DEUXIEME PARTIE
DIFFERENTS OBJETS MANIPULES EN MAPLE

I - Constantes, variables système, variables libres, variables assignées

1. Constantes

Il n'y a pas de type "constante" en Maple et seules quelques constantes importantes en mathématique sont reconnues.

Pi	π
I	imaginaire pur
gamma	constante d'Euler
E	base des log népériens
± infinity	concept traité comme une constante

2. Variables système

Les variables système sont des variables globales et sont toujours assignées (elles ont des valeurs par défaut).

Digits	variable système donnant la précision des nombres flottants Valeur initialisée à 10.
Order	ordre des développements limités Valeur initialisée à 6.

3. Variables

Contrairement aux autres langages, une variable Maple peut être utilisée de façon formelle.

q:= cos(x) + 2* sin(y)	x, y et q sont formels
x:= 3	la variable est assignée
x:= evaln(x)	désassigne la variable x
x:= 'x'	désassigne la variable x
x:=`x`	x est une chaîne de caractère et vaut "x"
restart	désassigne tout

II - Nombres

Tous les langages de calcul codifient les nombres sur des champs de mémoire fixes (généralement 32 ou 64 bits), les nombres sont donc nécessairement limités en précision et en grandeur. Maple représente les objets comme des listes, la précision des nombres peut donc être quasi infinie (il y a tout de même une limite physique qui est la taille de la mémoire tampon disponible).

1. Les entiers

100!	retourne factorielle de 100
ifactor(n)	décomposition en facteurs premiers de n
isprime(n)	fonction booléenne vraie si le nombre est premier
igcd (n_1, n_2)	recherche le PGCD

Les conversions de base d'entiers positifs peuvent être obtenues par la fonction convert.

```
> convert(247,binary);
```

$$11110111$$

```
> convert(1023,hex);
```

$$3FF$$

2. Rationnels

Maple peut effectuer des calculs exacts avec des rationnels, même avec des termes très grands au numérateur et au dénominateur. La simplification est automatique.

3. Nombres réels exacts

Maple utilise les irrationnels **sans passer par leur valeur approchée**.

Exemples

```
> a:=sqrt(3);
```

$$a := \sqrt{3}$$

```
> a*a;
```

$$3$$

```
> b:=2^(2/3);
```

$$b := 2^{2/3}$$

```
> b^3;
```

$$4$$

```
> sin(Pi/12)
```

$$\frac{1}{4}\sqrt{6}\left(1-\frac{1}{3}\sqrt{3}\right)$$

4. Nombres réels approchés et fonction evalf

On peut évaluer tout nombre rationnel ou réel en virgule flottante avec une précision définie à l'aide de l'instruction evalf.

Le nombre de chiffres significatifs est celui qui est donné par la dernière valeur de la variable Digits (variable système par défaut initialisée à 10)

```
> evalf(a);
```

$$1.732050808$$

```
> Digits:=30;
```

$$Digits := 30$$

```
> evalf(a);
```

$$1.73205080756887729352744634151$$

Le nombre de digits peut également être précisé dans la fonction evalf

```
> evalf (Pi, 40)
```

$$3.1415926535897932384626433832795028841 97$$

5. Nombres complexes et fonction evalc

Le nombre imaginaire pur est noté I

```
> sqrt(-1);
```

$$I$$

Les fonctions **Re, Im, abs, argument** fournissent respectivement la partie réelle, imaginaire, le module et l'argument d'un nombre complexe.

polar permet de définir le nombre en représentation polaire.

evalc(z) entraîne l'évaluation sous la forme a + Ib, a et b étant des **rationnels ou des réels exacts**.

```
> evalc(ln(1+I));
```

$$\frac{1}{2}\ln(2) + \frac{1}{4} I \pi$$

evalf (z) entraîne l'évaluation par un couple de flottants à la précision donnée par Digits.

```
> evalf(ln(1+I),20);
```

$$0.34657359027997265471 + 0.78539816339744830962\ I$$

On évalue l'expression en flottant avec 20 digits.

III - Expressions

1. Syntaxe des expressions

La syntaxe des expressions algébriques est proche de la syntaxe mathématique.

a) Opérateurs arithmétiques et parenthèses

+ - * / ^ ou **	pour la puissance
( )	mêmes parenthèses quel que soit le niveau d'imbrication
[ ]	utilisé pour les indices et les listes
{ }	utilisé pour les ensembles
x/y/z $\rightarrow \frac{x}{yz}$	pour programmer un trait de fraction principal, utiliser des parenthèses
(x/y)/z	entraîne x/(y*z) qui est différent de x/(y/z)
&*	produit matriciel

b) Opérateurs logiques et relationnels

Lesopérateurs logiques sont : and or ;xor , not implies et FAIL (non vrai et pas sûrement faux)
Lesopérateurs relationnels sont : <, <=, =, <>
Attention, il n'y a pas d'opérateur > et >= , il faut programmer not < ou not <= ou modifier le sens des tests

c) Fonctions

ln, sqrt, exp, log[10]	logarithme népérien, racine carrée, exponentielle, logarithme décimal
sin, cos, tan, ...	fonctions trigonométriques
sinh, cosh, tanh, ...	fonctions hyperboliques
arcsin, arccos, arctan, ...	fonctions trigonométriques inverse (attention on n'utilise pas la syntaxe mathématique pour les fonctions hyperboliques et leurs inverses)
arcsinh, arccosh, ...	fonctions hyperboliques inverses
erf, erfc	fonction d'erreur (intégrale de la fonction de Gauss)
bessel	fonction de Bessel
GAMMA	fonction d'Euler
etc...	

Une liste complète des fonctions est donnée en annexe

d) Sommes, produits et factorielles

On peut introduire des sommes Σ ou des produits Π dans les expressions mathématiques.

Ces signes de sommation et de produits existent **sous forme inerte et sous forme active**, lorsque l'instruction est écrite avec une majuscule, il s'agit de la forme inerte (Sum, Product), c'est à dire que l'expression est affichée mais non exécutée.

```
> Sum(1/i^6,i=1..infinity);
```

$$\sum_{i}^{\infty} \frac{1}{i^6}$$

Le signe somme est simplement affiché.

```
> sum(1/i^6,i=1..infinity);
```

$$\frac{1}{945}\Pi^6$$

ici la somme a été calculée (si possible)

```
> Product(k+n,k=0..n-1);
```

$$\prod_{k=0}^{n-1} (k+n)$$

(Aucune ambiguïté avec le nombre Π)

```
> product(k+n),k=0..n-1);
```

$$\frac{\Gamma(2n)}{\Gamma(n)}$$

2. Manipulation d'expressions

a) Simplification : fonction simplify

La simplification peut être effectuée à l'aide des **règles intrinsèques** de Maple ou en indiquant des **règles de simplification particulières**.

```
> expr:=(x^2-1)/(x^2+2*x+1);
```

$$expr := \frac{x^2-1}{x^2+2x+1}$$

```
> simplify(expr);
```

$$\frac{x-1}{x+1}$$

```
> b:=4*cos(x)**2*cos(2*x)+4*sin(x)**2;
```

$$b := 4\cos(x)^2\cos(2x) + 4\sin(x)^2$$

```
> simplify(b);
```

$$8\cos(x)^4 - 8\cos(x)^2 + 4$$

La simplification utilise par défaut toutes les règles de trigonométrie, les règles sur les exponentielles et effectue les factorisations élémentaires.

Il est possible de préciser un ensemble de règles et les variables par rapport auxquelles on veut simplifier.

simplify (expr, {ensemble de relations},{ensemble de variables})

b) Décomposition des fractions rationnelles en éléments simples

convert (fraction, parfrac, variable)

```
> convert(expr,parfrac,x);
```

$$1+2\frac{1}{(x+1)^2}-2\frac{1}{x+1}$$

c) Développement d'une expression : fonction expand

```
> expand(x*(x-1)**3+2);
```

$$x^4 - 3\,x^3 + 3\,x^2 - x + 2$$

```
> expand(sin(3*x));
```

$$4\,\sin(x)\,\cos(x)^2 - \sin(x)$$

```
> expand(log((x-1)*(x+1)));
```

$$\ln(x - 1) + \ln(x + 1)$$

Ici, c'est le logarithme qui est développé et non la fonction polynôme.

par contre

```
> log(expand((x-1)(x+1)));
```

$$\ln(x^2 - 1)$$

Ici c'est le polynôme qui est développé.

d) Regroupement d'expression et factorisation : les fonctions combine et factor

En gros, la fonction combine réalise l'opération inverse de la fonction expand. Elle peut également être utilisée pour regrouper des intégrales sous le même signe somme. Dans le cas des polynômes, il faut utiliser l'opérateur factor, mais on ne peut factoriser que des expressions simples.

e) Opérateurs et opérandes d'une expression, arborescence d'une expression

op(expression)	donne les opérandes d'une expression
nops(expression)	donne le nombre d'opérandes
op(o, expression)	donne l'opérateur utilisé
op(j, expression)	fournit le $j^{ème}$ opérande de l'expression

Exemples

```
> expr:=x*y+z*t;
```

$$expr:=xy+zt$$

```
> op(expr);
```

$$xy,zt$$

```
> nops(expr);
```

$$2$$

```
> op(0,expr);
```

$$+$$

```
> op(2,expr);
```

$$zt$$

```
> op(0,op(2,expr));
```

$$*$$

On peut ainsi déterminer l'arborescence d'une expression.

f) Affectation d'une expression, désassignation

On peut donner un nom symbolique à une expression de deux manières :

nom:= expression
assign (nom, expression)
nom:= 'nom' désassigne l'expression nom

g) Evaluation d'une expression

Les fonctions evalf et evalc s'appliquent aux expressions lorsque toutes les variables ont reçu une valeur.

Lorsque certaines variables symboliques ont été valorisées, la fonction eval calcule et simplifie l'expression compte tenu de toutes les variables connues.

```
> y:= 4 :
> eval(x+x+2*(y+1));
```

$$2x + 10$$

```
> y:='y' :
> eval(x+x+2*(y+1));
```

$$2x + 2y + 2$$

L'instruction y:= 'y' a pour effet de **désassigner** y, c'est à dire **y redevient une variable symbolique.**

h) Substitution d'une expression dans une autre expression

subs (var = objet, expression (var))

L'expression devient expression (objet). Il faut éventuellement faire des simplifications ultérieurement.

```
> subs(x=z+t,x+y*cos(x));
```

$$z + t + y * cos(z + t)$$

```
> subs(u=ln(z),exp(u));
```

$$e^{ln(z)}$$

```
> simplify('');
```

$$z$$

En réalité, l'expression initiale reste inchangée bien que la substitution apparaisse à l'écran. Si on veut garder l'expression substituée, il faut la stocker sur une nouvelle variable.

IV - Séquences, listes, ensembles

Ces trois objets sont des structures comprenant plusieurs expressions (ou variables). Il ne faut pas confondre ces trois objets, ils ont chacun leurs propriétés et leur utilité dans Maple. Les propriétés de ces objets ne sont souvent pas compatibles, mais sont complémentaires d'où leur utilité.

1. Séquences

a) Définition et construction

C'est une suite ordonnée d'expressions (les nombres et les variables sont également considérés comme des expressions).

On construit une séquence de différentes manières :

- Ecriture des différentes expressions dans l'ordre et séparées par des virgules

```
> s:=x,x^2,x-y,z-1,2,3cos(x),cos(y);
```

$$s := x, x^2, x - y, z - 1, 2, 3\,cos(x), cos(y)$$

- En utilisant un constructeur ($)

```
> s:=$4..10;
```

$$s := 4, 5, 6, 7, 8, 9, 10$$

```
> u:=x^i,$i=1..4;
```

$$u := x, x^2, x^3, x^4$$

```
> v:=x[i],$i=1..4;
```

$$x_1, x_2, x_3, x_4$$

♦ En utilisant le constructeur seq

```
> u:=seq(x^i,i=1..4);
```

$$u := x, x^2, x^3, x^4$$

```
> s1:=seq(cos(2*Pi/i,i=1..4);
```

$$s1 := 1, -1, -\frac{1}{2}, 0$$

b) Opérations sur les séquences

La concaténation permet de mettre bout à bout deux séquences. On écrit simplement les deux séquences séparées par une virgule. **Cette opération n'est possible qu'avec les séquences** :

```
> w:=u,v;
```

$$x, x^2, x^3, x^4, x_1, x_2, x_3, x_4$$

c) Extraction des éléments d'une séquence, partie d'une séquence

```
> w[2];
```

$$x^2$$

```
> w[4..6];
```

$$x^4, x_1, x_2$$

2. Listes

a) Définition et construction

Une liste est une séquence entourée de crochets. Une liste se différencie d'une séquence par ses propriétés.

On construit une liste

♦ En écrivant les éléments de la liste dans leur ordre.

```
> l:=[a,b,c,x,y,cos(x),cos(y)] ;
```

$$l := [a, b, c, x, y, \cos(x), \cos(y)]$$

♦ Il n'y a pas de **constructeurs de listes**, mais on peut facilement **convertir une séquence en liste**. Il suffit de mettre la séquence entre crochets.

```
> v:=[s];
```

$$[4, 5, 6, 7, 8, 9, 10]$$

b) Eléments d'une liste

Un élément isolé peut être référencié par son rang dans la liste.

l[3]	c
liste[i..j]	fournit la séquence des éléments depuis le rang i jusqu'au rang j, i et j inclus
l[3..6]	c, x, y, cos(x)
op(liste)	fournit la séquence de tous les éléments de la liste
nops(liste)	nombre d'éléments de la liste
op(o, liste)	donne le type (liste)
op(j, liste)	identique à liste [j]
member(élément, liste)	teste si l'élément se trouve dans la liste. C'est une fonction à résultat booléen
has(liste, élément)	test identique mais beaucoup plus approfondi. Si on traite une liste de listes, la fonction *has* cherche l'élément dans toute l'arborescence de la liste

c) Concaténation de deux listes

Elle n'est pas possible directement. Il faut d'abord transformer les deux listes en séquences (op(liste)), effectuer la concaténation des séquences, puis retransformer en liste .

```
> l1:=[a,b,c,d];
```

$$l1 := [a, b, c, d]$$

```
> l2:=[x,y,z,t];
```

$$l2 := [x, y, z, t]$$

```
> l3:=l1,l2;
```

$$l3 := [a, b, c, d], [x, y, z, t]$$

On obtient la séquence formée par les deux listes et non pas la liste concaténée

```
> l4:=[op(l1),op(l2)];
```

$$l4 := [a, b, c, d, x, y, z, t]$$

De cette façon, on obtient une seule liste qui est la concaténation des listes l1 et l3.

```
> l5:=[op(l1),l2];
```

$$l5 := [a, b, c, d, [x, y, z, t]]$$

Observer la différence avec l4

d) Application d'une fonction sur les éléments d'une liste

On utilise l'instruction

map (fonction, liste)

Exemple

```
> s:=Pi/i $i = 1..6;
```

$$s := \pi, \frac{1}{2}\pi, \frac{1}{3}\pi, \frac{1}{4}\pi, \frac{1}{5}\pi, \frac{1}{6}\pi$$

```
> l:=[s];
```

$$l := [\pi, \frac{1}{2}\pi, \frac{1}{3}\pi, \frac{1}{4}\pi, \frac{1}{5}\pi, \frac{1}{6}\pi]$$

```
> map(sin,l);
```

$$[0, 1, \frac{\sqrt{3}}{2}, \frac{\sqrt{2}}{2}, \frac{1}{4}\sqrt{2}\sqrt{5-\sqrt{5}}, \frac{1}{2}]$$

Le résultat est une liste des valeurs de sin(π), sin($\frac{\pi}{2}$), etc... Les trois instructions ci-dessus peuvent être combinées :

```
> map(sin,[Pi/i, $i=1..6]);
```

$$[0, 1, \frac{\sqrt{3}}{2}, \frac{\sqrt{2}}{2}, \frac{1}{4}\sqrt{2}\sqrt{5-\sqrt{5}}, \frac{1}{2}]$$

3. Ensembles

a) Définition et construction

En Maple, un ensemble est un objet correspondant à la définition des ensembles en mathématiques et qui en possède les propriétés.

Dans un ensemble, les éléments ne sont ni ordonnés, ni dupliqués. En Maple, un ensemble est formé en écrivant la séquence de ses éléments entre accolades.

ens:= {a, b, c, d, x, y, z}

On peut également convertir une liste ou une séquence en un ensemble. **Dans cette conversion, les éléments multiples sont supprimés.**

Exemple

```
> l:=[x,x^2,x,x^2-1,x];
```

$$l := [x, x^2, x, x^2 - 1, x]$$

```
> ens:=convert(l,set);
```

$$ens := \{x, x^2 - 1, x^2\}$$

b) Eléments d'un ensemble

Toutes les opérations possibles sur les éléments d'une liste sont également possibles sur les éléments d'un ensemble, mais il faut remarquer que :

- l'extraction d'un élément ou d'une suite d'éléments à l'aide d'un indice ou d'un champ d'indices n'a pas de sens puisque les éléments ne sont pas rangés,
- les instructions l[1] ou l[2..3] donneront des éléments suivant leur **rangement en mémoire**, mais celui-ci peut être modifié à chaque manipulation de l'ensemble en question,
- la fonction map ne s'applique pas aux éléments d'un ensemble.

c) Opérations spécifiques sur les ensembles

L'union	union
L'intersection	intersect
La différence	minus
Sous-ensembles	on peut déterminer des sous-ensembles à l'aide du paquetage combinat.

d) Structure d'un ensemble

Un ensemble n'est pas d'emblée doté d'une structure. On peut définir des opérateurs et des structures particulières avec des bibliothèques spécialisées.

V - Tableaux, matrices, vecteurs

1. Tableaux

a) Définition et déclaration

Les tableaux sont des collections d'objets rangés sous un nom unique et accessible par des indices. Tous les objets d'un tableau ne sont pas nécessairement du même type.

Les tableaux doivent être déclarés (l'ordinateur doit réserver de la place en mémoire et créer des pointeurs pour gérer les tableaux).

nom:= array (inf1..sup1, inf2..sup2,..[init])

inf et sup sont les bornes du tableau. Ces bornes doivent être des entiers et inf < sup. Les bornes peuvent être négatives, positives ou nulles.

Le nombre d'indices est la dimension du tableau.

[init] est une structure de listes imbriquées permettant d'initialiser le tableau au moment de sa déclaration. Le niveau d'imbrication des listes est égal à la dimension du tableau et les listes les plus internes de cette structure sont les lignes du tableau.

b) Initialisation des tableaux

♦ Soit au moment de déclaration

```
> A:=array(-1..0,1..2,[[2,3],[4,5]]);
```

A:= array(-1 .. 0, 1 .. 2, [(-1, 1) = 2 (-1, 2) = 3 (0, 1) = 4 (0, 2) = 5])

A est un tableau bidimensionnel dont les bornes sont respectivement 1;0 pour les lignes et 1;2 pour les colonnes. La structure d'initialisation est une liste de listes [2, 3] initialise la première ligne du tableau, respectivement [4, 5] pour la deuxième ligne.

$$\begin{array}{c} \\ 1 \\ 2 \end{array}\begin{array}{c} \begin{array}{cc} -1 & 0 \end{array} \\ \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \end{array}$$

donc A(-1, 1) = 2 A(0, 1) = 3 A(-1, 2) = 4 et A(0, 2) = 5

♦ Soit par simple affectation

```
> A:=array(1..2,1..2);
```

A:= array (1..2, 1..2, [ ])

```
A[1, 1]:=x:   A[1,2]:=x^2:   A[2,1]:=x^3:   A[2,2]:=1:
> evalm(A);
```

$$\begin{bmatrix} x & x^2 \\ x^3 & 1 \end{bmatrix}$$

♦ Soit par un des constructeurs : sparse, identity, symmetric, antisymmetric.

sparse	tous les éléments sont nuls
identity	les éléments diagonaux sont égaux à 1
symmetric ou antisymmetric	l'initialisation d'un élément extradiagonal entraîne automatiquement l'initialisation de l'élément conjugué

```
> T:=array(symmetric,1..2,1..2);
```

T:= array (symmetric, 1..2, 1..2, [ ])

```
> T[1,2]:=2;
```

T1,2 := 2

```
> evalm(T);
```

$$\begin{bmatrix} ?_{1,1} & 2 \\ 2 & ?_{2,2} \end{bmatrix}$$

```
> I3:=array(identity,1..3,1..3);
```

I3:= array (identity, 1..3, 1..3 [ ])

```
> evalm(I3);
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

c) Opérations sur les tableaux

Il n'y a pas d'algèbre sur les tableaux sauf si ceux-ci sont déclarés comme des matrices. Les seules opérations sont la copie d'un tableau et la conversion dans d'autres types.

B:= A B:= copy (A)

Ces deux instructions ne sont pas équivalentes. Dans le premier cas, les tableaux B et A restent liés, si l'on modifie un élément de A, l'élément correspondant de B est modifié. **Utiliser copy si on veut créer des objets indépendants mais initialement égaux**.

convert (tableau1, list)	convertit un tableau unidimensionnel en liste
convert (tableau2, list list)	convertit un tableau bidimensionnel en liste de listes
convert (tableau2, matrix)	convertit un tableau bidimensionnel en matrice
convert (tableau, set)	convertit un tableau en ensemble

2. Matrices

a) définition et déclaration

Dans le sens de Maple, une matrice est un tableau bidimensionnel auquel on a associé les propriétés des matrices (structure d'espace vectoriel).

En Maple natif (sans l'appel d'aucun paquetage), l'objet matrice ne peut pas être déclaré, mais on peut convertir les tableaux en matrices ou les évaluer comme des matrices.

```
> a:=array(1..2,1..2,[[1,3],[0,-1]]);
```

$$a := \begin{bmatrix} 1 & 3 \\ 0 & -1 \end{bmatrix}$$

```
> m:=convert(a,matrix);
```

$$m := \begin{bmatrix} 1 & 3 \\ 0 & -1 \end{bmatrix}$$

```
> evalm(a);
```

$$\begin{bmatrix} 1 & 3 \\ 0 & -1 \end{bmatrix}$$

Pour déclarer directement une matrice, il faut avoir chargé préalablement le paquetage linalg. La déclaration des matrices diffère légèrement de celle des tableaux parce que les indices débutent toujours par 1. Il suffit donc de déclarer la borne supérieure.

matrix (sup1, sup2, [init])

[init] est une liste simple (pas une liste de listes) contenant les éléments de la matrice **rangés par ligne**.

```
> with(linalg):
> m:=matrix(2,2,[1,3,0,-1]);
```

$$m := \begin{bmatrix} 1 & 3 \\ 0 & -1 \end{bmatrix}$$

b) Initialisation d'une matrice

- Soit par une liste d'initialisation au moment de la déclaration.
- Soit par simple affectation.
- Soit à l'aide d'une fonction génératrice

```
> Z:=matrix(3,3,(i,j)→x^(i+j-2));
```

$$Z := \begin{bmatrix} 1 & x & x^2 \\ x & x^2 & x^3 \\ x^2 & x^3 & x^4 \end{bmatrix}$$

- Soit en convertissant un tableau bidimensionnel préalablement initialisé.
- Le paquetage linalg permet également d'initialiser directement des matrices en bande ou des matrices diagonales en blocs.
- En donnant la dénomination de matrices particulières ayant des noms reconnus

```
> C:=hilbert(3);                # crée une matrice de Hilbert
```

$$C := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

```
> V:=vandermonde([1,x,y])     # crée une matrice de van der Monde
```

$$V := \begin{bmatrix} 1 & 1 & 1 \\ 1 & x & x^2 \\ 1 & y & y^2 \end{bmatrix}$$

c) Opérations élémentaires sur les matrices

Multiplication par un scalaire	*
Multiplication matricielle (non commutative)	&*
Transposée	transpose(m)
Inverse	inverse(m) ou m**(-1) ou 1/m
Opérations sur les lignes et les colonnes	utiliser l'aide en ligne

Nota : **Utiliser evalm(resultat)** pour afficher le résultat d'une opération matricielle. On ne donne ici que les opérations élémentaires. Pratiquement tous les calculs concernant les matrices (valeurs propres, vecteurs propres, réduction sous forme de Gauss-Jordan) sont possibles grâce au paquetage linalg. Consultez l'aide en ligne dans ces domaines.

3. Vecteurs

a) Définition et déclaration

Pour Maple, un vecteur est un tableau unidimensionnel auquel on a associé les propriétés des vecteurs (addition, multiplication par une constante, produit scalaire, produit vectoriel pour les vecteurs dans R^3). Maple reconnaît les tableaux unidimensionnels correctement déclarés comme des vecteurs mais seulement pour les opération élémentaires.

Pour déclarer directement un vecteur ou pour utiliser des possibilités de calcul plus étendues, il faut avoir préalablement chargé le paquetage d'algèbre linéaire.

v:= vector (dim, [liste])

dim est la dimension du vecteur (nombre de composantes)
liste est la liste des composantes.

```
> with(linalg);
> v:=vector(3,[1,1,1]);
```

$$v := [1\ 1\ 1]$$

On peut également convertir un tableau en vecteur

```
> a:=array(1..3,[1,-1,1]);
```

$$a := [1\ -1\ 1]$$

```
> v:=convert(a,vector):
```

b) Opérations sur les vecteurs

L'addition et la multiplication par un scalaire sont possibles directement sur les tableaux à condition d'évaluer le résultat par l'instruction evalm (tout se passe comme si on travaillait sur des matrices ligne).

```
> u:=array(1..3,[a,b,c]);
```

$$u := [a\ b\ c]$$

```
> v:=array(1..3,[x,y,z]);
```

$$v := [x\ y\ z]$$

```
> m:=evalm(u+lambda*v);
```

$$m := [a + \lambda x \quad b + \lambda y \quad c + \lambda z]$$

Par contre, les opérations propres aux vecteurs se trouvent dans le paquetage linalg.

Combinaison linéaire

```
> add(u,v,λ,μ);
```

$$[\lambda a + \mu x \quad \lambda b + \mu y \quad \lambda c + \mu z]$$

réalise la somme λu + μv (double emploi avec evalm, mais add **conserve la dimension du résultat**, même si ce résultat est le vecteur nul)

Produit scalaire

```
> dotprod(u,v);
```

$$ax + by + cz$$

Produit vectoriel (uniquement dans R^3)

```
> crossprod(u,v);
```

$$[bz - cy \quad cx - az \quad ay - bx]$$

Angle de deux vecteurs

```
> angle(u,v);
```

$$\arccos\left(\frac{ax + by + cz}{\sqrt{a^2 + b^2 + c^2}\sqrt{x^2 + y^2 + z^2}}\right)$$

Norme d'un vecteur

norm(u, n)

Elle est définie par

$|u| = (|a|^n + |b|^n + |c|^n)^{1/n}$

```
> norm(u,1);
```

$$|a| + |b| + |c|$$

```
> norm(u,2);
```

$$\sqrt{|a|^2 + |b|^2 + |c|^2}$$ (norme Euclidienne)

```
> norm(u,infinity);
```

$$max(|a|, |c|, |b|)$$ (norme de Tchebychev)

Attention, par défaut, la fonction norm retourne la norme de Tchebychev

VI - Fonctions

1. Déclaration

a) A l'aide d'une flèche

nom:= (seqvar) → expression

seqvar est une variable ou une séquence de variables. L'expression doit être valide et contenir les variables données dans la séquence. Toutes les autres variables sont considérées comme des paramètres.

Exemple :

```
> f:=(x,y)→a*x**2+b*y**2+2*c*x*y;
```

$$f := (x, y) \rightarrow ax^2 + by^2 + 2cxy$$

f est une fonction de x et de y. Les variables a, b, c sont des paramètres.

```
> g:=x→a*x**2+b*y**2+2*c*x*y;
```

$$g := x \rightarrow ax^2 + by^2 + 2cxy$$

Cette fois, g est une fonction de x tandis que y est considéré comme un paramètre.

b) Notation en crochet

nom:= <expression | var>

```
> w:=<exp(x**2+y**2)|x,y>;
```

$$w := \langle e^{(x^2+y^2)} \mid x, y \rangle$$

c) Transformation d'une expression en fonction

nom:= unapply (expression, seqvar)

Cette déclaration est intéressante lorsque l'expression qui définit la fonction existe auparavant.

```
> z:=unapply((x**2+y**2),x,y);
```

$$z := (x, y) \rightarrow x^2 + y^2$$

d) Définition à partir d'autres fonctions

Φ:= f*g	produit de fonctions
Φ:= f@g	fonction de fonction : Φ = f(g(x))

Exemples

```
> Y:=tan(x)*sin(x);
```

$$Y := \tan(x)\sin(x)$$

```
> diff(Y,x);
```

$$(1 + \tan(x)2)\sin(x) + \tan(x)\cos(x)$$

```
> Z:=tan(x)@sin(x);
```

$$Z := \tan(x)@\sin(x)$$

```
> diff(Z,x);
```

$$D_1(@)(\tan(x), \sin(x))\,(1 + \tan(x)^2) + D_2(@)(\tan(x), \sin(x))\cos(x)$$

e) Expression sous forme de procédure

```
nom:= proc (arguments);
    séquence d'instructions;
    RETURN (valeur de la fonction)
end;
```

La construction des procédures sera étudiée en détail ultérieurement.

2. Manipulation des fonctions

a) Valeurs particulières

Il suffit de donner le nom de la fonction et de valoriser correctement la liste de variables. Dans l'exemple précédent :

```
> f(0,1);
```

$$b$$

```
> f(1,1);
```

$$a + b + 2c$$

```
> f(u,v);
```

$$au^2 + bv^2 + 2cuv$$

b) Tracé d'une fonction

plot (fonc(var), var = deb..fin) ou bien plot (fonc, deb..fin)

On peut effectuer le tracé directement sur une expression (la conversion en fonction n'est pas nécessaire). Le tracé n'est possible que si tous les **paramètres ont reçu une valeur numérique et si le résultat du calcul de la fonction conduit à une valeur réelle**.

Une ou les deux bornes peuvent être infinies.

On peut également spécifier les bornes sur y ainsi que certaines options (voir l'aide en ligne et le cours sur les graphiques).

plot(fonc(var) = deb..fin, fonc = deb..fin, options)

Dérivée d'une fonction ou d'une expression

nom:= diff (expression, var) dérivée première
nom:=diff (expr, var$n) dérivée $n^{ème}$

Si la fonction comporte plusieurs variables, cette expression retourne la dérivée partielle simple ou multiple d'ordre n par rapport à la variable var.

Pour calculer les dérivées partielles par rapport à plusieurs variables, il faut faire figurer ces variables et éventuellement les ordres de dérivation dans le sens correct.

diff (expr, var1, var2)
diff (expr, var1$ordre1, var2$ordre2..)

Exemples :

```
> f:=(x,y)→a*x**2+b*y**2+2*c*x*y:
> diff(f(x,y),x);
```

$$2\,ax + 2\,cy$$

```
> diff(f(x,y),x$2):
```

$$2\,a$$

```
> diff(f(x,y),x,y);
```

$$2\,c$$

d) Intégration

int (fonction, variable)

Cette instruction fournit une primitive de la fonction. La fonction peut être une simple expression.

int (fonction, var = deb..fin)

fournit l'intégrale de la fonction entre les bornes deb et fin. Certaines bornes peuvent être infinies.

e) Limites d'une fonction

limit (fonc, var = borne, direction)

```
> limit(f(x,y),x=0);
```

$$b\,y^2$$

La borne peut être infinie.
La direction peut être left, right, real ou complex. Elle peut être omise dans le cas des fonctions continues puisque dans ce cas les limites à droite et à gauche sont égales.

f) Développements limités, développements asymptotiques

Le calcul peut être appliqué directement à une expression (la conversion en fonction n'est pas nécessaire).

taylor (expression, var = borne, ordre)

Le développement limité est calculé au voisinage de la borne (celle-ci peut être infinie).

Si l'ordre est omis dans l'instruction, il est fixé par la variable système Order. Cette variable vaut 6 par défaut, mais peut être modifiée par l'utilisateur.

Exemple :

```
> taylor(exp(x**2+y**2),x=0,3);
```

$$e^{(y^2)} + e^{(y^2)} x^2 + O(x^4)$$

Un développement asymptotique équivaut à un développement de Taylor au voisinage de l'infini, mais l'instruction asympt n'agit que sur les fonction.

asympt (fonct, var)

L'instruction series est plus générale que Taylor (series choisit la meilleure série possible (Taylor ou Laurent)).

order → l'ordre du développement limité
Order → variable système qui impose l'ordre des développements ultérieurs.

L'instruction convert (serie, polynom) est indispensable pour supprimer le reste du développement en série et pouvoir tracer la courbe.

Exemple

```
> Order:=9:

> taylor(tan(sin(x))-sin(tan(x)),x=0);
```

$$\frac{1}{30} x^7 + O(x^9)$$

$O(x^9)$ indique que les termes sont omis à partir de l'ordre 9. Les termes omis sont appelés le "reste" du développement en série.

VII - Equations, inéquations, systèmes

Ce chapitre traite de la syntaxe et de l'écriture des équations, leur résolution fera l'objet d'un chapitre spécifique.

1. Ecriture d'une équation algébrique

expression 1 = expression 2

Noter que l'on utilise le signe d'égalité. On peut affecter une équation à une variable (utilisation du **symbole d'affectation**).

eq:= expression 1 = expression 2

```
> eq:=x**2-6*x+3=0;
```

$$eq := x^2 - 6x + 3 = 0$$

2. Système d'équations algébriques

Un système d'équations doit obligatoirement être **affecté à un ensemble** (l'ordre des équations n'intervient pas au moment de la résolution).

Exemples :

```
> eq1:=x+y+z=0;
```

$$eq1 := x + y + z = 0$$

```
> eq2:=x-2*y=3;
```

$$eq2 := x - 2y = 3$$

```
> syst:={eq1,eq2, x**2-y**2=0};
```

$$syst := \{x + y + z = 0, x - 2y = 3, x^2 - y^2 = 0\}$$

3. Inéquations

Même syntaxe que pour les équations.

expression 1 > expression 2
expression 1 < expression 2

4. Equations différentielles

Il faut préciser la variable par rapport à laquelle est effectuée la dérivation.

diff (fonction, variable, $ordre) = expression (fonction, variable)

ou bien

expression (fonction, derivées, variable) = 0

Exemple

```
> ed:=diff(x(t),t)=-k*x(t)^2;
```

5. Systèmes différentiels, conditions initiales

Comme dans le cas des systèmes algébriques, il faut les rassembler dans un ensemble.

Exemples :

```
> ed1:=diff(x(t),t)=k1*(a-x(t));
```

$$ed1 := \frac{\partial}{\partial t} x(t) = k1(a - x(t))$$

```
> ed2:=diff(y(t),t)=k2*y(t)-k1*x(t);
```

$$ed2 := \frac{\partial}{\partial t} y(t) = k2\,y(t) - k1\,x(t)$$

On peut adjoindre les conditions initiales dans le même ensemble.

```
> cond1:=x(0)=0; cond2:=y(0)=0;
```

$$cond1 := x(0) = 0$$
$$cond2 := y(0) = 0$$

```
> sysd:={ed1} union {ed2} union {cond1} union {cond2};
```

$$sysd := \left\{ \frac{\partial}{\partial t} x(t) = k1(a - x(t)), \frac{\partial}{\partial t} y(t) = k2\,y(t) - k1\,x(t), x(0) = 0, y(0) = 0 \right\}$$

VIII - Chaînes de caractères

1. Utilité

L'utilité des chaînes de caractères n'est pas évidente dans un langage de calcul formel. Toutefois, le traitement des chaînes de caractères est indispensable dans les cas suivants :

- Manipulation de fichiers
- Lecture de données dans un fichier

Maple comporte toutes les opérations classiques sur les chaînes de caractères plus des opérations de conversions de chaînes en instructions et inversement.

2. Syntaxe

Une chaîne de caractères s'écrit entre quotes inverses. Dans les versions récentes on peut utiliser indifféremment les cotes inversées ou les doubles cotes

`` `ceci est un exemple` ou “ceci est un exemple” ``

Le caractère \ doit être doublé pour qu'il apparaisse comme caractère. Les blancs sont significatifs. La chaîne nulle s'écrit ``.

3. Opérations sur les chaînes

a) Concaténation

L'opérateur de concaténation est le "."

chaîne . objet

Objet est une autre chaîne, un nombre, une expression ou une liste :

```
> i:=5:
> p.i
```

p5

```
> p.(2*i+3)
```

p13

Il existe également une fonction réalisant la concaténation de deux ou de plusieurs objets.

cat(nom1, nom2, nom3)

La fonction cat est plus puissante que l'opérateur "." parce qu'elle peut agir sur des noms d'objets.

Exemple

```
> a:=`tri-`:
> b:=`nitro`:
> c:=`benzène`:
> a.b.c
```

abc

```
> cat(a,b,c)
```

tri-nitrobenzène

b) Conversion d'expressions en chaînes

convert (expression, string)

c) Conversion d'une chaîne en instruction

parse(chaîne)

Cette instruction transforme la chaîne de caractères en instruction Maple si elle est valide.

parse(chaîne, statement)

L'option statement déclenche l'exécution de l'instruction définie par la chaîne de caractères.

d) Extraction d'une sous-chaîne

substring(chaîne, début..fin)

début et fin indiquent les bornes de la sous-chaîne à extraire.

```
> x:= `champ`:
> y:= `pignon`:
> cat(x, substring(y,2..6))
```

champignon

e) Recherche d'une sous-chaîne dans une chaîne

SearchText(sous-chaîne, chaîne, domaine)
searchText(sous-chaîne, chaîne, domaine)

Cette fonction retourne le rang de la première occurrence de la sous-chaîne dans la chaîne testée (par défaut on retourne 0). Si on spécifie un domaine, la recherche est limitée à ce domaine.

SearchText effectue la recherche en dinstinguant les majuscules et les minuscules, tandis que l'instruction searchText ne fait pas cette distinction.

4. Longueur d'un objet Maple

length(objet)

L'objet peut être une chaîne, un nombre ou une expression. La fonction retourne le nombre de caractères dans la chaîne.

5. La structure TEXT

On peut rassembler dans une structure plusieurs lignes de texte.

TEXT(chaîne1, chaîne2, chaîne3, ...)

A l'impression, chaque chaîne figure sur une ligne (intéressant pour insérer du texte dans une image)