TP 0 : Présentation de Xcas

1 Fonctionnalités utiles dans Xcas

Xcas possède déjà les éléments suivants :

1. Les opérations usuelles : addition, multiplication, etc., les sommes et les produits (finis ou non) (`sum` et `product`), la racine carrée (`sqrt`), la factorielle (`!`).
2. Quelques constantes universelles : π (`pi`), e (`e`), i (`i`), l'infini (`infinity`), l'infini signé (`inf` ou `-inf`), etc.. Et on prendra garde à ne pas utiliser la lettre i pour des variables.
3. Des fonctions usuelles :
   (a) fonctions liées au réels : arrondi au n-ème chiffre après la virgule (`evalf(t,n)`), signe (`sign`), minimum (`min`) et maximum (`max`), partie entière (`floor`) et fractionnaire (`frac`). ;
   (b) fonctions liées au complexes : partie réelle (`re`) et imaginaire (`im`), argument (`arg`), valeur absolue ou module (`abs`), conjugué complexe (`conj`) ;
   (c) fonctions logarithme et exponentielle : exponentielle (`exp`), logarithme népérien (`log` ou `ln`), logarithme en base 10 (`log10`) ;
   (d) fonctions circulaire, hyperboliques, et leurs réciproques : `cos`, `acos`, `cosh` et `acosh` (et pareil pour les fonctions sin et tan) ;
   (e) fonctions d'origine arithmétique : a modulo p (`a mod p` ou `a%p`), reste et quotient de la division euclidienne (`irem` et `iquo`), plus grand commun diviseur et plus petit commun multiple (`gcd` et `lcm`), primalité d'un entier (`is_prime`).

2 Objets représentés dans Xcas

2.1 Les nombres

On peut choisir de représenter les nombres de manière exacte ou approchée. Les notations exactes se font en général à l'aide de rationnels et des constantes prédéfinies. Les fonctions `evalf` et `exact` permettent de passer d'une représentation à l'autre. Plus précisément, la commande `evalf(t,n)` donne l'approximation de t à n chiffres près. Si l'on veut changer la précision par défaut de Xcas (qui est normalement de 14 chiffres), on utilise la commande `Digits`.

Certaines expressions exactes sont plus simples que d'autres. La fonction `simplify` permet en général d'avoir une expression plus simple, tout en conservant une expression exacte.

Exemple : simplifiez les expressions suivantes et en donner une valeur approchée

$$\sqrt{3+2\sqrt{2}}, \qquad \frac{1+\sqrt{2}}{1+2\sqrt{2}}, \qquad e^{i\pi/6}, \qquad 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right).$$

Remarque : on prendra garde au fait que les expressions apparemment simples peuvent être compliquées par une mauvaise écriture. Et il est difficile pour Xcas de comparer différentes écritures d'une même quantité. On peut le voir en faisant les commandes suivantes :

```
sqrt(2)+1-sqrt(2)
sqrt(2)-sqrt(2)+1
simplify(sqrt(2)+1-sqrt(2)
1==(sqrt(2)+1-sqrt(2))
1==simplify(sqrt(2)+1-sqrt(2))
```

2.2 Les variables

Xcas considère que toutes les variables sont initialement formelles : elles n'ont pas de valeur, mais on peut les manipuler. On peut les affecter, c'est-à-dire leur assigner une valeur, qui peut être elle-même une variable formelle. L'affectation se fait avec la commande `:=`. Pour revenir à une variable formelle, on utilise la commande `purge`.

Exemple : effectuer les commandes suivantes :

```
a==b
a:=b
a==b
a
b
b:=2
a
b
purge(a)
a
a:=b
b:=3
a
b
```

Remarque : On fera bien attention à distinguer `:=` (qui sert à l'affectation), `==` (qui rend une valeur de vérité d'une égalité) et `=` (qui sert à écrire une égalité, par exemple une équation).

On peut rajouter des contraintes à une variable formelle, grâce à la commande `assume`, qui écrase une affectation déjà existante. Moralement, la commande `:=` affecte une variable avec un signe $=$, tandis que la commande `assume` l'affecte avec un signe $\in$. Ceci permet de simplifier certaines expressions.

Exemple : effectuer les commandes suivantes :

```
a-abs(a)
assume(a>0)
a-abs(a)
simplify(a-abs(a))
simplify(atan(a)+atan(1/a)
assume(a<0)
simplify(atan(a)+atan(1/a)
```

2.3 Les expressions

Les expressions sont des combinaisons de nombres et de variables (formelles ou non). Une variable ayant été affectée sera remplacée par sa valeur dans l'expression où elle apparaît.

Exemple : effectuer les commandes suivantes :

```
purge(a); purge(b);
a^2+sqrt(b)+12*a-4
a:=2; b:=2*a^2;
a^2+sqrt(b)+12*a-4
simplify(a^2+sqrt(b)+12*a-4)
```

Xcas sait simplifier ou développer des expressions. Il sait aussi donner des formes standardisées dans certaines cas particuliers d'expressions :

- `expand` développe une expression (avec la distributivité de la multiplication sur l'addition) ;
- `normal` simplifie une fraction rationnelle sous forme de fraction irréductible sous forme développée ;
- `factor` factorise les polynômes, et réécrit une fraction rationnelle sous forme factorisée ;
- `partfrac` donne la décomposition d'une fraction rationnelle en éléments simples ;
- `simplify` essaie de donner une forme plus simple à une expression ;
- `convert` transforme une expression sous un format spécifique.

Exemple : effectuer les commandes suivantes :

```
expand((x+1)^3*(x-3)^4)
normal(((x-1)^5*(x-2)^2)/(x^5+4*x^4-11*x^3-26*x^2+64*x-32))
factor(x^5+4*x^4-11*x^3-26*x^2+64*x-32)
factor(((x-1)^5*(x-2)^2)/(x^5+4*x^4-11*x^3-26*x^2+64*x-32))
factor(x^2-a);
purge(b); a:=b^2; factor(x^2-a)
partfrac(((x-1)^5*(x-2)^2)/(x^5+4*x^4-11*x^3-26*x^2+64*x-32))
convert(exp(i*x),sincos)
convert(1/(x^4-1),partfrac)
```

2.4 Les fonctions

Plutôt que d'écrire une expression faisant intervenir une variable formelle, qu'on évalue après avoir affecté notre variable, on peut directement voir notre expression comme une fonction en les variables formelles qu'elle fait intervenir. La fonction f, qui associe à x la quantité $x \cdot \exp(x)$ peut par exemple être définie de l'une des manières suivantes :

— `f(x):=x*exp(x)`
— `f:=x->x*exp(x)`
— `f:=unapply(x*exp(x),x)`

Une fonction peut avoir plusieurs variables. Par exemple, on peut définir la fonction $h : (r, t) \mapsto r \cdot \exp(t)$ par la commande :

```
h(r,t):=r*exp(t)
```

et une fonction peut avoir des valeurs dans des espaces de dimensions arbitrairement grandes.

Certaines fonctions ont des expressions un peu plus complexes, et sont ce qu'on appelle des "fonctions non algébriques". Auquel cas, on peut faire différentes manipulations de quantités locales, et il faut spécifier l'image de la fonction avec la commande `retourne`. On donne plus loin quelques exemple de telles fonctions.

On peut facilement passer d'une fonction à deux variables $h(r, t)$ à la fonction en une variable $k(t)$, qui prend r comme paramètre, ce qui se fait de la manière suivante :

```
k(t):=unapply(h(r,t),r)
```

et on peut effectuer la commande Xcas : `h(r,t)==k(t)(r)` pour vérifier que tout va bien.

Exemple : définir la fonction f dans Xcas qui à deux réels a, b rend le quadruplet formé de leur somme, leur différence, leur maximum et leur minimum.

Dans le cas de fonctions définies sur $\mathbb{R}$ et à valeurs réelles, il est possible de calculer la dérivée de f, ce qui se fait de l'une des manières suivantes :

— `g:=f'`
— `g:=function_diff(f)`
— `g:=unapply(diff(f(x),x),x)`

On peut aussi avoir accès aux développements limités ou asymptotiques de fonctions, à l'aide de la commande `series` ou `taylor`, ou à la limite en un point, avec la fonction `limit`.

Exemple : définir la fonction $f : x \mapsto \frac{\sin(x)}{x} \cdot \exp(x)$. Calculer sa dérivée, puis son développement limité à l'ordre 6 en 0 et sa limite en $-\infty$.

On peut également définir la primitive d'une fonction, avec la commande `int`, ce qui permet ainsi de faire des calculs, au moins approchés, de certaines intégrales.

Exemple : effectuer les commandes suivantes :

```
f(x):=x*exp(-x);
g:=int(f)
limit(g(x),x=inf)-g(0)
g:=unapply(int(sin(x)*exp(-x),x),x)
g(2*pi)-g(0)
```

Enfin, on peut faire des composées de fonctions. La composée $f \circ g$ se note `f@g` tandis que la composée f^n se note `f@@n`.

2.5 Listes et séquences

Il sera parfois intéressant, quand on fait varier une quantité de manière itérative, de garder stockées les différents résultats intermédiaires (ne serait-ce que pour avoir une idée du comportement convergent ou divergent de la suite associée à ces quantités).

On écrit une séquence "à la main" (entre des parenthèse), ou avec la commande `seq`. On écrit une liste "à la main" (entre des crochets) ou avec la commande `makelist`. On passe d'une séquence à une liste en la mettant entre crochets, et d'une séquence à une liste avec la commande `op`.

Le nombre d'éléments d'une liste ou d'une séquence est donné par la commande `size` ou `nops`, et le i-ème terme de la liste ou de la séquence l est donné par `l[i-1]` (on prendra garde au fait que les éléments d'une liste de taille n sont numérotés de 0 à $n-1$). La séquence à 0 éléments est notée `NULL`, et la liste à 0 éléments est notée `[ ]`.

Pour ajouter un élément à une séquence, il suffit de l'écrire précédé de la séquence et d'une virgule, tandis qu'il faut utiliser la commande `append` pour ajouter un élément à une liste.

Les fonctions `sum`, `product`, `min` et `max` peuvent être directement appliquée à une liste. Pour appliquer une fonction aux éléments d'une liste, on utilise les commandes `apply` ou `map`. Enfin, on peut passer d'un polynôme à la liste de ses coefficients (et inversement) avec les commandes `poly2symb` et `symb2poly`.

Exemple : effectuer les commandes suivantes :

```
s:=(0,1,0,2,0,4);
l:=[s];
nops(s);
s[4];
l[5];
l2:=makelist(x->2^x,2,4,1/2);
s2:=seq(2^(x/2),x=4..8);
[s2]==l2;
apply(x->(ln(x)/ln(2)),l2)==[seq(x/2,x=4..8)];
evalf(simplify(apply(x->(ln(x)/ln(2)),l2))-[seq(x/2,x=4..8)]);
l1:=append(l,l2);
l1:=l; (l1:=append(l1,l2[j]))$(j=0..3);
l1:=l1.append(l2[4]);
```

```
poly2sym(l)
expand(poly2symb(l))
```

3 Éléments de programmation

3.1 Les assertions

On a vu lors des différents exemples qu'il fallait bien prendre garde lorsque l'on utilise le signe `==`, qui sert à comparer deux quantités. Pour faire court, disons simplement que, **si** Xcas affiche `True` lorsqu'on lui fait évaluer `a==b`, **alors** les quantités a et b sont égales, mais on n'a pas d'équivalence ! Le problème vient de la pluralité des écritures d'une même quantité : on peut avoir deux expressions qui sont mathématiquement équivalentes, mais dont les représentations en machine diffèrent.

Pour ne pas trop compliquer nos situations, on pourra dire en général que deux quantités a et b sont égales si, et seulement si, Xcas renvoie `True` quand on lui demande d'évaluer : `simplify(a-b)==0`.

Il faudra tout de même faire attentions au sens de a et b. Il y a certaines quantités que Xcas ne saura pas comparer. Par exemple, Xcas ne sait pas comparer deux fonctions, mais sait comparer les expressions qui leur sont associées. De même, on ne peut pas comparer des séquences, mais il est possible de comparer les listes associées.

Exemple : On considère les fonctions $f : x \mapsto (x+1)^3$ et $g : x \mapsto x^3 + 3 \cdot (x^2 + x) + 1$. Montrer avec Xcas que ces fonctions sont égales.

À part l'égalité (représentée dans Xcas par `==`), les autres assertions qui nous seront particulièrement utiles sont celles liées aux inégalités larges ou strictes $<, \leq, >, \geq$, représentées dans Xcas par `<`,`<=`,`>`,`>=`.

On a déjà vu un intérêt de ces assertions, avec la commande `assume`. Un autre intérêt réside dans l'utilisation d'alternative. L'idée est de donner à Xcas une commande de la forme :

si on a la condition 1*, alors faire l'action A, sinon faire l'action B*

où l'action B est optionnelle.

Les commandes Xcas associées à cette procédure sont les suivantes, selon que l'on a ou pas une action B :

— `si <cond1> alors <actA> fsi;`

— `si <cond1> alors <actA> sinon <actB> fsi;`

Exemple : décrire simplement la fonction suivante :

```
f(x):= si x>=0 alors x sinon -x fsi
```

3.2 Les boucles

Un des intérêt de l'utilisation de l'ordinateur est pour les calculs simples mais répétitifs. Par exemple, si on souhaite répéter de très nombreuses fois une tâche assez élémentaire.

Il y a essentiellement deux manière de répéter une tâche pour un ordinateur :

— avec des boucle "pour" : si l'on sait exactement combien d'itérations sont nécessaires ;

— avec des boucles "tant que" : si on s'arrête seulement lorsqu'une condition est remplie.

Le point important derrière est que notre programme doit s'arrêter à un moment. Et il faudra s'assurer que c'est bien le cas lorsque l'on utilise ce type de boucles (et on pourra d'ailleurs s'interroger sur le nombre d'étapes avant la fin du programme, ce qui rejoint la notion de "complexité abordée dans le paragraphe suivant).

Les commandes Xcas associées aux boucles

— *pour j allant de a à b par sauts de p faire l'action A* ;

— *tant qu'on a la condition 1 faire l'action A ;*

sont les suivantes :

— `pour j de a jusque b pas p faire <actA> fpour;`

— `tantque <cond> faire <act> ftantque;`

Il n'est pas nécessaire de définir le "saut" dans la boucle "pour" (auquel cas il est par défaut de +1). De plus, l'action A peut ou non dépendre de j.

Exemple : On définit la suite de Syracuse de paramètre a, pour $a \in \mathbb{N}$, comme étant la suite définie par $u_0 = a$ et : $u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3 \cdot u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$.

Définir dans Xcas la fonction f telle que $u_{n+1} = f(u_n)$. Et définir la fonction g qui à a associe le plus petit indice n tel que $u_n = 1$. On pourra avoir recours à la commande `retourne`. Et en déduire la plus petite valeur de a pour laquelle cet indice est d'au moins 100, et donner la plus grande valeur prise par les termes de la suite associée.

Au passage, la justification pour dire que la boucle "tant que" se termine est lié à la conjecture de Syracuse. Elle n'est qu'à l'état conjecturel à l'heure actuelle, mais personne n'a encore trouvé de valeur de a qui l'invaliderait pour le moment.

Exemple : Définir la fonction qui à un entier n associe la liste des n premiers nombres premiers. On pourra utiliser la commande `is_prime`, ou alors définir une fonction qui dit si un nombre m est premier ou non.

3.3 Programmation efficace

Il existe de nombreux problèmes liés au calcul formel. On a déjà vu qu'il faut faire attention aux expressions que l'on manipule (notamment avec des `==`). C'est un problème qui est cependant lié au langage de programmation.

Un problème qui n'est pas lié au langage de programmation est ce que l'on appelle la "complexité algorithmique". Celle-ci augmente lorsque l'on utilise beaucoup de mémoire (lorsque l'on conserve des résultats intermédiaires), et que l'on augmente le nombre de calculs.

On peut reprendre les deux derniers exemples de fonction à implémenter :

— pour la suite de Syracuse : la mémoire nécessaire ne va pas beaucoup grandir (comme on n'a pas besoin d'avoir à tout moment tous les termes de la suite jusqu'à un certain

rang), mais le nombre de fois où l'on utilise la fonction f grandit avec a. Donc la complexité augmente avec a ;
— pour la liste des nombres premiers : aussi bien la mémoire que le nombre d'opération augmentent avec n, donc la complexité grandit avec n.

Dans les deux cas, on peut voir concrètement comment se traduit l'augmentation de la complexité, avec le temps de calcul de nos fonctions. La commande `time` permet d'avoir le temps de calcul.

Exemple : comparez les programmes suivants, en terme de temps de calcul, pour calculer le n-ème terme de la suite de Fibonacci. Et dire ensuite comment grandissent avec n la mémoire allouée, ainsi que le nombre d'opérations, pour chacun des cas.

```
f1(n):= si n==1 ou n==2 alors 1 sinon f1(n-1)+f1(n-2) fsi
```

```
f2(n):= {
   local a:=[1,1];
      pour j de 1 jusque n-2 faire
         a:=append(a,a[nops(a)-2]+a[nops(a)-1]);
         fpour
   retourne a[n-1];
}
```

```
f3(n):= {
   local a,b,j;
   a:=1;b:=1;
   pour j de 1 jusque n-2 faire
      a,b:=b,a+b;
   fpour
   retourne b;
}
```

3.4 Un peu d'aide à la programmation

Xcas possède quelques éléments pour aider à programmer facilement.

Dans Xcas desktop, lorsque l'on fait `Prg>Nouveau programme`, on a directement accès aux syntaxes pour créer une fonction, un test, ou une boucle, grâce aux boutons `Fonction`, `Test` et `Boucle`. Il y a aussi une documentation qui permet de retrouver les syntaxes.

Pour chercher un erreur dans un programme, on peut utiliser la commande `debug` qui permet d'exécuter un programme Xcas pas a pas en visualisant les variables. Cela permet de voir à quel endroit une fonction ne fait pas ce que l'on voudrait.

Exemple : reprendre la fonction `f3` définie précédemment, et entrer la commande `debug(f3(6))` pour voir comment à chaque itération évoluent les quantités a et b.