Programme Commun Mathématiques Informatique
PCMI

CALCUL FORMEL

MAPLE & CAML

http://lguezouli.univbatna.com/
ou
http://fac-sciences.univ-batna.dz/cs/enseignants/guezouli_larbi_site/index.html

Présenté par Dr Larbi GUEZOULI

Calcul formel

- Chapitre I: Introduction [1 cours]
  - Définition
  - Utilité du calcul formel
  - Structures de base
  - Calcul rapide
  - Bibliographie
- Chapitre II: Rappels d'arithmétique [2 cours]
  - PGCD et l'algorithme d'Euclide
  - Fonction d'Euler
  - Distance de Hamming
  - Cryptographie (César, Hill)
- Chapitre III: MAPLE [4 cours]
  - Introduction
  - Premiers pas avec MAPLE
  - Les variables
  - Arithmétique en nombre entiers
  - Nombres complexes
- Chapitre IV: Objectif CAML O'CAML & CAML Light [6 cours]
  - Introduction
  - Instructions, testes et boucles (let, if, for)
  - Les types (float, int, ref)
  - Fonctions
  - Tableaux et chaînes de caractères (array, char, string)
  - Valeurs booléennes et aléatoires (bool, random)
  - Boucle while et récursivité (while, rec)

Chapitre I

Chapitre I: Introduction
- Définition
- Utilité du calcul formel
- Structures de base
- Calcul rapide
- Bibliographie

Chapitre I
Introduction

Définition

Le **calcul formel** traite les **grands nombres**, sur lesquels les opérations sont de plus en plus **longues** à être **exécutées**.

Exemple: pour le calcul du déterminant d'une matrice $A = (a_{i,j})_{1 \le i,j \le 25}$

Il faut **25!** opérations de produit

Tel que $25! \approx 1{,}55.10^{25}$

Sur un ordinateur qui fait 10 milliards de produits par secondes, il faut 498×10^6 années pour calculer ce déterminant.

Chapitre I
Introduction

Utilité du calcul formel

Avec cet exemple on peux voir l'utilité du calcul formel:

Calculabilité

Trouver des algorithmes qui permettent de faire de grands calculs d'une façon plus rapide.

Efficacité

Les calculs établis devraient être sans erreurs

Chapitre I
Introduction

Donc un système de calcul formel permet:

- d'utiliser des grands nombres (entiers, réel…)
- d'utiliser des polynômes à plusieurs variables
- de calculer les limites, les dérivées…
- de simplifier de formules
- de résoudre des équations différentielles
- …

Chapitre I
Introduction

Il existe plusieurs logiciels de calcul formel, on peut citer:

Mathématica, Mupad, Derive, Maple, Objectif Caml…

Nous choisissons pour notre cours Maple et O'Caml

Chapitre I
Structures de base

Les structure de base les plus utilisées sont:

Les Entiers

Un entier N écrit dans une base B est représenté comme suit:

$$N = \sum_{i=0}^{k} a_i \cdot B^i \Leftrightarrow N = a_0 + a_1 B + a_2 B^2 + ... + a_k B^k$$

avec $0 \leq a_i < B$

Exemple:

$N = 1 + 2.10 + 3.10^2 + 4.10^3 = 4321$

Les Rationnels

Un nombre rationnel N est stocké comme une paire <numérateur, dénominateur>:

$$N = \frac{a}{b}$$

Exemple:

$$N = \frac{5}{2}$$

sera stocké sous forme de couple <5, 2>

Décomposition d'un nombre sur une base

Soit **N** le nombre et **B** la base. Trouver les $\mathbf{a_i}$.

Le quotient de la division **N**/**B** et le reste est **N mod B**.

Algorithme:

```
i ← 0
M ← N
tant que M ≠ 0 faire
   a_i ← M mod B
   i ← i + 1
   M ← M / B
Fin tan que
```


Les Vecteurs et Matrices

Une matrice ou un vecteur est représenté comme un **pointeur** vers un tableau en mémoire.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

En mémoire:

Un pointeur vers tableau

Ptr *p = a_{11}, a_{12}, … , a_{33}

Les opérations de bases sont: **l'addition** de matrices, **multiplication** avec un scalaire, **inverse** d'une matrice…

Les Polynômes

Un polynôme peut être stocké de plusieurs manières. Les plus utilisés sont:

- **Représentation dense**: Le polynôme est représenté par un pointeur vers un tableau contenant les cœfficients.

  Exp: $P_0 = 5.x^3 + 3.x + 2$

  En mémoire: ptr *p = [5, 0, 3, 2]

- **Représentation creuse**: Le polynôme est représenté par une liste de paires <cœfficient, exposant> triée par les exposants.

  Exp: $P_1 = x^5 + 2.x^3 + 5.x$

  En mémoire: liste (<1, 5>, <2, 3>, <5, 1>)

Chapitre I
Calcul rapide

En pratique, l'écriture d'un algorithme est facile si on ne cherche pas à l'optimiser.

Par contre, le problème du calcul formel, la solution donnée doit prendre en compte la **calculabilité** et la **rapidité**.

Une solution doit être optimisé au maximum pour faire le **minimum d'opérations** et **gagner du temps**.

Chapitre I
Calcul rapide

Exemple:

Multiplication d'une matrice diagonale par un vecteur $\begin{bmatrix}1&0&0\\0&2&0\\0&0&3\end{bmatrix}\bullet\begin{bmatrix}1\\2\\3\end{bmatrix}=\begin{bmatrix}1\times1\\2\times2\\3\times3\end{bmatrix}$

au lieu de:

$$\begin{bmatrix}1&0&0\\0&2&0\\0&0&3\end{bmatrix}\bullet\begin{bmatrix}1\\2\\3\end{bmatrix}=\begin{bmatrix}1\times1+0\times2+0\times3\\0\times1+2\times2+0\times3\\0\times1+0\times2+3\times3\end{bmatrix}$$

On gagne le temps de 6 opérations de multiplication et 6 opérations d'addition

Chapitre I
Bibliographie

- Granlund (Torbjörn). - GNU Multiple Precision Arithmetic Library. - http://swox.com/gmp, 2006
- SAUX PICCART P., *Cours de Calcul Formel. Algorithmes fondamentaux* Ellipses, 1999
- *Maple V Language Reference Manual,* by B. Char & al. Springer-Verlag 1991. *First Leaves. A tutorial Introduction to Maple V*, by B. Char & al. Springer-Verlag 1991.
- *Maple Calculus Workbook,* by K. O'Geddes & al., Disponible directement chez Waterloo Maple Software.
- INRIA http://caml.inria.fr/ocaml/index.fr.html

Chapitre II
Rappels d'arithmétique

1. PGCD et l'algorithme d'Euclide
2. Fonction d'Euler
3. Codes et distance de Hamming
4. Cryptographie (César, Hill)

Chapitre II
Rappels d'arithmétique

1. PGCD et l'algorithme d'Euclide

Le plus grand diviseur commun de deux nombres naturels ∈ **N** est calculé en utilisant l'algorithme d'Euclide comme suit:

Soit A ∈ N et B ∈ N et A ≥ B,

$A = a_0.B + r_0$ avec $a_0 \geq 1,\ 1 \leq r_0 < B$

$B = a_1.r_0 + r_1$ avec $a_1 \geq 1,\ 1 \leq r_1 < r_0$

$r_0 = a_2.r_1 + r_2$ avec $a_2 \geq 1,\ 1 \leq r_2 < r_1$

….

$r_{n-5} = a_{n-3}.r_{n-4} + r_{n-3}$ avec $a_{n-3} \geq 1,\ 1 \leq r_{n-3} < r_{n-4}$

$r_{n-4} = a_{n-2}.r_{n-3} + r_{n-2}$ avec $a_{n-2} \geq 1,\ 1 \leq r_{n-2} < r_{n-3}$

$r_{n-3} = a_{n-1}.r_{n-2}$ avec $a_{n-1} \geq 2,\ r_{n-1} = 0$

Le PGCD de **A** et **B** est le dernier reste non nul (r_{n-2}), et nous avons fait **n** divisions ($n \geq 2$)

Chapitre II
Rappels d'arithmétique

exp: soit A=24 et B=10

$24 = 2 \times 10 + 4$ r_{n-5}

$10 = 2 \times 4 + 2$ r_{n-4}

$4 = 2 \times 2 + 0$ $r_{n-3} = r_0 \Rightarrow n=3$

Le PGCD de **A** et **B** est $r_{n-2} = 2$, et nous avons fait **3** divisions.

Chapitre II
Rappels d'arithmétique

Algorithme d'Euclide (version itérative):

```
fonction PGCD(A,B)
   n ← A;
   d ← B;
   r ← n mod d;
   tant que r ≠ 0 faire
         n ← d;
         d ←  r;
         r ← n mod d;
   fin tant que;
   retourner d;
Fin fonction
```

Exp: PGCD(24,10)

n	d	r
24	10	4
10	4	2
4	2	0

Retourner 2

Chapitre II
Rappels d'arithmétique

Algorithme d'Euclide (version récursive):

```
fonction PGCD(A,B)
   si B = 0 alors
         retourner A;
   sinon
         retourner PGCD(B, A mod B);
   fin si
Fin fonction
```

Exp: PGCD(24,10)
PGCD(10,4)
PGCD(4,2)
PGCD(2,0)

Retourner 2

2. Fonction d'Euler

La fonction d'Euler est représentée par le symbole mathématique φ(phi) et définie par :

$\varphi : N \to N$

$n \mapsto \varphi(n) = card(\{a \mid 1 \le a \le n, PGCD(a,n)=1\})$

Exp:

$\varphi(10) = card(\{1, 3, 7, 9\}) = 4$

Théorème

Pour tout $n \in N$:

$$\sum_{d|n} \varphi(d) = n$$

tel que: d|n veut dire « d divise n »

Exp:

n = 10;

$d \in \{1, 2, 5, 10\}$

$\varphi(1) = card(\{1\}) = 1$

$\varphi(2) = card(\{1\}) = 1$

$\varphi(5) = card(\{1, 2, 3, 4\}) = 4$

$\varphi(10) = card(\{1, 3, 7, 9\}) = 4$

$$\Rightarrow \sum_{d|10} \varphi(d) = 10$$

3. Codes et distance de Hamming

a) Introduction

Soit le message binaire $M=(m_1, m_2, \ldots, m_k)$ à envoyer de A vers B. Le message est arrivé à B altéré (avec erreurs) sous la forme $M' =(m'_1, m'_2, \ldots, m'_k)$.

Pour corriger les erreurs de transmission on envoie des messages redondants.

Exemple:

On duplique chaque lettre **r** fois. Donc si **k=1** et **r=3** on aura:

$M=(m_1, m_1, m_1) \to M' =(m'_1, m'_1, m'_1)$

a) Introduction (suite)

Le bit reçu correspond à la majorité des 3 bits comme suit:

$000 \to m=0$

$001 \to m=0$

$011 \to m=1$

$111 \to m=1$

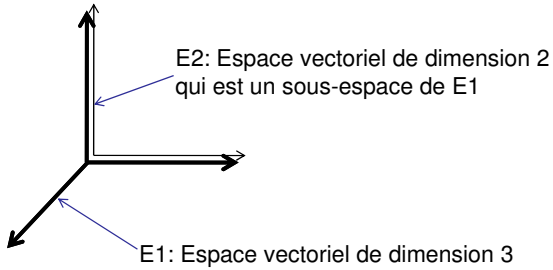
L'inconvénient de cette méthode est que le **message** est **très long**.

Chapitre II
Rappels d'arithmétique

b) Définitions

- $\mathbf{F_2^n}$ est un espace vectoriel de $\mathbf{\{0,1\}^n}$. (*de dimension n*)
- **Un code linéaire C** de longueur **n** et de dimension **k** est un sous-espace vectoriel de dimension **k** de $\mathbf{F_2^n}$.

Exp:



Chapitre II
Rappels d'arithmétique

c) Code de Hamming 7-4

C'est un **code linéaire** de longueur **7** et de dimension **4**.

Dans ce code on utilise **4 bits** à **envoyer** et **3 bits** de **redondance**.

L'algorithme de décodage de Hamming 7-4 utilise une matrice appelée **Matrice de contrôle:**

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Chapitre II
Rappels d'arithmétique

c) Code de Hamming 7-4 (suite)

Le « code de Hamming 7-4 » noté **C** est une partie de $\mathbf{F_2^7}$. Le message à envoyé **M=(x, y, a, z, b, c, d) ∈ C ⇔ H.M = 0**

avec: x, y, z sont les bits de redondance, et a, b, c, d sont les bits à envoyer, tel que:

x = (a+b+d) mod 2

y = (a+c+d) mod 2

z = (b+c+d) mod 2

Quand le receveur reçoit le message **M'** il calcule le **syndrome S**:

S=H.M'

Si S=0 alors il n'y a pas d'erreurs

Si $S = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix} \neq 0$ alors il y a **une erreur** sur le i^ème^ bit tel que **$i=4.s_1+2.s_2+s_3$**.

Chapitre II
Rappels d'arithmétique

c) Code de Hamming 7-4 (suite)

Exp.

Soit le message à envoyer : 1011

M=0110011

Et soit M'=0110111 le message reçus par le récepteur avec une erreur.

$$S = H.M' = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

c) Code de Hamming 7-4 (suite)

Exp. (suite)

Comme $S \neq 0$ donc il y a une erreur dans le i^{ème} bit, tel que:

$i = 4.s_1+2.s_2+s_3 = 4.1 + 2.0 + 1 = 5$

Donc il suffit d'inverser le 5ème bit de M' pour récupérer le bon message:

M' = 0110011 = M

Remarque:

Le nombre de messages qu'on peut envoyer avec le code de Hamming 7-4 est 2^4=16.

d) Code de Hamming 8-4

Ce code permet de **corriger** une erreur et de **détecter** 2 erreurs.

Dans ce code on utilise **4 bits** à **envoyer** et **4 bits** de **redondance**.

L'algorithme de décodage de Hamming 8-4 utilise la matrice de contrôle précédente en ajoutant une **ligne de 1**, et une colonne contenant les **bits de parité** de chaque ligne:

$$H = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

d) Code de Hamming 8-4 (suite)

Le « code de Hamming 8-4 » noté **C** est une partie de $\mathbf{F_2^8}$. Le message à envoyé **M=(w, x, y, a, z, b, c, d) ∈ C ⇔ H.M = 0**

avec: x, y, z sont les bits de redondance, et a, b, c, d sont les bits à envoyer, tel que:

x = (a+b+d) mod 2

y = (a+c+d) mod 2

z = (b+c+d) mod 2

w = (a+b+c) mod 2

Remarque: Le « **w** » est un bit de parité des 7 derniers bits.

Quand le receveur reçoit le message **M'** il calcule le **syndrome S=H.M'**

Si S=0 alors il n'y a pas d'erreurs.

d) Code de Hamming 8-4 (suite)

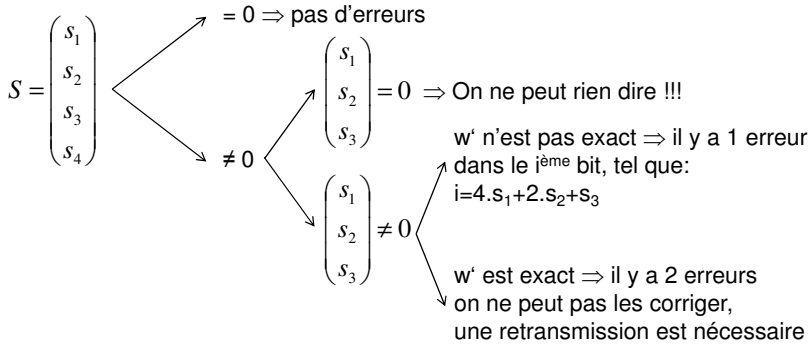
Si $S = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} \neq 0$ alors:

- Si **$(s_1,s_2,s_3) \neq 0$**, alors il y a **une (01) erreur** (selon le code de Hamming 7-4);
- S'il y a **une (01) erreur** sur les **sept derniers bits** et le bit de **parité "w"** n'est pas exact (signal un nombre impair d'erreurs), alors l'erreur est localisée sur le i^{ème} bit tel que **$i=4.s_1+2.s_2+s_3$** (sans compter le bit de parité **"w"**)
- Si le bit de **parité "w"** est exact (signal un nombre pair d'erreurs) et (s_1, s_2, s_3) ne sont pas tous nuls, alors il y a **deux (02) erreurs** sur les **sept derniers bits**, et une **retransmission** est **nécessaire**.

Chapitre II
Rappels d'arithmétique

d) Code de Hamming 8-4 (suite)

Conclusion:



Chapitre II
Rappels d'arithmétique

d) Code de Hamming 8-4 (suite)

Exp.

Soit le message à envoyer : 1011 ⇒ M=00110011

et soit M'=00110111 le message reçus par le récepteur avec une erreur.

$$S = H.M' = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Chapitre II
Rappels d'arithmétique

d) Code de Hamming 8-4 (suite)

Exp. (suite)

w=0 donc il y a une erreur dans le bit numéro:

$i = 4.s_1+2.s_2+s_3 = 5$

Si M'=00100111 nous aurons:

Le bit de parité w=0 qui est exact et $(s_1,s_2,s_3) = (1,1,0) \neq 0$ donc il y a 2 erreurs, et une retransmission est nécessaire.

Chapitre II
Rappels d'arithmétique

e) Distance de Hamming

C'est une distance entre deux vecteurs de même taille $V = (v_1, v_2, \ldots, v_n) \in F_2^n$ et $W = (w_1, w_2, \ldots, w_n) \in F_2^n$. Elle est définie comme suit:

$$\delta(V,W) = \sum_{i=1}^{n} |v_i - w_i|$$

Tel que: F_2^n est l'ensemble des n-tuple avec des **éléments binaires**.

Exp:

V1=(101010) W1=(101010)

δ(V1,W1)=0 (vecteurs égaux)

V2=(101010) W2=(010101)

δ(V2,W2)=6=n (distance maximale)

Chapitre II
Rappels d'arithmétique

Quelques définitions

- **Poids d'un élément** V d'un code **C** tel que: $V = (v_1, v_2, \ldots, v_n) \in C$ est définie comme suit:

$$w(V) = \sum_{i=1}^{n} v_i = \delta(V,0)$$

Exp:

V=(101010) w(V)=3

- **La distance minimale d'un code C** est définie comme suit:

$$d = \min_{V \neq W \in C} \delta(V,W)$$

Exp:

La distance minimale du code C suivant:

C={(001),(010),(100),(111)}

est d=2.

Chapitre II
Rappels d'arithmétique

- **Distance minimale d'un code linéaire**

Soit **C** un code linéaire. La distance minimale de ce code est définie comme suit:

$$d = \min_{V \neq W \in C} \delta(V,W) = \min_{V \in C - \{0\}} w(V)$$

Exp.

La distance minimale du code linéaire C suivant:

C={(000),(001),(010),(011),(100), (101),(110),(111)}

est d=1.

Chapitre III
MAPLE

1. Introduction
2. Premiers pas avec MAPLE
3. Les variables
4. Arithmétique en nombre entiers
5. Nombres complexes
6. Les polynômes
7. Séquences, listes et ensembles
8. Les tests
9. Les boucles
10. Les Procédures
11. Résolution d'équations
12. Vecteurs et matrices
13. Les graphiques

Chapitre III
MAPLE

II.1) Introduction

Maple est créé en 1980 dans une université à Canada.

C'est un langage **interprété**, c.à.d. sans compilateur.

II.1.1) Calcul numérique vs calcul symbolique

Il existe deux catégories de calcul:

Calcul numérique

Comme dans Matlab, Gauss et Scilab, ils possèdent des **algorithmes puissants** pour faire des calculs complexes.

Calcul symbolique

Comme dans Maple, Mathematica ou Mupad, ils manipules des **objets mathématiques** (fonction, polynômes, ..)

Exp. Sur Maple, le rationnel 2/3 est manipulé comme un objet, et représenté en mémoire par un couple <2,3>

et sur Matlab il est évalué en 0,6666667.

II.2) Premiers pas avec MAPLE

Une commande Maple se termine avec ';' ou ':'

Le ';' affiche le résultat dans la ligne suivante

Le ':' exécute l'instruction mais n'affiche pas le résultat

On peut taper plusieurs commandes Maple dans une seule instruction séparées par ';' ou ':', le retour à la ligne « Shift+Return » et la validation de l'instruction « Return »

II.2.1) Les commandes

- Les opérations de calcul algébrique +, -, *, /

Exp. 2*(5+6) ou 9*x

- L'**exposant** se fait par ^ ou **

Exp. 3**2 == 3^2 == 3^2 ou 3**(1/2)

La racine carrée a**(1/2) == sqrt(a)

- **Valeur absolue** : abs(x)
- **Affectation**: x := 5
- Le caractère % représente le **dernier résultat calculé** (et pas le dernier affiché) et %% pour l'avant dernier
- La commande **restart** : redémarre Maple pour effacer tous les calculs précédant
- La commande **evalf** : donne une évaluation proche de ce qui est entre les parenthèses. Exp: evalf(3**(1/2)) == 1,732…
- On peut fixer la précision par **Digits** (par défaut 10) comme:

Digits := 2; ou evalf(3**(1/2), 2)

II.2.1) Les commandes (suite)

- Les lettres grecques : α, β, λ dans Maple *alpha, beta, lambda*…
- L'infinie ∞ dans Maple *infinity*.
- Le nombre π dans Maple Pi, et e est exp(1)
- Les fonctions mathématiques: exponentielle exp(), logarithme log(), trigonométriques sin(), cos(), tan()…

II.3) Les variables

Les variables dans Maple sont des objets mathématiques.

Maple différencie entre **majuscule** et **minuscule** dans les noms de variables.

Le contenu d'une variable peut être calculé en fonction d'une autre variable, comme:

II.3) Les variables (suite)

```
y := 2:
x := 5+y:    (x contient 7)
y := 3;      (x contient toujours 7 et ne
              change pas si y change)
```

unassign('x') permet de vider le contenu d'une variable comme **restart** qui vide tout.

II.4) Utilisation des nombres entiers

Rappel:

x est le **PPCM** (Plus Petit Commun Multiple) de **a** et **b** si **a** divise **x**, **b** divise **x** et s'il n'existe pas de **x'** < **x** dont **a** et **b** divisent **x'**

exp: a=10, b=12, x=60

x est le **PGCD** (Plus Grand Commun Diviseur) de **a** et **b** si **x** divise **a**, **x** divise **b** et s'il n'existe pas de **x'** > **x** qui divise **a** et **b**

exp: a=10, b=12, x=2

Un **nombre premier** est un nombre entier positif qui n'a pas de diviseur autre que 1 et lui-même.

exp: a=7

Deux nombres entiers positives sont **premiers entre eux** s'ils n'ont pas de diviseur commun.

exp: a=7, b=3

II.4) Utilisation des nombres entiers (suite)

Dans une division entière, le quotient dans Maple **iquo(m,n,'r')**, et le reste est **irem(m,n,'q')**

exp: iquo(5,2)=2 et irem(5,2)=1 ou iquo(5,2, 'x')=2 et x=1 le reste.

Le PGCD dans Maple **igcd(x_1, x_2,…)**, et PPCM est **ilcm(x_1, x_2,…)**

exp: igcd(10,12)=2 et ilcm(10,12)=60

Pour vérifier si un nombre est premier **isprime()**

exp: isprime(5)=true et isprime(10)=false

Pour trouver le prochain et le précédent nombre premier par rapport à x **nextprime(x)**, **prevprime(x)**

exp: nextprime(5)=7 et prevprime(5)=3

Pour trouver le $x^{ième}$ nombre premier **ithprime(x)**

exp: ithprime(4)=7 (2, 3, 5, 7)

Pour trouver les facteurs premiers d'un nombre entier: **ifactor(x)**

exp: ifactor(12); → $(2)^2(3)$

II.5) Les nombres complexes

Un nombre complexe (a+ib) dans Maple **a+I*b;**

tel que I == (-1)**(1/2) == sqrt(-1)

Parties imaginaire et réelle sont **Im(c)** et **Re(c)**

Le **module** de **c** est **abs(c)** = sqrt(a^2+b^2);

L'**argument** de **c** est **argument(c)**

argument(x) = t ssi x = abs(x) * e^{I*t}, $-\pi < t <= \pi$

Le **conjugué** de c est **conjugate(c)** = a-I*b

Pour évaluer un complexe: **evalc(x)** pour l'afficher sous forme a+I*b

II.6) Les polynômes

Un polynôme est un objet arithmétique dans Maple.

Deux façons pour définir un polynôme:

- Par une expression: exp. $2x^2+3x+1$

Dans Maple: $P := 2*x**2+3*x+1$

x doit être une variable non affectée.

Pour évalué le polynôme pour (x =1 par exemple) on utilise **subs(x=1, P)**

II.6) Les polynômes (suite)

- Par une fonction polynomiale: exp. $2x^2+3x+1$

Dans Maple: $P := (t) -> 2*t**2+3*t+1$

t est une variable de la fonction et on ne peut pas l'utiliser en dehors du polynôme.

Pour évalué le polynôme pour (t =1 par exemple) on appel le polynôme **P(1);**

II.6) Les polynômes (suite)

Quelques fonction Maple:

- **simplify()** permet de simplifier le polynôme

exp: P:=(x+1)*(x-1); simplify(P); → x^2-1

- **factor()** permet de factoriser le polynôme

exp: P:=2*x^3+3*x^2+x; factor(P); → x(x+1)(2x+1)

- **expand()** permet simplifier et ordonner une expression

exp: expand(cos(a+b)); → cos(a)cos(b)-sin(a)sin(b)
expand(sin(a+b)); → sin(a)cos(b)+cos(a)sin(b)

- **sort()** permet de trier une expression par ordre décroissant des exposants

exp: sort([3,2,1]); → [1,2,3] sort(1+x+x^2); → x^2+x+1

- **solve()** donne les racines du polynôme

exp: solve(f=m*x, x); → f/m solve(x^4-5*x^2+6*x=2, x); → $1,1,\sqrt{3}-1,-\sqrt{3}-1$

- **degree() et ldegree()** donne le degré et le petit degré du polynôme

exp: degree(2/x^2+5+7*x^3,x); → 3 ldegree(2/x^2+5+7*x^3,x); → -2

Chapitre III
MAPLE

II.7) Séquences, listes et ensembles

Une variable dans Maple peut contenir des **objets** comme par exemple les racines d'un polynôme, les nombres premiers entre 1 et 50, …

Ces variables peuvent être des **séquences**, des **listes** ou des **ensembles**.

Chapitre III
MAPLE

II.7.1) Les séquences

Une séquence est une **succession d'objets** séparés par des virgules

On **crée** une séquence par **seq()**

exp. S:=seq(x**3+1,x=1..5); → S=2,9,28,65,126

On **concatène** des séquences en les écrivant séparées par des virgules

exp. S:=seq(x**3+1,x=1..5),seq(2*x,x=1..10); → S = 2, 9, 28, 65, 126, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20

Chapitre III
MAPLE

II.7.2) Les listes

Une liste contient une séquence dont les éléments sont **indexés** et peuvent se **répéter**.

exp. L:=[1,2,3,1,2,3,1,2,3]

- **Créer** une liste en utilisant les **crochets**

exp. L:=[seq(x**2,x=1..5)]; → L=[1,4,9,16,25]

- **nops(L)** retourne le nombre d'éléments dans la liste

exp. nops(L); → 5

- **op(L)** retourne le contenu de la liste

exp. op(L); → 1,4,9,16,25

- **op(n, L)** retourne le n^{ième} élément de la liste

exp. op(2,L); → 4

- **member(x, L)** vérifie si la liste contient l'élément x

exp. member(25,L); → true

Attention:

L := [op(L1),op(L2)]	crée une liste contenant les éléments de L1 suivis de ceux de L2
L := op(L1),op(L2)	crée une séquence (concaténation)
L := [L1,L2]	crée une liste contenant 2 éléments de type liste

Chapitre III
MAPLE

II.7.2) Les ensembles

Un ensemble contient une séquence mais les éléments **ne sont pas ordonnés** (on ne peut pas chercher le n^{ième} élément dans l'ensemble) et ne se **répètent pas**.

- **Créer** un ensemble par des accolades **E:={…}**
- **nops(E)** retourne le nombre d'éléments de l'ensemble
- **op(E)** retourne le contenu de l'ensemble
- **member(x, E)** vérifie si la liste contient l'élément x
- **union, intersect, minus** opérations ensemblistes classiques.
- **Réunion** de deux ensembles **E:={op(E1),op(E2)}**

```
exp. E1:={seq(x^2,x=1..5)};   → E1={1,4,9,16,25}
E2:={seq(x^2,x=1..4)};        → E1={1,4,9,16}
E1 union E2;                  → {1,4,9,16,25}
E:={op(E1),op(E2)};           → {1,4,9,16,25}
E1 intersect E2;              → {1,4,9,16}
E1 minus E2;                  → {25}
```

II.8) Les tests

II.8.1) Les tests booléen

a=b renvoie 'true' si a et b sont égaux, 'false' sinon.

a<>b renvoie 'true' si a et b sont différent, 'false' sinon.

a>b (a≥b) renvoi 'true' si a est strictement supérieur (supérieur ou égale) à b, 'false' sinon

a<b (a≤b) renvoi 'true' si a est strictement inférieur (inférieur ou égale) à b, 'false' sinon

On peut combiner ces tests élémentaires. Soit A et B des tests élémentaires.

(A) and (B) renvoie 'true' si A et B sont vrais, 'false' sinon.

(A) or (B) renvoie 'true' si A ou B sont vrais, 'false' sinon.

not(A) renvoie 'true' si A est faux, 'false' sinon.

II.8.1) Les tests booléen(suite)

exp.

```
(isprime(a))or((irem(a,b)=0)and(not(isprime(b))))
```

Si a est un nombre premier ou si b divise a et b n'est pas premier.

II.8.2) Les tests conditionnels

if *condition* **then** *instruction1* **else** *instruction2* **fi**;

exp: `a:=2:b:=3:if (a=b) then a:=1 else b:=1 fi;` → `b=1`

II.8.2) Les tests conditionnels (suite)

On peut imbriquer des tests conditionnels comme suit:

if *condition1* **then** *instruction1* **elif** *condition2* **then** *instruction2* **elif** condition3 **then** *instruction3…* **else** *instruction* **fi**;

exp.

```
if (x1<x2) then y:=1 elif (x1=x2) then y:=2 elif
  (x1>x2) then y:=3 else y:=4 fi;
```


II.9) Les boucles

II.9.1) Boucle 'for'

for *compteur* **from** *ini* **to** *fin* **by** *incrément* **do** *instruction* **od;**

exp: `a:=0:for c from 0 to 10 by 1 do a:=a+1 od;`

II.9.2) Boucle 'while'

while *condition* **do** *instruction* **od;**

exp: `a:=0:while a<10 do a:=a+1 od;`

II.10) Les Procédures

Une procédure permet de regrouper plusieurs instructions. Elle est stockée dans une variable.

Syntaxe:

```
nom := proc(parameters)
local varlocs;
global varglobs;
options ops;
instructions
end;
```

tel que:

nom: le nom de la procédure.

varlocs: séquence de variables locales. Leurs portée est à l'intérieur de la procédure

varglobs: séquence de variables (séparées par des virgules) définies à l'extérieur de la procédure

options: indique certains modes de fonctionnement (exp. remember pour garder les résultats partiels en mémoire…)

II.10) Les Procédures (suite)

Pour exécuter une procédure on l'appel avec son nom et ses paramètres :

`nom(param1, param2,…);`

II.10) Les Procédures (suite)

exp.

Une procédure pour afficher 'k' fois le mot « toto »

```
totoproc:=proc(k)
local i;
for i from 1 to k do
print("toto");
od;
end;
```

On l'appel comme suit: `totoproc(25);`

II.10) Les Procédures (suite)

exp.

Une procédure qui prend en argument une liste

```
somme:=proc(L)
local i,s;
s:=0;
for i from 1 to nops(L) do
s:=s+op(i,L);
od;
print("s=",s);
end;
```

On l'appel comme suit: `somme([1,1,2,3,5,8]);`

→ `"s=",20`

II.11) Résolution d'équations

II.11.1) Équations simples

La résolution d'une équation simple se fait avec Maple par la fonction:

`solve(équation, variable);`

tel que:

équation: est l'équation à résoudre, et l'égalité est prise par défaut = 0 si aucune égalité n'est précisée.

variable: est la variable (ou les variables) à déterminer pour que l'équation soit vérifiée. Si aucune variable n'est spécifiée, Maple utilisera toutes les variables de l'équation non affectées.

exp.

```
solve(2*x^2+x-1);
```

→ ½,-1

II.11.2) Système d'équations

La résolution d'un système d'équation se fait encore avec Maple par la fonction:

`solve({`equ_1`,…,`equ_n`}, {`var_1`,…,`var_n`});`

tel que:

equ_1,…,equ_n : sont les équations qui forment le système.

var_1,…,var_n : sont les variables des équations.

exp.

```
solve({x1+1,x2+2,x3+3},{x1,x2,x3});
```

→ `{x1=-1,x2=-2,x3=-3}`

II.11.3) Interprétation graphique

Pour visualiser graphiquement les solutions d'une équation on utilise l'une des commandes:

```
implicitplot(equ,x=a..b,y=c..d,options);
```

ou

```
implicitplot3d(equ, x=a..b,y=c..d,z=e..f,options);
```

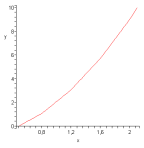
qui se trouvent dans la bibliothèque '**plots**'

Cette bibliothèque devrait être chargée avec `with(plots);`

tel que **options**: est optionnel, permet de définir la couleur, le titre, les axes… du graphe.

exp.

```
implicitplot(2*x^2+x-1=y,x=0..10,y=0..10);
```



II.12) Vecteurs et matrices

II.12.1) Vecteurs

Pour créer un vecteur on utilise les commandes de la librairie '**linalg**' suivantes:

`vector([x1,…,xn]);` pour un vecteur de 'n' éléments

`vector(n,[x1,…,xn]);` pour un vecteur de 'n' éléments

`vector(n);` pour un vecteur de 'n' éléments nuls

`vector(n,f);` pour un vecteur de 'n' éléments avec le $i^{ème}$ élément = f(i)

Chapitre III
MAPLE

II.12.1) Vecteurs (suite)

exp.

```
with(linalg):
vector([1,2,3]);
vector(3,[1,2,3]);
vector(4,[1,2,3]);
vector(3);
f:=(t)->t+1:vector(3,f);
```

[1, 2, 3]

[1, 2, 3]

[1, 2, 3,?$_4$]

[?$_1$, ?$_2$, ?$_3$]

[2, 3, 4]

Chapitre III
MAPLE

II.12.2) Matrices

Pour créer une matrice on utilise les commandes de la librairie '**linalg**' suivantes:

`matrix(L);` pour une matrice dont les lignes sont les éléments de la liste 'L'.

`matrix(m,n);` pour une matrice de 'm' lignes et 'n' colonnes et les éléments sont indéfinis.

`matrix(m,n,L);` pour une matrice de 'm' lignes et 'n' colonnes et les lignes sont les éléments de la liste 'L'.

`matrix(m,n,f);` pour une matrice de 'm' lignes et 'n' colonnes avec l'éléments M[i,j] = f(i,j)

Chapitre III
MAPLE

II.12.1) Matrices (suite)

exp.

```
with(linalg):
matrix([[1,2,3],[4,5,6]]);
matrix(2,3);
matrix(2,3,[[1,2,3],[4,5,6]]);
f:=(t1,t2)->t1+t2:matrix(2,3,f);
```

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}$$

$$\begin{bmatrix}?_{1,1} & ?_{1,2} & ?_{1,3}\\ ?_{2,1} & ?_{2,2} & ?_{2,3}\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}$$

$$\begin{bmatrix}2 & 3 & 4\\ 3 & 4 & 5\end{bmatrix}$$

Chapitre III
MAPLE

II.12.3) Opérations sur les matrices

- La somme, le produit et le produit par un scalaire des matrices se fait tout simplement on utilisant les opérations habituelles:

A+B; A*B; A*k;

- **vectdim(V)** permet de calculer la dimension d'un vecteur.
- **rowdim(M)** et **coldim(M)** permet de calculer le nombre de lignes et colonnes d'une matrice.
- **transpose(M)** permet de calculer la transposée d'une matrice.
- **det(M)** permet de calculer le déterminant d'une matrice.
- **trace(M)** permet de calculer la trace d'une matrice. $\sum a_{ii}$
- **inverse(M)** permet de calculer la matrice inverse d'une matrice si possible. $A \cdot A^{-1} = I$
- **evalm(M)** permet de d'afficher M en évaluant ces éléments.
- **linsolve(A,B)** permet de résoudre le système linéaire A.X=B

Exp: $\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}.\begin{bmatrix}x_1\\ x_2\end{bmatrix}=\begin{bmatrix}1\\ 1\end{bmatrix} \Leftrightarrow \begin{cases}x_1+2\cdot x_2=1\\ 3\cdot x_1+4\cdot x_2=1\end{cases} \Rightarrow linsolve(\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix},\begin{bmatrix}1\\ 1\end{bmatrix});\rightarrow [-1,1]$

II.12.3) Opérations sur les matrices(suite)

exp.

```
with(linalg):
A:=matrix([[2,3],[4,5]]);
```
→ $A=\begin{bmatrix}2 & 3\\ 4 & 5\end{bmatrix}$

```
B:=matrix([[7,8],[9,10]]);
```
→ $B=\begin{bmatrix}7 & 8\\ 9 & 10\end{bmatrix}$

```
C:=A+B;
C:=A.B;
C:=A*3;
rowdim(A);coldim(A);
```
→ 2, 2

```
transpose(A);
```
→ $\begin{bmatrix}2 & 4\\ 3 & 5\end{bmatrix}$

```
det(A);
```
→ −2

```
trace(A);
```
→ 7

```
inverse(A);
```
→ $\begin{bmatrix}-5/2 & 3/2\\ 2 & -1\end{bmatrix}$

```
C;evalm(C);
```
→ 3A, $\begin{bmatrix}6 & 9\\ 12 & 15\end{bmatrix}$

```
linsolve(A,B);
```
→ $\begin{bmatrix}-4 & -5\\ 5 & 6\end{bmatrix}$

II.13) Les graphiques

Avant d'utiliser les commandes graphiques de Maple il faut charger la librairie 'plots'

II.13.1) Les graphes classiques

Les commandes classiques pour les graphiques sont:

```
plot(f, h);
plot3d(f, h, v);
```

tel que:

f: la fonction ou l'expression à dessiner

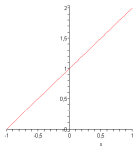
h: l'intervalle horizontal

v: (optionnel) l'intervalle vertical

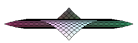
II.13.1) Les graphes classiques(suite)

exp.

```
plot(x+1, x=-1..1);
```



```
plot3d(sin(x+y), x=-1..1, y=-1..1);
```



II.13.2) Les graphes implicites

Les commandes pour les graphes implicites sont:

```
implicitplot(équation,intervalle1,intervalle2,options);
implicitplot3d(équation,intervalle1,intervalle2,
  intervalle3,options);
```

tel que:

équation: l'équation à dessiner ses racines

Intervalle1/2/3: simultanément l'intervalle des x/y/z

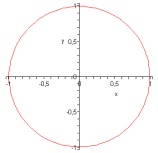
options: (optionnel) pour définir les options du graphe (couleur, titre, axes, légende…)

Chapitre III
MAPLE

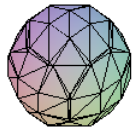
II.13.2) Les graphes implicites (suite)

exp.

```
implicitplot(x^2+y^2=1,x=-2..2,y=-2..2);
```



```
implicitplot3d(x^2+y^2+z^2=1,x=-2..2,y=-2..2,z=-2..2);
```



Chapitre IV
La langage CAML (O'CAML & CAML Light)

1. Introduction
2. Instructions, testes et boucles (let, if, for)
3. Les types (float, int, ref)
4. Fonctions
5. Tableaux et chaînes de caractères (array, char, string)
6. Valeurs booléennes et aléatoires (bool, random)
7. Boucle while et récursivité (while, rec)

Chapitre IV
CAML

III.1) Introduction

CAML est créé en début 1980 par l'équipe INRIA suite à la rencontre du langage de programmation ML et la machine CAM de Guy Cousineau.

C'est un langage **compilé**, c.à.d. le **code source** est écrit dans un éditeur, puis le **compilateur** CAML est appelé avec le code source pour générer le **code machine** (le programme exécutable).

Chapitre IV
CAML

III.1) Introduction (suite)

Pour utiliser CAML, il faut écrire le **code source** dans un fichier à l'aide d'un éditeur de texte, puis **compiler** ce fichier en utilisant la commande suivante dans **l'invite de commande** de windows :

`ocamlc -o output.exe source.ml` (o'caml)

`camlc -o output.exe source.ml` (caml light)

Puis il suffit **d'appeler** le fichier exécutable pour exécuter le programme et voir le résultat.

Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for)

Instructions d'affichage:

Texte: `print_string "bonjour";` → bonjour

Nouvelle ligne: `print_newline ();` → saut de ligne

Entiers: `print_int 10*2;` → 20

Entiers négatif: `print_int (-10);` → -10

Ne pas oublier les parenthèses dans le cas des nombre négatifs et des expressions.

Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

Déclarations:

On déclare une valeur comme suit:

```
let x = (... la valeur de x ...) in (... instructions
    utilisant x ...)
```

exp.

```
let x = 238 in
print_int x;
print_string " a pour carré ";
print_int (x * x);
→ 238 a pour carré 56644
```


Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

Règle pour les noms des variables:

Un nom de variable doit être formé de **lettres** de bases (de 'a' à 'z' ou de 'A' à 'Z'), de **chiffres**, et **d'underscores** (le Symbole _)

Il doit **commencer** par une lettre **minuscule**. Il ne faut **pas** mettre **d'espaces**, ni de **lettres accentuées** ou tout autre **caractère spécial**.

Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

Lecture d'un entier:

Ça se fait à l'aide de la commande `read_int()`.

exp.

```
let x = read_int () in
```
(sans point-virgule)

Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

Lecture d'un texte:

Ça se fait à l'aide de la commande `read_line()`.

exp.

```
let t = read_line () in
print_string t;
```

Remarque: après le « in » on ne met pas de ';'

Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

Conditions:

```
if (condition) then
   begin
   (instructions)
   end
else
   begin
   (instructions)
   end
;
```


Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

Conditions (suite):

exp:

```
let mon_nombre = read_int () in
if mon_nombre >= 0 then
   begin
   print_string " positif ";
   end
else
   begin
   print_string " négatif ";
   end
;
```


Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

Remarque:

Si un bloc contient une seule instruction et qu'on à supprimer le `begin` et `end`, il ne faut pas mettre le ';'

exp:

```
let x = read_int () in
if x = 4
then print_string "égale à 4"; → erreur
else print_string "différent de 4" → pas d'erreur
;
```

Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

Conditions (suite):

if imbriqué:

```
if (...1ere condition ...) then
(... bloc du 1er then ...)
else if (...2ème condition ...) then
(... bloc du 2ème then ...)
else if (...3ème condition ...) then
(... bloc du 3ème then ...)
...
...
else
( ... bloc du else ...)
;
```


Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

Conditions (suite):

if imbriqué (suite):

exp.

```
let prenom = read_line () in
if prenom = "toto" then
   print_string "c'est moi"
else if prenom = "TOTO" then
   print_string "C'EST MOI"
;
```


Chapitre IV
CAML

III.2) Instructions, testes et boucles (let, if, for) (suite)

La boucle 'for':

```
for compteur = val_initiale to val_finale do
    (... corps de la boucle for ...)
done; (*compteur croissant*)
for compteur = val_initiale downto val_finale do
    (... corps de la boucle for ...)
done; (*compteur décroissant*)
```

exp:

```
for i = 1 to 3 do
print_string "bonjour";
print_newline ();
done ;
```


Chapitre IV
CAML

III.3) Les types (float, int, ref)

III.3.1) Les réels

Les opérations de lecture/écriture sur les réels sont:

`print_float()` pour afficher un réel

`read_float()` pour lire un réel

exp.

```
print_float(1); erreur
print_float(1.);
print_float(1.5);
print_newline();
let x = read_float() in
print_float(x);
```

Chapitre IV
CAML

III.3.1) Les réels (suite)

Les opérations arithmétiques de base qui traitent les réels sont:

l'addition +.

la soustraction -.

La multiplication *.

la division /.

exp:

```
let x = read_float () in
print_float (3.4 *. x);
print_newline ();
print_float (-. x);
```


Chapitre IV
CAML

III.3.1) Les réels (suite)

La valeur absolue et la puissance en utilisant les réels se font avec les commandes suivantes:

`abs_float x`	pour la valeur absolue
`sqrt x`	pour la racine carrée
`x ** y`	pour la puissance

exp.

```
let x = read_float () in
let y = read_float () in
let r = abs_float (x -. y) in
print_string "x - y = "; print_float r;
print_newline();
print_string "racine carré de x = "; print_float ( sqrt x);
print_newline();
print_string "x puissance 3 = "; print_float(x ** 3.0);
```


Chapitre IV
CAML

III.3.1) Les réels (suite)

Le maximum et le minimum de deux valeurs réels se calculent avec les deux commandes suivantes:

`max x y`	pour le maximum
`min x y`	pour le minimum

exp.

```
let x = read_float () in
let y = read_float () in
let r1 = max (x +. 5.) (y -. 5.) in
print_float r1; print_newline();
let r2 = min (x +. 5.) (y -. 5.) in
print_float r2; → 7. -2.
```


Chapitre IV
CAML

III.3.2) Les entiers

Les opérations arithmétiques de base qui traitent les entiers sont:

l'addition +

la soustraction -

La multiplication *

la division /

Le modulo x mod y

exp:

```
let x = read_int() in
print_int(3 * x);
print_newline ();
print_int(- x);
print_newline ();
print_int(x mod 2);
```

Chapitre IV
CAML

III.3.2) Les entiers (suite)

La valeur absolue en utilisant les entiers se fait avec la commande suivante:

`abs x` pour la valeur absolue

Le maximum et le minimum de deux valeurs entières se calculent avec les deux commandes suivantes:

`max x y` pour le maximum
`min x y` pour le minimum

exp.

```
let x = read_int() in
let y = read_int() in
let r1 = max (x + 5) (y - 5) in
print_int r1; print_newline();
let r2 = min (x + 5) (y - 5) in
print_int r2; print_newline();
let r3 = abs (x - y) in
print_string "abs (x - y) = "; print_int r3;
```


Chapitre IV
CAML

III.3.2) Les entiers (suite)

La conversion d'un entier vers un réel se fait par le biais de la commande suivante:

```
float_of_int x
```

exp.

```
let x = 2 in
print_int x;
print_newline ();
let y = float_of_int x in
print_float y;
```


Chapitre IV
CAML

III.3.2) Les entiers (suite)

La conversion d'un réel vers un entier se fait avec une perte d'information en utilisant la commande suivante:

```
int_of_float x
```

Ça permet de **supprimer** tout simplement les **chiffres après la virgule**.

exp.

```
let y = 4.2 in
print_int (int_of_float y);   → 4
let y = -4.2 in
print_int (int_of_float y);   → -4
let y = 4.9 in
print_int (int_of_float y);   → 4
```


Chapitre IV
CAML

III.3.2) Les entiers (suite)

Attention:

Un entier est limité et il a une valeur comprise entre -1073741824=-(2^{30}) et 1073741823 = 2^{30}-1

CAML ne donne pas d'erreur si on dépasse ces deux valeurs.

exp.

```
print_int (1073741823 + 1);
    → -1073741824
```

III.3.3) Les références

Problématique:

On veut écrire un programme qui demande 'n' nombres à l'utilisateur, et en affiche la somme.

Solution fausse:

```
let somme = 0 in
for i = 1 to n do
   let x = read_int () in
   somme := somme + x; → erreur
done;
print_int somme;
```

Cette solution est **fausse** car la déclaration de somme=0 **ne permet pas** de changer la valeur de 'somme'.

'somme' est une constante dans ce cas.

III.3.3) Les références

Pour résoudre ce problème il faut utiliser les **références**.

Au lieu de déclarer une **constante** 'somme' on déclare une **référence** vers une case mémoire qui s'appel 'somme' comme suit:

```
let somme = ref 0 in
```

Pour **obtenir le contenu de la référence** créé, on utilise le caractère '!' comme suit:

```
print_int !somme;
```

Pour **changer la valeur du contenu de la référence** on utilise l'affectation ':=' comme suit:

```
somme := nouveau_contenu;
```

La solution de notre problème devient:

```
let somme = ref 0 in
for i = 1 to 3 do
     let x = read_int () in
     somme := !somme + x;
done;
print_int !somme;
```


III.3.3) Les références

1er cas exceptionnels:

exp:

```
let s = ref 5 in
let t = s in
t:=10;
print_int !s; print_newline();
print_int !t;
```

Quel est l'affichage obtenu?

On aura: 10
10

Ça explique que 's' et 't' contiennent l'adresse de **la même case mémoire**.

III.3.3) Les références

2ème cas exceptionnels:

exp:

```
let x = ref 5 in
let y = ref !x in
y := 7;
print_int !x ; print_newline();
print_int !y ;
```

Quel est l'affichage obtenu?

On aura: 5
7

Ça explique que 'x' et 'y' contiennent les adresses de deux cases mémoire différentes.