

Utilisation d'un logiciel de calcul formel: Maple

Raphaël Giromini

Table des matières

1 Premiers pas avec Maple.	4
1.1 Commandes de base.	4
1.2 Calculs numériques.	6
1.3 Calcul symbolique.	8
1.4 Quelques fonctions de base.	9
1.5 Représentations graphiques	9
1.6 Exercices.	9
2 Arithmétique.	11
2.1 Nombres complexes.	11
2.1.1 Représentation usuelle et représentation polaire.	12
2.2 Opérations d'arithmétique.	13
2.2.1 Calcul modulaire.	13
2.2.2 pgcd, ppcm et factorisation d'entiers.	13
2.2.3 Nombres premiers.	14
2.3 Calculs sur les polynômes.	15
2.3.1 La commande <code>factor(...)</code>	16
2.3.2 La commande <code>collect(...)</code>	16
2.3.3 Autres commandes sur les polynômes.	17
2.4 Équations.	19
2.4.1 Systèmes d'équations.	20
2.4.2 Solutions numériques.	21
2.4.3 Les commandes <code>isolve(...)</code> et <code>msolve(...)</code>	22
2.5 Exercices.	23
3 Graphiques	25
3.1 Graphiques en deux dimensions.	25
3.1.1 Introduction aux graphiques	26

3.1.2	La librairie "plots"	26
3.2	D'autres type de graphes en 2-d.	27
3.2.1	courbes paramétrées.	27
3.2.2	Champs de vecteurs en 2-d.	28
3.2.3	Fonctions implicites et graphes de points.	29
3.2.4	D'autres systèmes de coordonnées.	29
3.2.5	Inégalités.	31
3.2.6	Echelles logarithmiques.	31
3.3	Graphiques en trois dimensions.	32
3.4	Animations.	34
3.5	Exercices.	35
4	Fonctions, limites, dérivées et intégrales.	37
4.1	Fonctions.	37
4.1.1	Fonctions à plusieurs variables.	37
4.1.2	Fonctions composées.	38
4.1.3	Les Suites définies par des fonctions.	38
4.2	Limites.	38
4.3	Dérivées.	39
4.3.1	Dérivées partielles.	40
4.3.2	La commande D.	40
4.4	Intégrales.	41
4.5	Exercices	42
5	Type de données en langage Maple.	44
5.1	Quelques exemples de types.	44
5.2	Types de données fondamentaux.	46
5.2.1	Suites et listes.	46
5.2.2	Les ensembles.	52
5.2.3	Tableaux.	54
5.2.4	Les tables.	58
5.3	Exercices.	58
6	Algèbre linéaire.	60
6.1	Algèbre matriciel élémentaire.	60
6.1.1	Matrices et vecteurs en tant que listes.	60
6.1.2	Les commandes vector et matrix.	63
6.1.3	Opérations algébriques sur des matrices ou des vecteurs.	64
6.2	Operations diverses sur les vecteurs et sur les matrices.	66
6.2.1	Operations sur les vecteurs.	66
6.2.2	Opérations sur les matrices.	67

6.3	Solutions des systemes linéaires.	69
6.4	Exercices.	70
7	Introduction à la programmation.	72
7.1	Les Expressions logiques.	72
7.1.1	Connecteurs logiques.	72
7.1.2	Operateurs booléens et commandes conditionnelles. . .	74
7.2	Operations répétées et suites.	77
7.3	Introduction aux procedures.	79
7.4	Exercices.	81
8	Programmation.	83
8.1	Les suites.	83
8.1.1	Suites définie par une fonction.	83
8.1.2	Les suites définies par récurrence.	84
8.2	Procedures iteratives.	85
8.2.1	Un algorithme de tri.	85
8.2.2	La suite de Syracuse en itératif.	86
8.3	Procédures récursives.	86
8.4	Exercices.	87
9	Programmation avancée.	88
9.1	Lire et écrire dans des fichiers.	88
9.1.1	Sauver des fichiers avec Maple.	88
9.1.2	Sauver, écrire et lire des données dans un fichier	88
9.2	Conversion vers d'autres langages et génération de code. . . .	90
9.2.1	Conversion des commandes et procedures en Fortran et en C.	90
9.2.2	Conversions de Maple en L ^A T _E X	91

1 Premiers pas avec Maple.

1.1 Commandes de base.

À la fin de chaque commandes Maple, il y a un soit un point-virgule (;) soit deux points (:). Le point-virgule affiche le résultat du calcul, alors qu'avec un deux point, Maple effectue le calcul, mais ne l'affiche pas.

Si l'on ne mets rien à la fin d'une ligne et qu'on la valide (en appuyant sur la touche [entrée]), Maple affiche alors un **warning** («attention» en anglais) signalant une fin prématurée de la commande.

Si l'on appuie sur [entrée] à la fin d'une ligne, Maple essaie de calculer ce qu'on a écrit. On peut cependant écrire plusieurs lignes de calcul en appuyant à la fin d'une ligne simultanément sur les touches [shift] et entrée (ce que l'on note dans la littérature par [shift]+[entrée]).

```
> 1+2;  
25*34:
```

3

Rappel du dernier calcul. Dans le langage Maple, le caractère pourcent (%) permet de rapeler la valeur du dernier calcul effectué par la machine. Ce n'est pas nécessairement le dernier calcul affiché. De même, %% permet de revenir deux calculs en arrière et %%% permet de rapeler...3 calculs.

```
> 100! :  
50! :  
length(%);  
length(%%);
```

65

82

Ici, le premier calcul effectué par Maple est 100! (qu'il n'affiche pas, puisqu'il est suivi par deux-points). Le deuxième calcul est 50! (qu'il n'affiche pas). Ensuite Maple calcul la longueur (le nombre de chiffres, commande

`length(...)` du dernier résultat qu'il a calculé (donc $50!$). Une fois cela effectué, le dernier résultat calculé est la longueur, l'avant-dernier est $50!$ et l'anté-penultième est $100!$. lorsqu'on lui demande `'length(%%%)'`, il va chercher la longueur de l'antépenultième résultat (donc la longueur de $100!$).

Multiplication. Il faut toujours mettre une étoile (*) entre les facteurs.

```
> 5*4;  
5*x;
```

20

$5x$

Ici, si l'on avait tapé $5x$ à la place de $5 * x$, Maple aurait retourné un 'warning' signalant qu'il manque un opérateur ou un point-virgule. Notez que lorsque Maple affiche un 'warning', le curseur se place à l'endroit où l'erreur à été rencontrée.

Aide en ligne. Maple dispose d'une aide en ligne (très pratique lorsqu'on a oublié comment se servir d'une fonction ou d'une commande). Celle-ci s'obtient en tapant un point d'interogation avant le nom de la fonction. Notez qu'il n'y a pas de point-virgule après cette commande.

```
> ?sin
```

Ré-initialisation. Lorsqu'on veut ré-initialiser une feuille de calcul, il suffit de se servir de la commande `"restart;"`.

```
> restart;
```

Commentaires. Lorsqu'on veut écrire un commentaire dans ses calculs, on écrit une dièse (#) suivi du commentaire. Tout ce qui sera après la dièse ne sera pas pris en compte dans l'évaluation par Maple de la feuille de calcul.

```
> # Ceci est un commentaire.  
sin(x) + cos(y); # Ceci est un autre commentaire.  
# sin(x); <- c'est un commentaire, ça ne sera pas évalué.
```

$$\sin(x) + \cos(y)$$

Attribution des variables. Pour attribuer une valeur à une variable, on se sert de la commande `:=` (comme dans le langage Pascal). Par exemple, supposons que l'on veuille donner à x la valeur de 5, on procède comme suit :

```
> x:=5;
  x^2; # Quel est le carré de x?
```

$$x := 5$$

$$25$$

Jusqu'à la fin des calculs, x sera égal à 5 (sauf si on ré-initialise la feuille de calcul, par la commande `restart`). Pour supprimer la valeur de x , on se sert de la commande `unassign`, comme dans l'exemple suivant :

```
> unassign('x');
```

Je vous encourage à lire l'aide en ligne sur cette commande, dont l'utilisation n'est pas si triviale que ça.

1.2 Calculs numériques.

La commande `evalf(...)`, qui signifie «evaluate floating» permet d'évaluer une approximation numérique des calculs effectués.

```
> 5/3;
  evalf(%);
```

$$\frac{5}{3}$$

$$1.666666667$$

Maple n'aime pas les calculs numériques et préfère donner les résultats dans leur forme la plus formelle.

```
> 1/2 + 3/10;
```

$$\frac{4}{5}$$

Cependant, si l'on ajoute un point (.) à la fin d'un nombre, Maple pense alors que le nombre en question est un réel (et non plus un entier ou un rationnel) et donnera le résultat sous forme d'un nombre réel.

Lorsqu'on écrit 1., cela signifie pour Maple le nombre 1.0.

```
> 1./2 + 3/10;
```

$$.8000000000$$

Pour parler plus mathématiquement, Maple calcul un résultat dans le plus petit corps contenant les coefficients du calcul¹.

Notez que Maple dispose de certaines constantes en son sein, telle que les nombres π , $\ln(2)$, $\exp(1)$. Pour obtenir des approximations numériques de ces constantes, on se sert de la fonction `evalf(...)`.

```
> evalf(Pi);  
ln(2);  
evalf(%);  
exp(1);  
evalf(%);
```

$$3.141592654$$
$$\ln(2)$$
$$.6931471806$$
$$e$$

¹Si cette dernière phrase vous semble obscur, ce n'est pas grave, loin de là.

2.718281828

Notez que le nombre e (c'est à dire le nombre tel que $\ln(e) = 1$) ne s'écrit pas e , mais $\exp(1)$. De même, la «racine carré de -1 » ne s'écrit pas "i", mais "I" («i majuscule»).

Il existe une commande Maple pour fixer le nombre de chiffre à afficher, il s'agit de la commande `Digits`. Par défaut, Maple affiche 15 chiffres au total (partie entière et partie décimale).

```
> sin(0.2);  
Digits:=3;  
sin(0.2);
```

.1986693308

Digits := 3

.199

1.3 Calcul symbolique.

Pour développer une expression on utilise la commande `expand(...)`, pour la simplifier, on utilise la fonction `simplify(...)` et pour la factoriser, on utilise la commande `factor(...)`.

```
> (1+x)^2;  
expand(%)  
factor(%)  
simplify(cos(x)^2 + sin(x)^2);
```

$(1 + x)^2$

$1 + 2x + x^2$

$(1 + x)^2$

1

1.4 Quelques fonctions de base.

Maple connaît un grand nombre de fonctions mathématiques de base (certainement beaucoup plus que vous même en connaissez) et il est possible d'évaluer ces fonctions à un point donné.

```
> ln(1);  
sin(Pi/4);  
ln(Pi);  
evalf(%);
```

0

$\frac{1}{2}\sqrt{2}$

$\ln(\pi)$

1.144729886

1.5 Représentations graphiques

Maple permet également de tracer n'importe qu'elle fonction, aussi bien sur le plan que dans l'espace.

```
> plot( x^2, x=-5..5);  
plot3d( x * sin(y), x=-5..5, y=-5..5 );
```

Nous verrons plus en détail comment tracer des graphiques, dans un prochain cours.

1.6 Exercices.

Exercice 1.

1. Évaluer le cosinus de $\frac{3\pi}{5}$, en affichant 5 chiffres.
2. Évaluer $\cos(\ln(3) \sin(\frac{3\pi}{7}) + 5)$, en affichant 7 chiffres.

Exercice 2.

1. Simplifier l'expression suivante :

$$(\tan(a + b) + \tan(a - b))(\cos(b)^2 + \cos(a)^2 - 1)$$

2. Développer l'expression $\sin(2a)$. Y-a-t'il un rapport entre les deux ?

3. Développer l'expression suivante $\frac{\cos(a-b) - \cos(a+b)}{2}$.

Exercice 3.

Tracer la courbe représentative de $5x^2 + 3x - 5$ pour x variant entre -10 et 5 .

2 Arithmétique.

2.1 Nombres complexes.

Maple sait faire à peu près n'importe quel calcul sur les nombres complexes. En langage Maple, "I" («i majuscule») désigne «une racine carré de -1 », c'est à dire le nombre tel que $I^2 = -1$.

```
> (2+5*I) + (1-I) + 5*(3+6*I);  
(4+6*I) * (7 - 9*I);  
(1+I)/(3-2*I);  
(2+8*I)^2;
```

$$18 + 34I$$
$$82 + 6I$$
$$\frac{1}{13} + \frac{5}{13}I$$
$$-60 + 32I$$

Notez que Maple décompose toujours le résultat sous la forme habituelle $a + ib$. De plus, Maple possède des commandes spécifiques aux calculs sur les nombres complexes :

- La plus usuelle est la commande `evalc(...)`, qui simplifie une expression complexe sous la forme $a + ib$.
- La commande `Re(...)` donne la partie réelle d'une expression complexe.
- La commande `Im(...)` donne la partie imaginaire d'une expression.
- La commande `abs(...)` donnera le module d'un complexe.
- La commande `argument(...)` donnera son argument.
- La commande `conjugate(...)` donnera son conjugué.

```
> z:= 2 + 3 * I;  
Re(z);  
Im(z);  
abs(z);  
argument(z);  
conjugate(z);  
unassign('z');
```

$$z := 2 + 3I$$

$$2$$

$$3$$

$$\sqrt{13}$$

$$\arctan\left(\frac{3}{2}\right)$$

$$2 - 3I$$

2.1.1 Représentation usuelle et représentation polaire.

Un complexe peut également être représenté sous la forme ρe^{θ} , où ρ est son module et θ son argument.

Pour avoir une telle représentation, on utilise la commande `polar(ρ, θ)`.

Pour mettre un nombre de la forme $a + ib$ en forme polaire, on utilise la commande `convert(..., polar)`.

```
> polar(2, Pi/4);
convert(3-I*4,polar);
```

$$\text{polar}\left(2, \frac{1}{4}\pi\right)$$

$$\text{polar}(\sqrt{17}, -\arctan(4))$$

Réciproquement, pour passer de la forme polaire à la forme «usuelle», on utilise la commande `evalc(...)`, que nous avons déjà vu.

```
> polar(2, Pi/4);
evalc(%);

convert(-5+12*I,polar);
evalc(%);
```

$$\begin{aligned} & \text{polar} \left(2, \frac{1}{4}\pi \right) \\ & \sqrt{2} + I\sqrt{2} \\ & \text{polar} \left(13, -\arctan \left(\frac{12}{5} \right) + \pi \right) \\ & -5 + 12I \end{aligned}$$

2.2 Opérations d'arithmétique.

2.2.1 Calcul modulaire.

```
> 3+22 mod 4;
1/23 mod 7;
modp(14,5);
mods(14,5);
```

1
1
4
-1

Les commandes `mods(...)` et `modp(...)` donnent les représentations symétriques et positives d'un calcul modulaire.

2.2.2 pgcd, ppcm et factorisation d'entiers.

le Plus Grand Commun Diviseur (pgcd) de deux, trois, ou plus, nombres entiers se calcule à l'aide de la commande `igcd(...)` (pour «Integer Greater Common Divisor»). Le Plus Petit Commun Multiple (ppcm) de deux, trois ou plus, nombres entiers se calcule grâce à la commande `ilcm(...)` (pour «Integer Less Common Multiple»).

```
> igcd(24,4);
igcd(6160,8372,56);
```

```
ilcm(38,341);  
ilcm(6,9,18);
```

```
4  
28  
12958  
18
```

La commande `ifactor(...)` permet de factoriser des nombres entiers en facteurs premiers². Notez que cette commande a un certain nombre d'options qui permettent de choisir la méthode de factorisation. Regardez l'aide-en-ligne à ce sujet.

```
> ifactor(2850375285);
```

```
(3)3(5)(631)(33461)
```

2.2.3 Nombres premiers.

Un nombre premier est un nombre entier dont les diviseurs positifs sont 1 et par lui-même. Par convention, 1 n'est pas premier. 24 n'est pas premier car il est divisible (au moins) par 2. 23, par contre, est un nombre premier.

La commande `ithprime(...)` permet d'afficher le *i*-ème nombre premier. Notez que le premier nombre premier est 2. Pour savoir si un nombre est premier, on se sert de la commande `isprime(...)`.

Pour connaître le plus petit nombre premier qui succède un nombre donné, on se sert de la commande `nextprime(...)`.

```
> isprime(12);  
isprime(35532119);  
ithprime(100);  
ithprime(1000);  
nextprime(2000);
```

²le «i» étqnt là pour «integer», nombre entier.

false

false

541

7919

2003

2.3 Calculs sur les polynômes.

Nous allons commencer par travailler sur les expressions symboliques et par les plus simples d'entre-elles : les polynômes.

Voici un certains nombres de calculs élémentaires sur les polynômes :

```
> P1:= 3 * x^2 + 4 * x + 7; # On assigne à P1 un polynôme.  
P2:= x^3 -1; # On assigne à P2 un polynôme.  
P1^2; # On calcule le carré de P1.  
P1/P2; # On calcule le rapport de P1 / P2.  
eval(P1, x=2); # On évalue P1 au point x=2.  
eval(P2, x=-4); # On évalue P2 au point x=-4.
```

$$P1 := 3x^2 + 4x + 7$$

$$P2 := x^3 - 1$$

$$(3x^2 + 4x + 7)^2$$

$$\frac{3x^2 + 4x + 7}{x^3 - 1}$$

$$27$$

$$-65$$

2.3.1 La commande `factor(...)`.

La commande `factor(...)` permet de factoriser un polynôme.

```
> factor(21*x^3+151*x^2+324*x+180);
```

$$(3x + 10)(7x + 6)(x + 3)$$

La factorisation se fait dans le plus petit corps qui contient les coefficients du polynôme. Étudiez la différence entre les deux expressions suivantes :

```
> factor(x^2 - 2);  
factor(x^2 - 2.0);
```

$$x^2 - 2$$

$$(x + 1.414213562)(x - 1.414213562)$$

Cette commande permet également de factoriser un polynôme à plusieurs variables.

```
> factor(x^3-y^3);
```

$$(x - y)(x^2 + xy + y^2)$$

2.3.2 La commande `collect(...)`.

La commande `collect(...)` permet de grouper les termes d'une expression suivant une variable donnée.

```
> collect(a*x^3 + y*x - (x^2+1)*(y-2)+4 , x);  
collect(a*x^3 + y*x - (x^2+1)*(y-2)+4 , y);
```

$$ax^3 + (-y + 2)x^2 + xy + 6 - y$$

$$(x - x^2 - 1)y + ax^3 + 2x^2 + 6$$

Bien sûr, on peut factoriser une expression suivant plusieurs variables, en respectant l'ordre des variables.

```
> collect(a*x^3 + y*x - (x^2+1)*(y-2)+4, [x,y]);
collect(a*x^3 + y*x - (x^2+1)*(y-2)+4, [y,x]);
```

$$ax^3 + (-y + 2)x^2 + xy + 6 - y$$

$$(x - x^2 - 1)y + ax^3 + 2x^2 + 6$$

2.3.3 Autres commandes sur les polynômes.

`normal(...)`. La commande `normal(...)` permet de mettre des fractions de polynômes au même dénominateur.

```
> 1/(1+2*x^2) + 2*x/(1+x) + 3*x^2/(4-x^3) - 2*x^3/(3+x+x^4);
normal(%);
```

$$\frac{1}{1+2x^2} + 2\frac{x}{x+1} + 3\frac{x^2}{4-x^3} - 2\frac{x^3}{3+x+x^4}$$

$$\frac{-12 - 40x - 21x^2 - 23x^4 + 17x^6 - 17x^5 - 7x^8 - 16x^7 - 10x^9 + 4x^{10} - 49x^3}{(1+2x^2)(x+1)(-4+x^3)(3+x+x^4)}$$

`simplify(...)`. La commande `simplify(...)` permet de simplifier une expression. Attention cependant, cette commande ne donne pas toujours le résultat voulu (Maple oublie parfois certaines simplifications «évidentes», mais c'est relativement rare).

```
> (x^5-3*x^3-x^4+2*x+4) / (x^4+x-2*x^3-2);
simplify(%);
```

$$\frac{x^5 - 3x^3 - x^4 + 2x + 4}{x^4 + x - 2x^3 - 2}$$

$$\frac{x^4 + x^3 - x^2 - 2x - 2}{x^3 + 1}$$

`sort(...)`. La commande `sort(...)` permet de classer les coefficients d'un polynôme suivant la valeur croissante des puissances. Si l'on a un polynôme à plusieurs variables, on peut bien sûr préciser l'ordre dans lequel on veut classer le polynôme.

```
> # Un polynôme à une seule variable.
poly:=x+x^2 - x^3+7 - x^6 + x^22:
sort(poly); # On trie poly.

# Un polynôme à plusieurs variables.
poly2:= x*y^3 - 2*x^2*y^5 + 4*x*y^4 + y^3:
sort(poly2,[x,y]); # On trie poly2 suivant x puis y.
sort(poly2,[y,x]); # On trie poly2 suivant y puis x.
```

$$x^{22} - x^6 - x^3 + x^2 + x + 7$$

$$-2x^2y^5 + 4xy^4 + xy^3 + y^3$$

$$-2y^5x^2 + 4y^4x + y^3x + y^3$$

Degrés et coefficients. La commande `degree(...)` permet d'obtenir le degré d'un polynôme. Si le polynôme a plusieurs variables, on peut préciser suivant quelles variable on veut le degré. Si l'on ne précise pas la variable, on obtient le degré total de l'expression.

```
> P:= 4*x^5 + 3*x^4 - 8*x + 2;
degree(P);

Q:= 4*y^2*x^4 + 12*y^5 + 7*x^2*y^2 + 7;
degree(Q,x);
degree(Q,y);
degree(Q);
```

$$P := 4x^5 + 3x^4 - 8x + 2$$

5

$$Q := 4y^2x^4 + 12y^5 + 7x^2y^2 + 7$$

4

5

6

La commande `coeff(...)` permet d'obtenir la valeur d'un coefficient du polynôme considéré. La commande `lcoeff(...)` permet d'obtenir le coefficient du terme de plus haut degré³.

```
> P:= 4*x^5 + 3*x^4 - 8*x +2;
coeff(P, x^4);
coeff(P, x^2);
lcoeff(P);
```

$$P := 4x^5 + 3x^4 - 8x + 2$$

3

0

4

2.4 Équations.

Maple sait résoudre presque n'importe quelle équation. La principale commande pour résoudre une équation est `solve(...)`.

```
> eqn1:= x^2 - 4 = 0;
solve(eqn1, x);
solve(eqn1, {x});
```

³le «l» initial étant là pour «lead»

$$\begin{aligned} \text{eqn1} &:= x^2 - 4 = 0 \\ &2, -2 \\ &\{x = 2\}, \{x = -2\} \end{aligned}$$

Notez que si l'on met l'inconue entre accolades (`{ et }`), alors Maple précise de quelle variable on parle.

On peut même résoudre des équations plus théoriques, par exemple la solution générale d'un trinôme du second degré (pensez à préciser les inconues lorsque vous travaillez avec des expressions théoriques, sinon Maple peut réserver de drôles de surprises...)

```
> # Une équation.
eqn := a * x^2 + b * x + c = 0;

solve(eqn, x);          # La solution générale du trinôme.
solve(eqn)              # Attention...
```

$$\begin{aligned} \text{eqn} &:= ax^2 + bx + c = 0 \\ &\frac{1-b+\sqrt{b^2-4ac}}{2} \frac{1-b-\sqrt{b^2-4ac}}{2} \\ &\{x = x, c = -ax^2 - bx, a = a, b = b\} \end{aligned}$$

2.4.1 Systèmes d'équations.

```
> solve({x+2*y=3, x+y =1 }, {x,y});
```

$$\{y = 2, x = -1\}$$

Maple ne se limite pas à la résolution des systèmes linéaires, mais bien au contraire, il sait résoudre des systèmes nettement plus compliqués.

Parfois, plutôt que de calculer une solution, Maple donne un résultat en `RootOf(...)`. Si l'on veut toutes les solutions, il suffit de se servir de la commande `allvalues(...)`.

```
> # Un systeme un peu plus complexe:
solve({x^2+y=1,x^2-y^2=2},{x,y});
# On veut toutes les solutions.
allvalues(%);
```

$$\{x = \text{RootOf}(-\text{RootOf}(-3_Z + _Z^2 + 3) + _Z^2), \\ y = -\text{RootOf}(-3_Z + _Z^2 + 3) + 1\}$$

2.4.2 Solutions numériques.

La commande `fsolve(...)` permet d'obtenir des solution numériques⁴.

Pour une équation qui a plusieurs solutions (les racines d'un polynôme, par exemple), `fsolve(...)` ne donnera pas obligatoirement toutes les solutions⁵.

```
> poly1:=3*x^4-16*x^3-3*x^2+13*x+16; # Un polynome.
fsolve(poly1,x); # Ses racines.

poly2:=23*x^5+105*x^4-10*x^2+17*x; # Un autre polynome.
fsolve({poly2},{x}); # Ses racines
```

$$poly1 := 3x^4 - 16x^3 - 3x^2 + 13x + 16$$

$$1.324717957 \quad 5.333333333$$

$$poly2 := 23x^5 + 105x^4 - 10x^2 + 17x$$

$$x = -4.536168981, x = -.6371813185, x = 0$$

⁴le «f» de «fsolve» signifie «floating», comme dans la commande «evalf», que nous avons vu la semaine dernière.

⁵Dans le cas des racines d'un polynôme, 'fsolve' ne donne que les racines réelles.

`fsolve(...)` a beaucoup d'option⁶. Par exemple, on peut spécifier dans quel interval on veut la solution, ou bien spécifier que l'on veut les racines complexes.

```
> poly:=3*x^4-16*x^3-3*x^2+13*x+16; # un polynome.
fsolve(poly, x, 4..8); # Ses racines entre 2 et 8.
fsolve(poly, x, complex); # Ses racines complexes.
```

$$poly := 3x^4 - 16x^3 - 3x^2 + 13x + 16$$

$$5.333333333$$

$$-.6623589786 - .5622795121I - .6623589786 + .5622795121I$$

$$1.324717957 5.333333333$$

2.4.3 Les commandes `isolve(...)` et `msolve(...)`.

Ces commandes sont utiles lorsqu'on cherche des solutions entières de certaines équations (pour la relation de Bezout, par exemple).

La commande `isolve(...)` permet de donner des solutions entières d'une équation. La commande `msolve(..., m)` permet de donner les solutions d'une équation modulo un certain entier m .

```
> # Solutions entieres de 5x - 3y = 4.
isolve({5*x-3*y=4});

# Solutions modulo 17 d'un système.
msolve({x+7*y=1,3*x+2*y=2},17);
```

$$\{x = 2 + 3_N1, y = 2 + 5_N1\}$$

$$\{y = 9, x = 6\}$$

⁶Regardez l'aide sur cette commande, via `?fsolve`

2.5 Exercices.

Exercice 4.

1. Donnez une représentation polaire de $z_1 = 6 + i\pi$.
2. Donnez les parties réelles et imaginaires de $z_2 = (3 - 6i)^{3i}$.
3. Soient $z_3 = i - 1$ et $z_4 = 2i$. Donnez les modules et arguments de z_3 et z_4 .
4. Mettez $\frac{z_3}{z_4}$ sous forme algébrique, puis sous forme polaire.
5. (facultative) : Donnez les module et argument de z_2 . Cela vous semble-t-il logique ? Trouvez un moyen de contourner le «problème.»

Exercice 5.

1. Soit $a = 1049427$ et $b = 17493$. Quel est le pgcd de a et b ? Ce pgcd est-il un nombre premier ?
2. Décomposez a et b en nombres premiers. Quelle doit être la décomposition du ppcm en nombre premier ?
3. Calculez le résultat de $773^{3570} \pmod{3571}$, de $123^{3570} \pmod{3571}$. Soit n un nombre entier entre 1 et 3570. Quel doit être le résultat de $n^{3570} \pmod{3571}$?
4. Quel est le 500^è nombre premier ? Quel est le nombre premier qui succède 40000 ?
5. (facultative) Donnez la décomposition en facteur premier de $200!$ D'après vous, par combien de zéro le nombre $200!$ se termine-t-il ?

Exercice 6.

1. Factorisez les deux polynômes suivants : $P_1 = x^4 - 1$ et $P_2 = x^3 + x^2 - 5x - 5$. Que vaut $\frac{P_1}{P_2}$?
2. Donnez le nom de 'poly' à l'expression suivante :

$$(x^2 - a)(xy + b)(x^4 - y^4)(3x + z)(z - x)$$

Developpez, simplifiez et ordonnez ce polynôme, suivant x , puis suivant y et enfin suivant z .

3. Quelle est la valeur du coefficient de x^5 dans l'expression précédente ? Même question pour y^3 .

Exercice 7.

Determinez toutes les solutions de l'équation suivante : $\exp(x) - 3x = 0$.

Exercice 8.

1. Résoudre le système suivant :

$$\begin{cases} x + 2y + 3z + 4t + 10 = 41 \\ 8x + 4z + 3t + 4 = 11 \\ x + y + z + t + 2 = 9 \\ 3y + 4z - 8t + 4 = 125 \end{cases}$$

2. Résoudre le système suivant :

$$\begin{cases} x + 2y + 3z + 4t + 10 = 41 \\ 8x + 4z + 3t + 4 = 11 \\ x + y + z + t + 2 = 9 \end{cases}$$

3. Résoudre le système suivant :

$$\begin{cases} x + y + 2z = 2 \\ 2x + 3y + z = 4 \\ x - y + 5z = 7 \end{cases}$$

4. Trouvez toutes les solutions du système suivant :

$$\begin{cases} x^2 + y^2 = 3 \\ x^2 + 2y^2 = 3 \end{cases}$$

Exercice 9.

1. Trouvez une solution réelle strictement positive à l'équation

$$\tan(x) - x = 0$$

2. Trouvez une solution approximative de l'équation

$$\tan(\sin(x)) = 1$$

lorsque x varie entre $-\pi$ et π

3 Graphiques

Maple dispose de nombreuses commandes et fonctions pour tracer des fonctions, des graphiques ou des animations en deux ou trois dimensions.

La syntaxe générale pour tracer un graphique avec Maple est la suivante :

$$\text{plot_type}(expression, intervalle, options)$$

Où :

- `plot_type` est le type de graphique que l'on veut tracer.
- *expression* est une expression mathématique, où un ensemble d'expressions définissant la (ou les) fonction(s) à tracer.
- *intervalle* est de la forme ' $x=a..b$ ' avec a et b des nombres réels et $a < b$.
Ce peut également être une liste d'intervalles de la même forme.
- *Option* correspond à une ou plusieurs spécifications optionnelles qui déterminent l'apparence du graphe (tels que le titre, les échelles, etc.).

La commande de base pour tracer un graphique est la commande '`plot`'. Cependant, de nombreuses commandes existent pour tracer différents types de graphiques. Il existe même une librairie, appelée '`plots`' pour tracer certains graphiques.

Pour pouvoir charger cette librairie de fonction en mémoire, il faut taper la commande '`with(plots) :`'. Si l'on remplace le deux-point final par un point virgule, Maple va afficher l'ensemble des nouvelles fonctions et commandes qu'il a chargé en mémoire.

3.1 Graphiques en deux dimensions.

Le type le plus simple de graphiques en deux dimensions est obtenu par une expression de la forme :

$$\text{plot}(f(x), x=a..b, y=c..d, opt1, opt2, \dots)$$

Où les termes en italiques sont optionnels. $f(x)$ est une fonction dépendante de la variable réelle x et le domaine où l'on va tracer la fonction est dénoté par $x=a..b$ (avec $a < b$). Nous verrons les options les plus souvent utilisées. Pour les autres options vous pouvez regarder l'aide, via la commande '`?plot[options]`'. Voici un exemple (le plus simple possible) pour tracer un graphique.

```
> plot(x^2-3,x=-4..4);
```

3.1.1 Introduction aux graphiques

Lorsqu'on essaie de tracer une fonction discontinue⁷, Maple va essayer de joindre les points de discontinuité, sauf si on lui précise de ne pas le faire, via l'option 'discont=true'.

Par défaut, Maple calcule un minimum de 50 points par graphe, mais il peut-être utile d'augmenter ce nombre de points, via l'option 'numpoints'.

L'option 'scaling=constrained' permet de spécifier si l'on veut la même échelle sur les deux axes du graphe. Attention, par défaut ce n'est pas le cas !

```
> # Une fonction discontinue.
plot(x-floor(x),x=-3..3,scaling=constrained);
plot(x-floor(x),x=-3..3,discont=true,scaling=constrained);

# Une fonction d'amplitude très petite.
plot(x^3*sin(exp(1/x)),x=(.2)..(.3));
plot(x^3*sin(exp(1/x)),x=(.2)..(.3), numpoints=500);
```

Bien entendu, on peut tracer plusieurs graphes à la fois et grâce à la seconde notation, on peut imposer une couleur particulière à chaque graphe.

```
> plot({x^2-1,sin(x)},x=-Pi..Pi,y=-1..1);
plot([x^2-1,sin(x)],x=-Pi..Pi,y=-1..1);
plot([x^2-1,sin(x)],x=-Pi..Pi,y=-1..1, color=[green,blue]);
```

De manière générale, plusieurs courbes peuvent être tracées simultanément avec la commande plot, avec différentes options tels que la couleur, le style etc. pour chacune des courbes. La syntaxe générale est alors :

```
> plot([expr1,expr2, ... , exprn], fenetre,
      color=[c1, c2, ...], style=[s1,s2, ... ], ... )
```

3.1.2 La librairie "plots"

Un autre moyen d'afficher plusieurs graphes en même temps est d'utiliser la commande 'display(...)'. Cette commande fait partie de la librairie 'plots'. Ainsi, cette commande peut être utilisée soit sous la forme 'plots[display]' soit en chargeant en mémoire la librairie en entier, via la commande 'with(plots)'.

⁷Une fonction continue est une fonction «que l'on peut tracer à la main sans lever le crayon.» Vous verrez une définition plus rigoureuse en deuxième année de Deug.

Comme nous allons étudier de nombreuses fonctions de cette librairie, nous choisissons ici de la charger en entier.

```
> # On ne charge que la commande display en mémoire.
with(plot,display):

# Deux graphes (pas affichés).
plot1:=plot(sin(x),x=-2*Pi..2*Pi,color=red):
plot2:=plot(cos(x), x=-2*Pi..2*Pi,color=green):

# Affichage des deux graphes.
display([plot1,plot2]);
```

On peut aussi placer du texte à un endroit précis sur un graphe, en utilisant la commande `textplot`.

```
> a:=plot(sin(x),x=-Pi..Pi):
b:=textplot([Pi/2,1,'Maximum local'],font=[TIMES,ITALIC,12]):
c:=textplot([-Pi/2,-1,'Minimum local'],
            font=[TIMES,ITALIC,12]):
display([a,b,c],title="f(x) = sin(x)",
        titlefont=[HELVETICA, BOLD, 18]);
```

3.2 D'autres type de graphes en 2-d.

3.2.1 courbes paramétrées.

Nous allons voir comment tracer une courbe qui est définie paramétriquement. La syntaxe pour tracer une courbe définie paramétriquement par le vecteur $(f(t), g(t))$ est : `'plot([f(t), g(t), t=a..b])'`

Notez que les crochets ('[' et ']') sont autour de toute l'expression. C'est la seule particularité qui distingue une expression paramétrique d'un graphe normal.

```
> # On trace la courbe de vecteur (sin(t), cos(t)).
plot([sin(t),cos(t),t=0..2*Pi]);
```

Voici ce que donne le graphique de la courbe paramétrée définie par $(f(t), g(t))$, où $-\pi \leq t \leq \pi$ et

$$\begin{aligned} f(t) &= (\sin(t) - \cos(t))^3 - \sin(t) + \cos(t) \\ g(t) &= \cos(t)^2 - \sin(t)^2 \end{aligned}$$

```
> plot([(sin(t)-cos(t))^3-(sin(t)-cos(t)),
        cos(t)^2-sin(t)^2,t=-Pi..Pi],
        -2..2,-1..1,scaling=constrained); # Les options.
```

On peut tracer plusieurs courbes paramétrées dans le même graphe. De même, les courbes paramétrées et non-paramétrées peuvent être représentées sur un seul graphe.

```
> # Plusieurs courbes paramétrées sur un même graphe.
plot([[t^2,t,t=-1..1], # Courbe 1
      [2*sin(t),cos(t),t=0..2*Pi]], # Courbe 2
      scaling=constrained,color=[red,green]); # Les options

# Une fonction et une courbe paramétrée.
plot([
      (x-x^3)/4, # Une fonction.
      [(sin(t)-cos(t))^3-(sin(t)-cos(t)), # Une courbe
       cos(t)^2-sin(t)^2, t=-Pi..Pi] # paramétrée.
],
      x=-2..2, color=[red,blue]); # Les options.
```

3.2.2 Champs de vecteurs en 2-d.

En physique, il peut être utile de tracer un champs de vecteur (par exemple le champ de vecteur d'une force magnétique, électrique ou gravitationnelle). Maple peut tracer un champs dont les composantes sont de la forme $(f(x,y), g(x,y))$. La commande pour tracer un tel champs est définie par : 'fieldplot([f(x,y), g(x,y)], x=a..b, y=c..d, options)' («field» signifie champs en anglais).

Notez que la commande 'fieldplot(...)' fait partie de la librairie 'plots', attention donc lors de l'utilisation de cette commande.

```
> # Un champs tournant.
fieldplot([y,-x],x=-10..10,y=-10..10,arrows=SLIM);
```

L'option 'arrows=SLIM', qui a été utilisée ici, permet de spécifier le type de flèche («arrow» dans la langue de Dickens). Les autres types de flèches sont : 'LINE' («ligne»), 'THIN' («maigre») and 'THICK' («épais»). Par défaut c'est l'option 'THIN' qui est utilisée.

3.2.3 Fonctions implicites et graphes de points.

Fonctions implicites. Les fonctions implicites en deux dimensions sont les fonctions définies par une relation de la forme $f(x, y) = 0$. Elles peuvent être tracées en utilisant la commande 'implicitplot(...)'. Encore une fois, cette commande fait partie de la librairie 'plots', et subit donc les mêmes restrictions que d'habitude.

```
> # Une conique.  
implicitplot(x^2 / 25 + y^2 / 9 = 1,  
             x=-6..6,y=-6..6,scaling=CONSTRAINED);
```

Graphes de points. La commande plot peut aussi être utilisée pour relier des points entre eux, avec la syntaxe suivante : 'plot([[a1,b1], [a2,b2], ..., [an,bn]])', où [a1, b1], [a2, b2],... représente les deux composantes d'un point en coordonnées cartésiennes. En mettant [a1,b1] = [an,bn], on peut fermer la courbe ainsi obtenue.

Si l'on ne veut que représenter les points sans les rejoindre, il suffit d'utiliser l'option 'style=point' comme dans le second exemple.

```
> # Une courbe fermée.  
plot([[[-12,-1],[20,7],[21,3],[-11,-5],[-12,-1]]]);  
  
# Une série de points.  
plot([[[-12,-1],[20,7],[21,3],[-11,-5],[-12,-1]],  
      style=point);
```

3.2.4 D'autres systèmes de coordonnées.

En dehors des coordonnées cartésiennes, On peut également tracer des graphes dans d'autres coordonnées, telles que polaires, hyperboliques, elliptiques, etc. En ajoutant l'option 'coords=type'. Je vous encourage à regarder l'aide à ce sujet, via les commande '?plot[coords]' ou '?coords'.

Graphes en coordonnées polaires. Avant d'expliquer ce que sont les coordonnées polaires, il faut bien comprendre ce que sont les coordonnées cartésiennes. On se donne un repère de centre O et un point M du plan. En projetant ce point orthogonalement sur les axes du repère, on obtient ses coordonnées (que l'on notera x et y).

On se donne alors une fonction f qui depend de la variable x . Une representation de cette fonction, dans un tel repère est de la forme $y = f(x)$.

Dans un système de coordonnées polaire, on se donne un point du plan M et on trace le vecteur OM . Les coordonnées (polaires) de M sont alors repérées par la longueur du vecteur OM (sa norme, noté r) et l'angle formé par xOM (noté θ et appelé argument de M).

Une fonction f , dans un tel repère depend de la variable θ et une représentation de cette fonction est de la forme $r = f(\theta)$.

Il y a bien sûr une transformation évidente entre les coordonnées polaires et les coordonnées cartésiennes, qui est donnée par les formules suivantes :

$$\begin{aligned} x &= r \cos(\theta) \\ y &= r \sin(\theta) \\ r &= \sqrt{x^2 + y^2} \\ \cos(\theta) &= \frac{x}{r} \\ \sin(\theta) &= \frac{y}{r} \end{aligned}$$

Voici quelques exemples de graphes en coordonnées polaires.

```
> # Tout d'abord un cercle de rayon 2.
plot(2, t=0..2*Pi, coords=polar, scaling=constrained);

# On veut représenter la fonction f définie par:
# f(theta) = 6/sqrt(4*cos(theta)^2+9*sin(theta)^2);
plot( 6/sqrt((4*cos(t)^2+9*sin(t)^2)),
      t=0..2*Pi, coords=polar,scaling=constrained);
```

Dans le premier cas, la fonction est une fonction constante, égale à 2. On va donc obtenir tous les points qui sont à une distance constante du centre du repère, donc un cercle. Dans le second cas, si vous repassez la fonction en coordonnées cartésiennes, vous allez obtenir l'équation d'une ellipse.

Il existe également une commande 'polarplot(...)' incluse dans la librairie 'plots', qui est pratiquement équivalente à la commande 'plot' avec les options 'coords=polar', (si ce n'est que l'intervalle où varie l'angle n'est pas à spécifier). Voici ce que cela donne pour l'exemple précédent :

```
> polarplot(6/sqrt(4*cos(t)^2+9*sin(t)^2));
```

Des courbes paramétrées (en coordonnées polaires) peuvent être obtenues avec la commande 'polarplot(...)', mais l'intervalle sur lequel varie le para-

metre doit absolument être spécifié. Ici aussi, les courbes paramétrés et non paramétrés peuvent être tracés ensemble.

```
> # Une courbe paramétrée.
  polarplot([sin(t),2*Pi*sin(t),t=0..Pi/2]);

# Deux courbes, une paramétré, l'autre non.
  polarplot({[sin(t),2*Pi*sin(t),t=0..Pi/2],sin(2*t)+sin(t)},
            t=-Pi/2..Pi/2,scaling=constrained);
```

Notez qu'on peut utiliser la commande 'display(...)' pour afficher plusieurs graphes dans la même fenêtre.

```
> a:=polarplot([sin(t),2*Pi*sin(t),t=0..Pi/2],color=blue):
  b:=polarplot(sin(2*t)+sin(t),color=red):
  c:=plot(x-floor(x),x=-2..2,discont=true,color=green):
  display({a,b,c});
```

3.2.5 Inégalités.

On peut tracer des courbes représentatives d'inégalités en utilisant la commande 'inequal(...)' comme suit :

```
> inequal({x+y<5,0<x,x<4},
          x=-1..5,y=-10..10,
          optionsexcluded=(color=yellow));
```

3.2.6 Echelles logarithmiques.

La commande 'logplot(...)' permet d'obtenir des échelles logarithmiques sur l'axe des y. La commande 'semilogplot(...)' permet la même chose sur l'axe des x. Et la commande 'loglogplot(...)' permet la même chose sur les deux axes.

```
> logplot(10^x,x=0..10);
  semilogplot(2^(sin(x^1/3)),x=1..100);
  loglogplot(x^17,x=1..7);
```

3.3 Graphiques en trois dimensions.

Maplesait aussi bien tracer les graphes en 3-d que les graphes dans le plan. La commande à utiliser est la commande `plot3d`. La syntaxe est la suivante : `'plot3d(f(x,y), x=a..b, y=c..d, options)'`

De même que dans les graphes en 2d, on peut afficher plusieurs graphes sur une même fenêtre ou afficher une courbe paramétrée.

Notez que Maple ne montre pas d'axes par défaut. Mais on peut préciser les options suivantes : `'axes=frame'`, `'boxed'` ou `'normal'`.

```
> # Un graphe en 3d.
plot3d((x^2-y^2)/(x^2+y^2),x=-2..2,y=-2..2);

# Le même avec d'autres axes.
plot3d((x^2-y^2)/(x^2+y^2),x=-2..2,y=-2..2,axes=frame);

# Deux graphes en 3d.
plot3d({x+y^2,-x-y^2},x=-2..2,y=-2..2,axes=boxed);

# Une courbe paramétrée.
plot3d([s*sin(s),s*cos(t),s*sin(t)],
        s=0..2*Pi,t=0..Pi, axes=normal);
```

La commande `'plot3d(...)'` utilise les coordonnées cartésiennes par défaut. Mais en utilisant l'option `coords`, on peut changer ce système de coordonnées en un autre. Les plus familiers étant les coordonnées cylindriques et sphériques. Regardez l'aide à ce sujet, via l'aide-en-ligne (`'?plot3d[coords]'` ou `'?coords'`)

La syntaxe générale d'une commande est de la forme :

```
'plot3d(arguments obligatoires et optionnels,
        coords=système_de_coordonnée)'
```

Lorsqu'on veut utiliser un système de coordonnées sphériques (soit l'option `'coords=spherica'`), on peut également se servir de la commande `'spherplot(...)'`, incluse dans la librairie `'plots'`. De même, si l'on veut tracer un graphe dans un système de coordonnées cylindriques (`'coords=cylindrica'`), on peut se servir de la commande `'cylindricalplot(...)'` (également dans la librairie `'plots'`).

En bref, voici les syntaxes minimales pour tracer le graphe de la fonction f dépendant des variables x et y (avec $a < x < b$ et $c < y < d$) :

```
plot3d(f(x,y), x=a..b,y=c..d) ou plot3d(f, a..b,c..d)
```

Et la syntaxe pour les courbes paramétrées par le vecteur (f, g, h) dépendant de deux variables s et t (avec $a < s < b$ et $c < t < d$) :

```
plot3d([f(s,t),g(s,t),h(s,t)], s=a..b,t=c..d) ou plot3d([f(s,t),g(s,t),h(s,t)],
a..b, c..d)
```

Voici quelques exemples :

```
> # Une première courbe en 3d.
plot3d( (1.3)^x*sin(y),          # La Fonction.
        x=-1..2*Pi, y=0..Pi,    # La fenêtre.
        coords=spherical,style=PATCH); # Les options.

# Une deuxième courbe en 3d.
plot3d( sin(x*y),              # La Fonction.
        x=-Pi..Pi, y=-Pi..Pi,  # La fenêtre.
        style=HIDDEN);         # Les options.

# Une courbe paramétrée en 3d.
plot3d([                          # Le vecteur.
        x*sin(y)*cos(y),
        x*cos(y)*cos(y),
        x*sin(y)
    ],
        x=0..2*Pi,y=0..Pi);     # La fenêtre.

# Plusieurs graphes en même temps.
plot3d({sin(x*y),x+2*y},        # Les courbes.
        x=-Pi..Pi,y=-Pi..Pi    # La fenêtre.
        axes=FRAME);           # Les options.

# Plusieurs courbes paramétrées en même temps.

# On se donne 4 vecteurs de courbes paramétrées:
c1:=[cos(x)-2*cos(0.4*y),sin(x)-2*sin(0.4*y),y]:
c2:=[cos(x)+2*cos(0.4*y),sin(x)+2*sin(0.4*y),y]:
c3:=[cos(x)+2*sin(0.4*y),sin(x)-2*cos(0.4*y),y]:
c4:=[cos(x)-2*sin(0.4*y),sin(x)+2*cos(0.4*y),y]:

# On trace les 4 courbes en même temps.
plot3d( {c1,c2,c3,c4},          # Les courbes
        x=0..2*Pi, y=0..10,    # La fenêtre.
        grid=[25,15],style=PATCH); # Les options.
```

```

# Un graph en coordonnées spheriques.
sphereplot(1, # La courbe
            theta=0..2*Pi, phi=0..Pi, # La fenetre.
            scaling=constrained, # Les options.
            title="La Sphere de rayon 1",
            titlefont=[HELVETICA,BOLD,24]);

# En coordonnées cylindriques.
# On charge la commande cylinderplot en mémoire.
with(plots,cylinderplot)

cylinderplot(theta,theta=0..4*Pi,z=-1..1);
cylinderplot(z,theta=0..2*Pi,z=0..1);

# Et une courbe paramétrée en coordonnées cylindriques :
cylinderplot([s*t,s,cos(t^2)],s=0..Pi,t=-2..2);

```

3.4 Animations.

Une suite de graphes en 2 ou 3 dimensions peut être animé grâce à la commande 'animate(...)' qui a la syntaxe suivante en 2-d : 'animate(f(x,t),x=a..b,t=c..d)'

Où t représente le temps, i.e.: le paramètre à faire varier.

Bien sûr, on peut animer aussi bien une fonction, qu'une courbe paramétrée, qu'en courbe en 3d...Attention cependant, ces fonction font parties de la librairie 'plots'. Vérifiez qu'elles sont en mémoire avant de les utiliser à tout va.

```

> # On charge en mémoire les commandes que l'on va utiliser.
with(plots,animate,animate3d)

# L'animation d'une fonction.
animate(sin(x*t),x=-10..10,t=1..2);

# L'animation d'une courbe paramétrée.
animate([a*cos(u),sin(u),u=0..2*Pi],a=0..2);

# Une animation en 3d.
animate3d(cos(t*x)*sin(t*y),x=-Pi..Pi,y=-Pi..Pi,t=1..2);

```

3.5 Exercices.

Exercice 10.

1. Sur l'intervalle $[0, +\infty[$, tracez les fonctions $x \rightarrow \sqrt{|x|}$ et $x \rightarrow x^2$, dans un même graphe.
2. Tracez la fonction $x \rightarrow 1 - \text{floor}(x)$ (attention à la continuité, ou non, de cette fonction).
3. Tracez la fonction $x \rightarrow x \exp(-x)$, pour $x > -1$. Pensez-vous que la fonction dérivée de cette fonction s'annule en un point de l'intervalle considéré ?

Exercice 11.

1. Tracer sur le même graphe les fonctions définies par $f_1(x) = 1 - x$, $f_2(x) = 1 - x^2$, $f_5(x) = 1 - x^5$ et $f_{10}(x) = 1 - x^{10}$, pour tout $x \in [0, 1]$.
2. Tracer l'animation de la fonction f_n définie par $f_n(x) = 1 - x^n$, (pour $x \in [0, 1]$), lorsque n varie entre 1 et 10.

Exercice 12.

Sur un même graphe, tracez en bleu les fonctions $x \rightarrow x$, $x \rightarrow -x$ et $x \rightarrow x \sin(ax)$, où le paramètre a varie entre 0 et 100 et x varie entre -2π et 2π . Passez l'option 'numpoints' à 200 et nommez votre graphe «Un signal sinusoïdal» (avec la fonte Helvetica et la police 14).

Exercice 13.

1. Tracer la courbe définie par $\theta \rightarrow \sin(3\theta)$, en coordonnées polaires et pour θ variant de 0 à 2π .
2. Tracer la spirale définie par $\theta \rightarrow \theta$, pour θ de -4π à 4π .
3. Tracer la courbe (polaire) paramétrée donnée par le vecteur

$$(r, \theta) = (\sin(t), \cos(t))$$

pour t variant de 0 à 2π .

4. Tracer la courbes polaire correspondant à l'équation $\theta = \exp(\rho)$

Exercice 14.

1. Tracer la courbe donnée par l'équation $y^2 = 2x + 4$, dans la fenêtre $-2 < x < 1.5$ et $-3 < y < 3$

2. Tracer entre 0 et 4 la fonction définie par

$$x \rightarrow \begin{cases} x & \text{pour } 0 < x < 1 \\ 1 & \text{pour } 1 < x < 2 \\ 2x - 4 & \text{pour } 2 < x < 3 \\ 3 & \text{sinon} \end{cases}$$

Exercice 15.

Quelle est la région du plan qui satisfait ces inégalités : $x \leq 4$, $x > 0$ et $x + y \leq 5$.

Exercice 16.

Dans les questions suivantes, x et y varient entre -5 et 5 .

1. Tracer la fonction f , définie par : $f(x, y) = \exp(-x^2) + \exp(-4y^2)$.
2. Tracer la fonction f , définie par : $f(x, y) = \sin(\sqrt{x^2 + y^2})$.
3. Tracer la fonction f , définie par : $f(x, y) = \frac{x^2 y^2 \exp(-x^2 - y^2)}{x^2 + y^2}$.
4. Tracer la fonction f , définie par : $f(x, y) = -xy \exp\left(\frac{-x^2}{2} - \frac{y^2}{2}\right)$.

4 Fonctions, limites, dérivées et intégrales.

4.1 Fonctions.

Une fonction sous Maple se définit de la façon suivante :

nom := variable(s) -> definition

Comme le montre l'exemple suivant :

```
> f:=x->2*x^2-3*x+4; # Définition de la fonction.
f(2); # Valeur de la fonction en 2.
plot(f(x),x=-5..5); # Graphe de la fonction entre -5 et 5.
```

$$f := x \rightarrow 2x^2 - 3x + 4$$

6

[graphe]

4.1.1 Fonctions à plusieurs variables.

De même que pour une fonction à une seule variable, on peut définir une fonction à plusieurs variables. Et donc la tracer dans le plan⁸, ou bien évaluer sa valeur à un point donné (x, y) du plan, comme dans l'exemple suivant.

```
> f := (x,y) -> sin(x)*cos(y); # Définition de la fonction.
plot3d(f(x,y),x=-5..5,y=-5..5); # Graphe de la fonction
'f(15,2)' = f(15,2); # valeur en (15,2)
'f(Pi,7)' = f(Pi,7); # valeur en (Pi,7)
'f(8,Pi/2)' = f(8,Pi/2);
```

$$f := (x, y) \rightarrow \sin(x) \cos(y)$$

[graphe]

$$f(15, 2) = \sin(15)\cos(2)$$

$$f(\pi, 7) = 0$$

$$f(8, \pi/2) = 0$$

⁸N'oubliez pas que c'est un graphe en trois dimensions.

4.1.2 Fonctions composées.

Maple sait composer des fonctions entres-elles. On se sert de l'opérateur @. On peut également composer une fonction par elle-même. On utilise alors l'opérateur @@.

```
> (sin@cos)(x);  
g := x -> x^3 + 2;  
h := y -> ln(y):  
'g(h(x))' = (g@h)(x);  
'h(g(x))' = (h@g)(x);  
'(g@@2)(x)' = (g@@2)(x);  
'(g@@3)(x)' = (g@@3)(x);  
'(g@@4)(x)' = (g@@4)(x);
```

TODO

4.1.3 Les Suites définies par des fonctions.

Les suites sont des fonctions «comme les autres», si ce n'est qu'elles sont définies dans \mathbb{N} au lieu de \mathbb{R} .

```
> u := n -> 3*n + 4; # Définition de la suite u_n.  
u[3]; # Calcul du troisième terme.
```

$$u := n \rightarrow 3n + 4$$

13

4.2 Limites.

La syntaxe pour calculer la limite de la fonction f au point a est de la forme suivante :

```
limit(f(x), x = a)
```

Notez que si l'on veut juste écrire la limite, sans la calculée, alors il suffit de mettre un 'L' majuscule au mot limit; comme le montre les exemples suivants :

```
> Limit(sin(x)/x,x=0)=limit(sin(x)/x,x=0);
Limit((1+a)^(1/a),a=0)=limit((1+a)^(1/a),a=0);
limit((1+x^2+3*x^3)/(2-3*x+x^3),x=infinity)=
    limit((1+x^2+3*x^3)/(2-3*x+x^3),x=infinity);
```

TODO

De nombreuses options sont disponible concernant la directions où vous voulez calculez la limite (si vous voulez calculer la limite à gauche ou à droite par exemple)⁹.

```
> limit(floor(x),x=1);
    limit(floor(x),x=1,right);
    limit(floor(x),x=1,left);
```

TODO

4.3 Dérivées.

Les dérivées «ordinaires» ou «partielles» peuvent être faite en utilisant la commande `diff`. Notez que si vous écrivez `Diff` à la place de `diff`, Maple n'évaluera pas la dérivée, mais affichera l'expression que vous avez écrit, comme le montre l'exemple suivant :

```
> Diff(sin(x),x) = diff(sin(x),x);
Diff((1+x^2+4*x^3)/(1+2*x-x^2),x) =
    diff((1+x^2+4*x^3)/(1+2*x-x^2),x);
```

⁹

maple à le courage de vous dire si une limite n'est pas définie.

TODO

Les dérivées multiples peuvent être évalués en répétant la variable plusieurs fois. Ou bien, de manière équivalente en écrivant x^n , où n est le nombre de fois où l'on dérive.

```
> diff(sin(x^2), x, x);  
diff(sin(x^2), x\^2);
```

TODO

4.3.1 Dérivées partielles.

De manière similaire, on peut évaluer les dérivées partielles d'une fonction à plusieurs variables : Notez que l'ordre dans lequel Maple dérive les fonctions est de gauche à droite (dans l'ordre où vous les écrivez). Parfois ¹⁰ cet ordre peut être important.

```
> diff(cos(x*tan(y)), x);  
diff(cos(x+y^2), x, y);  
diff(cos(x+y^2), y, x);  
diff((x^2+y^2)*(ln(x)-ln(y)), x\^2, y);  
diff((x^2+y^2)*(ln(x)-ln(y)), y, x\^2);
```

TODO

4.3.2 La commande D.

Un autre moyen pour dériver les fonctions est d'utiliser le symbole D , qui peut être appliqué à une fonction sans avoir à spécifier ses arguments.

```
> D(sin);  
D(f@g);
```

¹⁰assez rarement, cependant

```
D(f*g);
D(exp*cos(ln));
```

TODO

Cela peut bien sur etre itéré en utilisant le symbole $@@n$ (où n est un entier) dans la composition de D . Ou encore en utilisant l'indexation en i , dans la forme $D[i]$, ce qui donnera une dérivée partielle par rapport à la i ème variable.

Pour des dérivées partielles, on utilise une idexation par rapport à l'ordre dont on veut deriver les fonctions :

```
> (D@@2)(sin);
D[2]((x,y)->x*y^3);
D[2,1]((x,y)->x*y^3);
D[1,2]((x,y)->x*y^3);
```

TODO

4.4 Intégrales.

Maple sait calculer les integrales, grace à la commande `int`. La syntaxe pour les intégrales indefinie (les «primitives») est :

$$\text{int}(\text{expr}, x)$$

où `expr` est l'expression que vous integrez et `x` la variable par rapport à laquelle vous integrez.

Pour les integrales définies, la syntaxe est la suivante :

$$\text{int}(\text{expr}, x=a..b)$$

où `expr` est l'expression que vous integrez et `a..b` l'intevale sur lequel vous integrez.

Comme pour la dérivée, si vous ecrivez `Int` à la place de `int`, vous aurez la forme non évaluée.

```
> Int( tan(x), x ) = int( tan(x), x );
Int( (1-x^2)/(1+x^2), x ) = int( (1-x^2)/(1+x^2), x );
Int( x^2*exp(-x^2), x ) = int( x^2*exp(-x^2), x );
```

TODO

Voici quelques integrales définies et impropre :

```
> Int( cos(x), x=-Pi/2..Pi/2 ) = int( cos(x), x=-Pi/2..Pi/2 );  
Int( x^2*exp(-x^2), x=0..infinity ) = int( x^2*exp(-x^2),  
x=0..infinity );
```

TODO

Les intégrales multiples sont faites par itération de l'intégrale :

```
> Int(Int(Int(exp(-x-y-z),x=0..y), y=0..z), z=0..1)  
= int(int(int(exp(-x-y-z),x=0..y), y=0..z), z=0..1);
```

TODO

4.5 Exercices

Exercice 17.

1. Définissez la fonction f_1 par $f_1(x) = \sin(x) \cos(x)$. Tracer la fonction entre 0 et π . Calculez $f_1(0)$ et $f_1(\pi)$ f_1 est-elle injective ou surjective de $[0, \pi]$ dans \mathbb{R} ?
2. Définissez la fonction f_2 par $f_2(x, y) = \exp(xy) \cos(x) \sin(y)$ Tracer sa courbe representative pour x et y variants de $-\pi$ à π . f_2 est-elle injective ? Si non, trouvez un contre-exemple.
3. Définissez la fonction g par $g(x) = 3x/\ln(x)$ Que vaut la fonction $\exp(g)$? Et $g(\exp)$?
4. Soit la suite u_n définie par : $u_0 = 3$ et pour tout entier n non-nul, $u_n = \ln(u_{n-1}) + 5$. Calculez les termes 5, 10 et 100 en fonction de u_0 .
Refaire le calcul avec $u_0 = 10$.

Exercice 18.

1. Trouvez les limites en $+\infty$ et $-\infty$ de f_3 définie par $f_3(x) = \sin(x) \cos(x)/x$.
2. Quelle est la limite en 0 de la fonction f_4 définie par $f_4(x) = 4 \cos(x)/(1 - x^2)$?
3. Quelle est la limite en 1 à droite de $\text{floor}(x)$? Quelle est la limite en 1 à gauche de cette fonction ?

Exercice 19.

1. Quelle est la dérivée de $u(x)/v(x)$? Calculez la dérivée seconde de ce quotient.
2. Donnez toutes les dérivées partielles (d'ordre 1) de la fonction f_2 définie par :

$$f_2(x, y) = \exp(xy) \cos(x) \sin(y)$$

Donnez toutes les dérivées partielles d'ordre 2 de f_2 .

3. Définissez la fonction g par $g(x) = x^2 + \ln(3x)$. Puis, en une seule ligne de commande, tracez sur un même graphe la fonction g , ainsi que ses deux premières dérivées, pour x compris entre 0.5 et 2

Exercice 20.

1. Soit $f_5(x) = \exp(-x^2)$ calculez l'intégrale de f_5 entre 0 et $+\infty$, puis sur tout \mathbb{R} .
2. Tracez les deux courbes : $y = \sqrt{4x}$ et $y = x/2$, lorsque x varie entre 0 et 16. Par un calcul d'intégrale, donnez l'aire entre ces deux courbes.
3. Cherchez une primitive en x de la fonction $f_6(x, y, z) = \exp(x + y) - 3 \sin(xz) + \ln(z)/x$. Calculez une primitive en y et en z de cette fonction.

5 Type de données en langage Maple.

5.1 Quelques exemples de types.

Maple contient un grand nombre «d'unité» de base, apellés «type» (de l'anglais «data type»), à partir desquels des expressions plus compliquées peuvent être créés.

La commande `whattype(...)` permet de connaître le type d'une expression Maple. Appliqué à une expression «simple», cette commande donnera le type de l'expression considérée. Mais appliqué à une expression composée de plusieurs types différents, la commande `whattype(...)` retournera le type de la dernière étape réalisée (le «top-level» de l'expression, dans un français approximatif).

Types numériques. Ils se divisent en trois grands types distincts : les entiers ('integer'), les fractions ('fractions') et les réels ¹¹.

```
> whattype(3);
whattype(1/2);
whattype(123.432114);
```

integer

fraction

float

Types algébriques. Maple différencie trois types de relations algébrique : les additions (+), les multiplications (*) et les puissances (^).

```
> whattype(x-y);
whattype(x+y);
whattype(x*y);
whattype(x/y);
whattype(x^y);
```

¹¹considérés par Maple comme des nombres à virgule «flottante» (d'où leur nom de 'floats').

+
+
*
*
^

Relations d'égalités. Maple considère qu'il y a quatre types de relations «égalitaires» : l'égalité, la différence, la supériorité stricte et la supériorité large. Notez qu'en langage Maple, la différence se note $\langle \rangle$, la supériorité large se note \geq et l'infériorité large \leq .

```
> whattype(a=b);  
whattype(a<>b);  
whattype(a<=b);  
whattype(a>=b);
```

=
 $\langle \rangle$
 \leq
 \geq

Relations logiques. Les expressions logiques se regroupent en trois types, et (**and**); ou (**or**) et la négation (**not**).

```
> whattype(A and B);  
whattype(A or B);  
whattype(not A);
```

and
or
not

Autres types. Il existe d'autres types en langage Maple qui sont moins utilisés, mais qu'il peut toujours être utile de connaître ¹².

```
> whattype(sin(x));  
whattype(proc(x) x^2 end);  
whattype(a[1]);  
whattype(series(1/(1-x),x=0));  
whattype(x::float);  
whattype(a..b);  
whattype(''sin(x)''');
```

function

procedure

indexed

series

::

..

uneval

5.2 Types de données fondamentaux.

Nous allons voir les quatre types fondamentaux en langage Maple, dans le sens où ils font toute la structure du langage. Ces types sont : les suites ('exprseq', pour «expression sequence», littéralement : «suite d'expression» en anglais), les listes ('list'), les ensembles ('set') et les tableaux ('array').

On peut ajouter à ces quatre types les tables ('table'), que vous utiliserez peu dans le cadre de ce cours, mais qui sont une donnée importante de la structure intrasèque du langage Maple.

5.2.1 Suites et listes.

Suites. Elles sont les expressions composées les plus simples en langage Maple. Pour être le plus général possible, une suite est composée d'un ensemble d'expressions, écrite dans un ordre donné et séparé par une virgule.

¹²pour se la raconter au café après les cours, etc.

```
> a,3,c,3,4,-4,0.9; # Ceci est une suite.
whattype(%);
4,5,d,u,zxs,sin(x); # Ceci est une autre suite.
whattype(%);
```

```
a,3,c,3,4,-4,0.9
      exprseq
4,5,d,u,zxs,sin(x)
      exprseq
```

On peut assigner rapidement (i.e.: en une seule ligne de commande) des valeurs à des variables, grâce aux listes :

```
> p,q,r := 3,12,-5;
```

```
p,q,r := 3,12,-5
```

Listes. Une liste est une «suite entourée de crochets». la liste a les mêmes propriétés que la suite (ordre et possibilité de répétition d'un élément).

```
> [2,4,4,-5,d,2,8]; # Une liste
whattype(%);
```

```
[2,4,4,-5,d,2,8]
      list
```

Notez que l'on ne peut pas assigner des valeurs aux variables à l'intérieur d'une liste, la commande suivante donnera une erreur précisant que la partie gauche de l'assignement n'est pas valide¹³ comme le montre l'exemple suivant :

```
> [a,b,c] := 1,3,5;
```

```
Error, invalid lhs of assignment
```

¹³lhs signifie «Left Hand Side»

Quelques commandes sur les listes et les suites. Pour passer d'une suite à une liste, il suffit d'ajouter des crochets ('[' et ']'). Pour faire l'inverse (i.e.: passer d'une liste à une suite), on se sert de la commande `op(...)`. Cette commande a un grand nombre d'utilité et nous reviendrons sur son utilisation.

```
> # On attribut à la variable 'suite' la valeur a,b,c.
suite := a,b,c;

# On attribut à la variable 'liste' la valeur [a,b,c].
liste := [suite];

# On retombe sur une suite.
op(liste);
```

suite := a,b,c

liste := [a,b,c]

a,b,c

Pour créer une suite générique, c'est à dire du type $f(1), f(2), f(3), \dots$ on se sert de la commande `seq(...)` dont voici l'utilisation.

```
> # on crée une suite arbitraire de f(1) à f(5).
seq( f(i), i=1..5);

seq( j^3, j=4..8);          # la suite des cubes de 4 à 8.
seq( sin(k*pi/5), i=5..10) # une autre suite.
```

f(1), f(2), f(3), f(4), f(5)

64, 125, 216, 343, 512

sin(π), sin($6/5\pi$), sin($7/5\pi$), sin($8/5\pi$), sin($9/5\pi$), sin(2π)

Pour obtenir le n-eme terme d'une suite ou d'une liste, on procède comme suit :

```

> liste:=[4,5,3,2,6,9];      # On definit une liste.
  liste[4];                  # On veut le quatrième terme.
  suite:= sin(x),cos(y),x*y; # On definit une suite.
  suite[1];                  # On veut son premier terme.

```

```

liste := [4, 5, 3, 2, 6, 9]

2
suite := sin(x), cos(y), xy
sin(x)

```

De même, on peut obtenir une sous-suite (ou une sous-liste) à partir d'une suite ou d'une liste donnée.

```

> # On définit une suite.
  s1:=3,5,g,t,sin(x),u,8,3,5;
  # On veut la sous-suite du 4è au 7è terme.
  s1[4..7];
  # On définit une liste.
  l1:=[q,w,e,r,t,y,u,i,o,p];
  # On veut ses 6 premiers éléments.
  l1[1..6];

```

```

s1 := 3, 5, g, t, sin(x), u, 8, 3, 5

t, sin(x), u, 8
l1 := [q, w, e, r, t, y, u, i, o, p]
[q, w, e, r, t, y]

```

On peut bien sûr définir des listes de listes, ou bien concatener plusieurs suites ou listes ensemble. Pour les listes de listes, il suffit de proceder comme suit :

```

> # On défini une liste de liste.
  LdL:= [[a1,a2,a3,a4],[b1,b2,b3,b4],[c1,c2,c3,c4]];

```

```

# On veut son troisième terme.
LdL[3];
# On veut le 4è élément de son premier terme.
LdL[1,4];

```

$$LdL := [[a1, a2, a3, a4], [b1, b2, b3, b4], [c1, c2, c3, c4]]$$

$$[c1, c2, c3, c4]$$

$$a4$$

La concatenation de suite est triviale : il suffit d'accoler deux (ou plus) suites, séparées par une virgule.

```

> # On definit deux suites.
s1:=1,2,3;
s2:=a,b,c;
# s3 est la concatenation des deux premières suites.
s3:=s1,s2;
# s4 est la concatenation des trois premières suites.
s4:=s1,s3,s2;

```

$$s1 := 1, 2, 3$$

$$s2 := a, b, c$$

$$s3 := 1, 2, 3, a, b, c$$

$$s4 := 1, 2, 3, 1, 2, 3, a, b, c, a, b, c$$

Mais il n'en va pas tout à fait de même pour la concatenation de listes. Dans le cas où l'on veut concatener deux listes (il faudra suivre la même idée, si l'on veut concatener trois ou plus listes), si l'on accole les deux listes, nous allons obtenir une suite composée de deux éléments (qui sont des listes), et non pas une seule liste, comme on le voudrait. Il faut donc proceder comme suit :

```

> # On definit deux listes.
l1:=[1,2,3];
l2:=[a,b,c];

```

```
# On concatene et on stocke le résultat dans l3.
l3:=[ op(l1) , op(l2) ];
```

```
l1 := [1, 2, 3]
```

```
l2 := [a, b, c]
```

```
l3 := [1, 2, 3, a, b, c]
```

Les commandes `op(l1)` et `op(l2)` permettent de repasser des listes aux suites, que l'on concatène comme on l'a vu plus haut. Les crochets (au début et à la fin de l'expression) permettent de passer de la suite à une liste.

La commande `op(...)` est fondamentale et nous la reverrons plus tard. Dans le même style, il existe la commande `nops(...)` qui permet de connaître le nombre d'éléments d'une liste (mais pas d'une suite, pour connaître le nombre d'éléments d'une suite, il faut ruser...)

```
> liste:= [1,2,3,4,5,6]; # Definition d'une liste.
nops(liste); # Nombre d'éléments de la liste.
suite:=a,b,c,d # Definition d'une suite.
nops([suite]); # Nombre d'éléments de la suite.
```

```
liste := [1, 2, 3, 4, 5, 6]
```

```
6
```

```
suite := a, b, c, d
```

```
4
```

Ici, pour connaître le nombre d'éléments de la suite, on a transformé la suite en liste en ajoutant des crochets, puis on s'est servi de la commande `nops(...)`.

On peut également substituer un élément par un autre dans une liste, mais pas dans une suite, à l'aide des commandes `subs(...)` et `subsop(...)`.

`subs(...)` permet de substituer un élément par son nom, tandis que `subsop(...)` substitue un élément en fonction de sa place dans la liste. Notez qu'un tel remplacement n'est valide que pour la commande effectuée et non définitif.

```

> # On definit une liste.
  liste := [a,b,c,3,5,sin(x)];
  # On remplace c par 5 dans cette liste.
  subs(c=5, liste);
  # Notez que la liste n'a pas changée.
  liste;
  # On remplace le troisième élément par cos(x).
  subsop(3=cos(x), liste);

```

$$\begin{aligned}
 & \text{liste} := [a, b, c, 3, 5, \sin(x)] \\
 & [a, b, 5, 3, 5, \sin(x)] \\
 & [a, b, c, 3, 5, \sin(x)] \\
 & [a, b, \cos(x), 3, 5, \sin(x)]
 \end{aligned}$$

Enfin, on peut trier une liste numérique (i.e.: composée uniquement de nombres) grâce à la commande `sort(...)`.

```

> liste:=[21,.58,1,96,3,7];
  sort(liste);

```

$$\begin{aligned}
 & \text{liste} := [21, .58, 1, 96, 3, 7] \\
 & [.58, 1, 3, 7, 21, 96]
 \end{aligned}$$

5.2.2 Les ensembles.

Un autre type standard, en langage Maple est celui des ensembles («set», en anglais), qui est «une suite entourée par des accolades» ('{' et '}'). Un ensemble est une suite d'éléments qui ne se répètent pas et qui n'ont pas d'ordre particulier. Ne pensez surtout pas qu'un ordre peut être conservé dans les ensembles, cela peut-être une source de grave erreur.

```

> e1 := {1,3,5,3,1,4,5,2};          # On definit un ensemble.
  whattype(%);                     # Verifions son type.

```

```
e2 := {tan(x), sin(x), cos(x)}; # Voici un autre ensemble.
whattype(%);                    # Vérifions son type.
```

```
e1 := {1, 2, 3, 4, 5}
```

```
set
```

```
e2 := {sin(x), cos(x), tan(x)}
```

```
set
```

Bien sur, on peut passer d'une suite à un ensemble ne ajoutant des accolades, pour faire l'inverse (i.e.: passer d'un ensemble à une suite), on se sert de la commande `op(...)`.

```
> e1 := {1,3,5,3,1,4,5,2}; # On definit un ensemble.
op(e1);                    # On passe à la liste correspondante.
{%;}                       # Retour à l'ensemble.
```

```
e1 := {1, 2, 3, 4, 5}
```

```
1, 2, 3, 4, 5
```

```
{1, 2, 3, 4, 5}
```

On peut faire les opérations habituelles sur les ensembles (telles que union, interpartie et soustraction)

```
> {a,b,c} union {b,c,d};    # Union de deux ensembles
{1,2,3} intersect {3,4,5}; # Interpartie.
{seq(i, i=1..5)} minus {seq(i,i=4..8)}; # Soustraction.
```

```
{a, b, c, d}
```

```
{3}
```

```
{1, 2, 3}
```

La commande `select(...)` peut être appliquée à un ensemble pour identifier les éléments de celui-ci avec un critère spécial. La syntaxe est : `select(critère, ensemble)`

```
> select(isprime, {1,45,27,17,19,29874,109547});
```

```
{17, 19, 109547}
```

5.2.3 Tableaux.

Un tableau («array» en anglais) de taille $m_1 \times m_2 \times \dots \times m_p$ est une structure contenant $m_1 m_2 \dots m_n$ entrées, numérotées par le n -uplet (i_1, i_2, \dots, i_p) . Un tableau de dimension $1 \times m$ peut être vu comme un vecteur colonne de dimension m , un tableau de dimension $m \times 1$ peut être vu comme un vecteur de dimension m . Et un tableau de dimension $m \times n$ est une matrice de dimension $m \times n$. Nous verrons plus en détail les matrices dans le cours suivant.

La syntaxe pour déclarer un tableau nommé T, en langage Maple est la suivante :

```
T := array(type_de_tableau, fonction_index, liste_de_valeurs)
```

Où

- `type_de_tableau` est optionnel et permet de décrire le type de tableaux et est l'un des noms suivants : `symmetric`, `antisymmetric`, `sparse`, `diagonal`, et `identity`.
- `fonction_index` est une fonction décrivant les dimensions du tableau.
- `liste_de_valeur` est optionnel et est la liste des valeurs à donner aux éléments du tableau.

Pour afficher un tableau, on utilise la commande `print(...)`, ou bien, si les entrées ont été définies, la commande `eval(...)`. Si les entrées du tableau n'ont pas été définies, la commande `eval(...)` mettra un point d'interrogation (?) à la place.

```
> # On définit un tableau 2x2 générique.  
AA:=array(1..2,1..2):  
# On l'affiche.  
print(AA);  
eval(AA);  
# On définit un tableau 2x2 avec des entrées précises.  
AB:=array(1..2,1..2, [[1,2],[2,3]]):
```

```
# On l'affiche avec la commande eval.  
eval(AB);  
# De même avec la commande print.  
print(AB);
```

$$\begin{bmatrix} AA_{1,1} & AA_{1,2} \\ AA_{2,1} & AA_{2,2} \end{bmatrix}$$

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

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

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

Notez, cependant que si l'affichage («ouput», en anglais) est la même, cela ne veut pas dire qu'ils sont du même type :

```
> whattype(print(AB));  
whattype(eval(AB));
```

exprseq

array

Notez que l'on peut définir un tableau simplement en donnant ses entrées sous forme de listes de listes, sans avoir à préciser ses dimensions.

```
> array([[a,b],[c,d]]);
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Notez que ce n'est pas le même objet que l'élément suivant :

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

```

Tab:=array([[a,b],[c,d]]);
whattype(LdL);
whattype(Tab);
whattype( eval(Tab) );

```

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

$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

list

symbol

array

On peut convertir un tableau en liste de liste («listlist», en anglais) en utilisant la commande `convert(...)`. Réciproquement, une liste peut être converti en un tableau («array»).

```

> # On defini un tableau
Tab:= array([[1,2,3,4],[a,b,c,d],[q,2,3,r]]);
# On converti le tableau en liste de liste.
convert(Tab,'listlist');
# On vérifie le type de la précédente expression.
whattype(%);
# On défini une liste de liste.
LdL:=[[4,5,6],[sin(x),cos(x),x]];
# On converti la liste en tableau.
convert(LdL,'array');
# On verifie le type.
whattype(%);

```

$Tab := \begin{bmatrix} 1 & 2 & 3 & 4 \\ a & b & c & d \\ q & 2 & 3 & r \end{bmatrix}$

$[[1, 2, 3, 4], [a, b, c, d], [q, 2, 3, r]]$

list

$LdL := [[4, 5, 6], [\sin(x), \cos(x), x]]$

On peut sélectionner une entrée d'un tableau ou d'une liste de liste. Cependant, dans une liste de liste on peut aussi prendre les éléments au «top-level» (i.e.: des lignes), car ce sont des listes, mais ce n'est pas possible avec un tableau (d'où l'utilité des changements de types).

```
> # On defini un tableau et une Liste de liste.  
Tab:= array([[1,2,3,4],[a,b,c,d],[q,2,3,r]]);  
LdL:=[[4,5,6],[sin(x),cos(x),x]];  
Tab[1,2]; # On extrait un élément du tableau.  
Tab[2,2]; # On extrait un autre élément  
LdL[1,1]; # De même avec la Liste de liste.  
LdL[1,2];  
LdL[1]; # On peut extraire une liste de la LdL  
LdL[2];
```

$$Tab := \begin{bmatrix} 1 & 2 & 3 & 4 \\ a & b & c & d \\ q & 2 & 3 & r \end{bmatrix}$$

$[[1, 2, 3, 4], [a, b, c, d], [q, 2, 3, r]]$
 $LdL := [[4, 5, 6], [\sin(x), \cos(x), x]]$
2
b
4
5
[4, 5, 6]
[sin(x), cos(x), x]

5.2.4 Les tables.

Les tables sont une généralisation des tableau, si ce n'est que les indices sont n'importe quel type d'expression (chaîne, symbole, de type alphanumérique, etc.). Les tables sont utiles uniquement dans le cadre de la programmation avancée, en effet elles permettent de définir des librairies. Voici quelques exemples :

```
> data1:=table([masse='1 kg', accel='9.8 m/sec^2',
               temps='3 sec']):
data1[masse];
animaux:=table([mammifere1=marmote, mammifere2=blaireau,
               oiseau1=coucou, oiseau2=pigeon]):
animals[mammifere2];
```

1 kg

blaireau

5.3 Exercices.

Exercice 21.

1. Construisez une liste ordonnée des 100 premiers nombres premiers.
2. Divisez tous les éléments de cette liste par 23 et appliquez la fonction `frac(...)` à celle-ci pour obtenir une nouvelle liste de 100 fractions dont la valeur est entre 0 et 1.
3. Multipliez tous les éléments de cette liste par 23 et construisez une nouvelle liste, qui sera le résultat ordonné de la liste.
4. Convertissez cette liste en un ensemble et donnez le nombre d'éléments de cet ensemble.

Exercice 22.

1. Créez une liste, appelée N2, qui sera constituée des carrés compris entre 1000 et 10000 des nombres entiers .
2. Créez une liste appelée N3, qui sera constituée des cubes compris entiers entre 1000 et 10000 des nombres entiers.
3. De ces deux listes créez une seule liste. Combien y-a-t-il d'éléments qui sont à la fois dans N2 et dans N3 ?

4. Repetez la même chose avec N4, constitué des puissances 4è et N5, constitué des puissances 5è.
5. (Pour les informaticiens en herbes.) S'il y a des éléments en commun, quels sont-ils ?

Exercice 23.

1. Construisez un tableau 4×4 , appelé JJ, tel que toutes ses entrées soient nulles sauf sur la diagonale principale où elles valent toutes 1 (i.e.: $JJ_{i,i} = 1$, pour i de 1 à 4) et sur la seconde diagonale (juste au dessus) où elles valent -1 (i.e.: $JJ_{i+1,i} = -1$, pour i de 1 à 3).
2. Créez un tableau 3×3 , appelé AA. Convertissez le en liste de liste, puis faite de ces liste une seule liste de 9 elements, de telle sorte que les 3 premiers elements de la liste correspondent à la première ligne, les 3 suivants à la seconde et les trois derniers à la dernière ligne.
3. En utilisant les commandes `seq(...)`, `isprime(...)`, et `select(...)` trouvez tous les nombres premiers entre 1235 et 1270.

6 Algèbre linéaire.

6.1 Algèbre matriciel élémentaire.

La façon la plus simple de représenter les matrices et vecteurs dans Maple est d'utiliser la structure des listes («list»). On représente les vecteurs-ligne par une liste et les vecteurs-colonne (ou de manière plus générale, les matrices) par des listes de listes. Les opérations algébriques élémentaire (addition ou multiplications de matrices, multiplication par un scalaire, puissance n-ième d'une matrice-carré, etc.) peuvent être réalisés avec des listes.

Cependant, pour certains calculs, il vaut mieux représenter les matrices comme des tableaux («array»), ou de se servir de la librairie «linalg». Ce package introduit (entre autre) deux commandes : `matrix` et `vector`, qui permettent de définir des matrices et des vecteurs¹⁴. La librairie «linalg» permet de réaliser toutes les opérations connues d'algèbre linéaire, telles que choisir une base, trouver le rang d'une matrice, transposer, inverser ou diagonaliser une matrice, trouver les vecteurs propres et les valeurs propres d'une matrice...

6.1.1 Matrices et vecteurs en tant que listes.

Vecteurs. On peut définir des vecteurs comme des listes et faire des opérations élémentaires d'additions, de soustractions et de multiplication par un scalaire sur de telles listes.

```
> # On definit 4 "vecteurs", v1, v2, v3 et v4
v1:=[1,3,5];
v2:=[a,b,c];
v3:=[seq(v[i],i=1..3)];
v4:=[seq(a^i,i=1..4)];
v1+v3;           # Addition.
v2-v1;          # Soustraction.
2*v4;           # multiplication par un scalaire.
-6*v3;
```

Cependant, pour les multiplications par un scalaire symbolique et évaluer le résultat, il faut utiliser la commande 'evalm(...)' (pour «evaluate matrix», en anglais). Notez que cette commande converti la liste («list») en

¹⁴Nous verrons ici toutes les opérations que l'on apprend en 1^è année de Deug. Pour plus d'information sur la librairie `linalg`, nous vous invitons à consulter l'aide en-ligne

tableau («array»). Pour revenir à une liste, il faut appliquer la commande 'convert(...)'.

```
> vecteur:=[1,34,6,9]; # On defini un vecteur.
A*vecteur;           # On multiplie le vecteur par A.
evalm(%);
whattype(%);
convert(evalm(A*vecteur),list); # On convertit en liste.
whattype(%);
```

Comme avec toutes les listes, on peut extraire l'un des composant en fonction de sa position dans la liste :

```
> vecteur:=[1,34,6,9]   # On defini un vecteur.
vecteur[2];           # On extrait le 2è élément.
vecteur[3];           # De même avec le 3è.
```

Matrices. Dans la même veine, on peut définir les matrices comme des listes de listes. Et convertir la liste en tableau par la commande 'evalm(...)'. Et comme pour les vecteurs, on peut faire les opérations élémentaires sur les matrices ainsi définies.

```
> m1:=[[a,b,c],
      [d,e,f],
      [g,h,j]];# On definit une matrice.
m2:=[[0,9,8],
      [7,6,5],
      [4,3,2]];# Une autre matrice.

m1:=[[1,2,3],
      [4,5,6]];# une troisieme matrice.
evalm(m1);    # On evalue la matrice.

3*m3;        # Multiplication par un scalaire.
evalm(%);

evalm(x*m1); # Multiplication par un scalaire arbitraire.
evalm(x*m3);

m1 + m2;     # Addition de deux matrices de même dimension.
```

```

evalm(%);
m2 - m1;      # Soustraction
evalm(\%);

```

Un vecteur colonne peut être représenté comme une liste de liste avec une seule entrée sur chaque ligne (une matrice de dimension $n \times 1$).

```

> vecteur:=[[a],
            [b],
            [c]];
evalm(vecteur);

```

Deux matrices 'X' et 'Y' peuvent être multipliées entre-elles en utilisant la notation 'X &* Y', suivit de la commande 'evalm(...)'. N'oubliez pas que le résultat sera un tableau («array»). Pour revenir à une liste de liste («listlist»), il faut utiliser la commande 'convert(...)'.
 Par la même méthode, il est possible de multiplier matrices et vecteurs, ou de calculer des produits scalaires, ou les puissances d'une matrice :

```

> m1:=[[a,b,c],
      [d,e,f],
      [g,h,j]];      # Une matrice.
m2:=[[0,9,8],
      [7,6,5],
      [4,3,2]];      # Une autre matrice.
evalm(m1 &* m2);      # On multiplie les deux matrices.

# On convertit le tableau en liste de liste.
convert(%,listlist);

v1:=[[3],[5],[7]]    # Un vecteur colonne.

evalm(m1 &* v2);      # On multiplie m1 par v1.

evalm(m1^2);          # On calcule le carré de m1.
evalm([[1,2],[2,3]]^3); # Et le cube de cette matrice.

```

6.1.2 Les commandes `vector` et `matrix`.

Pour des calculs plus compliqués sur les matrices et les vecteurs, on se sert de la librairie «`linalg`», il faut soit charger en mémoire la totalité de la librairie, via la commande `'with(linalg) :'`¹⁵

Vectors. Une fois la package chargé, on peut très facilement construire des vecteurs grace à la commande `'vector(...)`', dont la syntaxe est la suivante : `'vector(n, [x1, ..., xn])'`, où n est la dimension du vecteur et $[x1, \dots, xn]$ ses composantes. Notez que l'un des deux argument (au choix) est optionnel.

En clair la syntaxe de cette commande peut être : `'vector(n)'` ou `'vector([x1, ... , xn])'`. Regardez l'aide sur cette commande pour plus de détails, cette aide s'obtient via la commande `'?linalg[vector]'`. Souvenez vous que le type du vecteur ainsi généré est un tableau («`array`») et non pas une liste («`list`»). Voici quelques exemples :

```
> V1:=vector(3);           # On definit un vecteur.
  print(V1);              # On l'affiche.
  V2:=vector(4,[6,3,7,4]); # On definit un autre vecteur.
  print(V2);              # On l'affiche.
  V3:=vector(4,x->1/x);   # Un troisième vecteur.
```

Dans le dernier cas, la fonction $x \rightarrow \frac{1}{x}$ a été appliquée aux index (1, 2, 3, 4) pour créer un vecteur.

Matrix. De même que pour les vecteurs, la commande `'matrix(...)`' peut-être utilisée pour créer une matrice $n \times m$. La syntaxe de cette commande est la suivante : `'matrix(m, n, L)'`, où m et n sont des dimensions de la matrice et L une liste de vecteurs, pour plus de détails, consultez l'aide à ce sujet, via la commande `'?linalg[matrix]'`. Notez que les matrices sont alors des tableaux. Pour revenir aux listes, on utilise comme toujours la commande `convert`. Voici quelques exemples :

```
> M1:=matrix(2,3);           # On definit une matrice.
  print(M1);                 # On l'affiche.
  M2:=matrix(2,4,[1,3,4,7,2,3,3,6]); # Une autre matrice.
  print(M2);
  M3:=matrix(4,4,(i,j)->a^(i+j)); # Une troisième matrice.
```

¹⁵il vaut mieux mettre deux-points (':') à la fin de cette commande, cela permet de ne pas afficher l'ensemble des commandes qui ont été mise en mémoire.

```

print(M3);
M4:=matrix([[a,b],[c,d],[e,f]]); # Une quatrième.
print(M4);
convert(M3,listlist);           # Conversion en liste de liste
convert(%,matrix);              # Retour aux matrices.

```

Pour afficher le résultat d'une multiplication de matrices ou de vecteurs (ou, plus simplement, pour afficher une matrice ou un vecteur) on utilise la commande 'evalm(...)'. De même, on peut évaluer les puissances d'une matrice ou d'un vecteur.

```

> M:=matrix(3,3,[1,3,4,7,2,3,3,6,8]); # Une matrice.
   evalm(M^2);                          # le carré de la matrice.
   evalm(M^3);                          # Son cube.

```

De nombreuses matrices «spéciales» peuvent être générées grâce à des commandes présentes dans la librairie «linalg», telle que . Les matrices diagonales, diagonales par blocs, Les matrices bandes, de Toeplitz, de Jordan, de Vandermonde ou de Hilbert.

```

> diag(a,b,c,d);                       # Une matrice diagonale.
   m1:=matrix(2,2);                     # Une matrice 2x2 quelconque.
   m2:=matrix(3,3);                     # Une matrice 3x3 quelconque.
   diag(m1,m2);                         # Une matrice diagonale par bloc.
   band([a,b,c],5);                     # Une matrice bande.
   toeplitz([a,b,c,d,e]);               # Une matrice de Toeplitz.
   JordanBlock(alpha,4);                # Une matrice de Jordan.
   vandermonde([a,b,c,d,e]);            # Une matrice de Vandermonde.
   hilbert(4,x);                         # Une matrice de Hilbert.

```

6.1.3 Opérations algébriques sur des matrices ou des vecteurs.

```

> # Définitions quelques matrices et vecteurs :
   m:=matrix([[1,2],[3,4]]):
   M:=matrix(2,2):
   N:=matrix(2,3):
   v:=vector([2,4]):
   V:=vector(2):

```

```

W:=vector([a,b,c]):

'm'=evalm(m); # On les affiche.
'M'=evalm(M);
'N'=evalm(N);
'v'=evalm(v);
'V'=evalm(V);
'W'=evalm(W);

# On peut les additionner
evalm(m + M);
evalm(m - M);
matadd(m,M);
matadd(m,-M);

# De même on peut multiplier matrices et vecteurs.
'm M'=evalm(m&*M);
'm N'=evalm(m&*N);
'm V'=evalm(m &* V);
'N W'= evalm(N &* W);

```

Maplesait différencier les vecteurs colonnes et les vecteurs ligne, ainsi la multiplication de deux matrices donne un produit scalaire : On peut aussi le faire via la commande `dotprod` en utilisant l'option 'orthogonal'. Notez que si l'option 'orthogonal' n'est pas mise, alors Maple calcul pense que les vecteurs ont des entrées complexes et calcul le produit hermitien de deux vecteurs.

```

> v:=vector([2,4]):
V:=vector(2):
'v.V'=evalm(v &* V); # Calcul du produit scalaire.
dotprod(v,V,orthogonal);
dotprod(v,V);

```

Une autre façon de calculer cela est d'utiliser la commande `'multiply(...)`, qui est équivalente à utiliser d'abord la commande `'&*' puis la commande 'evalm(...).`

```

> # Définitions quelques matrices et vecteurs :
m:=matrix([[1,2],[3,4]]):
M:=matrix(2,2):

```

```

N:=matrix(2,3):
v:=vector([2,4]):
V:=vector(2):
W:=vector([a,b,c]):

# On multiplie dans la joie et la bonne humeur.
'm M' = multiply(m,M);
'm N' = multiply(m,N);
'm V' = multiply(m,V);
'N W' = multiply(N,W);
'v.V' = multiply(v,V);

```

Dans la librairie «linalg», on trouve une commande spéciale pour multiplier une matrice par un scalaire : 'scalarmul(matrice, scalaire)'.

```

> # Definissions quelques matrices et vecteurs :
m:=matrix([[1,2],[3,4]]):
M:=matrix(2,2):

scalarmul(m,3);          # On multiplie par un scalaire.
scalarmul(m,x);
scalarmul(M,3);
scalarmul(m,x);

```

6.2 Opérations diverses sur les vecteurs et sur les matrices.

La librairie «linalg» contient un certain nombre de commandes pour faire tous les calculs standard sur les matrices.

6.2.1 Opérations sur les vecteurs.

Dimension, base et norme. La dimension d'un vecteur peut être déterminé par la commande 'vectdim(...)'.
 On peut déterminer une base pour un ensemble (fini) de vecteurs, à l'aide de la commande 'basis(...)'.
 La commande 'norm(...)' permet de calculer la norme. il existe de nombreuses options pour cette commande, de manière à déterminer quelle norme utiliser. Je vous invite à consulter l'aide à ce sujet, via '?linalg,norm'

```

> vectdim(vector([2,5,3,4,1]));          # Dimension du vecteur
basis([vector([1,2,2]),vector([2,1,3]),vector([5,4,8])]);
                                          # Une base.
norm(vector([a,b,c]),2);                 # Une norme.
norm(matrix([[a,b],[c,d]]),infinity);  # Une autre norme.

```

Vecteurs dans l'espace. La commande `crossprod` permet d'évaluer le produit vectoriel de deux vecteurs et la commande `angle` permet d'évaluer l'angle entre deux vecteurs.

```

> crossprod(vector([a,b,c]),vector([A,B,C]));
eval(angle(vector([0,1,1]),vector([0,1,0])));

```

Créer des matrices à partir de vecteurs. La commande `stackmatrix` permet de créer une matrice à partir d'une liste de vecteurs.

```

> MM:=stackmatrix( vector([1,2,5,3]), vector([1,5,2,3]),
                    vector([4,2,3,1]), vector([-2,5,4,5]));

```

6.2.2 Opérations sur les matrices.

Matrices et applications linéaire. Une matrice peut représenter une application linéaire d'un espace vectoriel dans un autre. On peut alors vouloir déterminer la dimension de l'image (le rang de la matrice), le noyau de l'application linéaire, etc. Maple permet de réaliser tout cela.

La commande `'rank(...)` permet de déterminer le rang d'une matrice. La commande `'kernel(...)` permet de calculer le noyau d'une matrice (vu comme une application linéaire). Les commandes `'rowSpace(...)` et `'colSpace(...)` donne la famille maximale libre des vecteur-colonnes ou des vecteurs-lignes d'une matrice. Ces commandes font partie de la librairie «linalg».

```

> MM := matrix([[1,2,5,3],[1,5,2,3],[4,2,3,1],[-2,5,4,5]]);
rank(MM);          # Rang de la matrice.
kernel(MM);       # noyau de la matrice.
rowSpace(MM);     # espace libre engendré par les vecteurs
                  # lignes.
colSpace(MM);     # espace libre engendré par les vecteurs
                  # colonnes.

```

Opération élémentaires du pivot de Gauss. On peut vouloir ajouter une ligne (ou une colonne) à une autre, interchanger deux lignes (ou colonnes), bref faire les opérations élémentaires qui permettent de réaliser le pivot de Gauss. Maple permet de faire ces opérations¹⁶.

La commande 'addrow(Mat, l1, l2, m)' permet d'ajouter m fois la ligne $l1$ de la matrice Mat à la ligne $l2$. De manière similaire 'addcol(Mat, c1, c2, m)' va ajouter m fois la colonne $c1$ à la colonne $c2$ dans la matrice Mat .

La commande 'swaprow(Mat, l1,l2)' échange les lignes $l1$ et $l2$ de la matrice Mat . De même, la commande 'swapcol(Mat, c1, c2)', échangera les colonnes $c1$ et $c2$ de la matrice Mat .

Les commandes 'mulrow(Mat, r, expr)' et 'mulcol(Mat, c, expr)' multiplient la ligne r (ou la colonne c) de la matrice Mat par la valeur de $expr$.

```
> MM := matrix([[1,2,5,3],[1,5,2,3],[4,2,3,1],[-2,5,4,5]]);
  addrow(MM,1,2,a);    # On ajoute a fois la ligne 1 à
                      # la ligne 2.
  addcol(MM,2,3,b);    # On ajoute b fois la colonne 2 à
                      # la colonne 3.

  swaprow(MM,1,3);     # On échange les lignes 1 et 3
  swapcol(MM,2,4);     # On échange les colonnes 2 et 4

  mulcol(MM,2,x);      # On multiplie la ligne 2 par x.
  mulrow(MM,3,y);      # On multiplie la colonne 3 par y
```

Concatenation. La commande 'concat(Mat1, Matr2, ..., Matr n)' permet de faire une seule matrice à partir des matrices $Matr1, Matr2, \dots$. Cette commande ajoute les matrices horizontalement donc les matrices doivent avoir le même nombre de ligne. On peut également réaliser cette opération via la commande 'augment(Mat1, Matr2, ..., Matr n).'

```
> u:=matrix(2,2);
  v:=matrix(2,3);
  w:=matrix(2,1);    # trois matrices.
  concat(u,v,w);     # On concatene.
  augment(u,v,w);    # On augmente.
```

¹⁶Encore une fois, cela ne marche que si l'on a chargé la librairie «linalg» en mémoire.

On peut sortir les colonnes ou les lignes d'une matrice par les commande 'row(...)' et 'col(...)'. Attention cependant au type qui est renvoyé par Maple.

```
> MM := matrix([[1,2,5,3],[1,5,2,3],[4,2,3,1],[-2,5,4,5]]);
row(MM,1);      # On extrait la ligne 1 de MM.
col(MM,2);      # On extrait la colonne 2 de MM.
```

Matrice transposée et inverse. La commande 'transpose(...)' remplace une matrice par sa transposée.

```
> u:=matrix(2,2):
v:=matrix(2,3):
w:=matrix(2,1): # trois matrices.
evalm(u);      # La matrice u.
transpose(u);  # transposé de u.
evalm(v);      # La matrice v.
transpose(v);  # transposé de v.
evalm(w);      # La matrice w.
transpose(w);  # transposé de w.
```

La commande 'det(...)' permet d'obtenir le déterminant d'une matrice. La commande 'inverse(...)' permet de donner l'inverse d'une matrice, lorsqu'il existe (i.e.: lorsque le déterminant n'est pas nul).

```
> mm:=matrix([[a,b],[c,d]]); # Une matrice.
det(mm);                  # Son déterminant.
inverse(mm);              # Son inverse.
```

6.3 Solutions des systemes linéaires.

L'un des interet principal des matrice est qu'elle permettent de résoudre des équations linéaires de la forme : $AX = B$, où A est une matrice $n \times n$ et B un vecteur colonne de dimension n . Maple permet de résoudre ce type de systeme en utilisant la commande 'linsolve(A,B)'.¹⁷

¹⁷La commande 'linsolve(...)' fait parti de la librairie «linalg» et elle a quelques options que je vous invite à regarder, dans l'aide-en-ligne, via la commande '?linsolve'.

```

> A:=matrix([[1,2,3],[2,1,1],[4,2,1]]); # Une matrice.
  B:=vector([1,4,0]); # Un vecteur.
  linsolve(A,B); # Solution de A X = B.

```

6.4 Exercices.

Exercice 24.

1. Soient $u = [8, 2, 9]$; $v = [2, 3, -5]$ et $w = [15, -3, 7]$. Calculez le produit vectoriel de u par v puis le produit scalaire du résultat par w . Calculez le produit vectoriel de v par w puis le produit scalaire du résultat par u .
2. Soient, maintenant U , V et W trois vecteurs quelconques de dimension 3.
3. calculez le produit vectoriel de U par V puis le produit scalaire de la résultante par W . Calculez le produit vectoriel de V par W puis le produit scalaire de la résultante par U . Concluez.

Exercice 25.

1. Calculez l'angle et la distance entre les vecteurs $[3, 2]$ et $[5, -7]$. Retrouvez ce resultat en passant par les nombres complexes.
2. Trouvez une base pour l'espace engendré par les vecteurs suivants : $v_1 = [2, 4, 5, 6]$, $v_2 = [1, 2, 5, 3]$, $v_3 = [3, 1, -1, 0]$ et $v_4 = [2, 4, 5, 6]$. Donnez la dimension de cet espace. Est-ce que le vecteur $w = [-2, 1, 1, 3]$ est un element de cet espace ? Si oui, exprimez le comme combinaison des vecteurs de la base.

Exercice 26.

1. en utilisant des matrices, determinez les solutions du systeme suivant :

$$\begin{aligned}
 3x + 2y - z &= 1 \\
 2x - 3y + 4z &= 3 \\
 x + y - 3z &= 2
 \end{aligned}$$

2. Trouvez le rang de la matrice définissant le systeme suivant :

$$\begin{aligned}
 3x + 2y + z &= 1 \\
 2x - 3y + 5z &= -8 \\
 x + y &= 1
 \end{aligned}$$

Y-a-t-il une solution ? Si oui, trouvez la.

Exercice 27.

Exercice 28.

1. Créez une liste des neuf premiers carrés. Convertissez cette liste en une matrice 3×3 , nommée C , dont les lignes coïncident avec cette liste. Trouvez le déterminant de cette matrice.
2. (Pour les matheux en herbe :) Trouvez l'inverse de cette matrice en utilisant la matrice étendue $(C|Id)$ et les opérations élémentaires sur cette matrice. Vérifiez votre résultat.

7 Introduction à la programmation.

7.1 Les Expressions logiques.

7.1.1 Connecteurs logiques.

Nous avons vu au chapitre 5 les différents type de données sous Maple. Nous nous intéresserons ici au type booléen («boolean») avec les connecteur logiques «et», «ou» et «non» (`and`, `or` et `not`).

Pour illustrer l'utilisation des connecteurs logique, il est pratique d'introduire la commande Maple `is(...)`, qui determine si une expression possède (ou non) certaines propriétés.

Par exemple, on peut savoir si un nombre est dans un interval donné (l'intervale se determine à l'aide de la commande `RealRange(...)`), ou bien si une fonction est continue, ou encore si un nombre est un entier, etc.

La commande `type(...)` permet de verifier si une expression est du type demandé. Cette commande retourne également «true» ou «false». Cette commandes ets moiuns complete que la commande `is(...)`, dans le sens où elle ne permet de vérifier que les type de donnée et pas autre chose.

```
> # Est-ce que 3 est dans l'intervale [2,4] ?
is(3,RealRange(2,4));
# Est-ce que la fonction sinus est continue ?
is(sin,continuous);
is(5/2,integer);      # Est-ce que 5/2 est un entier ?
type(1,integer);     # Est-ce que 1 est un entier ?
type(.5,fraction);   # Est-ce que .5 est uen fraction ?
# Est-ce que a*b est une multiplication ?
type(a*b,'*');
type(a*b,'+');      # Est-ce que a*b est une addition ?
```

true

true

false

true

false

true

false

Fort de ces commandes, nous pouvons disséquer un peu plus sur les connecteurs logiques en eux-même.

Les connecteurs logiques **and** et **or** sont appliqués à une paire de proposition alors que le connecteur **not** est appliqué à un élément unique. Ces opérateurs renvoient soit «true» (si l'expression considérée est vrai) soit «false» (dans le cas contraire).

Notons que c'est exactement la même chose que en logique mathématique, on a donc les tableaux de vérité suivants :

<i>and</i>	<i>V</i>	<i>F</i>	<i>or</i>	<i>V</i>	<i>F</i>
<i>V</i>	<i>V</i>	<i>F</i>	<i>V</i>	<i>V</i>	<i>V</i>
<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>V</i>	<i>F</i>

```
> # Est-ce que Pi n'est pas une fraction ?
not is(Pi, fraction);
# Est-ce que 5/2 n'est pas une fraction ?
not is(5/2, fraction);
# Est-ce que 5 est un entier ?
is(5, integer);
# Est-ce que -5 est un entier positif ?
is(-5, positive);
# Est-ce que -5 n'est pas un entier positif ?
not is(-5, positive);

# Est-ce que -5 est un entier positif ET
# est-ce que 5/2 est une fraction ?
is(-5, positive) and is(5/2, fraction);

# Est-ce que -5 est un entier positif OU
# est-ce que 5/2 n'est pas une fraction ?
is(-5, positive) or not is(5/2, fraction);

# Est-ce que -5 est un entier positif OU
# est-ce que 5/2 est une fraction ?
is(-5, positive) or is(5/2, fraction);
```

true

false

true

false

true

false

false

true

7.1.2 Opérateurs booléens et commandes conditionnelles.

Lorsqu'on a une expression et que l'on veut déterminer si celle-ci est vraie ou fautive (i.e.: la déterminer de façon booléenne), on utilise la commande `evalb(...)`. Voici quelques exemples :

```
> evalb(5>3);           # Est-ce que 5 est plus grand que 3 ?  
evalb(3>5);           # Est-ce que 3 est plus grand que 5 ?
```

```
# Est-ce que 5 > 3 et 3/2 est un entier ?  
evalb(5>3 and type(3/2,integer));
```

```
# Est-ce que 6 n'est pas un nombre premier.  
evalb(not(isprime(6)));
```

```
# Est-ce que Pi est un rationnel OU  
# est-ce que Pi est un entier positif ?  
evalb(type(Pi,rational) or type(evalf(Pi),positive));
```

true

false

false

true

true

Nous allons voir maintenant ce que l'on appelle en programmation les commandes conditionnelles, c'est à dire les commandes qui permettent d'exécuter une partie d'un programme si une condition est réalisée. Il s'agit de la commande `if`, dont la syntaxe en langage Maple est :

```
if condition_bouleenne
  then ...
  elif ...
  else ...
fi;
```

Les commandes `elif` et `else` sont optionnelles. Par contre, le `fi`, à la fin, est indispensable. Nous allons voir une application de la condition `if` dans un exemple.

La commande `dd`, qui depend de deux variables x et y . `dd` evalue le dénominateur commun à x et y (commande `denom(...)`) si x et y sont rationnels, sinon si x ou y est rationnel, `dd` evalue le dénominateur de celui qui est rationnel et enfin, si aucun n'est rationnel, `dd` renvoie l'infini.

```
> dd:=(x,y) ->
  if type(x,rational) and type(y,rational)
  then denom(x+y)
  elif not(type(x,rational)) and type(y,rational)
  then denom(y)
  elif not(type(y,rational)) and type(x, rational)
  then denom(x)
  else infinity
fi:

# Quelques exemples...
dd(7/5,5/3);
dd(exp(1),5/3);
dd(7/5,exp(1));
dd(evalf(Pi),exp(1));
```

15

3

5

∞

Voici un autre exemple, dependant d'une seule variable réelle x , La fonction `ent` retourne l'entier le plus proche du nombre x .

```
> ent:= x->
  if abs(x-round(x))<1/4
  then round(x)
  else (floor(x) +ceil(x))/2
fi:

# Quelques exemples...
ent(4.8);
ent(4.7);
ent(4.3);
ent(4.2);
```

5

9/2

9/2

4

S'il n'y a que deux alternatives dans une commande donnée, on peut alors utiliser la forme abrégée `'if'(expr, alt1, alt2)`, qui renvoie `alt1` si `expr` est vrai et `alt2` sinon.

```
> Moy:= x->
  if x > 0
  then 1/2
  else 0
fi :

# Deux exemples.
Moy(7.4);
Moy(-7.9);

# Une autre facon de programmer cette fonction.
aMoy:= x-> 'if'(x>0, 1/2, 0) :

# Ca marche pareil.
```

```
aMoy(7.4);  
aMoy(-7.9);
```

```
1/2  
0  
1/2  
0
```

7.2 Operations répétées et suites.

Nous avons déjà vu la commande `seq(...)` qui permet de créer une suite «arbitraire». Il existe également la commande `sum(...)` qui permet de faire la somme d'une suite donnée et qui a la même syntaxe que la commande `seq(...)`.

```
> seq( sin(n*theta), n=1..8 ); # La suite des valeurs.  
sum( sin(n*theta),n=1..8); # La somme des valeurs.
```

```
sin(theta), sin(2theta), sin(3theta), sin(4theta), sin(5theta), sin(6theta), sin(7theta), sin(8theta)  
  
sin(theta) + sin(2theta) + sin(3theta) + sin(4theta) + sin(5theta) + sin(6theta) + sin(7theta) + sin(8theta)
```

Le même résultat (generer une suite ou bien la sommer) peut etre obtenu en utilisant la commande `for ...while ...do ...od`. La syntaxe de cette commande est illustée par les exemples suivants :

```
> S:=sin(theta): # initialisation de S.  
for n from 2 to 8 # Boucle for.  
do  
  S:=S,sin(n*theta) # Action de la boucle.  
od:  
S; # Affichage de S.
```

$\sin(\theta), \sin(2\theta), \sin(3\theta), \sin(4\theta), \sin(5\theta), \sin(6\theta), \sin(7\theta), \sin(8\theta)$

Ici la variable S a une valeur initiale $\sin(\theta)$. Les termes suivants sont déterminés en ajoutant le terme $\sin(n\theta)$ à la suite précédente, pour n variant de 2 à 8. Comme le `fi` à la fin de la commande `if`, le `od` est indispensable à la fin de la commande `do`.

On peut choisir l'incrément, en plaçant un `by` entre le `from` et le `to` :

```
> S:=sin(theta):          # Initialisation de S.
  for n from 2 by 2 to 8 # Boucle for.
  do
    S:=S,sin(n*theta)
  od:                      # Fin de la boucle.

  S;                       # Affichage de S.
```

$\sin(\theta) + \sin(2\theta) + \sin(3\theta) + \sin(4\theta) + \sin(5\theta) + \sin(6\theta) + \sin(7\theta) + \sin(8\theta)$

Si l'on veut une somme, plutôt qu'une suite, il suffit de remplacer la virgule (,) par un plus (+). Ou bien si l'on veut une multiplication à la place, etc.

```
> S:=a[1]*sin(theta):    # Initialisation de S.
  for n from 2 to 8      # boucle for.
  do
    S:=S*sin(n*theta)
  od:                    # Fin de la boucle.

  S;                     # Affichage de S.
```

$\sin(\theta) \sin(2\theta) \sin(3\theta) \sin(4\theta) \sin(5\theta) \sin(6\theta) \sin(7\theta) \sin(8\theta)$

Voici un autre exemple qui introduit une autre façon de générer les index, ainsi qu'une autre façon d'initialiser les listes.

```

> S:=NULL:           # Initialisation de S.
  for n in {1,4,7,10,14} # Boucle for.
  do
    S:=S,sin(n*theta)
  od:                # Fin de la boucle.

S;                  # Affichage de S.

```

$\sin(\theta), \sin(4\theta), \sin(7\theta), \sin(10\theta), \sin(14\theta),$

La commande `while(...)` permet de faire une opération tant qu'une condition reste vraie.

```

> S:=NULL:           # Initialisation de S.
  n:=1:              # Initialisation de n.
  while n < 14       # Boucle while.
  do
    S:=S*sin(n*Pi/7);
    n:=n+1:
  od:                # Fin de la boucle while.

S;                  # Affichage de S.

```

7.3 Introduction aux procédures.

Les procédures Maple sont une séquence de commandes dépendantes d'un nombre fini de variables, qui permettent d'obtenir un résultat de manière simple. Cela peut être vu comme une généralisation des fonctions mathématiques, ou bien des "programmes" en langage Maple. Voici la structure générale d'une procédure :

```

proc (suite_d'argument)
  local suites_de_noms;
  global suite_de_noms;
  options suite_de_noms ;
  description suite_de_chaines;
programme
end;

```

Les parties indispensables sont le 'proc()' au début, le programme et le 'end' à la fin. S'il doit y avoir des variable à entrer, on les mets en tant qu'arguments. Si on veut spécifier le type de la variable, on mets deux ':' après le nom de celle-ci, suivi du type voulu.

Toutes les autres parties de la structure sont optionnelles. Les noms après 'local', permettent de définir des variables locales à la procedure. Les noms après 'global' permettent de définir des variables globales.

'options' et 'description' ne servent pas pour le programme en tant que tel, mais pour les fioritures que l'on peut mettre autour du programme.

Les options sont de la forme remember, builtin, system, operator, arrow, trace, package et Copyright. Nous détaillerons certaines options dans un prochain cours.

Voici un exemple de procedure :

```
> AP:= proc(x,y,z) # Debut de la procedure AP.
  (x+y)^z
end;                # Fin de la procedure.

# Quelques exemples...
AP(a,b,c);
AP(1,5,2);
```

Bien sur les procedures peuvent etre plus complexes que la precedente (nous verrons cela dans un prochain cours). Maple permet de lire certaines des procedures qui sont implémenté dans son code. pour cela il faut mettre la variable 'verboseproc' à 2, puis l'on peut voir comment sont faites certaines procedures (pas toutes cependant).

```
> interface(verboseproc=2):

# A quoi ressemble la fonction log.
print(log);
```

Voici une procedure qui determine si une expression appartient à uen liste donné et si c'est le cas, elle donne sa prosition dans la liste.

```
> MEMBER:= proc(x::anything,L::list) # Debut de la procedure
  local i;                          # Variable locale.
  for i from 1 to nops(L)           # debut boucle for.
  do
```

```

        if L[i]=x                # Est-ce que x = L[x] ?
            then RETURN("oui, à la position ",i)
        fi;
    od;                            # Fin boucle for.
    'non';
end;                                # fin de la procedure.

# Quelques exemples.
MEMBER(oiseau, [poissons,reptile,oiseau,animal,insecte]);
MEMBER(17, [seq(3*i,i=1..7)]);

```

Notez l'apparence de la commande return, qui entraine un arret immédiat de la procedure. Regardez l'aide à ce sujet, via l'aide en ligne '?RETURN'.

7.4 Exercices.

Exercice 29.

1. Trouvez toutes les racines (en nombres flottants) du polynome $x^3 + ax + b$ lorsque a et b sont des entiers compris entre -5 et 5 .
2. En utilisant les commandes 'seq(...)', 'isprime(...)', et 'select(...)', Trouvez tous les nombres entiers entre 1235 et 1270.
3. Faites une liste ordonnée des entiers inferieurs à 200 divisibles soit par 2, soit par 3 soit par 5.

Exercice 30.

1. Faites une procedure, appelée Racine(c,d) (avec a et b entiers) qui calcule les racines du polynome $x^3 + ax + b$. Pour a et b compris entre c et d .
2. Faites une procédure, appelée PrimeSeq(m,n) (avec $m < n$ entiers) qui donne la liste des nombres premiers compris entre m et n .
3. Faites une prcedure, appelée IntDiv(p,q,r) (avec p , q et r premiers) qui donne la liste des entiers inferieurs à 200 divisibles soit par p , soit par q , soit par r .

Exercice 31.

1. Trouvez tout les triplets d'entiers (a, b, c) tels que $a^2 + b^2 = c^2$ et $c < 50$. Repetez cette operation si la puissance est 3, 4 ou 5. Comparez ce resultat avec le dernier théorème de Fermat.

2. (Pour les informaticiens en herbe.) Tracez une serie de graphe de fonction à deux variables (appelées $f(x,t)$) tels que sur chaque graphe, la fonction est tracée suivant x avec t fixe. On tracera autant de graphe qu'il y a de nombres entiers entre a et b (a,b étant eux-meme entiers donnés). Chaque graphe doit porter le titre " $f(x,t)$ pour $t=t[0]$;". En clair, on a $a < t < b$ et $c < x < d$.
3. Testez votre procedure pour les fonctions : $f(x,t) = \sin(x - 2t)$ avec ($a = 0, b = 5, c = -3, d = 3$) et $f(x,t) = \exp(-x^2 + 2 * x * t)$ avec ($a = 0, b = 5, c = -10, d = 10$).

8 Programmation.

8.1 Les suites.

8.1.1 Suites définie par une fonction.

Nous avons déjà vu certaines suites, lorsque nous avons vu les fonctions. Les suites qui étaient définie par $u_n = f(n)$ où f est une fonction définie précédemment, comme dans l'exemple suivant :

```
> u:= n->sqrt(n)+3; # Definition de la suite.
    u(5);           # Calcul du 5e terme.
```

Cependant, nous n'avons pas défini une suite (c'est à dire une application de \mathbb{N} dans \mathbb{R}), mais une fonction (c'est à dire de \mathbb{R} dans \mathbb{R}). En effet, Maple accepte de calculer la valeur de u pour n'importe quel nombre réel, comme nous le voyons ici :

```
> u:= n->sqrt(n)+3;
    u(7.5);
```

Si l'on veut définir une suite proprement, il faut définir une procédure qui ne prendra que des entiers en arguments. Par exemple, pour définir la suite $v_n = \sqrt{n} + 3$, on procède comme suit :

```
> # On ne prend que des entiers comme argument.
    v:= proc(n::integer)
        sqrt(n)+3;
    end;
```

Notez qu'alors si on tente de calculer une valeur en un nombre non-entier, Maple renvoie un 'warning', comme le montre l'exemple suivant :

```
> # On ne prend que des entiers comme argument.
    v:= proc(n::integer)
        sqrt(n)+3;
    end;

# evaluation en un nombre pas entier.
v(7.5);
```

Notez également que v est définie pour tous les entiers (positifs et négatifs). Si on veut définir v uniquement pour les entiers strictement positifs, il suffit de remplacer 'integer' par 'positive'

De même, si l'on veut définir v pour les entiers positifs ou nuls (les entiers non-négatifs) il suffit de remplacer 'positive' par 'nonneg'.

8.1.2 Les suites définies par récurrence.

Nous allons maintenant voir comment programmer les suites définie de la façon suivante :

$$\begin{aligned} u_0 &= k \\ u_n &= f(u_{n-1}) \end{aligned}$$

Où k est un nombre réel donné et f une fonction de \mathbb{R} dans \mathbb{R} .

Par exemple, nous voulons définir la suite :

$$\begin{aligned} u_0 &= 3 \\ u_n &= \frac{u_{n-1}}{3} + 2 \end{aligned}$$

On procède comme suit : si $n = 0$ alors la procedure va renvoyer 3, sinon, on lui fait calculer $\frac{u_{n-1}}{3} + 2$.

```
> u:= proc(n::nonneg)      # Debut de la procedure.
  if n=0                  # Si n=0
    then 3;               # On retourne 3.
    else 1/3 * u(n-1) +4; # Sinon on calcule le terme
                          # précédent.
  fi;
end;                      # Fin de la procedure.

# Le calcul de u[5]; se fait alors comme suit :
u(5);
```

Parfois, mais c'est assez rare, on peut accler la vitesse de calcul d'une suite définie par recurence en ajoutant l'option «remember». Cette option fait générer une table de souvenir à Maple. Mais attention, cela prend beaucoup de place en mémoire vive (RAM), donc mieux vaut en avoir beaucoup.

De plus si l'on change (même légèrement) une suite avec qui contient cette option, la table de souvenir n'est pas oublié pour autant, donc cela peut entrainer des erreurs de calculs assez importantes¹⁸

¹⁸En fait, l'expérience montre que l'option remember est assez inutile. Je ne saurai que trop vous deconseiller de l'utiliser.

8.2 Procédures itératives.

On dénomme ainsi les procédures qui utilisent les boucles (de type `for` ou `while`). Nous avons déjà vu ce type de procédures dans l'introduction à la programmation en langage Maple.

8.2.1 Un algorithme de tri.

Un algorithme (assez lent cependant), consiste à rechercher l'élément le plus petit dans la liste et à échanger l'élément le plus petit avec l'élément en première position. Puis on recommence avec la liste privé de son premier élément, jusqu'au classement complet de la liste.

Note : Il peut arriver lorsque vous programmez que Maple accepte de compiler un code, mais refuse d'exécuter ce dernier avec l'erreur suivante «illegal use of a formal parameter». Cela arrive lorsque vous essayez de modifier (en lui attribuant une valeur, par exemple) un paramètre que vous avez passé en argument. En effet, Maple prend les données en argument comme des «valeurs» et non des «variables».

Par exemple, on veut écrire un programme qui prend un entier en argument, et qui le divise par 2 s'il est pair. Une (très mauvaise) idée est d'écrire le programme suivant :

```
> essai:=proc(n::integer)
  if n mod 2 = 0      # Si n est pair
    then n:=n/2;    # Alors on le divise par 2
  fi;
  n;                 # On retourne l'entier n.
end;

# On teste...
essai(12)

# ... Et ça ne marche pas.
```

On a procédé à un usage illégal d'un paramètre formel. C'est-à-dire que Maple prend l'argument de la procédure 'essai', non pas pour une variable à laquelle on peut affecter une valeur, mais uniquement, pour la valeur avec laquelle on a testé la procédure.

Un (bon) moyen pour éviter ce problème est de créer une variable locale, qui aura la même valeur que l'argument de la procédure, mais qui sera vu par

Maple comme une variable. Ainsi on corrige le programme précédent comme suit :

```
> essai_bis:=proc(n::integer)

# On defini une variable locale qui aura
# la même valeur que l'argument.

local parametre;
parametre:= n;

# On effectue le test avec le paramètre.
if parametre mod 2 = 0
  then parametre:=parametre/2;
fi;

# On affiche le parametre.
parametre;
end;

# On teste...
essai_bis(12);

# ... Et ca marche.
```

8.2.2 La suite de Syracuse en itératif.

J'explique ici le principe de la suite de Syracuse, vous aurez à la programmer vous même, en exercice. On part d'un entier positif (non-nul), s'il est pair on le divise par 2. Sinon on le multiplie par 3 et on ajoute 1. On recommence cette operation indefiniment.

On constate aisement que lorsqu'on part de 1, on obtient le parcours suivant : 1, 2, 4, 1, 2, 4, ... En fait une conjecture dit que tout nombre entier retombe sur la boucle 4, 2, 1.

8.3 Procédures récursives.

Les programmes recursifs sont les programmes qui se rapellent eux-meme (comme les suites definies par recusivites). Un exemple simple de programme recursif est la fonction factorielle. En effet l'on a la propriété suivante : $n! =$

$n(n-1)!$ et $0! = 1$ (ce qui est la condition de sortie du programme). Ce qui se programme comme suit :

```
> facto:=proc(n::nonneg)
  if n=0          # Si n=0 alors n!=1
  then 1;
  else n*facto(n-1) # Sinon on applique la condition
                  # de recurrence.
  fi;
end;
```

Une erreur souvent commise dans les procédures récursives, est l'oubli de la condition de sortie du programme. Attention, si vous oubliez cette condition, le programme se retrouve dans une boucle infinie, qui a comme effet de faire geler Maple (dans le meilleur des cas) ou l'ordinateur (dans le pire).

8.4 Exercices.

Exercice 32.

Programmer les suites suivantes :

1. $u_n = 5n + 3 - \sqrt{n}$
 2. $u_0 = 5$ et $u_{n+1} = u_n^3$;
 3. $u_0 = 1$ et $u_1 = 2$ et $u_{n+1} = \frac{u_n-1}{4} + 7$
 4. La suite de Fibonacci : $u_0 = 1$ $u_1 = 1$ et $u_n = u_{n-2} + u_{n-1}$.
- Reprogrammez cette suite avec l'option «remember».

Exercice 33.

1. Ecrire une procédure qui trie une liste par la recherche de son plus petit élément. Vérifiez votre code avec la liste $[2, 1, 4, 3, 17, 5]$.

Exercice 34.

1. Etant donné un entier n , programmez le «parcours» de Syracuse de l'entier à 1, en itératif.
2. Etant donné un entier n , programmez le «parcours» de Syracuse de l'entier à 1, en récursif.

9 Programmation avancée.

Il est possible de sauver des données dans des fichiers en utilisant Maple, de même que de lire des fichiers textes ou au format Maple. De plus, on peut convertir une grande partie des programmes Maple en d'autres langages (C, Fortran, etc.) ou formats (TEX, L^AT_EX, HTML, etc.)

Nous allons voir ici comment faire pour réaliser de telles opérations.

9.1 Lire et écrire dans des fichiers.

9.1.1 Sauver des fichiers avec Maple.

La façon la plus simple de sauver un document Maple est celle que vous utilisez depuis le début de ce cours, en cliquant sur l'icône «save», en haut à gauche de la barre de menu. Ainsi, vous obtenez un fichier avec l'extension ¹⁹ «.mws» pour «Maple Work Sheet» (soit «Feuille de travail Maple»).

De tels fichiers sont exportables d'un ordinateur à l'autre (et ce, quelque soit le système d'exploitation), pourvu que les ordinateurs concernés aient tous les deux Maple installé.²⁰

Lorsqu'on utilise cette méthode pour sauver des données, on peut également choisir de sauver au format «Maple Text», ce qui a pour effet de sauver le document au format texte brut, avec des signes mathématiques en ASCII-art. Notez que dans ce cas, si l'on édite le fichier ainsi sauvé, avec un éditeur de texte standard, le texte est précédé d'une dièse et d'une espace (#), tandis que les entrées Maple sont précédés d'un signe supérieur et d'une espace (>).

9.1.2 Sauver, écrire et lire des données dans un fichier

La commande 'save(...)' crée un fichier texte écrit en langage Maple. Il est préférable d'entourer le nom du fichier par des guillemets anglais ("), de manière à ce que Maple interprète le nom du fichier comme une chaîne (et non comme une variable, ou autre).

Les fichiers en langage Maple sont des fichiers codés spécialement pour retenir l'ensemble des informations dans un format le plus compact possible. De tels fichiers peuvent être lus (ou créés) avec un simple éditeur de texte.

Faites attention à l'endroit où est sauvé votre fichier (il vaut mieux mettre un chemin absolu, plutôt que relatif, pour plus de certitude). notez cependant

¹⁹Du moins, si votre système d'exploitation affiche les extensions des fichiers...

²⁰Notons cependant des problèmes au niveau de l'encodage des accents entre un fichier sauvé sur un Macintosh et un PC. Mais ce problème est générique et en rien lié à Maple.

que dans les exemples suivants, on utilisera un chemin relatif, la syntaxe des chemins absolus dépendant des systèmes d'exploitations.

Si l'on ajoute l'extension «.m» à la fin du nom du fichier de sauvegarde, le fichier sera enregistré dans un format spécial. À l'aide d'un éditeur de texte, comparez les fichiers «temp» et «temp.m» suivants :

```
> f:=x->x*sin(2*x);      # Une fonction f.
g:=exp;                 # Une fonction g.
h:=g@f;                 # La composé des deux fonctions.

# On crée une procédure...
OD:=proc(m::integer,n::integer)
  local i, istriple;
  istriplei := i ->
  'if'(type((i-1)/2,integer) and type(i/3,integer), i, NULL);
  [seq(istriple(i), i=m..n)]
end:

# On sauve f, g, h et OD dans le fichier temp
save f,g,h,OD,"temp";
# On sauve f, g, h et OD dans le fichier temp.m
save f,g,h,OD,"temp.m";
```

Pour récupérer des données sauvées dans un fichier, on se sert de la commande 'read(...)'. De même, il est préférable d'utiliser un chemin absolu, plutôt que relatif, pour être certain d'ouvrir le bon fichier.

```
> read("temp");        # On lit le fichier temp.
h(theta);              # On peut se servir des fonctions
OD(10,40);             # qui sont dedans...
```

À la place de lire le fichier en entier, il est possible de lire un fichier ligne par ligne, avec la commande 'readline(...)', ou d'écrire une ligne dans un fichier, à l'aide de 'writeline(...)'. Notez que si un fichier n'a pas encore été ouvert, il est lu depuis sa première ligne. Si un fichier a déjà été ouvert, Maple se souvient de la ligne où il était et lit la ligne suivante. Une fois arrivé à la fin du fichier, Maple reprend à la première ligne.

```
> # Lecture de temp
readline("temp");
```

```

readline("temp");
readline("temp");
readline("temp");

# Ecriture dans temp
writeline("emp,"hello", "goodbye");

```

Pour écrire l'ensemble des sorties de Maple dans un fichier, plutôt que le faire apparaître sur le terminal, on utilise la commande 'writeto(...)'. Si l'on veut rediriger la sortie vers le terminal, il suffit de taper la commande 'writeto(terminal)'.

```

> writeto("temp"); # On redirige la sortie vers temp.
aa:=i->a*r^i;
ss:=n->simplify(sum(aa(i),i=1..n));
ss(n);
writeto(terminal); # Retour vers le terminal.
ss(n);

```

Notez que la commande 'writeto(...)' crée un nouveau fichier. S'il existe déjà un fichier avec le nom indiqué dans la commande, son contenu sera écrasé au profit du nouveau. Pour ne pas écraser un fichier existant, et rajouter des données à la fin, on se sert de la commande 'appendto(...)'.
On peut générer des sorties préformatées, soit sur le terminal, soit dans un fichier, à l'aide de la commande 'writedata(...)', consultez l'aide-en-ligne pour plus de détails à ce sujet.

9.2 Conversion vers d'autres langages et génération de code.

Maple permet de générer d'autres langages, tels que le Fortran, le C ou d'autres présentations, tels que T_EX, L^AT_EX ou du HTML.

9.2.1 Conversion des commandes et procédures en Fortran et en C.

Pour convertir des commandes en fortran, on se sert de la commande 'fortran(...)'. Pour plus de détails sur ce langage historique, je vous invite à consulter l'aide en-ligne.

Une commande similaire existe pour les conversions en langage C, mais il faut avant tout utiliser la commande 'readlib(C)'. Encore une fois, pour plus de détails consultez l'aide en ligne. Voici quelques exemples :

```
> S:=proc(x,y)      # Une procedure.
      x*log(y)
end;

fortran(S);        # Conversion en Fortran

readlib(C);        # Lecture de la bibliothèque C.
C(ss,optimized);  # Conversion en C (optimisée)
```

9.2.2 Conversions de Maple en L^AT_EX

Une simple expression peut être convertie en L^AT_EX en utilisant la commande ...'latex(...)'. Encore une fois, plutôt que d'avoir la sortie sur le terminal, on peut l'envoyer dans un fichier.

```
> # Une integrale compliquée.
EI:=Int(1/sqrt(x^3+1),x=1..X)=int(1/sqrt(x^3+1),x=1..X);
latex(EI);          # On traduit la sortie en LaTeX
# On sauve la sortie dans un fichier nommé temp.
latex(EI, "temp");
```

Notez qu'il est également possible, avec l'interface graphique, de sauver toute une feuille de travail Maple en un fichier L^AT_EX (via le menu «Export As...»).

Via ce menu, on peut également sauver un tel fichier en HTML.

Il faut cependant être réaliste : le code généré par Maple (que ce soit en HTML ou en L^AT_EX) est illisible avec un éditeur de texte, comme avec tous les éditeurs «WYSIWYG»²¹.

Si vous voulez compiler un document en L^AT_EX, converti à partir d'un feuillet Maple, vous aurez besoin de certaines bibliothèques développées pour (et par) Maple.

Lors de la conversion en HTML, c'est encore pire : toutes les formules mathématiques deviennent des images et ainsi une simple page HTML peut atteindre des centaines de ko...

²¹WYSIWYG est l'acronyme de What You See Is What You Get, soit «Ce que vous voyez est ce que vous obtenez.»

Bref, même si c'est possible, je ne saurais que trop vous deconseiller de faire de telles conversions. Apprenez plutôt à écrire du HTML (et du L^AT_EX) cela vous sera nettement plus utile.