Calcul symbolique

Généralités

G. Vinsard

Gerard.Vinsard@univ-lorraine.fr

3 janvier 2016

La structure de Maxima / wxMaxima / Imaxima

- Maxima comporte
  - un ensemble de fonctions écrites en Lisp dont l'objectif est de transformer des expressions Lisp en autres expressions Lisp ;
  - une interface traduisant des expressions écrites en langage de Maxima en expressions Lisp et vice-versa et cela à partir d'une ligne de commande.

- wxMaxima est une interface qui fournit la ligne de commande nécessaire à Maxima et qui traduit les expressions renvoyées en l'idéographie utilisée pour les mathématiques.

- Imaxima rend le même service que wxMaxima mais à partir de d'Emacs.

- Et il existe encore TeXEmacs et d'autres
`https://en.wikipedia.org/wiki/Maxima_(software)`

Le cœur du calcul symbolique

► C'est la transformation d'expressions en autres expressions. Par exemple si l'expression de départ est

$$"(x+1)^2"$$

il est possible de demander au logiciel de calcul symbolique de la développer, soit de renvoyer

$$"x^2+2*x+1"$$

► De façon interne les étapes suivies par le logiciel seront quelque chose comme : 1) $"(x+1)*(x+1)"$; 2) $"(x+1)*x+(x+1)*1"$; 3) $"x*x+x+x+1"$; 4) $"x^2+2*x+1"$.
Mais il n'est pas nécessaire de connaître ce détail.

► Par contre il faut comprendre globalement ce que contient ce genre de demande : « développer l'expression. » c'est à dire maîtriser le vocabulaire mathématique dans le contexte du calcul effectif.

Pourquoi Maxima ?

▶ Les logiciels de calcul symbolique sont : Mathematica, Maple, Mupad (dans Matlab), Xcas/Giac, Sage, Calc (sous Emacs) et d'autres encore dont Maxima.

▶ Maxima présente l'avantage d'être un logiciel libre (comme Xcas/Giac, Sage, Calc) ;

▶ Il a cependant le défaut de sa qualité, n'étant pas commercial (comme Mathematica, Maple, Mupad) ses interfaces sont moins raffinées ;

▶ Mais ce défaut devient une qualité s'il s'agit de comprendre les mécanismes sous-jacents au calcul symbolique.

Téléchargement et Documentation

- Le site de Maxima est `http://maxima.sourceforge.net/` qui donne notamment l'accès au manuel `http://maxima.sourceforge.net/docs/manual/maxima.pdf` ainsi qu'à d'autres documentations dont celles de `http://michel.gosse.free.fr/documentation/index.html`

- Le site de wsMaxima est `http://andrejv.github.io/wxmaxima/index.html`

- Celui de Imaxima `https://sites.google.com/site/imaximaimath/`

- Le sites de téléchargements contiennent des binaires tout prêts pour Maxima et wxMaxima sous Windows, Mac-os et certaines distributions de Linux ;
- Sinon il est également possible de compiler Maxima (et wxMaxima) à partir des sources, en gros c'est without agonizing pain.

Objectif des transparents suivants

- Il existe un véritable cours en français assez exhaustif de Maxima

`http://bdesgraupes.pagesperso-orange.fr/UPX/Tutoriels/presentation_maxima.pdf`

- Les transparents suivants n'ont pas pour objectif de le remplacer mais plutôt de fournir une introduction brève au vocabulaire (par imprégnation [1]) et aux concepts (par description et exemples) du calcul symbolique en général mais incarné par Maxima.

- La description exacte de la syntaxe du langage n'est donc pas fournie ; elle devra être apprise lors des séances pratiques.

1. Les mots qui dénotent les choses peuvent être introduits avant même que les choses soient définies et associées à ces choses

Variable

► Une variable est l'association d'un nom et d'une valeur : de façon interne un logiciel de calcul symbolique maintient une liste de la forme

$$"\{(x_1, v_1), (x_2, v_2), \ldots, (x_N, v_N)\}"$$

où les "x_n" sont les noms et les "v_n" les valeurs associées à ces noms.

► L'instruction qui affecte une valeur à un nom s'appelle l'affectation.

► L'invocation d'une variable dans une instruction se fait par son nom et c'est la valeur qui est utilisée dans l'action.

► Si une variable est invoquée sans qu'elle ait été préalablement affectée, sa valeur sera son nom et on l'appelle un symbole.

Instructions

▶ Une instruction est une suite de mots composés d'opérateurs, de fonctions et de variables qui est interprété par le logiciel pour fournir un résultat. Par exemple (avec la syntaxe de Maxima) :

▶ l'affectation "$x : 1$"

▶ Les opérations arithmétiques "$x + 1$", "$x - y$", "$x * y$", "x/y", $x\hat{}2$, $sqrt(x)$ ...
(une suite d'opérations arithmétiques est appelée une expression)

▶ La désaffectation "$x : 'x$" ou "$kill(x)$" ou "$remove(x)$" ou "$remove(x, value)$" (chacune de ces instructions réalisant la fonction avec des nuances plus ou moins différentes.)

▶ La factorisation d'une expression "$factor(x\hat{}2 - 1)$"

▶ la simplification "$ratsimp((x\hat{}2 - 2)/(x - 1))$"

▶ La dérivation par rapport à une variable (non affectée) "$diff(x\hat{}2, x)$"

▶ La primitivation par rapport à une variable (non affectée) "$integrate(2 * x, x)$"

▶ et bien d'autres

Ici les opérateurs sont les symboles "$+ - */\hat{}$" et les fonctions "$sqrt$" "$kill$", "$remove$", "$factor$", "$ratsimp$", "$diff$" et "$integrate$"

Fonctions

- Une fonction est l'association d'un nom à un processus effectuant une suite d'instructions ; ce processus utilise des données (les arguments de la fonction) et fournit (retourne) un résultat (la valeur de la fonction qui dépend de celle des arguments).
- Une métaphore utile est celle des entrées/sorties : la fonction admet des entrées et une sortie ; les arguments sont placés dans ces entrées et la valeur en ressort à la sortie.
- La syntaxe la plus simple de Maxima pour l'invocation d'une fonction (on dit l'appel) est

$$"f(x_1, x_2, \ldots, x_N)"$$

où f est le nom de la fonction et $x_1, x_2, \ldots, x_N$ ses arguments. Mais il y en a une autre

$$"apply(f, [x_1, x_2, \ldots, x_n])"$$

où *"apply"* est une fonction qui sélectionne le processus associé au nom $"f"$, place les arguments $"x_1, x_2, \ldots, x_N"$ dans les entrées, et retourne une valeur en sortie.

Opérateurs

► Certaines fonctions sont très souvent utilisées et il est commode de modifier la syntaxe de langage pour introduire des opérateurs qui peuvent leur être substitué ;

► Par exemple "+" : la fonction "*plus*" n'existe en fait pas nominativement dans Maxima mais elle peut être invoquée par

$$"apply("+", [a, b, c, d])"$$

Il est plus commode d'écrire comme "$a + b + c + d$" que d'utiliser la fonction "+" ;

► Les opérateurs les plus commmuns sont arithmétiques "+", "−", "*", "/", "^" ; mais il existe également des opérateurs qui peuvent être utilisés sur d'autre type de données que des scalaires comme "." pour le produit scalaire de vecteurs définis par composantes.

Types de données : les atomes

- Les atomes (ou scalaires) sont
  - soit des nombres entiers "3", décimaux "3.4", flottants (à virgule flottantes) "$3.2e+4$" où "3.2" peut être appelé la mantisse et "+4" l'exposant, bigfloat "$3.2d+600$" qui sont des flottants admettant des exposants plus grands ;
  - soit des symboles comme "x" ;
  - soit des chaines de caractère comme ""mot"" (attention au double sens du signe "" ')

leur caractéristique est d'être atomique dans le sens où ils ne comportent pas plusiseurs parties. Ce qui peut se discuter mais il y a un arbitre dans Maxima qui est la fonction "$atom()$"

"$atom(3)$ renvoie $true$"

et c'est vrai pour tous les exemples donnés.

Types de données : les quasi-atomes

► Il y a les fractions rationnelles "3/2" ; et le mécanisme interne de Maxima fait qu'elle sont simplifiées automatiquement

"9/6 renvoie 3/2"

► les complexes : la racine carrée de -1 se note "%i" et donc un complexe $a + i\ b$ s'écrit "$a + \%i * b$"

Types et Fonctions : exemple de "*sqrt*"

► L'argument de "*sqrt*" peut être : un nombre entier, rationnel, décimal, décimal flottant ; un symbole ; une expression. Chacun de ces cas apporte une sortie calculée par un processus différent.

► Entier carré : "*sqrt*(4)" retourne "2"

► Entier non carré : "*sqrt*(7)" retourne "*sqrt*(7)", "*sqrt*(6)" retourne "*sqrt*(6)"

► Entier produits de carrés et de non carré "*sqrt*(12)" retourne "2 ∗ *sqrt*(3)" (il y a factorisation de l'entier et application de la racine carré aux éléments de cette factorisation)

► Rationnel : "*sqrt*(3/4)" retourne "*sqrt*(3)/2" (toujours la décomposition)

► nombre décimal : "*sqrt*(1.44)" renvoie "1.2" ce qui est exact ; mais bien entendu il y a troncature des décimales dans d'autres cas.

► "x^2" renvoie "*abs*(x)" sauf si préalablement "x" est déclaré réel positif par "*assume*($x > 0$)"

Une ≪ simple ≫ fonction se réfère à des algorithmes différents suivant le type de l'argument qui lui est soumis.

Types de données : les listes

▶ Une liste est une structure de données permettant de rassembler en un seul objet une collection de données. Sa syntaxe est

$$"[a, b, c]"$$

Ce n'est pas un atome et elle peut être modifiée. Par exemple si la liste "l" a été affectée par "$l : \{a, b, c\}$" son premier élément "a" s'obtient par "$first(l)$" ; le reste des éléments "$\{b, c\}$" par "$rest(l)$".

▶ Les éléments d'une liste peuvent être d'un type quelconque, notamment une autre liste et les types des éléments d'une liste peuvent être hétérogènes par exemple dans

$$"[a, [b, c], d]"$$

le second élément est la liste "$[b, c]$".

Listes et tableaux

▶ Du point de vue de la structure interne (au logiciel) des données une liste et un tableau forment deux objets qui ont en commun la propriété énoncée au transparent précédent pour la liste et ont pour différence leur mode d'adressage :

- Tout élément d'une liste est associé à un ≪ pointeurs ≫ qui permet de localiser l'élément suivant ;
- Les éléments d'un tableau sont eux rangés dans des cases qui sont celles du tableau.

Cette différence a de l'importance vis-à-vis de l'utilisation de l'objet : les éléments du tableaux nécessitent *a priori* moins d'opérations internes pour être manipulés que ceux de la liste ; a contrario il est plus facile de transformer la structure de la liste que celles du tableau (ajouter ou retrancher des éléments).

▶ Maxima possède le type de donnée de tableau (array) mais sauf s'il s'agit de réaliser des optimisationas particulières il est bien plus commode d'éviter de s'en servir. D'autant que les opérations typiques de tableaux peuvent être réalisées sur les listes.

Opérations sur les listes

Si $"l = [a, b, c, d]"$ et $"p = [e, f]"$

Operations de type liste	Operations de type tableau
$"first(l)" \longrightarrow "a"$;	$"l[2]" \longrightarrow "b"$
$"rest(l)" \longrightarrow "[b, c, d]"$	$"[a, b, c] + [d, e, f]" \longrightarrow$
$"append(l, p)" \longrightarrow "[a, b, c, d, e, f]"$	$"[a + d, b + e, c + f]"$
$"cons(x, l)" \longrightarrow "[x, a, b, c, d]"$	$"[a, b, c].[d, e, f]" \longrightarrow$
$"reverse(l)" \longrightarrow "[d, c, b, a]"$	$"a * d + b * e + c * f"$
$"sort([b, x, a])" \longrightarrow "[a, b, x]"$	...
...	

Condition

- *"if"* : Des syntaxes possibles sont

$$"if < cond > then\ block([], < inst1 >, \ldots, < instN >)"$$

$$\begin{array}{rl} „\ if < cond > then & block([], < inst1 >, \ldots, < instN >) \quad " \\ else & block([], < inst'1 >, \ldots, < inst'N' >) \end{array}$$

$$\begin{array}{rl} if < cond > then & block([], < inst1 >, \ldots, < instN >) \\ " else\ if < cond' > then & block([], < inst'1 >, \ldots, < inst'N' >) \quad " \\ else & block([], < inst''1 >, \ldots, < inst''N'' >) \end{array}$$

et on peut empiler autant de cas qu'on le souhaite.

- C'est l'unique instruction de condition qui est fournie, mais on peut construire (cf. infra) avec une fonction qui rend plus simple la manipulation de cas multiples.

Boucles

▶ Une boucle est une instruction qui permet de répéter un nombre défini ou indéfini de fois une opération.

▶ *"while"* : une syntaxe parmi celles qui sont valides est

$$"while < cond > do\ block([], < inst1 >, \ldots, < instN >)"$$

le nombre de répétition n'est pas nécessairement défini comme par exemple pour

$$"while\ not(random(10) = 2)\ do\ block([], print("ok"))"$$

▶ *"for"* : une syntaxe possible est

$$"fori : 1 thru N do block([], < inst1 >, \ldots, < instN >)"$$

Là le nombre de répétition est défini, c'est N.

Exemple : fabrication du 'case' lisp ou 'Piecewise' Mathematica

- La fonction utilise *"if"* et *"while"*, c'est

```
case(l,sinon):=block([n:0,N:length(l),res],
 res:while (n<N) do
 block([],n:n+1,if is(l[n][1]) then return(l[n][2])),
 if res=done then sinon else res)
```

Cette fonction permet par exemple de définir une expression donnée par morceau comme la fonction chapeau :

$$\text{"}case([[x<-1,0],[x<0,1+x],[x<1,1-x]],0)\text{"}$$

mais elle peut également être utilisée dans bien d'autres situations.

- Un objectif de l'enseignement est de faciliter le passage à l'état mental où la fabrication de telles fonctions va de soi.

Le mapping et les lambda-expressions

▶ Une fonction des plus utiles dans la pratique est "*map*" qui retourne la liste des valeurs d'une fonction pour les arguments correspondant aux éléments d'une liste.

$$\text{"}map(f, [a, b, c]) \longrightarrow [f(a), f(b), f(c)]\text{"}$$

▶ Comme cela a été dit une fonction est l'association d'un nom à un processus effectuant une suite d'instructions ; mais (notamment) dans le mapping il est pratique de disposer du processus sans lui donner de nom : ça s'appelle une lambda-expression et ça a la syntaxe

$$\text{"}lambda([arguments], instruction)\text{"}$$

Par exemple "*map*(*lambda*([*u*], *u*^2), [1, 2, 3])" $\longrightarrow$"[1, 4, 9]"

Echantillon de ce que le mapping (et lambda-expressions) peuvent faire

▶ Ça n'a l'air de rien comme cela, mais le mapping permet de réaliser des opérations qui seraient plus complexes sans lui.

▶ Par exemple la matrice correspondant au produit tensoriel d'un vecteur "[b1, b2, b3]" se fait par

"map(lambda([u], map(lambda([v], u * v), b)), b))"

qui renvoie

"[[b1^2, b1 b2, b1 b3], [b2 b1, b2 b2, b2 b3], [b3 b1, b3 b2, b3^2]]"

▶ L'alternative avec des boules "for" ou "while" est évidemment possible mais c'est beaucoup plus lourd !

La récursivité

▶ Les fonctions de Maxima peuvent être écrites sous forme récursive : c'est à dire qu'elles peuvent s'appeler elles-même, évidemment pas systématiquement parce que sinon ça ne s'arrêterait pas.

▶ Par exemple la recherche récursive du zéro d'une expression par la méthode du point fixe s'écrit

```
point_fixe(f,x,x0,eps):=block([fx0,res],
 fx0:subst(x0,x,f),
 if (abs(fx0-x0) > eps) then
  res:point_fixe(f,x,fx0,eps)
 else res:fx0,
 res);
```

Au travail !

- L'essentiel a été dit, les détails suivent...

- C'est à dire qu'il faut maintenant maîtriser pleinement la syntaxe de Maxima à l'aide de `Tutorat_Maxima.mac` et de la documentation figurant dans les liens.

- Puis s'attaquer aux problèmes qui font l'objet des leçons de l'enseignement.