Chapitre M V

Tableaux, vecteurs, matrices et tables

JM Janod

Les Tableaux

Principes de base des tableaux en Maple

Nous avons déjà vu des types de données structurées comme les séquences, les listes, et les ensembles dont le principe de base repose sur l'énumération. Les tableaux sont des types de données structurées par des indices d'entiers relatifs. En particulier un élément d'un tableau sera directement modifiable, contrairement aux séquences et ensembles.

En Maple, les tableaux sont hétérogénes (i.e. les éléments d'un tableau peuvent être de types différents). Ils sont statiques, unidimensionnels, bidimensionnels ou multidimensionnels.

Les tableaux unidimensionnels indexés à partir de 1 sont dits des vecteurs (*vector*).

Les tableaux bidimensionnels indexés à partir de 1 sont dits des matrices (*matrix*).

La règle de l'évaluation finale et la règle d'évaluation au premier nom (dite encore au premier niveau) ne s'appliquent pas aux tableaux qui sont régis par la règle de l'évaluation au dernier nom (voir ci-dessous) permettant le calcul symbolique matriciel.

Les exemples suivants exigent des explications qu'on abordera dans les paragraphes suivants.

```
> A:=array(1..2,1..2);print(A);C:=A;A[1,1];
```

```
> A[1,1]:=0:A[1,2]:='toto':A[2,1]:=x^2+1:A[2,2]:=2*x:print
  (A);
```

```
> print(C);
```

```
> x:=1:A[2,1];print(A);map(eval,A);
```

```
> B:=C+A;evalm(B);print(B);
```

```
> P:=array(-1..5);print(P);# P n'est pas un vecteur
```

```
> P[6]:=5;A[1,3];# les indices doivent être valides
Error, 1st index, 6, larger than upper array bound 5
Error, 2nd index, 3, larger than upper array bound 2
```

```
> nl:=2;nc:=3;E:=array(1..nl,1..nc);print(E);
```

```
> G:=array(1..n,1..m);
Error, non-integer ranges in array/table creation
```

Déclaration et initialisation des tableaux

La fonction **array** permet de déclarer les tableaux. Cette fonction réserve les emplacements mémoires des éléments du tableau, à partir de ses arguments qui sont de trois types.

1°) une ou des fonctions d'indices.
2°) la séquence des intervalles d'entiers fixant ses dimensions.
3°) l'initialisation de ses éléments en totalité ou partiellement sous forme de liste.

Chacun de ces types est optionnel, mais la fonction array doit pouvoir déterminer les dimensions du tableau à partir de ses arguments. Par exemple les dimensions seront déterminées en fonction de l'initialisation si aucune précision n'est donnée sur les dimensions, les indices commençant alors à 1.

Les fonctions d'indices permettent de gérer des tableaux ayant une configuration ou un modèle particulier, notamment des tableaux à deux dimensions carrés dont les éléments sont symétriques par rapport à la diagonale (ie A[i,j]=A[j,i]) qu'on appelle matrice symétrique en mathématiques. Citons encore les tableaux carrés antisymétriques (A[j,i]= - A[i,j] et A[i,i]=0), les matrices carrées diagonales ayant des 0 sauf sur la diagonale, la matrice identité ayant des 1 sur la diagonale et des 0 ailleurs, et les tableaux creux (***sparse*** pour clairsemé) dont les éléments sont nuls sauf un petit nombre d'entre eux. Ces fonctions d'indices sont prédéfinies en Maple et sont respectivement :

symmetric, antisymmetric, diagonal, identity, sparse

On peut définir ses propres fonctions d'indices que nous n'aborderons pas à ce niveau. Pratiquement toute référence au tableau ou à un de ses éléments passe par la fonction d'indice qui calcule par un algorithme la valeur pour les éléments fixés du tableau et n'utilise les emplacements des mémoires du tableau que pour les autres éléments.

La syntaxe générale est:

array(<fonctions d'indices>, <séquence des intervalles fixant les dimensions>, <liste des

initialisations>);

soit avec affectation:

Nom:=array(<fonction d'indices>**,** <séquence des intervalles fixant les dimensions>**,** <liste des initialisations>**)**

Nom est généralement une lettre majuscule par référence aux mathématiques sauf **D qui est l'opérateur différentiel**.

La syntaxe des intialisations est:

[valeur1,valeur2,..]

pour unidimensionnel à partir du premier indice

[i =valeur, j =valeur,..]

pour unidimensionnel initialisation des éléments i et j

Pour des tableaux bidimensionnels on a la syntaxe suivante:

[[valeur1,valeur2,..], [valeur12,valeur22,..] , [..]]

pour initialiser à partir des premiers indices de chaque ligne

[(i,j)=valeur, (i2,j2)=valeur,..]

initialisation des éléments d'indices (i,j) et (i2,j2)

Ces dernières syntaxes se généralisent pour des tableaux multidimensionnels.

Dans les exemples suivants on utilisera print pour afficher l'ensemble d'un tableau, ou evalm pour le résultat d'un calcul sur les vecteurs ou matrices.

```
> V:=array(-1..1,[12,5]);# V a une dimension avec une
  initialisation partielle des deux premiers
  print(V);
```

```
> V1:=array([1,2,3,4]);# La dimension est fixé par
  l'initialisation
  print(V1);
```

```
> V2:=array(1..4,[2=5]);#initialisation du deuxième
  élément
  V2[2];evalm(V1+V2);#evalm pour obtenir le tableau.
```

```
> Id:=array(identity,1..2,1..2);# pas d'initialisation
  print(Id);Id[2,2]:=5;
```

```
Error, cannot assign to an identity matrix
```

Dans l'exemple suivant on utilise deux fonctions d'indices qui, inversées, ne donneraient pas le même résultat.

```
> M:=array(symmetric,sparse,1..4,1..4,[seq(seq((i,j)=
```

```
   'x'^j,j=i+1),i=1..3)]);
```

```
>    [seq(seq((i,j)='x'^j,j=i+1),i=1..3)];#car
```

```
>    M[1,4]:=5;'M[4,1]'=M[4,1];# M est symétrique.
```

```
>    H:=array([ [1-lambda,2],[3,2-lambda] ]);
```

L'opérateur op appliqué aux tableaux

Un tableau n'a qu'une opérande lui-même et par contre op(<tableau>) a trois opérandes, les fonctions d'indices, les bornes des dimensions, les valeurs des éléments.

op(<tableau>) est équivalent à print.
op(1,op(<tableau>)) donne les fonctions d'indices**.**
op(2,op(<tableau>)) donne les bornes des dimensions.
op(3,op(<tableau>)) donne les éléments assignés.
indices(<tableau>) donne les indices des éléments assignés.
entries(<tableau>) donnent les valeurs effectives**.**

```
>    nops(M);op(M);nops(op(M));for i from 1 to 3 do op(i,op
   (M));od;indices(M);entries(M);
```

L'évaluation au dernier nom

Rappelons les exemples du premier paragraphe où on a posé A := array(1 .. 2,1 .. 2,[]), C:=A, puis les éléments de A ont été affectés, soit enfin on a initialisé x à 1. Posons F:=C et examinons les réponses fournies par Maple.

```
>    F:=C;`A`=A;'C'=C;'F'=F;print(whattype(A),whattype(C),
   whattype(F));
```

```
> print(type(A,array),type(C,array),type(F,array));
```

```
> print(A,C,F,x);
```

On remarque que F :=C a été évalué à A car C est évalué à A. Par contre les éléments de A sont toujours évalués en fonction de x et non de sa valeur 1. Posons x:=y et toto:=titi. Examinons les réponses de Maple puis nous donnerons les règles d'évaluation des tableaux.

```
> x:=y;toto:=titi;eval(A);A[1,2];A[2,2];print(C);
```

Règles d'évaluation des tableaux.

1°) **Règle d'évaluation au dernier nom**
Pour les expressions construites à partir des noms des tableaux, l'évaluation est effectuée jusqu'au dernier nom (ainsi F est évalué à C qui est évalué à A, le dernier nom est A).

2°) **Règle d'évaluation d'un tableau passé comme paramètre effectif** d'une fonction (print, eval et evalm).
Les éléments de A **qui comprennent des noms** sont évalués au premier niveau, c'est-à-dire au premier nom.(A s'affiche toujours avec x et toto).

3°) **Règle d'évaluation des initialisations à la construction**
Les éléments initialisés par la fonction array sont évalués selon la règle classique de l'évaluation finale.

4°) **Règle d'évaluation des éléments indexés**
Les éléments (A[i,j]) sont toujours évalués selon la règle de l'évaluation finale.(A[2,2] est bien y).

Les fonctions indispensables et d'évaluation

Les fonctions présentées dans ce paragraphe sont les suivantes.

eval, print, map, evalm, copy

Fonctions print et eval

Un tableau étant évalué à son nom, on peut forcer l'évaluation au tableau des éléments par la fonction eval qui retourne le tableau des valeurs. La fonction print affiche le tableau des éléments, mais retourne la valeur NULL

Il existe encore une différence entre print et eval. Lorsque les éléments ne sont pas assignés print utilise le nom du tableau, eval utilise le ?

```
> A;eval(A);print(A);
```

```
> X:=array(1..3);print(X);eval(X);
```

```
> C1:=eval(A);type(C1,array);C1;print(C1);# le dernier
  nom pour C1 est C1
```

La fonction map appliquée aux tableaux

La fonction map permet d'appliquer à tous les éléments d'un tableau une même fonction sa syntaxe est

map(<fonction>,<tableau>)

elle retourne le tableau transformé

Application importante

map(eval,<tableau>)

évaluation finale des éléments du tableau

```
> map(exp,A);
```

```
> map(eval,A);# on retrouve enfin y et titi
```

```
> eval(A);y:=1;titi:=2;map(eval,A);
```

```
> C2:=map(eval,A);C2;eval(C2);# le dernier nom pour C2
  est C2
```

Affectaction et evalm

Dans le premier paragraphe nous avons introduit l'affectation C:=A mais aussi B:=C+A et ci-dessus C1:=eval(A), C2:=map(eval,A) . Posons encore C3:=eval(A)+eval(A). Ces affectations ne sont pas de même nature et doivent être distinguées. En effet B et C3 ne sont

pas des tableaux mais des expressions matricielles comme on peut le vérifier.

L'évaluation de B conduit à l'évaluation de C+A, or C est evalué à A, et A à A, donc le résultat est 2A; B et eval(B) sont donc équivalents, les tableaux étant évalués au dernier nom. Pour obtenir l'évaluation de B sous forme de tableau il faut là encore forcer l'évaluation.

L'évaluation de C3 conduit à l'évaluation de eval(A), A etant évalué à A, eval(A) donne le tableau donc le résultat est 2 fois le tableau des valeurs, mais C3 n'est pas un tableau mais une expression matricelle ainsi C3 et eval(C3) sont équivalents. Par contre C est un tableau et non une expression matriciel.

La fonction evalm (m comme matrice) permet d'obtenir l'évaluation sous forme de tableau. Sa syntaxe est

evalm(<expression matricielle>)

donne le tableau des valeurs.

Lorsque l'expression est un tableau evalm est équivalent à eval

```
> C3:=eval(A)+eval(A);print(type(C,array),type(C1,
  array),type(C2,array),type(B,array),type(C3,array));
```

```
> B,eval(B),C3,eval(C3);
```

```
> evalm(B);map(eval,%);# car x:=y et y:=2,toto:=titi et
  titi:=2
```

La fonction copy

Nous avons vu que <tableau1>:=<tableau> n'entraîne pas une copie du tableau dans tableau1 mais simplement un synonyme de tableau. La fonction copy crée une sauvegarde d'un tableau indépendante de l'original.. Sa syntaxe est

<sauvegarde>:=copie(<du tableau>)

```
> AA:=copy(A);eval(AA);
```

Le calcul symbolique

Dans ce paragraphe, nous utiliserons des tableaux donc les indices sont supérieurs à 1. Les tableaux unidimensionnels sont dits des vecteurs (vector), les tableaux bidimensionnels sont dits des matrices (matrix), ces types étant reconnus sous Maple.

```
> W:=array(1..4);print(whattype(W),type(W,array),type(W,
  vector),type(M,matrix));
```

Le calcul vectoriel

On peut directement effectuer du calcul vectoriel classique, comme l'addition de deux vecteurs, la multiplication d'un vecteur par un nombre, la norme d'un vecteur. La norme est une fonction de la librairie linalg (linear algebra) dont la syntaxe est

linalg[norm] (<vecteur>,2)

pour utiliser la seule fonction norme, 2 pour euclidienne

ou

with(linagl):
norm(<vecteur>,2);

chargement de toutes les fonctions de linagl puis utilisation de norm

ou

with(linalg,norm):
norm(<vecteur>,2);

chargement de la seule fonction norm de linagl puis utilisation de norm

On signale dans la librairie linalg les fonctions suivantes que le lecteur découvrira dans l'aide.

vector est équivalente à array mais offre plus de souplesse.
vectdim donne la dimension d'un vecteur.
matadd calcule des combinaisons linéaires de deux vecteurs.
angle calcule l'angle entre deux vecteurs.
dotprod calcule le produit scalaire de deux vecteurs.
normalize calcule le vecteur normé.

```
> eval(V1);eval(W);V3:=W/2+alpha*V1;evalm(V3);
```

```
> linalg[norm](V3,2);linalg[norm](V1,2);
```

La géométrie à deux dimensions

La librairie géométrique (geometry) fournit de nombreux outils permettant le calcul classique sur des points d'un plan et les vecteurs. Nous donnons simplement un exemple définissant un point, calculant la droite passant par ce point et l'origine, et les informations liées à cette droite dont son équation. Pour l'utilisation de cette librairie consultez l'aide.

```
> with(geometry):point(a1,1,1);line(h,[a1,point(``,0,0)]);
  detail(h);

   name of the object: h
   form of the object: line2d
   assume that the name of the horizonal and vertical
```

```
axis are _x and _y
   equation of the line: _x-_y = 0
```

Le calcul matriciel

Les matrices n'étant pas enseignées en première année de GEA, ce type de calcul ne sera pas abordé. Signalons toutefois que la librairie linalg contient toutes les procédures liées aux matrices (vecteurs propres, valeurs propres, déterminant etc).

On peut calculer directement la somme de deux matrices par le signe + et la multiplication par l'opérateur fonctionnel &*.

Les Tables

Principe des tables en Maple

Les tables sont présentées comme une généralisation des tableaux. Les tableaux sont souvent considérés comme un cas particulier des tables, les tableaux étant indicés par des entiers , les tables par des indices quelconques. Maple considère que tout tableau est aussi de type table. Poutant les tables présentent un caractère supplémentaire car elles sont dynamiques, leurs dimensions variant en fonction de son utilisation. C'est la structure de donnée la plus générale similaire aux structures des données des enregistrements qu'on trouve dans d'autres langages tels que le Pascal, ou le C.

Pratiquement tout ce qui concerne les tableaux s'applique aux tables, sauf le caractère statique, La fonction table remplace array et intègre le caractère dynamique de celle-ci.

Déclaration d'une table
Cette déclaration n'est pas obligatoire, sa syntaxe est:

table(<fonctions d'indices>**,** <liste des initialisations>**)**

soit avec affectation:

<nom>:=**table(**<fonctions d'indices>**,** <liste des initialisations>**)**

remarque
Un élément d'une table peut lui-même être une table.

```
>    T:=table(symmetric,[(paris,nantes)=400,(paris,lyon)=450]
   ) ;#avec initialisation
```

```
>    T[lyon,paris];
```

```
>    Vo[depart]:=8;Vo[arrivee]:=10;Vo[distance]:=410;Vo[prix]
   :=380;V0;eval(Vo);#creation dynamique par affectaction
```

```
> VT:=table([un=1,(trois)=3,5]);# pour avoir des indices
  toutes les initialisations doivent être définies par des
  indices et l'égalité ce n'est pas le cas pour 5
```

```
> Vt:=table([un=1,(trois)=3,cinq=5]);
```

```
> Vt:= table([un,deux,cinq]);
```

```
> AT:=table([0,'toto','x'^2+1,2*'x']);x:='y';op(AT);y,AT
  [3];
```

```
> eval(AT);x,eval(toto,1),titi;map(eval,AT);
```

```
> for i from 1 to 3 do i,op(i,op(AT));od;indices(AT);
  entries(AT);#op(3,op(<table>) n'existe pas
```

```
Error, improper op or subscript selector
```

Suppression d'un élément d'une table et initialisation d'une table vide

Pour supprimer un élément d'une table il suffit de l'affecter par son nom

<table>[<indice>]:= ' <table>[indice] '

Pour initialiser une table vide on peut utiliser la commande:

<table>**:=table();**

Rappelons les points vus sur les tableaux et signalons les différences si elles existent.

L'opérateur op appliqué aux tables

op(<table>) identique aux tableaux est équivalent à print.

op(1,**op**(<table>)) identique aux tableaux donne les fonctions d'indices.

op(2,**op**(<table>)) identique a **op**(3,**op**(<tableau>)) donne la liste des éléments assignés

indices(<table>) donne la séquence des indices des éléments assignés, chaque indice étant une liste.

entries(<tableau>) donne la séquence des valeurs assignées, chaque valeur étant une liste.

Les bornes n'ayant plus de signification, il n'y a pas d'équivalent de op(2,<tableaux>).

Les règles d'évaluation des tables

Elles sont identiques point par point aux tableaux.

Les fonctions indispensables

les fonctions **eval, print, map, copy** sont strictement identiques. evalm (table) est équivalent à <table>**;**

```
>    evalm(AT);AT;
```

Utilisation des tableaux et des tables dans les fonctions

Passage des paramètres

L'utilisation d'un tableau ou d'une table comme paramètre formel d'une fonction (proc) se traduit toujours par un passage par variable, comme si on indiquait ::name, quelque soit la déclaration du paramètre formel.

Une fonction peut évidemment retourner une table ou un tableau ce qui n'est pas le cas des langages classiques.

```
>    pro1:=proc(a,b)
   #b n'est pas déclaré name et pourtant les affectations ci-
   dessous seront acceptées
         local lesindices,j;
         lesindices:=indices(a);
         for j in lesindices do
              b[op(j)]:=eval(a[op(j)],1)+1;#sans le op on
   aurait a[[?]] et sans eval(..,1) on aurait une évaluation
   finale, b sera indépendant de a
         od;
   NULL;end;
```

```
>    indices(AT);AT[1];entries(AT);pro1(AT,CT);print(CT);
```

```
>    ATA:=copy(AT):AT[3]:='AT[3]';print(CT,AT);#CT est
   indépendant de A
   AT:=copy(ATA):;
```

```
> pro2:=proc(a::table,b::table)
    local lesindices,j;
    lesindices:=indices(a);
    for j in lesindices do
        b[op(j)]:=eval(a[op(j)],1)+2;
    od;
     a:=b;
#a est lié à b l'affectation est acceptée même si a est
déclaré comme table et non comme name
end;
```

```
> ATA:=copy(AT):CT:=table():pro2(AT,CT):print(CT);CT:=
'CT':print(AT,CT);AT:=copy(ATA):;
```

```
> pro3:=proc(a::table)
    local lesindices,j,b:
    lesindices:=indices(a);
    for j in lesindices do
        b[op(j)]:=eval(a[op(j)],1)+2;
    od;
     op(b):# pro3 retourne la dernière évaluation
end;
```

```
> R:=pro3(AT);type(R,table);
```

Les tables ou les tableaux de fonctions:les librairies

On peut construire des tables ayant pour éléments des noms de fonctions. On peut alors les calculer pour une valeur fixée. Cette possibilité remarquable est inexistante dans les langages classiques.

Donnons des exemples.

```
> mesfon:=table([lecos=cos,lesin=sin,lexpon=exp]);indices
(mesfon);
```

```
> for i in indices(mesfon) do mesfon[op(i)](Pi/6);od;
```

Remarque

On peut remplacer la table par un tableau; le principe est le même (voir ci-dessous).

Les librairies utilisent ce principe. Elles sont conçues comme des tables ayant pour indice le nom de la fonction initialisé par la commande readlib (commande pour lire un fichier) d'ou la notation lorsqu'on ne charge pas la librairie (with(<librairie>)).

<librairie> **[**<nom de la fonction>**] (**<valeur>**)**

```
> eval(geometry);# le résultat, trop long, a été supprimé
```

```
> ft:=array([1=cos,2=sin,3=exp]);op(2,op(ft));op(op(2,op
  (ft)));sup:=op(op(2,op(ft)))[2];#remarquer l'obtention de
  la dimension
```

```
> for i from 1 to sup do ft[i](Pi/2);od;
```

La fonction de conversion

Compte tenu de la richesse des implantations (sous Maple) de la notion de liste définie en algorithmique, une fonction de conversion est indispensable entre les séquences, les listes, les ensembles, les tableaux et les tables. La syntaxe est

convert(<objet>, <le type>)

retourne l'objet dans le nouveau type

```
> op(A);convert(A,listlist);# conversion d'un array 2,2 en
  liste de liste sans évaluation finale
```

```
> convert(%,list);#effectue une évaluation finale sur les
  éléments d'une liste de liste
```

```
> convert([[0,'toto','y'],['x'^2+1,2*'x','y'+2]],array);
```

```
> convert(A,set);#evaluation finale
```

```
> convert(A,list);#impossible pour un tableau
  multidimensionnel
```

```
Error, (in convert/list) can't convert array of dimension > 1
```

```
> print(V3,W,V1);op(V3);convert(V3,list);convert(evalm(V3),
  list);#conversion d'une expression vectoriel)
```

```
> convert(evalm(V3),listlist);
```

```
> convert(%,array);
```

```
> convert(AT,list);#conversion d'une table en liste, la
  conversion en liste liste est impossible pas d'evaluation,
  finale
```

```
> convert(convert(A,table),list);
```

```
> matrix(2,2,%);#evaluation des éléments
```

```
> convert(%%,table);
```