INTRODUCTION RAPIDE À MAPLE

par

Adrien Brochier

1. Introduction

Maple est un logiciel propriétaire de calcul formel édité par la société Maplesoft. Le terme *formel* s'oppose à *numérique* et indique que Maple est capable de manipuler des expressions mathématiques symboliquement, et donc de faire du calcul exact. Maple possède toutefois quelques fonctions qui permettent de faire du calcul numérique, c'est-à-dire de travailler avec des valeurs approchées.

D'une façon générale, la résolution d'un problème mathématique à l'aide d'un logiciel passe par deux étapes. La règle d'or est la suivante : *Maple est stupide, il ne « comprend » rien aux maths, il se contente d'exécuter bêtement les instructions données par l'utilisateur.* Il est donc nécessaire

(1) d'écrire une procédure générale qui résout le problème de façon complètement mécanique, en écrivant pas à pas chacune des étapes. Une telle procédure s'appelle un algorithme, c'est une recette qui détaille suffisamment la solution pur qu'elle puisse être appliquée sans réfléchir

(2) de traduire cette solution dans le langage du logiciel utilisé.

C'est évidemment la partie (1) qui demande réellement du travail et qui fait appel à un raisonnement mathématique. Une fois celle-ci effectuée il est relativement simple de traduire l'algorithme dans le langage de n'importe quel logiciel approprié. Ainsi, même si la suite de ce document présente la syntaxe de Maple, il faut garder à l'esprit que les concepts abordés sont des concepts généraux de programmation et d'algorithmique. Vous connaissez déjà des algorithmes : la division euclidienne, par exemple, ou la résolution des équations du second degré qui peut se décrire comme suit :

On se donne : $ax^2 + bx + c$
$\Delta := b^2 - 4ac$
si $\Delta > 0$ **alors**
    les solutions sont $\frac{-b \pm \sqrt{\Delta}}{2a}$

```
sinon
    si Δ = 0
        l'unique solution est -b/a
    sinon
        pas de solutions
    fin si
fin si
```

On dispose donc d'une liste d'instructions précises et sans ambiguïtés à suivre qui donnent la solution à coup sûr. On voit apparaître dans cet algorithme :

- les paramètres du problème (a, b, c)
- des *variables* qui permettent de stocker des valeurs (Δ)
- des *structures de contrôle* qui comme leur nom l'indiquent permettent de contrôler l'exécution de l'algorithme (les expression si...alors)

À cela s'ajoute en général des procédures qui sont déjà programmée dans Maple, ce qui évite de tout reprogrammer de zéro. Maple possède en effet une large gamme de fonctionnalités : fonctions mathématiques de base, routines d'algèbre linéaires, procédures pour tracer des graphiques, etc... C'est évidemment ce qui le rend plus intéressant qu'un langage de programmation classique, il "sait" déjà faire beaucoup de choses en maths.

Remarque 1.1. — Il existe de très bonnes alternatives libres et gratuites à Maple. Citons notamment SAGE et Maxima.

1.1. Comment obtenir de l'aide. — Toutes les commandes et procédures Maple sont largement documentées. Pour accéder à cette documentation, on peut soit ouvrir l'aide par le menu approprié ou en tapant ctrl+F1 puis faire une recherche, soit sélectionner une instruction dans la feuille de calcul et appuyer sur F2 pour accéder directement à la documentation de l'instruction en question.

2. Syntaxe de base

Toute séquence d'instructions doit terminer par " ;" (qui affiche le résultat) ou ":" (qui ne l'affiche pas). On peut utiliser Maple comme simple calculatrice, par exemple taper "2+2 ;" affichera 4.

L'affectation de variable se fait grâce à " :=".

```
>a:=2;
      2
>a^2;
      4
>a:=a+5;
      7
>a^2;
      49
```

Maple connait un grand nombre de fonctions mathématiques de base, comme sin, cos, exp, mais aussi abs (valeur absolue) floor (partie entière) et beaucoup d'autres. Les racines sont obtenues grâce à la commande "sqrt". Puisque Maple est un logiciel de calcul formel, il ne cherchera pas à évaluer ces expressions, et il est possible de le forcer à simplifier une expression grâce à la commande simplify :

```
>a:=sqrt(2);
```
$$\sqrt{2}$$
```
>(1-a)*(1+a);
```
$$(1-\sqrt{2})(1+\sqrt{2})$$
```
>simplify(%);
```
$$-1$$

On notera l'emploi de "%" qui est une variable définie automatiquement et qui contient le résultat de la dernière expression entrée dans la feuille de calcul. Il existe aussi deux procédure factor et expand qui permettent de factoriser ou développer une expression. Une variable ne contient pas forcement un nombre. Elle peut contenir une expression

```
>expr:=x+y-z:
>3*expr+2;
```
$$3x+3y-3z+2$$

et peut même contenir une équation. Attention : dans l'équation il s'agit bien d'un "=" et non d'un " :=". Ceci permet d'introduire la commande "solve" de Maple, qui comme son nom l'indique sait résoudre certaine équations de façon exacte

```
>eq:=(x^2-2=0):
>solve(eq,x)
```
$$[x=\sqrt{2}, x=-\sqrt{2}]$$

Maple connait aussi quelque constantes comme Pi (π), I (le i des nombres complexes) et e (e la base du logarithme népérien). Il connait aussi des opérateurs de comparaisons : en plus de "=" il y a <,>,<=,>= pour inférieur, supérieur, inférieur ou égal, supérieur ou égal respectivement. Maple connait aussi des procédures d'évaluations, citons par exemple evalb (évaluation booléenne, qui évalue une expression logique) et plus important evalf qui donne une approximation numérique d'une expression.

```
>sin(Pi/3);
```
$$\frac{\sqrt{3}}{2}$$
```
>evalb(cos(Pi)>0);
    false
>evalf(sqrt(2));
    1.414213562
```

3. Définition de fonctions

La syntaxe générale de définition de fonction est : paramètre(s) -> expression. Comme tout objet Maple, une fonction peut être stockée dans une variable pour la réutiliser. La syntaxe est ensuite la même qu'en maths :

```
>f:=x->x^2+1:
>f(3);
      10
>f(1+1);
      5
```

Maple sait dériver les fonctions :

```
>f:=x->x^2+1:
>diff(f(x),x);
      2x
```

Il sait aussi calculer des intégrales et primitives simples :

```
>f:=x->x^2+1:
> int(f(x),x=1..2);
       10/3
> int(f(x),x);
      1/3x^3 + x
```

4. Listes et tableaux

Comme leur nom l'indique, les listes permettent de stocker des listes d'éléments. Par exemple, si une équation possède plusieurs solutions, la commande solve que nous avons déjà vue renvoie la liste des solutions. Maple permet de déclarer des suites :

```
>s:=1,2,3,4;
      s := 1, 2, 3, 4
```

On obtient une liste en mettant une suite entre crochet, soit directement soit à partir d'une suite existante :

```
>s:=1,2,3,4;
>maListe:=[s]:
>maliste:=[1,2,3,4]:
```

On peut générer des suites grâce a la commande seq, qu'il faut encadrer de crochets si on veut en faire une liste :

```
>uneListe:=[seq(k^2,k=1..10)];
      uneListe := [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

On peut accéder à ou modifier un élément d'une liste en mettant son indice entre crochets :

```
>uneListe[2];
      4
>uneListe[1]:=5:
>uneListe;
      [5, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

On récupère le nombre d'éléments d'une liste grâce à la commande nops. On peut faire des listes de n'importe quoi, y compris de fonctions par exemple :

```
>L:=[x->x^2, x->x^3]:
>L[1](5);
      25
```

Les tableaux sont des listes qui peuvent avoir plusieurs dimensions. On peut construire un tableau soit en donnant directement des valeurs à ses cases, soit en le déclarant au préalable

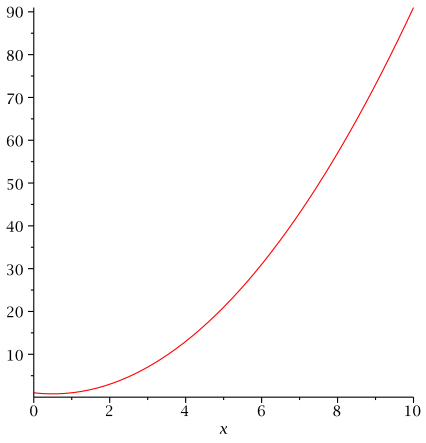
```
>Tab:=array(1..2,1..4):
>Tab[1,2]:=2:
```

5. Graphiques

Maple possède une procédure permettant d'afficher des graphiques. Celle-ci possède de très nombreuses options, il faudra donc se référer à la documentation pour une liste précise. Cette commande permet de tracer soit le graphe d'une fonction, soit un graphe donné par une liste de couples de points :

```
> f:=x-> x^2-x+1:
> plot(f(x), x = 0 .. 10);
```

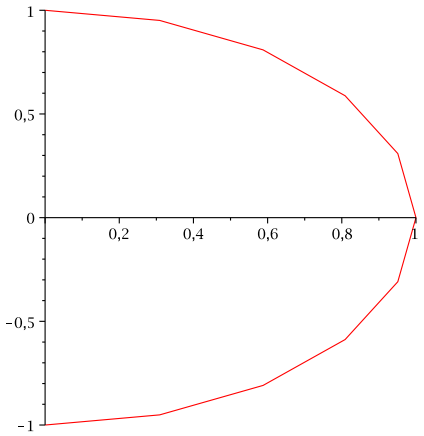
le résultat :



et

```
>Liste:=[seq([sin(k*Pi/10),cos(k*Pi/10)],k=1..10)]:
>plot(Liste);
```

le résultat :



Enfin, la commande display permet de regrouper des graphiques. Elle n'est pas disponible par défaut, puisque Maple ne charge que les commandes les plus courantes, il est donc fréquent de devoir lui demander de charger des bibliothèques spécifiques via la commande "with". On stocke ensuite les graphiques dans des variables, comme n'importe quel objet Maple, en ajoutant une option de couleur pour distinguer les graphiques :

```
>with(plots):
> f :=x-> x^2-x+1:
>Graph1:= plot(f(x), x = 0 .. 1):
>Liste:=[seq([sin(k*Pi/10),cos(k*Pi/10)],k=1..10)]:
>Graph2:=plot(Liste , color=blue):
>display(Graph1,Graph2);
```

6. Procédures et structures de contrôle

Une fois connue ces quelques instructions élémentaires, on voudrait pouvoir les combiner pour écrire de vrais programmes. C'est le rôle des procédures. La syntaxe générale de déclaration d'une procédure est :

```
>nomDeLaProcedure:=proc(param1,param2,...)
(Instructions)
end proc:
```

Remarque 6.1. — Attention, en principe la déclaration d'une procédure est considérée par Maple comme une seule expression, et donc tout devrait être tapé sur la

même ligne sous peine de déclencher une erreur. Pour améliorer la lisibilité, il est préférable de passer à la ligne, ce qui est possible en utilisant shift+entrée.

Remarque 6.2. — La valeur renvoyée par la procédure est celle de la dernière expression executée par la procédure. Renvoyée signifie ici que le résultat sera affiché (pour peu qu'on ait bien mis un ";" à la fin de cette ligne), et qu'en faisant

```
uneVariable:=nomDeLaProcedure;
```

le résultat de l'expression de la dernière ligne sera stockée dans la variable "uneVariable".

Un exemple idiot :

```
>double:=proc(x)
2*x;
end proc:
>double(3):
          6
>unNombre:=double(12):
>unNombre;
          24
```

On obtiendrait évidemment le même résultat en définissant la fonction "f :=x-> 2*x;". Un programme devient intéressant quand on peut contrôler la manière dont il se déroule suivant les situations qui se présentent. On dispose de deux types de structures : les tests conditionnels, et les boucles.

Les tests conditionnels correspondent au "si...alors". Sans surprise, la syntaxe Maple qui correspond est "if... then", assorti éventuellement d'un "else" ("sinon" en anglais) :

```
>if (test logique) then
(instructions)
fi;
```

Le test logique est une expression qui renvoie "vrai" ou "faux", par exemple en comparant deux valeurs, et le bloc "instructions" n'est exécutée que si l'expression renvoie "vrai". On peut prévoir une alternative si le résultat est "faux" :

```
>if (test logique) then
(instructions1)
else
(instructions2)
fi;
```

Ceci permet donc d'adapter l'exécution du programme en séparant des cas. Un exemple simple qui retourne le maximum de deux nombres :

```
>monMax:=proc(x,y)
if (x>y) then
    x;
```

```
else
    y;
fi;
end proc:
>monMax(5,7);
          7
```

Comme exemple à peine plus élaboré, on peut prendre l'algorithme présenté au début de ce document. Le travail mathématique est déjà fait il n'y a plus qu'à traduire :

```
>secondDeg:=proc(a,b,c)
delta:=b^2-4*a*c:
if (delta >0) then
    Soluce:=[(-b-sqrt(delta))/(2*a), (-b+sqrt(delta))/(2*a)];
else
    if (delta=0) then
        Soluce:=[-b/(2*a)];
    else
        Soluce:=[];
    fi;
fi;
Soluce;
end proc:
> expand((x-3)*(x-2));
                        x^2 - 5x + 6
> secondDeg(1, -5, 6);
                                              [2, 3]
> expand((x-2)^2);
                       x^2 - 4x + 4
> secondDeg(1, -4, 4);
                                                [2]
>expand((x-I)*(x+I));
                       x^2 + 1
> secondDeg(1, 0, 1);
                                                []
```

On a utilisé la commande expand() pour fabriquer des polynômes dont on connait les racines, notre programme fonctionne bien. Le dernier admet comme racine i et $-i$, et n'a donc pas de racine réelles. C'est une bonne habitude à prendre que de retourner quelque chose (ici une liste vide) même quand il n'y a pas de solutions. Ainsi le type d'objet retourné est toujours le même (ici une liste) quelle que soit la situation, ce qui évite des problèmes à l'utilisation du programme. Mélanger plusieurs types de valeurs de retour est souvent dangereux. On préfère aussi, pour la lisibilité du programme,

stocker le résultat dans une variable, et forcer l'affichage de celle-ci à la fin, juste avant le "end proc".

La deuxième notion importante pour les procédures est celle de *boucle*. Une boucle est un moyen de répéter une série d'instructions, soit un nombre fixé de fois, soit sur la base d'une condition logique. Dans le premier cas la syntaxe est :

```
>for (variable) from (valeur de depart)
      by (pas de la boucle) to (valeur d'arrivee) do
(instructions)
od:
```

Si la valeur de départ ou le pas ne sont pas précisé, ils seront pris par défaut égaux à 1. En fait, Maple est capable de créer des boucles en parcourant une liste quelconque grâce à la syntaxe

```
>for (variable) in (liste) do
(instructions)
od:
```

la valeur de la variable peut bien sûr être utilisée dans la boucle. Il est possible d'imbriquer des boucles (par exemple pour parcourir un tableau à deux dimensions). Un exemple classique d'utilisation des boucles est le calcul des termes de la suite de Fibonacci, définie par

$$\begin{cases} f_1 = f_2 & = 1 \\ f_{n+2} & = f_{n+1} + f_n \end{cases}$$

La procédure correspondante :

```
>fibo:=proc(N)
f[1]:=1:
f[2]:=1:
for i from 3 to N do
f[i]:=f[i-1]+f[i-2]:
od:
[seq(f[k],k=1..N)];
end proc:
```

On calcule ici chacune des valeurs, et on récupère une liste qui contient la totalité de ces valeurs. Ceci nous permet d'illustrer la seconde forme de la boucle for, en calculant la somme des 10 premiers termes de la suite de Fibonacci :

```
>L:=fibo(10);
                L := [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
> S := 0;
                                        S := 0
> for elt in L do
S := S+elt:
od:
```

```
> S;
                143
```

Ici la variable elt parcourt la suite L elle-même, et prend donc successivement les valeurs 1,1,2,3 etc... Enfin, il existe une boucle "tant que" ("while" en anglais) qui comme son nom l'indique répète des instructions tant qu'une certaine condition logique est vraie. Elle est pratique lorsque on ne peut pas prévoir à l'avance le nombre d'itération nécessaires, par exemple quand on effectue la division euclidienne d'un entier a par un entier b. On doit soustraire b à a *tant que* c'est possible, c'est-à-dire tant que le reste est supérieur ou égal à b, et à chaque soustraction le quotient incrémente de 1 :

```
>euclide:=proc(a,b)
reste:=a:
quotient:=0;
while (reste >= b) do
reste:=reste-b:
quotient:=quotient+1:
od:
[quotient,reste];
end proc:
>euclide(287,19);
                [15, 2]
```

On vérifie :

```
>15*19+2;
                287
```

7. Conclusion

Nous n'avons abordé dans ce document que la syntaxe de base de Maple qui correspond au dénominateur commun de la plupart des logiciels de calcul formel. Pour savoir utiliser des fonctions spécifiques il est nécessaire de se reporter à la documentation très complète de Maple. On pourra aussi consulter les documents suivants disponibles sur la toile :

- Un support de cours intégré par Michel Mehrenberger : `http://www-irma.u-strasbg.fr/~mehrenbe/polymaple.pdf`
- Une introduction "libre" à Maple sur le site de la Wikiversité : `http://fr.wikiversity.org/wiki/Introduction_à_Maple`

1er octobre 2010

Adrien Brochier, IRMA (CNRS), Strasbourg • *E-mail :* `brochier@math.unistra.fr`