

# Introduction à Maple

par PLU Julien ([page perso](#))

Date de publication : 02 mai 2008

Dernière mise à jour :

Cet article s'adresse aux personnes désirant apprendre un premier langage se rapprochant beaucoup du langage algorithmique.

1 - Introduction.....	3
1-1 - Présentation de Maple.....	3
1-2 - Présentation de l'outil de développement.....	3
2 - Les types de base en Maple.....	4
2-1 - Les entiers relatifs $Z$ .....	4
2-2 - Les fractions rationnelles $Q$ .....	4
2-3 - Les réels $R$ .....	4
2-4 - Les chaînes de caractères.....	4
2-5 - Les booléens.....	4
2-6 - Divers exemple.....	5
3 - Structure de Maple.....	6
3-1 - Procédure et fonction.....	6
3-2 - Affectation et séquence.....	6
3-3 - Instruction conditionnelles.....	6
3-4 - Itérations.....	7
3-5 - Les tableaux.....	7
4 - Traduction des algorithmes du premier cours en Maple.....	9
4-1 - Multiplication.....	9
4-2 - PGCD de 2 entiers.....	9
4-3 - Minimum d'un tableau d'entier.....	9
4-4 - Recherche dans un tableau.....	9
5 - Débuggage d'un code Maple.....	10
5-1 - Gestion d'erreur classique.....	10
5-2 - Autres possibilités.....	10
6 - Conclusion générale.....	11
6-1 - Epilogue.....	11
6-2 - Remerciements.....	11

## 1 - Introduction


### 1-1 - Présentation de Maple

Il faut choisir un langage de programmation car on écrit nos algorithmes dans un langage abstrait et indépendamment de toute machine. On veut ensuite tester et exécuter. Il nous faut donc un environnement d'exécution, composé d'une machine réelle, un langage de programmation et les règles de traduction qui permettent de passer du langage algorithmique au langage de programmation.

Franchement Maple n'est pas un super langage de programmation, mais il n'est pas fait pour cela. Maple est un remarquable environnement de prototypage pour faire du calcul symbolique nous allons donc faire des compromis et diverses contorsions pour traduire les structures de notre langage.

On n'écrit pas directement les algorithmes dans un langage de programmation car le langage algorithmique s'affranchit des contraintes des langages. Dans un langage d'algorithmique on définit précisément la méthode, et les données employées pour résoudre un problème.

### 1-2 - Présentation de l'outil de développement

Maple est un outil payant vous pourrez le trouver à cette adresse  <http://www.maplesoft.com> pour ceux qui ne voudraient pas l'acheter il y a deux alternatives qui sont les logiciels Mathématica et Mupad, ces deux logiciels sont bien, mais Mupad est celui qui doit se rapprocher le plus de Maple. Si vous l'avez acheté vous pouvez le télécharger et ensuite l'installer en suivant les instructions qui s'affichent sur votre écran, c'est exactement pareil qu'une installation de tout autre logiciel. Une fois Maple installé il vous faut ouvrir le logiciel qui s'appelle Classic Worksheet Maple, qui est la partie de Maple avec laquelle on peut développer.

## 2 - Les types de base en Maple

### 2-1 - Les entiers relatifs Z

- Nom du type: integer
- Représentation du type: avec une taille bornée, mais grande (268 435 448)
- Opérations/fonctions: classiques et nombreuses, +, \*, -, mod en notation infixée, d'autres comme abs, iquo, irem... en notation préfixée et ! en notation suffixée.

### 2-2 - Les fractions rationnelles Q

- Nom du type: fraction
- Pas de type de base, Maple fait du calcul formel en opérant sur des classes d'équivalences définies sur  $Z \times N \setminus \{0\}$ . Comme en maths, 20/12 et 25/15 sont deux éléments de la même classe, dont le représentant canonique est 5/3
- Attention, 5/3 n'est pas le flottant il faut faire evalf(5/3)
- Attention, 9/3 est un entier !
- Attention, les opérations dans Q se font en précision infinie, au sens que, avec tous les chiffres permettant de représenter des entier. 2<sup>3</sup>/4!; renvoie: 1/3
- Les opérations sur les fractions sont classiques: +, -, /,...

### 2-3 - Les réels R

- Nom du type: float
- La représentation est classique: un chiffre et un exposant qui sont des entiers qui ont chacun une taille bornée (268 435 448 pour le chiffre et 2 147 483 646 pour l'exposant). La base de la représentation est 10. Exemple: 41.87 est un flottant dont le chiffre est 4187 et l'exposant -2. On le note aussi 0.4187e2. Attention, le domaine est une partie finie des décimaux! Ce n'est évidemment pas un intervalle de R.
- Les opérations sont également classiques: +, -, /, sin, exp,...
- Les fonctions opèrent d'un type numérique à l'autre: evalf transforme son argument de type integer ou fraction en float. Exemple: evalf(1/3); renvoie 0.3333333333... Ensuite floor, ceil, trunc, frac, round transforment leur argument flottant en un entier: partie entière classique, par valeur supérieure, troncature, partie fractionnaire, arrondi. Exemple: floor(-6.789); renvoie -7.

### 2-4 - Les chaînes de caractères

- Nom du type: string
- C'est une suite d'un maximum de 268 435 439 caractères, entourés de guillemets". Le caractère guillemet est obtenu par /. Exemple: cat("Mon nom est ", "Personne.") renvoie la chaîne de caractères "Mon nom est "Personne." qui a pour longueur 23.
- Les fonctions du type string sont cat la concaténation, substring l'extraction d'une sous-chaîne, length qui renvoie l'entier longueur de son argument.

### 2-5 - Les booléens

- Nom du type: boolean
- Les valeurs du type sont true et false
- Les opérations sont and, or, not. Les règles d'évaluation sont celles vues dans l'article sur l'algorithmique =, >,... Attention: Maple utilise les opérations de comparaison pour faire du calcul formel. Il évalue donc une

expression comme  $(2 < 3)$ : un booléen normal, dans le contexte des conditions des structures de contrôle (si  $(2 < 3)$  ..?. Une égalité, inégalité en dehors de ce contexte. Dans ce cas, on forcera l'évaluation comme boolean en utilisant la fonction evalb: trouve := evalb( $2 < 3$ );

## 2-6 - Divers exemple

Langage d'algorithmme	variable : type;	variable := expression;	a, b: entier;	superieur : boolean;	a := 3; b := 4;	superieur := (b > 9);
variable :: type;	variable := expression;	a::integer, b::integer;	superieur :: boolean;	a := 3; b := 4;	superieur := evalb(b > 9);	

### Remarque:

- Les symboles en Maple différencient majuscule et minuscule. "Trouve" n'est pas "trouve". Certains noms sont réservés par Maple car alloués à des primitives du langage: Pi, I, D,...
- Par défaut d'affectation initiale, toute variable est initialisée automatiquement par Maple à la constante symbolique qu'est son nom. Ceci permet de faire du calcul symbolique mais nous n'en parlerons pas ici.

## 3 - Structure de Maple

### 3-1 - Procédure et fonction

On traduit la notion d'algorithme par une procédure Maple. La syntaxe est:

```
proc( param1::type1, param2::type2,...)::typeRès ;
description "Chaîne de caractère1", "Chaîne2",...,"Chaîne p" ;
local var1::type1, var2::type2, ... ;
instr1;
instr2;
... ;
instrn;
end proc;
```

#### Syntaxe d'une procédure Maple:

- les terminateurs ; sont obligatoires
- description ...; est facultative. Elle permet d'afficher un texte d'explication (de spécification) lorsqu'on demande l'affichage de la procédure.
- La liste des paramètres, typés, est éventuellement vide.
- typeRès; est le type du résultat de la procédure. On peut omettre cette déclaration, mais dans ce cas, il faut aussi omettre le ; qui suit le type du résultat.
- La dernière instruction exécutée doit être une instruction de type return expression; Ce n'est pas forcément instrn. Toute exécution de l'instruction return expression; termine l'exécution de l'algorithme en renvoyant la valeur de expression.

Un cas particulier d'algorithme est celui d'une fonction: il n'y a pas de variables déclarées dans le corps, il n'y a pas de structuration des instructions. Le corps est réduit à return expression; On traduira ainsi la fonction de math classique:

f: x1 : type1,...,xn : typen -> expression par (var1::type1,var2::type2,...,varn::typen) -> expression

C'est réellement une abréviation de la procédure Maple:

```
proc(var1::type1, var2::type2, ...,varn::typen) return expression; end proc;
```

On remarquera que dans la syntaxe Maple d'une fonction on ne peut donner un type au résultat contrairement à une procédure.

### 3-2 - Affectation et séquence

```
a := expression;
instr1;...;instrn;
```

Se note en Maple:

```
a := expression;
instr1;...;instrn;
```

### 3-3 - Instruction conditionnelles

si Cond alors Instr1; ...; Instrn;	et	si Cond alors Instr1;...;Instrk; sinon Instp;...;Instrn;	et	si Cond1 alors Instr1;...; sinon si Cond2 alors Instr2;...;
---	----	---	----	--

```

fin si                                fin si                                sinon
                                     Inst3;...;
                                     fin si
    
```

Se note en Maple:

```

if Cond then                          if Cond then                          if Cond1 then
  Inst1;                               Inst1; ...; Instk;                    Inst1; ...;
  ...;                                 et                                     elif Cond2 then
  Instn;                               Instp; ...; Instn;                    Inst2; ...;
end if;                                end if;                                else
                                     Inst3; ...;
                                     end if;
    
```

### 3-4 - Itérations

```

pour v de e1 à e2 par pas de e3 faire  pour v de e1 à e2 faire                tant que
  Cond faire                            Inst;...;                               et
  Inst;...;                              et                                     Inst;...;                               et
fin pour                                fin pour                                fin tant
  que
    
```

Se note en Maple:

```

for v from e1 to e2 by e3 do           for v from e1 to e2 do                while
  Cond do                               Inst; ...;                             et
  Inst; ...;                             et                                     Inst; ...;                             et
end do;                                 en do;                                 end do;
    
```

### 3-5 - Les tableaux

Nous utiliserons une petite partie des nombreuses variantes de array Maple:

- Déclaration du tableau T par l'instruction T:=array(1..taille); où T est le nom du tableau, taille sa taille entière. Attention on ne peut typer les éléments d'un array Maple dans la déclaration. Puis la déclaration n'est pas la même pour une variable tableau locale à une procédure. Enfin la déclaration du type est possible pour un tableau paramètre d'une procédure.
- On calcule la taille d'un tableau en utilisant la fonction taille. Attention cette fonction n'est pas une fonction standard Maple, je vais être gentil je vous la donne:

```

kernelopts(assertlevel=2):
taille := proc(t)::integer;
if type(t,array) then
  return op([2,2],eval(t));
else
  error " : le paramètre n'est pas de type array"
end if;
end proc;
    
```

- On accède à l'élément de rang i par T[i]. Attention un élément non initialisé a quand même une valeur: le nom. Ainsi si on fait: T:=array(1..3); T[1]:=4; T[2]:=8; T[3] renvoie T3. Bien sûr, ceci n'a aucun sens algorithmiquement parlant.
- On peut déclarer et initialiser un tableau en une seule instruction: T:=array(1..4, [2,5,7,1]);

Attention, un tableau s'évalue à son nom ! Après T:=array(1..3); T[1]:=4; T[2]:=8; T; renvoie T alors que eval(T); renvoie [4,8,?3]. Donc on s'interdira S := T; car après cette instruction, d'une certaine façon, S, T sont deux noms pour un même tableau. Ainsi S:=T; S[1]:=0; eval(T); renvoie [0,8,?3].

Pour un tableau comme paramètre d'une procédure on typera le paramètre `NomParam::array(TypeElements)`, où `TypeElements` est le type commun des éléments du tableau paramètre.

Pour un tableau comme résultat d'une procédure le type du résultat est `array(TypeElements)`. Attention à renvoyer la valeur du tableau déclaré dans la procédure, et non son nom: `local T::array;...; return eval(T);...`

<pre>T:array 1..5 of entier; entier; T[1]:=3; T[3]:=5; T[2]:=7; T[3]:=6; T[3];</pre>	et	<pre>T:array 1..3 of entier; T[1]:=8; T[2]:=7; T[3]:=6; T[1];</pre>	et	<pre>T:array 1..3 of entier; T[1]:=8; T[2]:=T[1] mod</pre>
--	----	---	----	--

Se note en Maple:

<pre>T::array; T:=array(1..5); T[1]:=3; T[3]:=5; T[3];</pre>	et	<pre>T::array; T:=array(1..3, [8,7,6]); T[1];</pre>	et	<pre>T::array; T:=array(1..3, [8,7,6]); T[2]:=T[1] mod</pre>
--	----	---	----	--

Exemple tableau comme paramètre d'un algorithme:

```
f := proc(T::array(integer))::integer;
description "renvoie la somme des éléments de T";
local i::integer, S::integer;
S:=0;
for i from 1 to taille(T) do
    S:=S+T[i];
end do;
return S;
end proc;

x::integer; T::integer;
T:=array(1..3, [8,7,6]);
x:=f(T);
```

Exemple tableau comme résultat d'un algorithme:

```
f := proc(N::integer)::array(integer);
description "renvoie le tableau des N premiers impairs";
local i::integer, T::array;
T:=array(1..N);
for i from 1 to N do
    T[i]:=2*i-1;
end do;
return eval(T);
end proc;

R:=array(1..3);
R:=f(3);
```

## 4 - Traduction des algorithmes du premier cours en Maple

Pour ceux qui n'auraient pas suivi le premier cours voici le lien: [ici](#)

### 4-1 - Multiplication

```
multiplication := proc(a::integer, b::integer)::integer;
local i::integer, x::integer;
x:=0;
for i from 1 to b do
    x:=x+a;
end do;
return x;
end proc;
```

### 4-2 - PGCD de 2 entiers

```
pgcd := proc(a::integer, b::integer)::integer;
local i::integer, x::integer, q::integer, r::integer;
i:=a; x:=b; r:=a mod b;
while r <> 0 do
    i:=x;
    x:=r;
    r:=i mod x;
end do;
return x;
end proc;
```

### 4-3 - Minimum d'un tableau d'entier

```
minTableau := proc(T::array(integer))::integer;
local i::integer, x::integer;
x:=T[1];
for i from 2 to taille(T) do
    if T[i] < x then
        x:=T[i];
    end if;
end do;
return x;
end proc;
```

### 4-4 - Recherche dans un tableau

```
rechTableau := proc(T::array(integer), x::integer)::boolean;
local i::integer;
for i from 1 to taille(T) do
    if T[i] = x then
        return true;
    end if;
end do;
return false;
end proc;
```

## 5 - Débuggage d'un code Maple

### 5-1 - Gestion d'erreur classique

La variable interne *printlevel* de Maple est une variable qui contrôle le niveau des instructions qui sont affichées par Maple. Cette variable vaut initialement 0, c'est à dire que Maple n'affiche que les résultats des instructions se déroulant au niveau le plus bas. Vous avez la possibilité de lui donner une valeur arbitraire, ce qui vous permettra de voir affichés les résultats de toutes les instructions effectuées par Maple jusqu'au niveau choisi, je vous conseille de la mettre malgré tout à 7 en tapant *printlevel := 7* . ATTENTION: ce sont les instructions qui sont affichées, les évaluations d'expressions sont également considérées comme des instructions, par contre les tests conditionnels et le calcul des bornes d'une structure itérative ne sont pas considérés comme des instructions et ne sont pas affichés.

### 5-2 - Autres possibilités

Voici différentes variable utiles pour le debugge:

- *option trace* : Une des facilités est de pouvoir observer certains appels de procédures ou de fonctions à l'aide de l'instruction *trace*. Si vous demandez la trace de l'une des procédure ou fonctions, à chaque appel de la procédure ou de la fonction, Maple affichera la valeur courante des paramètres à l'entrée et la valeur de retour à la sortie. C'est un moyen commode de recherche d'erreurs simples se situant au niveau du passage des paramètres, de valeurs et de retour de fonctions ou de boucles sans fin. Ceci permet aussi de suivre facilement les procédures récursives. On supprime cette facilité avec l'instruction *untrace*.

```
f := proc(x::integer)::integer;
option trace;
local i::integer;
if x mod 2 = 0 then
i:=x;
else
i:=0;
end if;
end proc;
```

- *trace g* : même effet que l'option *trace*
- *userinfo* : permet de programmer l'affichage de messages suivant le niveau *infolevel* . Exemple:

```
f := proc(x) userinfo(2, 'test', 'tested with ',x); x; end:
infolevel[test] := 3: f(2);
f:      tested with      2
                                     2
infolevel[test] := 1: f(2);
                                     2
```

Je vous ai donné quelques options de Maple pour pouvoir débbugger, mais le meilleur moyen d'avoir un code sans problème c'est de faire attention à ce que vous marquez et ceci est valable pour tous les langages.

## 6 - Conclusion générale

### 6-1 - Epilogue

Vous connaissez maintenant les bases de la programmation en Maple évidemment ceci n'est qu'une mince partie de tout ce que l'on peut faire en Maple, mais vous pouvez déjà faire pas mal de chose avec ceci.

### 6-2 - Remerciements

Remerciements à **gorgonite** pour son aide à la mise en place de cet article et à **jeepnc** pour sa vérification orthographique.