénient : une fois les affectations réalisées, il peut être difficile de revenir en arrière (on peut également noter qu'à cause du mécanisme de simplification de *Maple*, l'utilisateur peut avoir à

réevaluer son expression pour que toutes les transformations soient effectuées, ce qui peut être troublant). De plus, on peut être naturellement amené à manipuler plusieurs jeux d'hypothèses correspondant à des relations différentes entre de même vecteurs. Pour ces raisons, *Quaterman* offre à l'utilisateur une fonction particulière pour exprimer un ensemble de relations entre vecteurs et les appliquer à une expression contenant des quaternions. Cette fonction **`qtrans/side_relations`** est intégrée à la fonction plus générale **`qtrans`** qui est décrite dans la section suivante.

## 2.5 Les autres transformations

*Quaterman* implémente évidemment d'autres transformations que les règles par défaut que nous venons de voir. Ces transformations peuvent être brutalement classifiées en deux groupes : le premier pour les transformations "d'expansion" et le second pour les autres transformations.

Les transformations "d'expansions" sont l'expression de propriétés de distributivité (distributivité du produit des quaternions par rapport à la somme, du produit scalaire...). Elles sont implémentées directement à travers la fonction *Maple* **`expand`** et son mécanisme d'appels spécialisés. En effet, une expression comme **`expand(qadd(...,...))`** provoquera un appel à la fonction **`expand/qadd`**. Chaque opérateur (ainsi que l'opérateur **`Quater`**) a sa propre fonction d'expansion (aussi bien pour les opérations des quaternions que pour celles des vecteurs), ceci afin de pouvoir propager l'expansion à tous les niveaux d'une expression (c'est nécessaire pour pouvoir atteindre les opérations vectorielles à l'intérieur d'un quaternion).

Les autres transformations sont implémentés par une interface commune, la fonction **`qtrans`**. Elles ont la forme générale **`qtrans`**(<expression>, <regle>, $arg_1, \ldots, args_n$) où <regle> désigne une transformation particulière. Cette appel est transformé en un appel à la fonction **`qtrans/`**regle. Ceci permet en particulier à l'utilisateur de rajouter ses propres règles et d'y accéder par cette interface commune. Les transformations disponibles, en plus de **`side_relations`** qui est la transformation permettant de prendre en compte des relations entre vecteurs, sont **`double_cross_prod`** qui applique la règle du double produit vectoriel $(v_1 \wedge v_2) \wedge v_3 \longrightarrow - < v_3, v_2 > v_1 + < v_3, v_1 > v_2$ et **`elim_scal`** qui reconnait les quaternions dont la partie vectorielle est nulle et les transforme en scalaires.

# Chapitre 3

# Petit guide de l'utilisateur

## 3.1 Opérations sur les quaternions

Ce chapitre est consacré à la description des différentes fonctions disponibles dans la bibliothèque *Quaterman*. Le lecteur y trouvera de nombreux exemples qui illustrent les possibilités décrites au chapitre précédent. Cette description est disponible "en ligne" dans *Quaterman* sous forme de page de manuel *Maple* (consultables par la fonction `help` ou le caractère spécial `?`).

### qmake – création d'un quaternion

`qmake(`$x$`, `$y$`, `$z$`, `$t$`)` ou `qmake(`$x$`, `$e$`)`

Cette procédure crée un objet quaternion (un objet ayant le type Maple `function` et possédant le symbole `Quater` comme opérateur). Il existe une fonction `'type/quaternion'` permettant de tester ce type grâce à la fonction Maple `type`. Dans sa forme à quatre arguments, les quatre expressions $x$, $y$, $z$ et $t$ représentent les quatre composantes du quaternion (dans l'ordre canonique suivant la base $(1, i, j, k)$). Dans sa forme à deux arguments, $x$ est la partie scalaire et $e$ la partie vectorielle qui peut être sous la forme de la liste des trois composantes du vecteur ou sous la forme d'une expression (faisant intervenir les opérateurs d'algèbre vectorielle comme `vadd`, `vprod` *etc*).

```
> qmake(1,[x,y,z]);
                              Quater(1, [x, y, z])
> qmake(a,x,y,z);
                              Quater(a, [x, y, z])
> qmake(a, vadd(v1,v2));
                           Quater(a, vadd(v1, v2))
```

### qadd – addition $n$-aire des quaternions

`qadd`(<quaternion>, ..., <quaternion>)

- <quaternion>, ..., <quaternion> : `quater_expr,...,quater_expr`

```
> qadd(q1, Quater(0,[0,0,0]));
                                    q1
> qadd(q1, qsmult(2, q1));
                               qsmult(3, q1)
> qadd(q1, qsmult(2, q1),q2);
                           qadd(q2, qsmult(3, q1))
> qadd(q1,qadd(q2,q3));
                              qadd(q1, q2, q3)
> qadd(Quater(1, [2,3,4]), Quater(2,[0,1,1]));
                            Quater(3, [2, 4, 5])
> qadd(Quater(a, [2,3,4]), Quater(b,[0,1,1]));
               qadd(Quater(a, [2, 3, 4]), Quater(b, [0, 1, 1]))
```

### qsmult – multiplication d'un quaternion par un scalaire

`qsmult`(<scalaire>,<quaternion>)

- <scalaire> : `algebraic`
- <quaternion> : `quater_expr`

```
>  qsmult(0, q1);
                            Quater(0, [0, 0, 0])
>  qsmult(1, q1);
                                    q1
>  qsmult(2, qsmult(3, q1));
                               qsmult(6, q1)
> qsmult(3, Quater(1,[2,3,4]));
                           Quater(3, [6, 9, 12])
> qsmult(3, Quater(1,[x1,x2,x3]));
                     qsmult(3, Quater(1, [x1, x2, x3]))
```

## qmult – multiplication n-aire de quaternions

`qmult`(<quaternion>, ..., <quaternion>)

– <quaternion>, ..., <quaternion> : `quater_expr, ... , quater_expr`

```
> qmult(q1, Quater(0,[0,0,0]));
                              Quater(0, [0, 0, 0])
> qmult(q1, Quater(1,[0,0,0]));
                                      q1
> qmult(q1,qmult(q2,q3));
                               qmult(q1, q2, q3)
> qmult(qsmult(x1,q1), qsmult(x2,q2));
                          qsmult(x1 x2, qmult(q1, q2))

> qmult(Quater(1, [2,3,4]), Quater(2,[0,1,1]));
                             Quater(-5, [3, 5, 11])
> qmult(Quater(a, [2,3,4]), Quater(b,[0,1,1]));
               qmult(Quater(a, [2, 3, 4]), Quater(b, [0, 1, 1]))
> qmult(Quater(1, [2,3,4]), Quater(2,[x1,x2,x3]));
             qmult(Quater(1, [2, 3, 4]), Quater(2, [x1, x2, x3]))
```

## qinv – inverse d'un quaternion

`qinv`(<quaternion>)

– <quaternion> : `quater_expr`

```
> qinv(qinv(q));
                                      q
> qinv(qsmult(x,q));
                               qsmult(1/x, qinv(q))
> qinv(Quater(x,[1,2,3]));
                           qinv(Quater(x, [1, 2, 3]))
> qinv(Quater(1,[1,2,3]));
                      Quater(1/15, [-1/15, -2/15, -1/5])
```

## qconj – conjugué d'un quaternion

`qconj(`<quaternion>`)`

– <quaternion> : `quater_expr`

```
> qconj(qconj(q));
                                    q
> qconj(qsmult(x, q));
                           qsmult(x, qconj(q))
> qconj(Quater(x,[1,2,3]));
                       qconj(Quater(x, [1, 2, 3]))
> qconj(Quater(1,[1,2,3]));
                         Quater(1, [-1, -2, -3])
```

## qnorm – carré de la norme d'un quaternion

`qnorm(`<quaternion>`)`

– <quaternion> : `quater_expr`

```
> qnorm(qsmult(x, q));
                                 2
                                x  qnorm(q)
> qnorm(Quater(1,[1,2,3]));
                                    15
> qnorm(Quater(x,[1,2,3]));
                       qnorm(Quater(x, [1, 2, 3]))
```

## 3.2 Opérations sur les vecteurs

### vadd – addition n-aire de vecteurs

- `vadd(`<vecteur>, ..., <vecteur>`)` avec
  - <vecteur>, ..., <vecteur> : `vect_expr,...,vect_expr`

```
> vadd(v1, [0,0,0]);
                                    v1
> vadd(v1, vsmult(2, v1));
                                vsmult(3, v1)
> vadd(v1, vsmult(2, v1),v2);
                            vadd(v2, vsmult(3, v1))
> vadd(v1,vadd(v2,v3));
                                vadd(v1, v2, v3)
> vadd([1,2,3],[0,1,1]);
                                   [1, 3, 4]
> vadd([x1,x2,x3],[y1,y2,y3]);
                            [x1 + y1, x2 + y2, x3 + y3]
```

### vsmult – multiplication d'un vecteur par un scalaire

- `vsmult(`<scalaire>,<vecteur>`)` avec
  - <scalaire> : `algebraic`
  - <vecteur> : `vect_expr`

```
> vsmult(0, v1);
                                   [0, 0, 0]
> vsmult(1, v1);
                                    v1
> vsmult(2, vsmult(3, v1));
                                vsmult(6, v1)
> vsmult(3, [1,2,3]);
                                   [3, 6, 9]
> vsmult(3, [x1,x2,x3]);
                               [3 x1, 3 x2, 3 x3]
```

## vprod – produit vectoriel de deux vecteurs

- **vprod**(<vecteur>,<vecteur>) avec
  - <vecteur>,<vecteur> : **vect_expr,vect_expr**

```
> vprod(v1, [0,0,0]);
                                  [0, 0, 0]
> vprod(vsmult(x1,v1), vsmult(x2,v2));
                         vsmult(x1 x2, vprod(v1, v2))
> vprod([1,2,3], [0,1,1]);
                                 [-1, -1, 1]
> vprod([a,b,c], [x,y,z]);
                     [- c y + b z, c x - a z, - b x + a y]
```

## vscal – produit scalaire de deux vecteurs

- **vscal**(<vecteur>,<vecteur>) avec
  - <vecteur>,<vecteur> : **vect_expr,vect_expr**

```
> vscal(v1, [0,0,0]);
                                       0
> vscal(vsmult(x1,v1), vsmult(x2,v2));
                              x1 x2 vscal(v1, v2)
> vscal([1,2,3], [0,1,1]);
                                       5
> vscal([a,b,c], [x,y,z]);
                                 a x + b y + c z
> vscal(vprod(v1,v2),v1);
                                       0
```

## vnorm – norme d'un vecteur

- **vscal**(<vecteur>) avec
  - <vecteur> : **vect_expr**

```
> vnorm(vadd(v1,v2));
                              vnorm(vadd(v1, v2))
> vnorm([1,2,3]);
                                     1/2
                                   14
```

## 3.3 Evaluation des expressions

### qeval – evalue au deuxième niveau

- `qeval(<expression>)` avec
  - <expression> : `quater_expr`

Cette fonction évalue les expressions en développant les calculs à partir des composantes des quaternions – lorsqu'elle y a accès.

```
> qmult(q1,q2);
                      qmult(Quater(s1, v1), Quater(s2, v2))
> qeval(");
Quater(
                           s1 s2 - vscal(v1, v2),
                 vadd(vsmult(s2, v1), vsmult(s1, v2), vprod(v1, v2)))
> qeval(qmult(q1,q3));
                            qmult(Quater(s1, v1), q3)
```

On accède au troisième niveau d'évaluation (calcul sur les composantes d'un quaternion) naturellement à partir du niveau précédent et du mécanisme d'évaluation de *MAPLE* :

```
> exp:=qeval(qmult(q1,q2)):
> v1:=[x1,y1,z1];
                                  v1 := [x1, y1, z1]
> v2:=[x2,y2,z2];
                                  v2 := [x2, y2, z2]
> qeval(qmult(q1,q3));
                       qmult(Quater(s1, [x1, y1, z1]), q3)
> qmult(q1,q2);
            qmult(Quater(s1, [x1, y1, z1]), Quater(s2, [x2, y2, z2]))
> exp;
      Quater(s1 s2 - x1 x2 - y1 y2 - z1 z2,
          [s1 x2 + s2 x1 - z1 y2 + y1 z2, s1 y2 + s2 y1 - z1 x2 + x1 z2,
             s1 z2 + s2 z1 - y1 x2 + x1 y2])
> v1:='v1';
                                       v1 := v1
> v2:='v2';
                                       v2 := v2
> exp;
Quater(
                                 s1 s2 - vscal(v1, v2),
                  vadd(vsmult(s2, v1), vsmult(s1, v2), vprod(v1, v2)))
```

Les trois dernières commandes montrent la facilité à repasser du troisième au second niveau d'évaluation.

## 3.4 manipulation des expressions

### expand – développe les expressions

`expand(`<expression>`)` avec

- <expression> : `quater_expr` ou `vect_expr`

Il s'agit en fait de la fonction classique de *MAPLE*, qui lorsqu'elle prend pour argument une expression de type `quater_expr` ou de type `vect_expr` fait appel, selon le cas, à l'une des procédures internes suivantes :

- `expand/Quater`
- `expand/qadd`
- `expand/qsmult`
- `expand/qmult`
- `expand/qconj`
- `expand/qnorm`

ou alors :

- `expand/vadd`
- `expand/vsmult`
- `expand/vprod`
- `expand/vscal`

Ces procédures "transmettent" la commande `expand` à leurs arguments avant ou après les avoir manipulés. Ainsi, cette commande applique, de manière récursive, toutes les règles suivantes

- aux expressions de type `quater_expr` :

$$a(q_1 + q_2) \longrightarrow (aq_1) + (aq_2)$$

$$(q_1 + q_2)q_3 \longrightarrow q_1q_3 + q_2q_3$$

$$(q_1q_2)^{-1} \longrightarrow {q_2}^{-1}{q_1}^{-1}$$

- aux expressions de type `vect_expr` :

$$a(v_1 + v_2) \longrightarrow (av_1) + (av_2)$$

$$(v_1 + v_2) \wedge v_3 \longrightarrow v_1 \wedge v_3 + v_2 \wedge v_3$$

$$< (v_1 + v_2), v_3 > \longrightarrow < v_1, v_3 > + < v_2, v_3 >$$

## qtrans – applique des règles de transformation sur les expressions

`qtrans`(<expression>, <règle>, $arg_1, \ldots, arg_n$)

ou de manière équivalente :

`'qtrans/`règle`'`(<expression>, $arg_1, \ldots, arg_n$)

La procédure `qtrans` est une interface unifiée générale à l'application de fonctions de transformations. Un appel tel que `qtrans`(<expr>, <règle>, $arg_1, \ldots, arg_n$) s'évalue en le résultat de l'expression `'qtrans/`règle`'`( <expr>, $arg_1, \ldots, arg_n$) dès qu'il existe une procédure Maple de nom `'qtrans/`règle`'`. L'utilisateur peut donc, de manière externe, étendre cette fonction avec de nouvelles possibilités en écrivant la fonction correspondante à la règle qu'il désire implémenter.

Trois procédures internes associées à des règles classiques sont prédéfinies :

- `'qtrans/double_cross_prod'` qui applique la règle du double produit vectoriel à la composante vectorielle d'un quaternion :

$$(v_1 \wedge v_2) \wedge v_3 \longrightarrow - < v_3, v_2 > v_1 + < v_3, v_1 > v_2$$

- `'qtrans/elim_scal'` qui reconnaît les quaternions dont la partie vectorielle est nulle et les force à être considérés comme des scalaires,

- `'qtrans/side_relations'` qui applique un ensemble d'affectations définies par l'utilisateur pour exprimer notamment les propriétés de colinéarité ou d'orthogonalité de certains vecteurs.

Les deux premières procédures ne prennent pas d'arguments optionnels, alors que la troisième prend bien sûr comme argument l'ensemble des équations traduisant les affectations désirées :

- <arguments> : `{`<equation>`,...,`<equation>`}` avec
  - <equation> : `quater_expr = quater_expr`

Il existe des abréviations pour ces trois transformations (reconnues dans la fonction `qtrans`), ce sont respectivement `dcp`, `es` et `rel`. Nous les utiliserons dans les exemples suivants.

```
> qdcp := qmake(x, vprod(vprod(u, v), w));

                    qdcp := Quater(x, vprod(w, vprod(v, u)))

> qtrans(qdcp, dcp);

       Quater(x, vadd(vsmult(vscal(w, u), v), vsmult(- vscal(v, w), u)))

> expr:=qmult(q1, qmake(x, 0,0,0));

              expr := qmult(Quater(x1, v1), Quater(x, [0, 0, 0]))

> qtrans(expr,es);

                            qsmult(x, Quater(x1, v1))

> qtrans(Quater(x, vadd(v3, v4)), rel, {vadd(v3,v4)=v5});

                                  Quater(x, v5)

> qtrans(Quater(x, vsmult(a, V)), rel, {vsmult(a, V)=W});

                      Quater(x, vsmult(a, vsmult(1/a, W)))

> qtrans(vnorm(V1) + vscal(V1, V1), rel, {vscal(V1,V1) = 3});

                                      1/2
    3    + 3

> qtrans(vsmult(vscal(V1, V3),vscal(V1, V3), vprod(V1, V2)), rel,
```

```
                                                {vprod(V1, V2)=V3});

                                      [0, 0, 0]

>qtrans( vscal(V2, V3), rel, {vprod(V1,V2)=V3});

                                         0
```

## 3.5 Interface entre quaternions et rotations

### rot2quat et quat2rot – interface entre quaternions et caractéristiques géométrique des rotations

`rot2quat`($\theta$ , v)

`quat2rot`($q$) ou `quat2rot`($q$, `'force'`)

La procédure **rot2quat** rend le quaternion associé à la rotation d'angle $\theta$ (exprimé en radian s'il est numérique) et d'axe portant le vecteur $v$. **rot2quat** rend un résultat explicite utilisant les coefficients du vecteur si celui-ci est explicite (liste des trois composantes), une expression implicite faisant appel aux opérateurs vectoriels **vsmult** et **vnorm** si le vecteur n'est qu'un nom de variable.

La procédure **quat2rot** calcule l'axe et l'angle de la rotation associée au quaternion $q$ (sous la forme d'une liste). Dans le cas où le système ne dispose pas d'assez d'informations pour s'assurer que la norme de $q$ est bien 1, il renvoie une erreur. Si l'utilisateur le désire, il peut donner l'argument optionnel **force** qui permet de continuer le calcul dans ce cas.

```
> rot2quat(theta, v);

                                          sin(1/2 theta)
              Quater(cos(1/2 theta), vsmult(--------------, v))
                                            vnorm(v)

> quat2rot(");
Error, (in quat2rot) cannot compute norm

> rot2quat(theta, [Vx, Vy, Vz]);

    Quater(cos(1/2 theta),

          sin(1/2 theta) Vx     sin(1/2 theta) Vy     sin(1/2 theta) Vz
        [--------------------, --------------------, --------------------])
            2     2     2 1/2     2     2     2 1/2     2     2     2 1/2
           (Vx  + Vy  + Vz )     (Vx  + Vy  + Vz )     (Vx  + Vy  + Vz )

> quat2rot(");

                    Vx                    Vy                    Vz
  [theta, [--------------------, --------------------, --------------------]]
              2     2     2 1/2     2     2     2 1/2     2     2     2 1/2
             (Vx  + Vy  + Vz )     (Vx  + Vy  + Vz )     (Vx  + Vy  + Vz )

> subs(Vx^2+Vy^2+Vz^2=1, ");

                             [theta, [Vx, Vy, Vz]]
> qmake(sqrt(1/2),0,sqrt(1/2),0);

                                    1/2           1/2
                       Quater(1/2 2   , [0, 1/2 2   , 0])

> quat2rot(");

                              [1/2 Pi, [0, 1, 0]]

> rot2quat(op("));

                                    1/2           1/2
                       Quater(1/2 2   , [0, 1/2 2   , 0])

> quat2rot(Quater(s, [x, y, z]));
Error, (in quat2rot) cannot compute norm

>quat2rot(Quater(s, [x, y, z]), force);
                                 x             y             z
             [2 arccos(s), [-----------, -----------, -----------]]
                             2 1/2         2 1/2         2 1/2
                           (1 - s )      (1 - s )      (1 - s )
```

## mat2quat et quat2mat – interface entre quaternions et caractéristiques géométriques des rotations

`quat2mat(`$q$`)`

`mat2mat(`$M$`)`

La procédure **quat2mat** calcule la matrice de la rotation associée au quaternion unitaire $q$. Son résultat est une matrice carrée Maple à trois dimensions. Un calcul de la norme du quaternion est effectué pour s'assurer qu'elle vaut bien 1. Ce calcul utilise **qeval** et la procédure Maple **simplify**.

La procédure **mat2quat** calcule le quaternion unitaire associé à la matrice $M$, qui doit être de type Maple **matrix** de dimension 3. Le résultat n'a pas de sens a priori si $M$ n'est pas une matrice de rotation (on ne le teste pas).

```
> q:=qmake(0,1,0,0);

                          q := Quater(0, [1, 0, 0])

> quat2mat(q);

                                 [ 1  0   0 ]
                                 [          ]
                                 [ 0 -1   0 ]
                                 [          ]
                                 [ 0  0  -1 ]

> mat2quat(");

                             Quater(0, [2, 0, 0])

> q := qmake(cos(theta), sin(theta), 0, 0);

                 q := Quater(cos(theta), [sin(theta), 0, 0])

> quat2mat(q);

          [ 1             0                         0             ]
          [                                                       ]
          [                    2                                  ]
          [ 0     2 cos(theta)  - 1     - 2 cos(theta) sin(theta) ]
          [                                                       ]
          [                                          2            ]
          [ 0  2 cos(theta) sin(theta)      2 cos(theta)  - 1     ]

> mat2quat(");

                    Quater(cos(theta), [sin(theta), 0, 0])
```

## makerot – construction de la matrice de rotation correspondant à 3 rotations successives autour de 3 axes

`makerot`($l_{axes}$, $l_{angles}$)

Cette procédure construit la matrice de rotation correspondant à trois rotations successives autour des trois axes de la liste $l_{axes}$. Un axe est un symbole **x**, **y** ou **z** représentant les axes du repère euclidien de référence. La liste $l_{angles}$ est une liste de symboles qui vont représenter les angles respectifs de ces rotations.

```
# Les angles de Bryant
> makerot([x,y,z],[q1,q2,q3]);
          [cos(q2) cos(q3), - cos(q2) sin(q3), sin(q2)]

          [sin(q1) sin(q2) cos(q3) + cos(q1) sin(q3),

              - sin(q1) sin(q2) sin(q3) + cos(q1) cos(q3), - sin(q1) cos(q2)]

          [- cos(q1) sin(q2) cos(q3) + sin(q1) sin(q3),

                cos(q1) sin(q2) sin(q3) + sin(q1) cos(q3), cos(q1) cos(q2)]

> makerot([z,y,x],[Pi, Pi/2,Pi/4]);

                        [           1/2         1/2 ]
                        [  0  - 1/2 2     - 1/2 2   ]
                        [                           ]
                        [           1/2        1/2  ]
                        [  0  - 1/2 2      1/2 2    ]
                        [                           ]
                        [ -1       0          0     ]
```

`rotate`($q$, $v$)

Cette procédure calcule l'image du vecteur $v$ (qui doit être une liste ou un vecteur Maple) par la rotation donnée par le quaternion $q$ (aucun test n'est réalisé pour vérifier que $q$ est bien de norme 1). Le résultat est un quaternion.

```
> rotate(rot2quat(Pi/2, [1, 0, 0]), [x, y, z]);

                         1/2        1/2
       qmult(Quater(1/2 2   , [1/2 2   , 0, 0]), Quater(0, [x, y, z]),

                           1/2        1/2
           qinv(Quater(1/2 2   , [1/2 2   , 0, 0])))

> qeval(");

Quater(0,

          1/2       1/2          1/2           1/2       1/2          1/2
  [x, 1/2 2    (1/2 2    y - 1/2 2    z) - 1/2 2    (1/2 2    z + 1/2 2    y),

          1/2       1/2          1/2           1/2       1/2          1/2
      1/2 2    (1/2 2    z + 1/2 2    y) + 1/2 2    (1/2 2    y - 1/2 2    z)]

  )
```

## 3.6 calcul différentiel

### qdiff – dérivation d'un quaternion dépendant d'un paramètre réel

`qdiff`(*expr*, $v_1, \ldots v_n$)

La fonction `qdiff` réalise la dérivation de l'expression *expr* représentant un quaternion par rapport aux variables $v_1, \ldots v_n$ (qui sont des paramètres supposés être des réels). La dérivation se fait composante par composante. Dans le cas d'une partie vectorielle connue, on fait appel à `vdiff`.

Les possibilités de cette fonction extensibles par l'utilisateur sur le même modèle que pour `diff` : l'évaluation de `qdiff(f(x,y,z),t)` entraîne l'évaluation de `` `qdiff/f`([x,y,z], t) `` si la procédure de nom `` `qdiff/f` `` existe. Il faut noter que les arguments de `f` sont passés dans une liste, contrairement à la convention de `diff`, ceci afin de pouvoir traiter des fonctions à nombre variable d'arguments.

Contrairement à la fonction `diff` de Maple, cette fonction suppose une dépendance implicite des symboles. Pour déclarer qu'un quaternion ne dépend pas d'un paramètre, il faut donc le faire explicitement par assignation (par exemple `qdiff(q, t) := 0`).

```
> qdiff(qadd(q1, q2), x, y);

                    qadd(qdiff(q1, x, y), qdiff(q2, x, y))

> qdiff(qmake(sin(t), v), t);

                         Quater(cos(t), vdiff(v, t))

> v := [a*cos(t), b*sin(t), 1];

                         v := [a cos(t), b sin(t), 1]

> "";

                  Quater(cos(t), [- a sin(t), b cos(t), 0])

> qdiff(qadd(v1, v2), t);

                       qadd(qdiff(v1, t), qdiff(v2, t))

> qdiff(qmult(q1, q2, q3), t);

       qadd(qmult(q1, q2, qdiff(q3, t)), qmult(q1, qdiff(q2, t), q3),

           qmult(qdiff(q1, t), q2, q3))

> qdiff(qinv(q1), t);

             qsmult(-1, qmult(qinv(q1), qdiff(q1, t), qinv(q1)))

> qdiff(qconj(q), t);

                               qconj(qdiff(q, t))
```

## vdiff – dérivation d'un vecteur dépendant d'un paramètre réel

**vdiff**(*expr*, $v_1, \ldots v_n$)

Cette procédure différencie l'expression *expr* qui représente un vecteur, par rapport aux paramètres $v_1, \ldots v_n$. Cette différentiation est une différentiation composante par composante.

Cette procédure a un mécanisme d'extension semblable à celui de **qdiff**. Comme elle, il y a dépendance implicite des symboles.

```
> vdiff([exp(x), sin(x), sinh(x)], x);

                          [exp(x), cos(x), cosh(x)]

> vdiff(vadd(v1, v2), t);

                       vadd(vdiff(v1, t), vdiff(v2, t))

> vdiff(vscal(v1, v2), t);

             vadd(vscal(v1, vdiff(v2, t)), vscal(v2, vdiff(v1, t)))

> vdiff(vprod(v1, v2), t);

       vadd(vsmult(-1, vprod(v2, vdiff(v1, t))), vprod(v1, vdiff(v2, t)))
```

# Un exemple d'utilisation de Quaterman

```
# l'exemple de la composition des 3 rotations donn'e par le CNES

# les vecteurs sont norme's
> vnorm(n1) := 1;

                                vnorm(n1) := 1

> vnorm(n2) := 1;

                                vnorm(n2) := 1

> vnorm(n3) := 1;

                                vnorm(n3) := 1


> Q1 := rot2quat(a1(t), n1);

            Q1 := Quater(cos(1/2 a1(t)), vsmult(sin(1/2 a1(t)), n1))


> Q2 := rot2quat(a2(t), n2);

            Q2 := Quater(cos(1/2 a2(t)), vsmult(sin(1/2 a2(t)), n2))


> Q3 := rot2quat(a3(t), n3);

            Q3 := Quater(cos(1/2 a3(t)), vsmult(sin(1/2 a3(t)), n3))

# le truc au milieu
> O := rot2quat(o(t), n);

                                                 sin(1/2 o(t))
              O := Quater(cos(1/2 o(t)), vsmult(-------------, n))
                                                   vnorm(n)

> vnorm(n) := 1;

                                vnorm(n) := 1


> qmult(Q3, Q2, O, qconj(Q2), qconj(Q3));

        qmult(Quater(cos(1/2 a3(t)), vsmult(sin(1/2 a3(t)), n3)),

            Quater(cos(1/2 a2(t)), vsmult(sin(1/2 a2(t)), n2)),

            Quater(cos(1/2 o(t)), vsmult(sin(1/2 o(t)), n)),

            qconj(Quater(cos(1/2 a2(t)), vsmult(sin(1/2 a2(t)), n2))),

            qconj(Quater(cos(1/2 a3(t)), vsmult(sin(1/2 a3(t)), n3))))


# on ne fait plus qu'un seul quaternion
```

```
> P := qeval(");

P := Quater(

(%3 cos(1/2 a2(t)) + sin(1/2 a2(t)) vscal(n2, %4)) cos(1/2 a3(t)) +

    sin(1/2 a3(t)) vscal(n3,

    vadd(vsmult(- %3 sin(1/2 a2(t)), n2), vsmult(cos(1/2 a2(t)), %4),

        vsmult(sin(1/2 a2(t)), vprod(n2, %4)))                        ),

vadd(

vsmult(- (%3 cos(1/2 a2(t)) + sin(1/2 a2(t)) vscal(n2, %4)) sin(1/2 a3(t)), n3)

    ,

vsmult(cos(1/2 a3(t)),

    vadd(vsmult(- %3 sin(1/2 a2(t)), n2), vsmult(cos(1/2 a2(t)), %4),

        vsmult(sin(1/2 a2(t)), vprod(n2, %4)))                        ),

vsmult(sin(1/2 a3(t)),

  vprod(n3, vadd(vsmult(- %3 sin(1/2 a2(t)), n2), vsmult(cos(1/2 a2(t)), %4),

                vsmult(sin(1/2 a2(t)), vprod(n2, %4)))                        )

  )

)

)

%1 := vadd(vsmult(cos(1/2 a3(t)) sin(1/2 a2(t)), n2),

    vsmult(cos(1/2 a2(t)) sin(1/2 a3(t)), n3),

    vsmult(- sin(1/2 a3(t)) sin(1/2 a2(t)), vprod(n2, n3)))

%2 :=

    cos(1/2 a3(t)) cos(1/2 a2(t)) - sin(1/2 a3(t)) sin(1/2 a2(t)) vscal(n2, n3)

%3 :=           %2 cos(1/2 o(t)) - sin(1/2 o(t)) vscal(n, %1)

%4 := vadd(vsmult(%2 sin(1/2 o(t)), n), vsmult(cos(1/2 o(t)), %1),

    vsmult(- sin(1/2 o(t)), vprod(n, %1)))

# il faut maintenant exprimer le fait que les trois vecteurs
# forment une base orthonorme'e ...
# on donne d'abord les produits vectoriels, desquels on peut de'duire
# des produits scalaires nuls...

> RELv := {vprod(n1, n2) = n3, vprod(n2, n3) = n1, vprod(n3, n1) = n2};

  RELv :=

      {vprod(n1, n2) = n3, vprod(n2, n3) = n1, vsmult(-1, vprod(n1, n3)) = n2}

# les relations  vscal(n1, n2) = 0, vscal(n2, n3) = 0, vscal(n3, n1) = 0
# sont de'duites automatiquement a` partir de RELv.
# les relations {vscal(n1, n1) = 1, vscal(n2, n2) = 1, vscal(n3, n3) = 1};
# sont de'duites du "normage" de n1, n2 et n3;

# on ne veut pas voir...
> expand(P):

# un coup de transformation :
> qtrans(", rel, RELv):

# on applique la re`gle du double produit vectoriel :
```

```
> qtrans('', dcp):

# et finalement le resultat
> qtrans('', rel, RELv);

Quater(

                              2               2
cos(1/2 o(t)) cos(1/2 a3(t))  cos(1/2 a2(t))  + cos(1/2 a3(t)) cos(1/2 a2(t))

    sin(1/2 o(t)) sin(1/2 a3(t)) sin(1/2 a2(t)) vscal(n1, n)

                                    2               2
     + cos(1/2 o(t)) cos(1/2 a3(t))  sin(1/2 a2(t))  + cos(1/2 a3(t))

    sin(1/2 a2(t)) sin(1/2 o(t)) cos(1/2 a2(t)) sin(1/2 a3(t))

    vscal(n2, vprod(n3, n)) - cos(1/2 a3(t)) sin(1/2 o(t)) sin(1/2 a3(t))

                  2
    sin(1/2 a2(t))  vscal(n2, vprod(n1, n))

                    2                               2
     + cos(1/2 a2(t))  cos(1/2 o(t)) sin(1/2 a3(t))  + sin(1/2 a3(t))

                  2
    sin(1/2 a2(t))  sin(1/2 o(t)) cos(1/2 a3(t))

    vscal(n3, vadd(vsmult(vscal(n2, n), n2), vsmult(-1, n)))

                    2               2
     + sin(1/2 a3(t))  sin(1/2 a2(t))  cos(1/2 o(t))

                                                2
     + cos(1/2 a2(t)) sin(1/2 o(t)) sin(1/2 a3(t))  sin(1/2 a2(t)) vscal(n2, n)

                    2                               2
     - sin(1/2 a2(t))  sin(1/2 o(t)) sin(1/2 a3(t))  vscal(n2, n) vscal(n1, n3)

                                                2
     - cos(1/2 a2(t)) sin(1/2 o(t)) sin(1/2 a3(t))  sin(1/2 a2(t))

    vscal(n3, vprod(n1, n)) + 2 sin(1/2 a3(t)) vscal(n3, vprod(n2, n))

    cos(1/2 a2(t)) sin(1/2 o(t)) cos(1/2 a3(t)) sin(1/2 a2(t)),

vadd(

vsmult(

                                                                  2
- 2 sin(1/2 a3(t)) sin(1/2 o(t)) cos(1/2 a3(t)) sin(1/2 a2(t))  vscal(n2, n)

                                                               2
   - 2 sin(1/2 a2(t)) sin(1/2 o(t)) cos(1/2 a2(t)) sin(1/2 a3(t))  vscal(n3, n)

                                    2               2
   + sin(1/2 o(t)) sin(1/2 a3(t))  sin(1/2 a2(t))  vscal(n1, n),

n1),

vsmult(

                              2               2
sin(1/2 o(t)) cos(1/2 a3(t))  sin(1/2 a2(t))  vscal(n2, n) + 3 cos(1/2 a3(t))

    sin(1/2 a2(t)) sin(1/2 o(t)) cos(1/2 a2(t)) sin(1/2 a3(t)) vscal(n3, n)

                                                                  2
     - cos(1/2 a3(t)) sin(1/2 o(t)) sin(1/2 a3(t)) sin(1/2 a2(t))  vscal(n1, n)

                    2               2
     - sin(1/2 a3(t))  sin(1/2 a2(t))  sin(1/2 o(t)) vscal(n3, vprod(n1, n)) +

                                  2
```

```
    cos(1/2 a3(t)) sin(1/2 a2(t))  sin(1/2 o(t)) sin(1/2 a3(t))

    vscal(n3, vprod(n2, n)),

n2),

vsmult(

                              2               2
sin(1/2 o(t)) cos(1/2 a2(t))  sin(1/2 a3(t))  vscal(n3, n)

     - cos(1/2 a2(t)) sin(1/2 o(t)) sin(1/2 a3(t))^2 sin(1/2 a2(t)) vscal(n1, n)

                                                                    2
     - sin(1/2 a2(t)) sin(1/2 o(t)) cos(1/2 a2(t)) sin(1/2 a3(t))

    vscal(n2, vprod(n3, n))

                                    2               2
     + sin(1/2 o(t)) sin(1/2 a3(t))  sin(1/2 a2(t))  vscal(n2, vprod(n1, n)) +

    cos(1/2 a3(t)) cos(1/2 a2(t)) sin(1/2 o(t)) sin(1/2 a3(t)) sin(1/2 a2(t))

    vscal(n2, n),

n3),

                                    2               2
vsmult(sin(1/2 o(t)) cos(1/2 a3(t))  cos(1/2 a2(t)) , n),

                    2               2
vsmult(cos(1/2 a3(t))  sin(1/2 a2(t))  sin(1/2 o(t)),

    vadd(vsmult(vscal(n2, n), n2), vsmult(-1, n))),

vsmult(

                                    2
    2 cos(1/2 a3(t)) cos(1/2 a2(t))  sin(1/2 o(t)) sin(1/2 a3(t)), vprod(n3, n)

    ),

vsmult(

                                                2
    2 cos(1/2 a2(t)) sin(1/2 o(t)) cos(1/2 a3(t))  sin(1/2 a2(t)), vprod(n2, n)

    ),

                    2               2
vsmult(sin(1/2 a3(t))  cos(1/2 a2(t))  sin(1/2 o(t)),

    vadd(vsmult(vscal(n3, n), n3), vsmult(-1, n))),

vsmult(

  - cos(1/2 a3(t)) cos(1/2 a2(t)) sin(1/2 o(t)) sin(1/2 a3(t)) sin(1/2 a2(t)),

  vprod(n1, n))

)

)
```

# Références


[Cas87] P. Casteljau. *Les quaternions*. Hermes, Paris, Londres, Lausanne, 1987.

[GC89] M. Gerardin and A. Cardona. Kinematics and dynamics of rigid and flexible mechanisms using finite elements and quaternion algebra. *Computational Mechanics*, 4:p 115–135, 1989.

[LB87] Michel Le Borgne. Quaternions et controle sur l'espace des rotations. Rapport de recherche 751, INRIA, November 1987.

[Mou91] Bernard Mourrain. *Approche effective de la théorie des invariants des groupes classi ques*. PhD thesis, Ecole Polytechnique, September 1991.

[Mou92] Bernard Mourrain. Computable identities in the algebra of formal matrices. *Theoretical Computer Science*, 1992. to appear.

[RA90] Luis Reyes-Avila. Quaternions : une représentation paramétrique systématique des rotations finies. Rapport de recherche 1303, INRIA, October 1990.

[RA91] Luis Reyes-Avila. Quaternions : une représentation paramétrique systématique des rotations finies. partie 2 : Quelques applications. Rapport de recherche 1303, INRIA, 1991.