Introduction à la programmation scientifique

MuPAD

Damien Olivier

25 octobre 2007

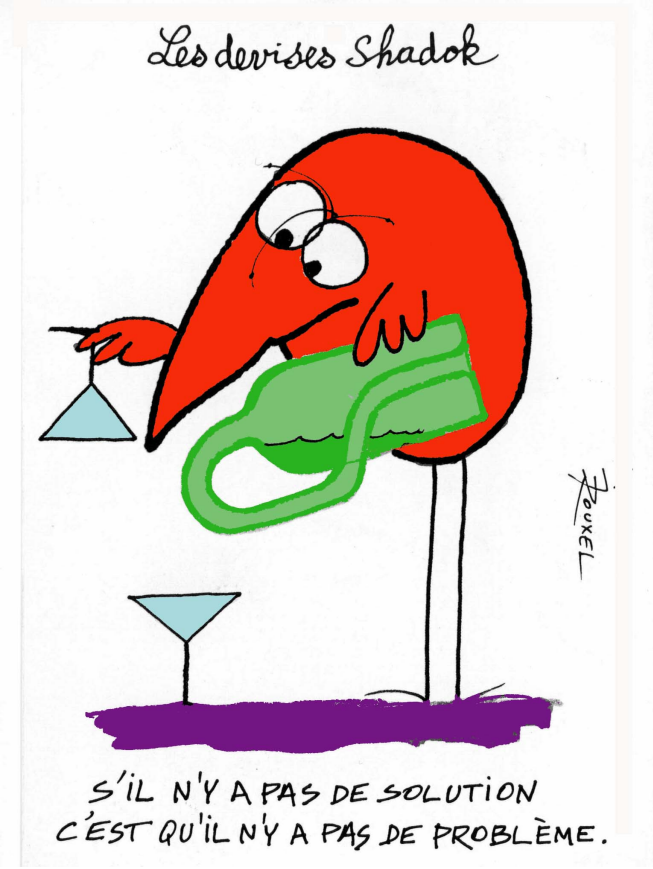


Table des matières

Avertissement

Ce document est en cours de rédaction, il est donc incomplet et contient sans aucun doute des coquilles et des erreurs. Je compte à la fois sur votre indulgence et sur votre collaboration.

Ce document ne reflète pas avec exactitude le contenu des cours proposés et dispensés lors de la première partie du semestre 1. "Ses" oublis seront ajoutés au fil du temps et ses suppléments sont là pour ceux qui souhaiteraient ou auraient besoin d'approfondir.

Bon courage à tous.

Ce document s'est largement inspiré de deux sources, la page web de Sylvain Damour [1] et le tutorial MuPAD [3].

Remerciements

Merci à tous les relecteurs attentifs qui ont corrigé les coquilles, les fotes et autres maladresses.

1 Résolution de problèmes à l'aide de l'informatique

Le processus de résolution d'un problème ne commence pas par écrire un programme ou utiliser directement un logiciel, mais passe par des étapes d'analyses. On peut le décomposer idéalement en cinq étapes :

- Identifier le problème ;
- Poser le problème sous la forme d'un modèle mathématique/informatique ;
- Déterminer une méthode informatique pour résoudre le modèle (algorithme) ;
- Implanter (programmer) le modèle et la méthode ;
- Évaluer les résultats.

Il est clair que ces différentes étapes ne sont pas indépendantes, et qu'elles ne sont pas irrévocables. En effet, l'évaluation des résultats peut remettre en cause le modèle par exemple.

Intéressons nous au problème posé par Leornado Pisano connu sous le nom de Fibonacci (1170-1250) dans son ouvrage Liber Abaci. Dans une population idéale de lapins, on suppose que :

- Toutes les saisons un couple de lapins met au monde un couple de lapins ;
- Le nouveau couple grandit durant la saison ;
- Il est en âge de procréer la saison d'après.

Quelle sera la population à la saison n ?

1.1 Identification du problème

Au début (année $n = 0$) nous avons un couple de lapins qui grandit l'année suivante ($n = 1$) et procrée l'année d'après ($n = 2$) et ainsi de suite, on peut donc construire le tableau suivant :

n	**Lapereaux * 2 =**L_n	**Adolescents * 2 =** a_n	**Adultes * 2 =** A_n	**Total * 2 =** T_n
0	1	0	0	1
1	0	1	0	1
2	1	0	1	2
3	1	1	1	3
4	2	1	2	5
5	3	2	3	8
6	5	3	5	13
7	8	5	8	21
8	13	8	13	34

1.2 Modèle mathématique

La population est constituée des lapereaux, des adolescents et des adultes

$$T_n = L_n + a_n + A_n \tag{1}$$

et les lapereaux de la saison précédente sont les adolescents de la saison courante.

$$a_n = L_{n-1} \tag{2}$$

La population d'adultes est constituée des anciens et des adolescents de la génération précédente :

$$A_n = A_{n-1} + a_{n-1} \tag{3}$$

et il y a autant de lapereaux que de parents.

$$A_n = L_n \tag{4}$$

$$\begin{aligned}
\text{(2) et (3) dans (1)} \quad T_n &= L_n + \underbrace{L_{n-1} + a_{n-1} + A_{n-1}}_{T_{n-1}} && (5)\\
\text{(3) et (4)} \quad L_n &= A_{n-1} + a_{n-1} && (6)\\
\text{De (3)} \quad A_{n-1} &= A_{n-2} + a_{n-2} && (7)\\
\text{Dans (6)} \quad L_n &= A_{n-2} + a_{n-2} + L_{n-2} = T_{n-2} && (8)\\
\text{Dans (5)} \quad T_n &= T_{n-2} + T_{n-1} && (9)
\end{aligned}$$

Ceci définit donc une suite de Fibonacci dont la relation de récurrence est la suivante :

$$\begin{cases} \forall n > 0, T_{n+2} = T_n + T_{n+1} \\ \\ \text{Conditions initiales : } T_1 = T_2 = 1 \end{cases} \tag{10}$$

1.3 Algorithme

```
Fonction Fibonacci( n : entier) : entier
    [n ≥ 3]
    fibNPlus2, fibN, fibNPlus1, i : entier
    fibN ← 1
    fibNPlus1 ← 1
    i ← 3
    Tant que (i ≤ n) faire
        fibNPlus2 ← fibN + fibNPlus1
        fibN ← fibNPlus1
        fibNPlus1 ← fibNPlus2
        i ← i + 1 ;
    Fait
    Retourner fibNPlus2 ;
Fin
```

Algorithme 1: Suite de Fibonacci

Le programme sous MuPAd ou une méthode de résolution sera détaillé dans la suite.

```
Suite de fibonacci


>> fibonacci := proc(n:Type::NonNegInt) : Type::Integer
local fibNPlus2, fibN, fibNPlus1, i;
begin
  fibN := 1;
  fibNPlus1 := 1;
  i := 3;
  while i <= n do
    fibNPlus2  := fibN + fibNPlus1;
    fibN       := fibNPlus1;
    fibNPlus1  := fibNPlus2;
    i          := i + 1;
  end_while;
  fibNPlus2;
end_proc:
```

1.4 Analyse des résultats

Ainsi que l'a remarqué Johannes Kepler, le taux de croissance $\frac{T_n}{T_{n-1}}$ des nombres de Fibonacci converge vers le nombre d'or noté Φ. Le nombre d'or est la racine positive de l'équation du second degré :

$$x^2 - x - 1 \ = \ 0$$

De façon approchée $T_n \approx 1,618 * T_{n-1}$. Le modèle ne prend pas en compte les phénomènes de décès, pour cela on pourrait utiliser une loi logistique.

1.5 Quelques conseils

Il y a bien entendu de nombreuses façons de résoudre un problème. Toutefois, quelques stratégies et méthodes générales peuvent être conseillées d'après Miguel de Guzmań [2] :

- avant d'agir, essayez de comprendre ;
- recherchez des stratégies ;
  - cherchez des ressemblances avec d'autres problèmes ;
  - commencez par ce qui est facile ;
  - faîtes des expériences et cherchez des traits communs, des règles ;
  - faîtes un schéma ;
  - modifiez le problème pour trouver un autre chemin possible ;
  - choisissez une bonne notation ;
  - exploitez les propriétés (symétrie, ...) ;
  - raisonnez par l'absurde ;
  - pensez à des techniques générales : récurrence, induction ...

  .
- menez à terme votre stratégie ;
  - menez à terme les meilleures idées ;
  - ne vous découragez pas ;
  - analysez bien les résultats avant de les admettre ;
- tirez profit de votre expérience ;

– analysez votre propre stratégie ;
– essayez de comprendre la solution et son pourquoi ;
– regardez si l'on ne peut pas faire plus simplement ;
– examinez si la méthode peut être utilisée à d'autres fins ;
– réfléchissez sur votre manière de raisonner.

1.6 Exercice

I *Exercice n°1*

Une grenouille saute sur des nénuphars, elle saute soit sur le nénuphar suivant ou sur celui d'après. Il y a n nénuphars alignés, sachant que la grenouille part du premier quel est le nombre de possibilités pour atteindre le dernier ?

2 Découverte de MuPAD

MuPAD[1] est un logiciel de *calcul symbolique* ou encore de *calcul formel*. La majorité des logiciels mathématiques travaillent avec des valeurs numériques pour toutes les variables alors que MuPAD manipule des symboles, cela permet en particulier d'obtenir des solutions analytiques à de nombreux problèmes. Attention, cependant cela ne résout pas tout ! D'autre part MuPAD possède des possibilités de visualiser des fonctions en deux ou trois dimensions ainsi qu'un langage de programmation (figure 1). L'objectif de cette présentation est de découvrir ces différentes fonctionalités.

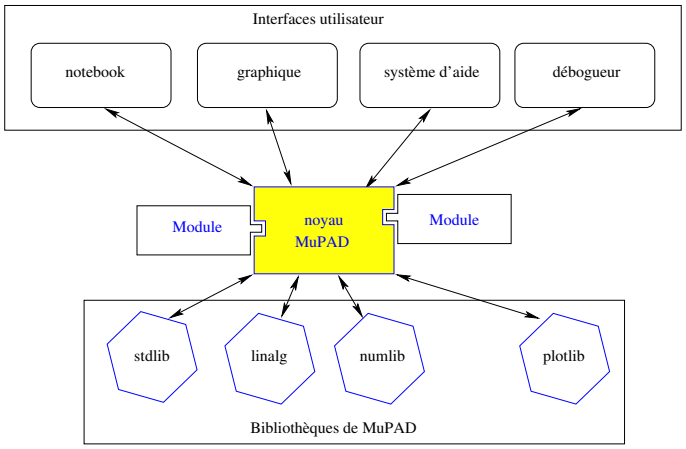


FIG. 1 – Architecture de MuPAD

2.1 Commandes de base

Dans l'environnement Linux, la première opération consiste à lancer MuPAD, ceci est possible soit en mode console (`$ mupad`) ou en mode graphique (`$ xmupad &`) avec possibilité d'utilisation de la souris, c'est cette dernière méthode que nous choisissons pour la suite de cette présentation.

[1] Il existe des logiciels similaires à MuPAD comme Maple, Mathematica, MatLab ...

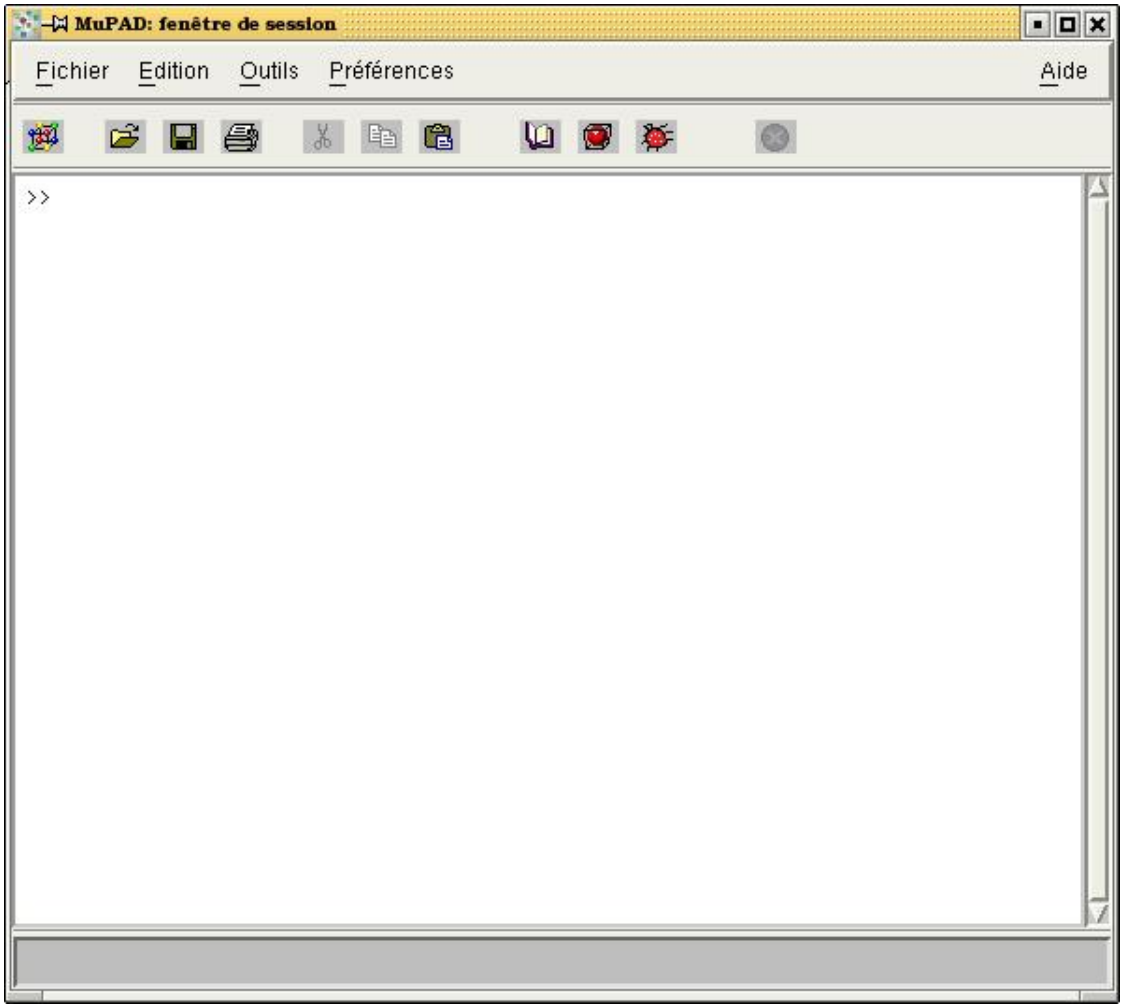


FIG. 2 – Ecran de démarrage de xmupad

```
damien@astrolabe2:~/enseignement/l1/prg_scien/Mupad$ xmupad &
```

Une ligne de commande se termine soit par un `;` (facultatif) ou par `:` et vous validez votre ligne par ↩ (entrée). Le point virgule (ou son absence) affiche le résultat alors qu'avec deux points le calcul est effectué mais n'est pas affiché, cela peut être utile lorsque l'on a de longues feuilles de calcul. Si une ligne de calcul ne tient par sur une seule ligne ↑↩ (shift entrée) permet de passer à la ligne suivante sans effectuer le calcul.

```
Exemple :


>> 111!

176295255109024466387216104710707578876140953602656551604157406334734695508724831643655557459846231577319604766283797891314584749719987162332009625414533120000000000000000000000000000
>> 111!;

176295255109024466387216104710707578876140953602656551604157406334734695508724831643655557459846231577319604766283797891314584749719987162332009625414533120000000000000000000000000000
>> 111:
>> 111
!

176295255109024466387216104710707578876140953602656551604157406334734695508724831643655557459846231577319604766283797891314584749719987162332009625414533120000000000000000000000000000
>>
```

Le dernier calcul est représenté par `%`, l'avant dernier par `%%` (ou `%2`), l'antépénultième par `%%%` (`%3`) ...et un commentaire figure entre deux `#` ou `/*` et `*/` . Si vous voulez tout réinitialiser, il faut utiliser `reset()`.

```
Exemple :


>> 100:
>> length(%)

                                      3
>> 100!:
>> length(%) #longueur de factorielle 100#

                                    158
>>
```

Attention MuPAD fait une différence entre les majuscules et les minuscules. Si vous avez besoin d'aide `?` viendra à votre secours en particulier pour une fonction, par exemple on pourra taper `?sin`.

2.2 Calculer avec MuPAD

MuPAD permet de calculer à la fois avec des nombres, de réaliser soit des calculs exacts ou approchés et d'effectuer des calculs symboliques.

2.2.1 Calcul numérique

Il est possible d'utiliser MuPAD comme une calculette retournant des résultats exacts ou approchés. MuPAD privilégie les résultats exacts.

```
Exemple :

>> 1 + 7/3;

                                  10/3
>> float(%)

                               3.333333333
>>
```

Pour montrer la différence, nous nous proposons de traiter un problème simple : la population de la Terre sera d'environ (estimation) de 7 516 000 000 d'habitants en 2020, si l'on décide de partager la surface des continents équitablement, quelle sera la taille du carré de terre que pourra occuper chaque habitant ? On suppose que la Terre est une sphère de rayon 6378 km occupée par 70% d'eau.

```
Solution exacte :

>> /* Surface des continents */
>> SurfaceContinents := 4 * PI * 6378^2 * 30/100

                              244073304 PI
                              ------------
                                     5
>> /* Chaque habitant aura un carré de coté : "
>> sqrt(SurfaceContinents/7516000000)*1000; /* m^2 */

                        1/2           1/2         1/2
                     PI    4697500000    30509163
                     --------------------------------
                                  4697500
>> float(%); /* On évalue la solution exacte */

                               142.842252
```

```
Solution numérique :

>> AireContinents := float(4 * PI * 6378^2 * 30/100);

                               153355779.8
>> /* On a obtenu un réel, le calcul est fait en réel */
>> sqrt(AireContinents/7516000000)*1000

                               142.842252
```

2.2.2 Les nombres

MuPAD peut manipuler des entiers, des rationnels, des réels et des complexes (cf. tab. 1, et 2). Les entiers ont une taille arbitraire limitée par la mémoire. Les rationnels sont définis par deux entiers (fraction)

sans diviseur commun dont le deuxième est strictement positif. Les réels (float) sont représentés par deux entiers : la *mantisse* et l'*exposant*. La valeur est $mantisse * 10^{exposant}$, la valeur de l'exposant est bornée par 2^{31}.

Entiers	!	factorielle d'un nombre	$12! = 479001600$
	^	puissance d'un entier	$3^0 = 1$
	`isprime()`	vérifie si un entier positif est premier	$isprime(2110805449) = true$
	`ifactor()`	factorisation en nombres premiers	$ifactor(123456789101112) = [2^3\,2437\,2110805449]$
Réels	`DIGITS`	fixe le nombre de chiffre décimaux	$DIGITS = 10$ (par défaut)
	`float()`	valeur de son arg. en flottant	$float(sqrt(2) = 1.414213562$

TAB. 1 – Quelques opérations concernant les entiers et les réels

MuPAD manipule également les nombres complexes, $\sqrt{-1}$ est représenté par I.

```
Nombres complexes :


>> sqrt(-1), I^2
                                  I, -1
>>
```

Complexes	`Re ()`	partie réelle
	`Im()`	partie imaginaire
	`conjugate()`	conjugué complexe
	`abs()`	module
	`rectform()`	met un nombre complexe sous la forme $a + ib$

TAB. 2 – Quelques opérations sur les nombres complexes

```
Exemple d'opérations sur les complexes :


>> z := (1+I)^2 / (1-2*I)
                                - 4/5 + 2/5 I
>> Re(z)
                                    -4/5
>> Im(z)
                                     2/5
>> abs(z)
                                       1/2
                                   (4/5)
>> conjugate(z)
                                - 4/5 - 2/5 I
>> w:=-1/2 * sqrt(2 + 2*I*sqrt(3))
                                          1/2     1/2
                               (2 I 3    + 2)
                             - -----------------
                                       2
>> rectform(w)
                                1/2
                               3
                             - ---- - 1/2 I
                                 2
>>
```

2.2.3 Calcul symbolique

Considérons un problème simple. Imaginons que nous ayons à notre disposition :

– Un tas de bouts de bois, tous d'égale longueur (10 cm par exemple) et d'égale densité uniforme ;

– Une table solidement ancrée dans le sol.

À l'aide de ces éléments il faut construire la plus large pile de bouts de bois possible à l'extérieur de la table, sachant que le premier bout de bois doit reposer sur la table et le suivant sur le précédent et ainsi de suite (fig 3). *Au $n^{ième}$ bout de bois posé, quelle est la largeur maximale de la pile à l'extérieur de la table ?*

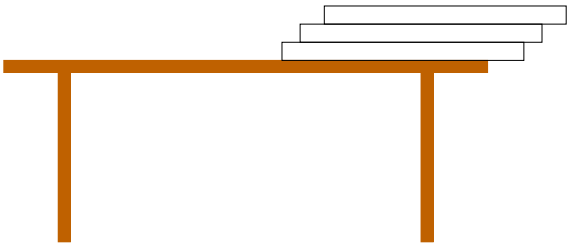
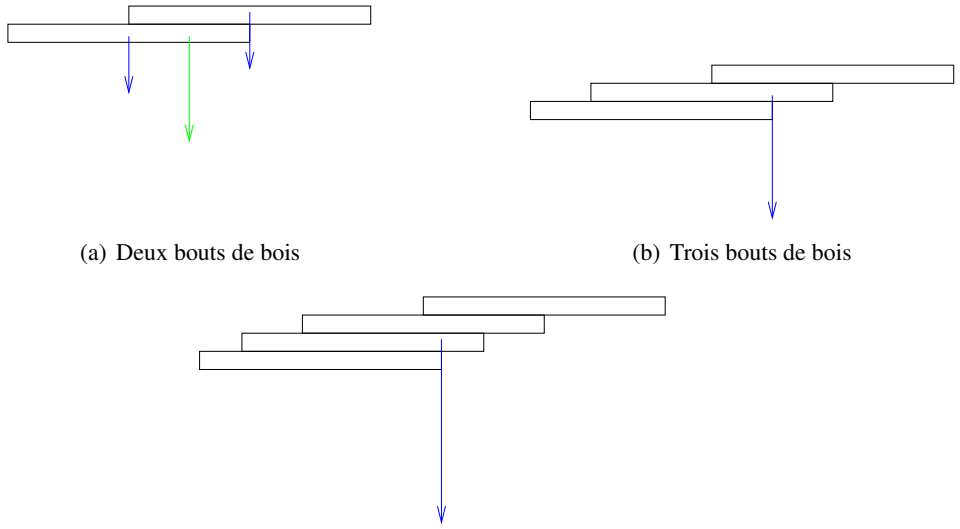


FIG. 3 – Trois bouts de bois en équilibre (non optimal)

Il s'agit d'un problème d'équilibre et donc de **position du centre de gravité** de chaque objet. Nos bouts de bois de taille et de densité uniforme ont leur centre de gravité au milieu. D'autre part un bout de bois seul est en équilibre tant que son centre de gravité est supporté par la table ou un autre bout de bois.

Cherchons tout d'abord la solution avec deux bouts de bois. Lorsque l'on a deux bouts bois posés l'un sur l'autre on peut les considérer comme un seul, mais alors quel est le centre de gravité de ce "nouvel



(a) Deux bouts de bois

(b) Trois bouts de bois

(c) Quatre bouts de bois

FIG. 4 – Position du centre de gravité

objet" ? Pour déterminer cela, il suffit simplement de faire la moyenne des centres de gravité puisqu'ils sont de même poids. Si les deux bouts de bois sont en équilibre, celui qui se situe au dessus doit être en équilibre sur celui au dessous et l'ensemble doit l'être également. Pour simplifier, on considére uniquement la coordonnée du centre de gravité des bouts de bois. On a donc établi que la partie qui dépasse à l'extérieur pourr deux bouts de bois de taille unitaire est :

$$\frac{1}{2} + \frac{1}{4} \tag{11}$$

On rajoute maintenant un troisième bout de bois à l'ensemble. Pour commencer l'extrémité droite de ce nouveau bout de bois (appellons le C) est située au bord de la table et on va chercher à déplacer l'ensemble le plus possible vers l'extérieur. Le centre de gravité de C est situé à $3/6$ du bord de la table. L'ensemble des bouts de bois (B et A) situés au-dessus pèse deux fois plus lourd que le bout de bois C. Le centre de gravité doit être proportionnellement plus proche de l'ensemble BC que de A.

$$\frac{3}{6} = \frac{2}{6} + \frac{1}{6}$$

d'où en complétant (11) $\quad \frac{1}{2} + \frac{1}{4} + \frac{1}{6}$

et en généralisant

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{8} + ... + \frac{1}{2 * n} \tag{12}$$

On a donc obtenu une suite (équation 12) qui diverge voisine de la suite harmonique (équation 13).

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + ... + \frac{1}{n} \tag{13}$$

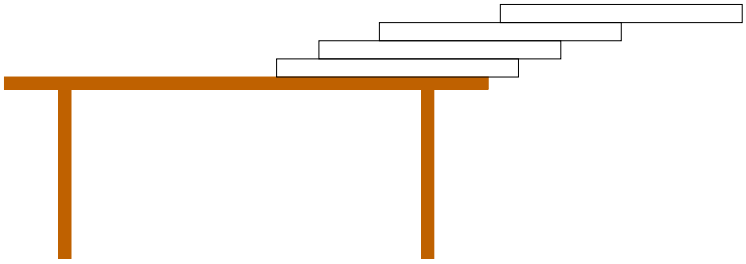


FIG. 5 – Equilibre optimal avec quatre bouts de bois

```
Etude da la suite :

>> sum(1/(2*i),i=1..infinity)

                                  infinity
```

ce que nous montre le calcul symbolique. À l'aide de cet exemple, on peut également mettre en évidence l'accumulation des erreurs d'arrondis.

```
Erreurs d'arrondis :

sum(float(1/(2*i)), i = 1..10000) - sum(1/(2*i), i = 1..10000)

                                   -9.3e-9
>> DIGITS:=10

                                     10
>> sum(float(1/(2*i)), i = 1..10000) - sum(1/(2*i), i = 1..10000)

                             -1.734723476e-18
```

2.2.4 Exercices

II *Exercice n°2*

II.1 Montrer que pour tout entier naturel $n \in \mathbb{N}, n(n^6 - 1)$ est divisible par 7 (raisonner par récurrence).

- Définir une fonction MuPAD `f := n -> n * (n^6 -1).`
- Factoriser à l'aide de la commande `factor().`

```
Exercice 2 :

>> f := n -> n * (n^6 -1)

                                n -> n*(n^6 - 1)
>> f(n-1) - f(n)

                                      6            6
                     (n - 1) ((n - 1)  - 1) - n (n  - 1)
>> factor(%)

                                              2     2
                       (-7) n (n - 1) (- n + n  + 1)
```

III *Exercice n°3*

Utiliser MuPAD pour déterminer les limites suivantes. Consultez au préalable l'aide concernant la fonction `limit`.

$$\lim_{x\to 0}\frac{\sin(x)}{x},\quad \lim_{x\to 0}\frac{1-\cos(x)}{x},\quad \lim_{x\to 0^+}\ln(x),$$
$$\lim_{x\to 0}x^{\sin(x)},\quad \lim_{x\to\infty}(1+\tfrac{1}{x})^x\quad \lim_{x\to\infty}\frac{\ln(x)}{e^x}$$
$$\lim_{x\to 0}x^{\ln(x)},\quad \lim_{x\to 0^-}\frac{2}{1+e^{-1/x}}\quad \lim_{x\to 0}\sin(x)^{1/x}$$

Expliquer le dernier résultat, que faut-il faire ?

```
Exercice 3 :

>> limit(sin(x)/x, x=0)

                                      1
>> limit((1 - cos(x))/x, x=0)

                                      0
>> limit(ln(x), x=0, Right)

                                  -infinity
>> limit(x^sin(x), x=0)

                                      1
>> limit((1 + 1/x)^x, x= +infinity)

                                   exp(1)
>> limit(ln(x)/exp(x),x=+infinity)

                                      0
>> limit(x^ln(x), x = 0)

                                  infinity
>> limit(2/(1 + exp(-1/x)), x = 0, Left)

                                      0
>> limit(sin(x)^(1/x), x = 0)

                                  undefined
>>
```

IV *Exercice n°4*

En utilisant la commande `sum` et la commande `binomial` simplifier les expressions :

IV.1 $\left(\begin{array}{c} n \\ 0 \end{array}\right) + \left(\begin{array}{c} n \\ 1 \end{array}\right) + \left(\begin{array}{c} n \\ 2 \end{array}\right) + \ldots + \left(\begin{array}{c} n \\ n \end{array}\right)$

IV.2 $\left(\begin{array}{c} n \\ 1 \end{array}\right) + 2\left(\begin{array}{c} n \\ 2 \end{array}\right) + 3\left(\begin{array}{c} n \\ 3 \end{array}\right) + \ldots + n\left(\begin{array}{c} n \\ n \end{array}\right)$

IV.3 $\left(\begin{array}{c} n \\ 1 \end{array}\right) - 2\left(\begin{array}{c} n \\ 2 \end{array}\right) + 3\left(\begin{array}{c} n \\ 3 \end{array}\right) + \ldots + (-1)^{n+1} n\left(\begin{array}{c} n \\ n \end{array}\right)$

avec

$$\left(\begin{array}{c} n \\ p \end{array}\right) = \frac{n!}{p!(n-p)!}$$

qui représente les coefficients du triangle de Pascal $\left(\begin{array}{c} i \\ j \end{array}\right)$ est le coefficient situé à la ligne i et à la colonne j :

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
```

et on a par exemple :

$$(x+y)^5 = x^5 + 5xy^4 + 10x^2y^3 + 10x^3y^2 + 5x^4y + y^5$$

```
Exercice 4

>> sum(binomial(n,k), k = 0..n)
                                      n
                                     2
>>
>> sum(k * binomial(n, k), k = 1..n)

                                       n
                                   n 2
                                   ----
                                     2
>> sum((-1)^(k+1) * k * binomial(n, k), k = 1..n)

                n                          2      n
        - n (-1)  binomial(n, n + 1) - n  (-1)  binomial(n, n + 1)
        ------------------------------------------------------------
                                   n - 1
>> expand(%)

                                     0
```

3 Fonctions

MuPAD manipule des *fonctions* et des *expressions*. On définit une fonction avec une notation proche de celle traditionnellement utilisée en mathématiques.

```
Fonction f et expression e

>> f := x -> x^2 + 1                    /* Fonction f */

                                  x -> x^2 + 1
>> f(4)                                 /* Evaluation de la fonction en un point */

                                       17
>> e := x^2 + 1                         /* Expression e */

                                      2
                                     x  + 1
>> subs(e, x=4)                         /* Evaluation de l'expression */

                                       17
```

C'est donc l'opérateur -> qui génère des objets qui représentent des fonctions mathématiques. La fonction peut alors être appelée comme n'importe quelle fonction prédéfinie.

```
Fonction f

>> f := (x,y) -> abs(x) - abs(sqrt(y))

                         (x, y) -> abs(x) - abs(sqrt(y))
>> f(a, a^2)

                                              2 1/2
                               abs(a) - (abs(a) )
>> simplify(%)

                                        0
```

La composition de fonction est réalisée à l'aide de l'opérateur `@` et l'opérateur `@@` réalise la composition itérée de la fonction avec elle-même.

```
Composition de fonction

>> f := x -> (1 / (1 + x))
                                  x -> 1/(1 + x)
>> g := sin(x^2)
                                        2
                                    sin(x )
>> f@g(a)
                           (x -> 1/(1 + x))@sin(x^2)(a)
>> h := f@g
                             (x -> 1/(1 + x))@sin(x^2)
>> h(a)
                                      1
                                  --------------
                                       2
                                  sin(x )(a) + 1
>> fff := f@@3
                (x -> 1/(1 + x))@(x -> 1/(1 + x))@(x -> 1/(1 + x))
>> fff(a)
                                      1
                                  -------------
                                      1
                                  --------- + 1
                                    1
                                  ----- + 1
                                  a + 1
>>
```

Il est possible de passer d'une fonction à une expression et inversement, il suffit dans le premier cas d'affecter la fonction à une expression et dans le second cas d'utiliser la fonction `fp::unapply()` [2].

```
fonction ↔ expression

>> f := x -> x^2
                                    x -> x^2
>> e := f(x)
                                       2
                                      x
>> g := fp::unapply(e)
                                    x -> x^2
>>
```

[2] fonction `unapply()` de la bibliothèque `fp`

3.1 Dérivées

La dérivée d'une fonction en un point peut être vue comme la mesure de la vitesse à laquelle cette fonction change lorsque sa variable change. La fonction croît lorsque la dérivée est positive et décroît lorsqu'elle est négative. Pour une fonction à plusieurs variables, on parle de dérivée partielle par rapport à l'une de ses variables.

Les dérivées partielles ou non peuvent être effectuées à l'aide de la fonction `diff`. L'appel `diff(expression, x)` calcule la dérivée de l'expression par rapport à `x`.

```
Dérivée d'une fonction


>> diff(sin(x),x)

                                    cos(x)
>> diff(f(x) + f(x)*sin(x),x)

            f(x) cos(x) + diff(f(x), x) + sin(x) diff(f(x), x)
```

On remarquera que la dérivée de `f` n'est pas connue, le résultat est retourné symboliquement. Il est également possible de réaliser des dérivées multiples en répétant la variable plusieurs fois ou en utilisant l'opérateur de séquence `$`.

```
Dérivées multiples


>> diff(sin(x^2),x,x)

                                2       2      2
                          2 cos(x ) - 4 x  sin(x )
>> diff(sin(x^2), x$2)

                                2       2      2
                          2 cos(x ) - 4 x  sin(x )
>> diff(diff(sin(x^2),x),x)

                                2       2      2
                          2 cos(x ) - 4 x  sin(x )
```

3.1.1 Dérivées partielles

A l'aide de `diff()` on peut également calculer une dérivée partielle d'une fonction à plusieurs variables.

```
Dérivées partielles

>> diff(cos(x*tan(y)),x)

                          -tan(y) sin(x tan(y))
>> diff(cos(x+y^2),x,y)

                                          2
                            - 2 y cos(x + y )
>> diff(cos(x+y^2),y,x)

                                          2
                            - 2 y cos(x + y )
>> diff((x^2 + y^2) * (ln(x) - ln(y)),x$2,y)

                                  2   2 y
                                - - - ---
                                  y     2
                                        x
>> diff((x^2 + y^2) * (ln(x) - ln(y)), y, x$2)

                                  2   2 y
                                - - - ---
                                  y     2
                                        x
>>
```

3.1.2 L'opérateur différentiel `D`

L'opérateur `D` s'applique à une fonction sans spécifier obligatoirement ses arguments.

```
L'opérateur différentiel

>> D(sin)

                                    cos
>> D(f+g)

                                D(f) + D(g)
>> D(f*g)

                              f D(g) + g D(f)
>> D(f@g)

                                D(g) D(f)@g
```

On peut itérer en utilisant `@@n` ou indexer en une forme `D[i]` qui retourne la dérivée partielle par rapport à la $i^{ème}$ variable.

```
Composition de D

>> f := (x,y) -> sqrt(x*y^5)

                              (x, y) -> sqrt(x*y^5)
>> D([2],f)

                         (x, y) -> 5/2*x*y^4/(x*y^5)^(1/2)
>> D([2,1],f)

          (x, y) -> 5/2*y^4/(x*y^5)^(1/2) - 5/4*x*y^9/(x*y^5)^(3/2)
>> D([1,2$2],f)

(x, y) -> 10*y^3/(x*y^5)^(1/2) - 35/2*x*y^8/(x*y^5)^(3/2) + 75/8*x^2*y^13/ (x*y^5)^(5/2)
>> (D@@1)(sin)

                                       cos
>> (D@@2)(sin)

                                      -sin
```

3.2 Intégrales

La fonction `int()` représente à la fois l'intégration définie et indéfinie. Ainsi pour le calcul d'une primitive, on ne spécifie pas les bornes :

```
Primitive

>> int(sin(x),x)

                                    -cos(x)
>> int(x/(x^2+x+1),x)
                                  /              /    1/2            \ \
                              1/2 |              | 2 3    (x + 1/2) | |
                  2           3   | PI - 2 arctan| ---------------- | |
     ln((x + 1/2)  + 3/4)         \              \         3        / /
     -------------------- + ------------------------------------------
               2                                 6
```

Pour le calcul d'une intégrale, on spécifie les bornes.

```
Intégrale

>> int(sin(x), x = 0..PI/2)

                                      1
>> int(x/(x^2+x+1), x = 0..1)

                                          1/2
                                ln(3)   PI 3
                                ----- - -------
                                  2        18
```

3.3 Développements limités

Soit $a \in \mathbb{R}$ et f une fonction définie dans un voisinage de a. On appelle développemement limité de f à l'ordre n au voisinage de a un polynôme P_n de degré inférieur ou égal à n tel que :

$$f(x) = P_n(x) + (x-a)^n \varepsilon(x) \text{ où } \lim_{x \to 0} \varepsilon(x) = 0$$

Avec les notations de Landau :

$$f(x) = P_n(x) + O((x-a)^n)$$

Il est possible de déterminer à l'aide de MuPAD le développement limité ou asymptotique d'une expression en un point. On utilise pour cela `taylor()` pour obtenir un développement en série de Taylor.

Développement limité

```
>> taylor(1/(1-x), x=0, 9)

                     2    3    4    5    6    7    8      9
            1 + x + x  + x  + x  + x  + x  + x  + x  + O(x )
```

Certaines fonctions ne possèdent pas de développement en série de Taylor, la fonction renvoie alors une erreur, la fonction `series()` permet de calculer des développements plus généraux.

Développement généralisé

```
>> taylor(cos(x)/x, x=0)
Error: does not have a Taylor series expansion, try 'series' [taylor]
>> series(cos(x)/x, x = 0)

                                3
                1   x   x       5
                - - - + -- + O(x )
                x   2   24
```

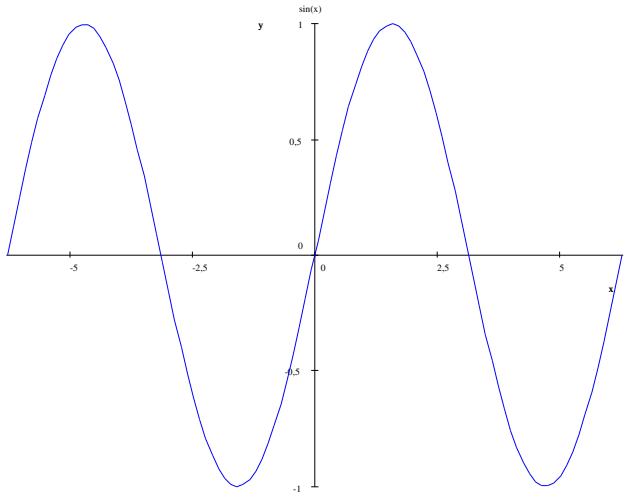
3.4 Graphiques

MuPAD offre de nombreuses possibilités pour montrer les objets mathématiques dans un espace à deux ou trois dimensions, nous nous contenterons d'aborder ici le tracé de fonctions et de champs de vecteur.

3.4.1 Tracés de fonctions

La fonction `plotfunc2d()` permet de tracer des fonctions à une variable.

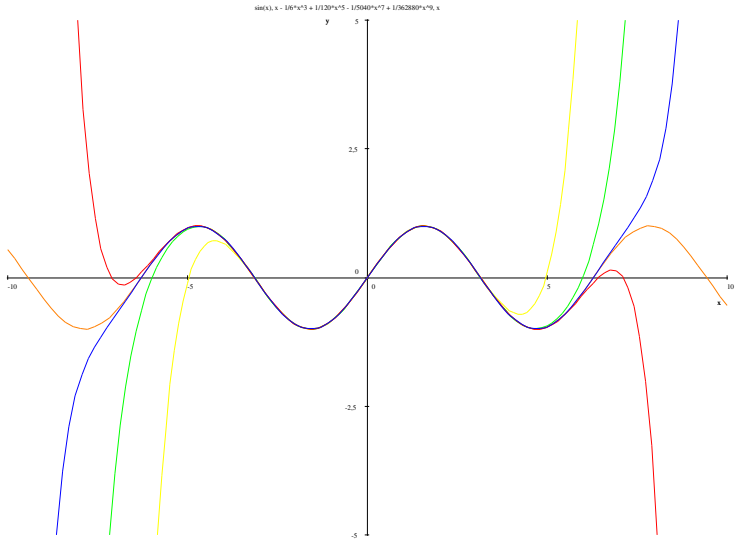
```
Tracé de fonctions à une variable

>> plotfunc2d(sin(x), x = -2*PI..2*PI)
```



De nombreuses fonctionalités sont offertes, entre autre la possibilité de tracer plusieurs fonctions sur un même graphe.

Tracé de plusieurs fonctions

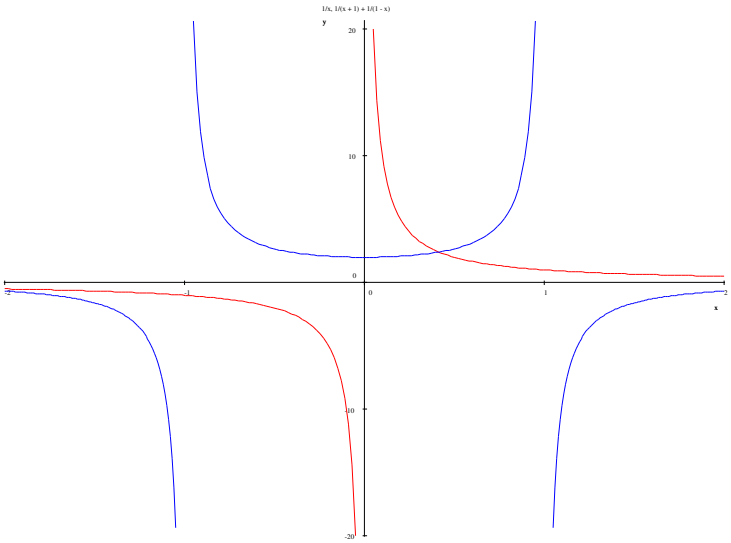
```
>> polynomeOrdre9 := expr(taylor(sin(x),x=0,9))

                           3     5     7       9
                          x     x     x       x
                      x - -- + --- - ---- + ------
                          6   120   5040   362880
>> polynomeOrdre13 := expr(taylor(sin(x),x=0,13))

                3     5     7       9         11          13
               x     x     x       x         x           x
           x - -- + --- - ---- + ------ - -------- + ----------
                6   120   5040   362880   39916800   6227020800
>> polynomeOrdre15 := expr(taylor(sin(x),x=0,15))

        3     5     7       9         11          13            15
       x     x     x       x         x           x             x
   x - -- + --- - ---- + ------ - -------- + ---------- - -------------
        6   120   5040   362880   39916800   6227020800   1307674368000
>> polynomeOrdre17 := expr(taylor(sin(x),x=0,17))

     3     5     7       9         11          13            15
    x     x     x       x         x           x             x
x - -- + --- - ---- + ------ - -------- + ---------- - ------------- +
     6   120   5040   362880   39916800   6227020800   1307674368000

          17
         x
   ---------------
   355687428096000
>> plotfunc2d(sin(x),polynomeOrdre9, polynomeOrdre13, polynomeOrdre15, polynomeOrdre17,
              x= -10..10,y=-5..5)
```



Il est possible de tracer des fonctions avec des singularités :

Fonctions avec singularités

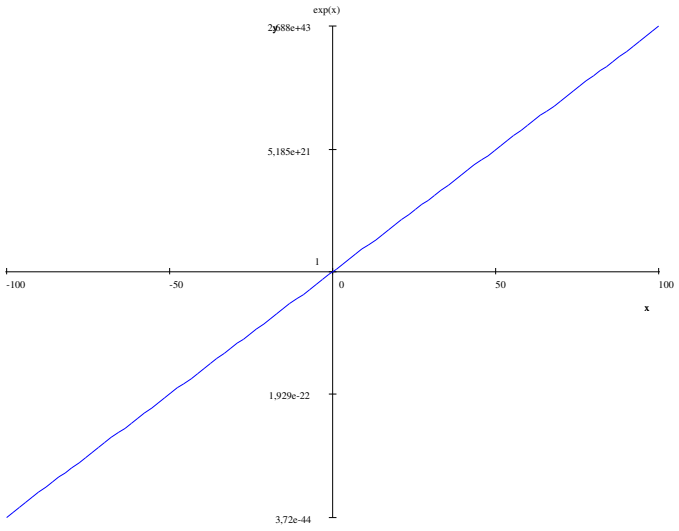
```
>> plotfunc2d(1/x, 1/(1 - x) + 1/(1 + x), x = -2..2)
```



On peut également choisir une échelle logarithmique.

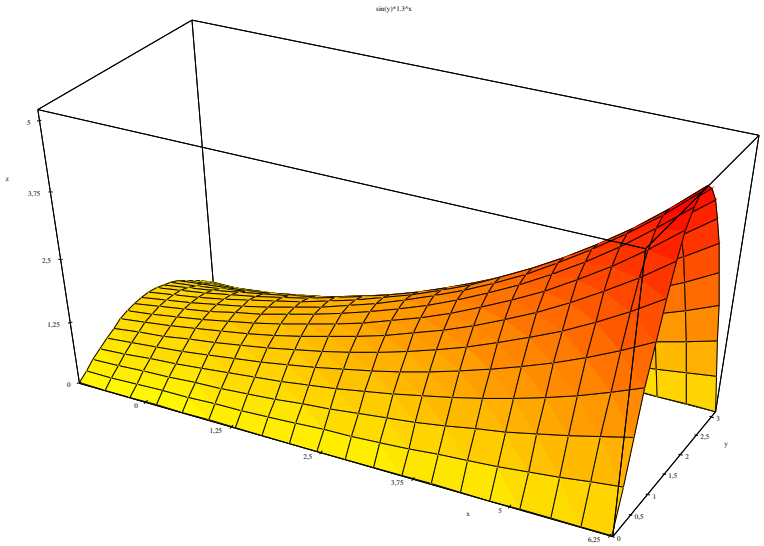
Echelle semi-log

```
>> plotfunc2d(exp(x), x = -100..100, CoordinateType = LinLog)
```



Il est également possible de tracer des fonctions de deux variables à l'aide de `plotfunc3d()`.

```
fonction à deux variables

>> plotfunc3d((1.3)^x*sin(y), x = -1..2*PI, YRange=0..PI)
```

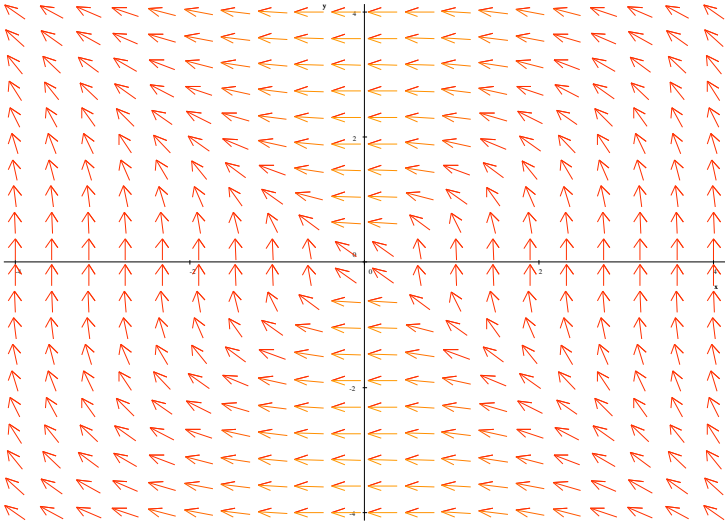


3.4.2 Champs de vecteurs

Pour les mécaniciens en herbe, il est également possible de tracer un champ de vecteurs. On définit au préalable le champ de vecteurs, puis on l'affiche. Pour l'exemple, on a :

$$(x, y) \to (-y^2, x^2)$$

```
Champ de vecteur

>> champ := plot::vectorfield([-y^2, x^2], x = -4..4, YRange = -4..4)

                                plot::Group()
>> plot(champ)
```



3.4.3 Représentation graphique à l'aide de points

MuPad permet de manipuler des listes (voir 5.2.4) qui sont constituées d'une séquence ordonnée d'objets MuPAD entre crochets.

```
Exemples de listes

>> liste1 := [1, 2, 3, 4, 5, 6]

                             [1, 2, 3, 4, 5, 6]
>> liste2 := [liste1, PI, cos(x)^2, Coucou ]

                                                 2
                  [[1, 2, 3, 4, 5, 6], PI, cos(x) , Coucou]
```

et l'on peut également générer une suite d'éléments à l'aide de $.

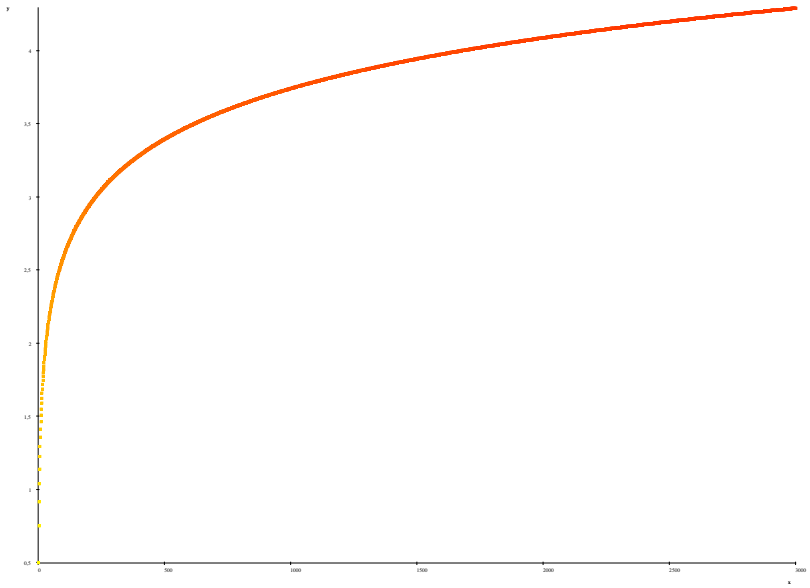
Utilisation du génèrateur d'éléments

```
>> premier := [ithprime(i) $i = 1..79] //utilisation de $
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
   71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139,
   149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,
   227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293,
   307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383,
   389, 397, 401]
>>
```

On peut alors construire des listes de points `plot::Pointlist()` que l'on affiche. Si on veut tracer l'évolution de la largeur en dehors des bouts de bois (voir 2.2.3), on peut tracer la suite en fonction de n.

Suite : $\sum_{i=1}^{n} \frac{1}{2*i}$

```
>> PlotPoints := plot::Pointlist([n, sum(1/(2*i), i=1..n)] $n = 1 .. 3000)
// Ce n'est pas la bonne méthode

                              plot::Pointlist()
>> plot(PlotPoints)
```



3.5 Exercices

V *Exercice n°5*

V.1 Calculer la limite de la suite :

$$u_n = 3 * 2^{\frac{1}{n}} - 2 * 3^{\frac{1}{n}}$$

Solution

```
>> u := 3 * 2^(1/n) - 2 * 3^(1/n)
```

$$3\,2^{\frac{1}{n}} - 2\,3^{\frac{1}{n}}$$

```
>> limit(u, n=infinity)
```

$$1$$

VI *Exercice n°6*

On considère la suite récurrente définie par :

$$u_n = u_{n-1} + u_{n-2}, u_0 = \sqrt{(5)} - 1 \text{ et } u_1 = \sqrt{(5)} - 3$$

VI.1 Calculer la valeur exacte de u_{100}. Vous utiliserez la composition de fonctions.

VI.2 Calculer les valeurs approchées de u_{100} avec 10, 20, 30, 40, 50 et 100 chiffres significatifs.

Solution :
Calculons quelques termes de la suite :

$$\begin{aligned} u_2 &= u_1 + u_0 \\ u_3 &= u_2 + u_1 \\ &= 2u_1 + u_0 \\ u_4 &= u_3 + u_2 \\ &= 3u_1 + 2u_0 \\ u_5 &= 5u_1 + 3u_0 \\ u_6 &= 8u_1 + 5u_0 \end{aligned}$$

Si l'on définit $f(u_1, u_0) = (u_1 + u_0, u_1)$

$$\begin{aligned} f(u_1, u_0) &= (u_1 + u_0, u_1) \\ f(f(u_1, u_0)) &= f(u_1 + u_0, u_1) \\ &= (2u_1 + u_0, u1 + u_0) = (u_3, u_2) \\ f(f(f(u_1, u_0) &= f(2u_1 + u_0, u1 + u_0) \\ &= (3u_1 + 2u_0, 2u_1 + u_0) = (u_4, u_3) \end{aligned}$$

```
Solution

>> f := (x, y) -> (x + y, x)

                              (x, y) -> (x + y, x)
>> u0 := sqrt(5) - 1

                                    1/2
                                   5    - 1
>> u1 := sqrt(5) - 3

                                    1/2
                                   5    - 3
>> u100 := (f@@100)(u1, u0)

                          1/2
92737269219307899917 6 5    - 2073668380220713167378,
                          1/2
57314784401381708410 1 5    - 1281597540372340914251
>> float(u100[2])

                                                                        -128.0
>> DIGITS := 20

                                                                          20
>> float(u100[2])

                                                    -0.00000002980232238769531 25
//etc
```

VII *Exercice n°7*

VII.1 Calculer la dérivée de la fonction f :

$$f : t \mapsto \arcsin(2t\sqrt{1-t^2})$$

VII.2 Simplifier l'expression en supposant $t > \frac{1}{\sqrt{2}}$, puis que $t \in]-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}[$. Vous utiliserez la commande `assume()`.

Solution

```
>> f := arcsin(2 * t * sqrt(1 - t^2))
```

$$\arcsin\left(2\,t\,\left(1-t^{2}\right)^{1/2}\right)$$

```
>> d := diff(f,t)
```

$$\frac{2\left(1-t^{2}\right)^{1/2}-\dfrac{2\,t^{2}}{\left(1-t^{2}\right)^{1/2}}}{\left(1-4\,t^{2}\left(1-t^{2}\right)\right)^{1/2}}$$

```
>> simplify(d)
```

$$\frac{2\left(1-t^{2}\right)^{1/2}-\dfrac{2\,t^{2}}{\left(1-t^{2}\right)^{1/2}}}{\left((2\,t^{2}-1)^{2}\right)^{1/2}}$$

```
>> assume(t, Type::Real) : assume(t > 1/sqrt(2), _and)
```

$$t > 1/2*2^{(1/2)}$$

```
>> simplify(d)
```

$$\frac{2\left(1-t^{2}\right)^{1/2}-\dfrac{2\,t^{2}}{\left(1-t^{2}\right)^{1/2}}}{2\,t^{2}-1}$$

```
>> assume(t, Type::Real) : assume(t < 1/sqrt(2), _and) : assume( t > -1/sqrt(2), _and)
```

$$t \in \left]-1/2*2^{(1/2)},\ 1/2*2^{(1/2)}\right[$$

```
>> simplify(d)
```

$$\frac{2\left(1-t^{2}\right)^{1/2}-\dfrac{2\,t^{2}}{\left(1-t^{2}\right)^{1/2}}}{1-2\,t^{2}}$$

VIII *Exercice n°8*

VIII.1 Tracer le graphe de la fonction f et de sa dérivée :

$$f : t \mapsto \arcsin\left(\frac{t+\sqrt{1-t^2}}{2}\right)$$

```
Solution

>> f := t-> arcsin((t + sqrt(1 - t^2))/2)

                                          t -> arcsin((t + sqrt(1 - t^2))/2)
>> d := diff(f(t),t)

                                                                  t
                                                        1/2 - -------------
                                                                     2 1/2
                                                              2 (1 - t )
                                                    --------------------------------
                                                    /     /           2 1/2 \2 \1/2
                                                    |     | t   (1 - t )    |  |
                                                    | 1 - | - + ----------- |  |
                                                    \     \ 2         2     /  /
 >> plotfunc2d(f(t),d(t), t=-1..1, y=-1..1)
```

IX *Exercice n°9*

A l'aide de MuPAD, étudiez la fonction :

$$f(x) = x^2 \arctan \frac{1}{1+x}$$

IX.1 Quel est son domaine de définition dans $\mathbb{R}$? Vous pourrez utiliser la fonction `discont()`.

IX.2 Donnez sous une forme simplifiée la dérivée de f.

IX.3 Calculer $f(2)$.

IX.4 Étudier les limites de la fonction f.

IX.5 Tracer la fonction à l'aide de MuPAD.

IX.6 Déterminer le développement de rang 3 de Taylor à droite et à gauche de -1.

IX.7 Tracer la fonction, sa dérivée et les développements limités sur un même graphe.

Pour résoudre cet exercice, on va créer un fichier contenant la résolution, la fonction qui permet de lire un fichier contenant des fonctions MuPAD est la fonction `read()`.

Solution

```
print(NoNL,"Fonction f :"): f:=x^2*arctan(1/(1+x));
//
// Domaine de définition
//
print(NoNL,"Positions des discontinuités :"): discont(f,x,Dom::Real);
// Dérivée
g:=diff(f,x):
print(NoNL,"Dérivée :"): simplify(g);
//
// Calcul de f(2)
//
print(NoNL,"valeur en 2 :"): eval(subs(f,x=2)),float(eval(subs(f,x=2)));
//
// Etude des limites
//
print(NoNL,"limite en +infini :"): limit(f,x=infinity);
print(NoNL,"limite en -infini :"): limit(f,x=-infinity);
print(NoNL,"limite en -1 :"): limit(f,x=-1);
print(NoNL,"limite à droite en -1 :"): limit(f,x=-1,Right);
print(NoNL,"limite à gauche en -1 :"): limit(f,x=-1,Left);
//
// Graphe de la fonction
//
plotfunc2d(f, x = -5..5);
input();
//
// Développement limités
//
print(NoNL,"Développement de Taylor à droite en x=-1,
      3 termes sont demandés  :"):
poly3d := expr(series(f,x=-1,3,Right));
print(NoNL,"Développement de Taylor à gauche en x=-1,
      3 termes sont demandés  :"):
poly3g := expr(series(f,x=-1,3,Left));
//
// Graphe de la fonction et des développements limités et de la dérivée
//
enMoins1 := piecewise([x < -1, poly3g], [x > -1, poly3d]);
plotfunc2d(f, enMoins1, g,  x = -5..5, y= -6..6);
```

```
Trace de la solution


>> read("ex9.mu")
Fonction f :
                                                        2       /   1   \
                                                       x  arctan| ----- |
                                                                \ x + 1 /
Positions des discontinuités :
                                                              -1
Dérivée :
                                                                          2
                                                          /   1   \      x
                                                2 x arctan| ----- | - ----------
                                                          \ x + 1 /         2
                                                                    2 x + x  + 2
valeur en 2 :
                                                    4 arctan(1/3), 1.287002218
limite en +infini :
                                                             infinity
limite en -infini :
                                                             -infinity
limite en -1 :
                                                             undefined
limite à droite en -1 :
                                                                PI
                                                                --
                                                                 2
limite à gauche en -1 :
                                                                 PI
                                                               - --
                                                                  2

Développement de Taylor à droite en x=-1, 3 termes sont demandés  :
                                                PI                                  2 / PI     \
                                                -- + (x + 1) (- PI - 1) + (x + 1)   | -- + 2 |
                                                 2                                  \  2     /
Développement de Taylor à gauche en x=-1, 3 termes sont demandés  :
                                                                      PI          2 /     PI \
                                                 (x + 1) (PI - 1) - -- + (x + 1)   | 2 - -- |
                                                                      2            \      2 /


        /   PI                              2 /   PI     \
piecewise| - -- + (x + 1) (PI - 1) + (x + 1)  | - -- + 2 | if x < -1,
        \    2                               \    2     /

   PI                                2 / PI     \           \
   -- + (x + 1) (- PI - 1) + (x + 1)  | -- + 2 | if -1 < x |
    2                                \  2     /           /
```

4 Résolution d'équations

MuPAD par l'intermédiaire de la fonction `solve()` permet de résoudre des équations ou même des inéquations. Cette résolution peut être formelle ou numérique.

```
Equation du 1er et 2nd degré


>> solve(a*x + b = c, x)

         / { - b + c }
piecewise| { ------- } if a <> 0, C_ if a = 0 and - b + c = 0,
         \ {    a    }

                                  \
   {} if a = 0 and - b + c <> 0 |
                                  /

>> solve(a*x^2 + b*x +c = 0, x)

         /
         |
         |
         |
piecewise| C_ if a = 0 and b = 0 and c = 0,
         \
                                     {   c }
   {} if a = 0 and b = 0 and c <> 0, { - - } if a = 0 and b <> 0,
                                     {   b }

   {                    2 1/2                      2 1/2 }          \
   {   b    (- 4 a c + b )       b    (- 4 a c + b )    }           |
   { - - - ----------------- - - + ----------------- }           |
   {   2           2             2           2         }           |
   { ----------------------, ---------------------- } if a <> 0 |
   {            a                        a           }          /
```

Si l'équation à un nombre infini de solutions le domaine est défini. Ainsi dans l'exemple suivant $X_2, X_4 \in \mathbb{Z}$.

```
>> solve(sin(x)+cos(x) = 1)

    x in { 2*X2*PI |  X2 in Z_ } union { 1/2*PI + 2*X4*PI |  X4 in Z_ }
```

Pour calculer numériquement les solutions d'une équation, on obtient toutes les solutions de l'équation, puis on peut utiliser la fonction `float()` pour obtenir une approximation. Dans l'exemple qui suit `RootOf` représente l'ensemble des racines solutions du polynome et la fonction `map()` applique la fonction `float()` à l'ensemble des solutions.

```
Solution numérique

>> solve(x^7 + x^2 + x, x)

                                              6
                    {0} union RootOf(X5 + X5  + 1, X5)
>> map(last(1),float)

{- 0.1547351445 + 1.038380754 I, - 0.1547351445 - 1.038380754 I,

   - 0.7906671888 - 0.3005069203 I, - 0.7906671888 + 0.3005069203 I,

   0.9454023333 + 0.6118366938 I, 0.9454023333 - 0.6118366938 I, 0.0}
```

Terminons par une inéquation :

```
Inéquation

>> solve(x^4 > 15, x)

             ]15^(1/4), infinity[ union ]-infinity, -15^(1/4)[
```

4.1 Système d'équations

Pour résoudre un système d'équation on spécifie les équations et les inconnues sous la forme d'un ensemble.

Considérons le problème suivant : Le père Olivier éleveur d'ânes de son état possède un terrain de forme carrée. Il décide de créer sur son terrain une carrière sous la forme d'un enclos carré pour entrainer ses ânes au saut d'obstacles, pour les prochains jeux olympiques. Pour clôturer le terrain et l'enclos, il lui faut 6000 mètres de lisses en bois et il y a 3 lisses par hauteur. La partie non occupée par la carrière est mise en herbe et cette surface ensemençée est de $150000m^2$, quelle est la taille du champ, quelle est la taille de la carrière ?

```
Système d'équations

>> equations := {4 * x + 4 * y = 2000, x^2 - y^2 = 150000}

                                      2    2
                   {4 x + 4 y = 2000, x  - y  = 150000}
>>  solve(equations, {x, y})

                           {[x = 400, y = 100]}
```

4.2 Equations différentielles

Même s'il n'est pas dans notre propos, d'étudier les équations différentielles commençons par situer le problème en termes de systèmes dynamiques qui conduisent génèralement à des modèles mathématiques utisant des équations différentielles. La notion de système dans ce cadre correspond au processus que l'on cherche à modéliser, il évolue dans le temps et le système est alors décrit par des variables qui varient en fonction du temps. Le système est ouvert s'il y a des échanges (matière/énergie) avec l'extérieur. Les échanges s'effectuent par l'intermédiaire des entrées et des sorties du système. Les entrées du système correspondent aux variations expérimentales que le système subit. Les sorties sont les réponses du système c'est-à-dire les échanges avec le milieu extérieur et également les variables modifiées.

Les paramètres d'un modèle interviennent dans les équations du modèle et sont des grandeurs fixées. Les fonctions utilisées sont fréquemment non-linéaires soit par rapport aux paramètres, soit par rapport aux variables :

– Par rapport aux paramètres, un exemple simple est donné par l'équation :

$$y = a^2 x \tag{14}$$

Si la valeur de a est modifiée d'un facteur n celle de y sera modifiée d'un facteur n^2. Il n'y pas proportionalité entre la cause qui est la modification de a et son effet qui se traduit par la modification de la valeur de y.

– Par rapport aux variables. On considère qu'une fonction est non linéaire si l'une au moins de ses dérivées premières n'est pas constante. Considérons par exemple l'équation

$$f(x_1, x_2, x_3) = ax_1 + bx_2x_3 \text{ avec } a, b \text{ constantes} \tag{15}$$

```
Exemple

>> f := (x1, x2, x3) -> a*x1 + b*x2*x3

                        (x1, x2, x3) -> a*x1 + b*x2*x3
>> diff(f(x1,x2,x3),x1)

                                          a
>> diff(f(x1,x2,x3),x2)

                                      b x3
>> diff(f(x1,x2,x3),x3)

                                      b x2
>>
```

$\frac{\partial f(x_1,x_2,x_3)}{\partial x_2} \neq$ constante l'équation n'est donc pas linéaire.

Schématiquement, un système dynamique est décrit par une loi d'évolution qui, à partir de conditions initiales permet de déterminer le futur ; l'évolution temporelle est décrite par une ou plusieurs équations différentielles. Ces équations peuvent être ordinaires (ODE ordinary differential equation) si elles font intervenir uniquement les variables et leurs dérivées par rapport par rapport au temps. Si vos équations différentielles contiennent des dérivées par rapport à plusieurs grandeurs, le temps et une distance par exemple, il s'agit d'une équations aux dérivées partielles. Enfin l'ordre d'une équation différentielle désigne l'ordre de la dérivée de plus haut rang qu'elle contient. Une équation du premier ordre ne contient que des dérivées

premières de ses variables. L'équation linéaire de premier ordre la plus simple s'écrit sous la forme :

$$\frac{dy}{dt} = ky \tag{16}$$

```
Equation différentielle ordinaire du premier ordre

>> equa_diff1 := ode(diff(y(t),t) = k*y(t), y(t))

                    ode(- k y(t) + diff(y(t), t), y(t))
>> solve(equa_diff1)

                                C1 exp(k t)
>> equa_diff2 := ode(y'(t) = k*y(t), y(t))

                    ode(- k y(t) + diff(y(t), t), y(t))
>> solve(equa_diff2)

                                C2 exp(k t)
```

La fonction `ode()` définit une équation différentielle ordinaire, on précise une équation et une fonction. On peut alors chercher la solution génèrale à l'aide de `solve()`.

Examinons maintenant un exemple, selon Thomas Malthus[3], la vitesse de croissance d'une population est proportionnelle à son effectif présent. Soit k la constante qui exprime le pourcentage d'augmentation de la population pendant la période considérée (ex : $k = 1,25$ si la population a connu une augmentation de 25%). Si l'effectif initial est de n_0 à $t = 0$, il devient à Δt défini comme unitaire :

$$n_1 = kn_0 \tag{17}$$

De même au pas de temps suivant :

$$n_2 = kn_1 \tag{18}$$

De proche en proche, on obtient l'équation (19) de Malthus :

$$n_{(t)} = kn_{(t-1)} \tag{19}$$

L'accroissement de la population pendant le temps Δt est de :

$$\Delta n_{(t)} = n_{(t+1)} - n_{(t)} = kn_{(t)} - n_{(t)} = (k-1)n_{(t)} \tag{20}$$

La figure 6 montre la suite des valeurs $n_{(t)}$. Un système dynamique évolue en fonction du temps mais la représentation proposée il n'apparaît pas de façon explicite. On le réintroduit en représentant (figure 7) la suite des valeurs successives de $n_{(t)}$ en fonction du temps (suite chronologique).

L'accroissement que nous venons d'établir (20) correspond à l'accroissement durant un pas de temps Δt. Si cette période est deux fois plus grande, la valeur obtenue pour $\Delta n_{(t)}$ serait deux fois plus grande et ainsi de suite. On peut écrire :

$$\frac{\Delta n_{(t)}}{\Delta t} = (k-1)n_{(t)} \tag{21}$$

[3]Le principe des populations, 1798

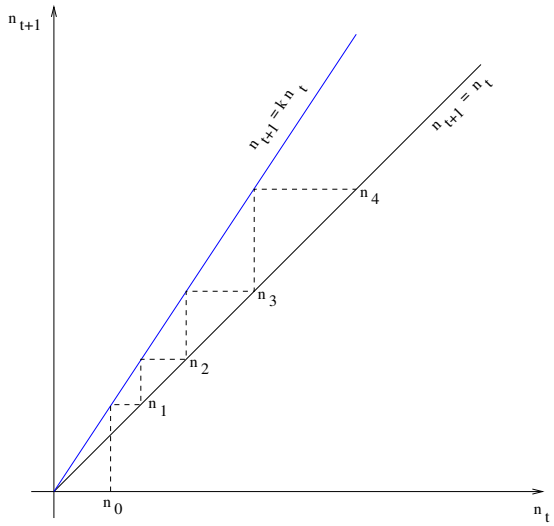


FIG. 6 – Équation de Malthus dans le plan (n_{t+1}, n_t)

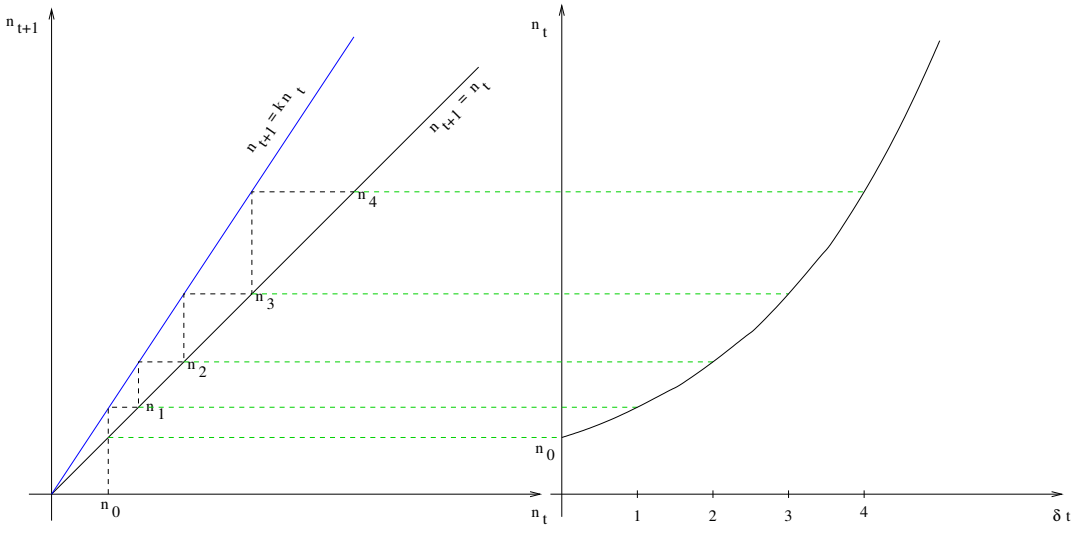


FIG. 7 – Série chronologique

En passant des différences finies aux différentielles :

$$\frac{dn_{(t)}}{dt} = (k-1)n_{(t)} \tag{22}$$

On résout cette équation différentielle en utilisant MuPAD :

```
Résolution de l'équation de Malthus

>> equa_malthus := ode({diff(n(t),t) = (k - 1) * n(t), n(0) = n0}, n(t))

           ode({n(0) = n0, diff(n(t), t) - n(t) (k - 1)}, n(t))
>> solve(equa_malthus)

                              {n0 exp(k t) exp(-t)}
```

Cela conduit donc à une croissance exponentielle $n_{(t)} = n_0 e^{(k-1)}t$, on parle alors de loi exponentielle.

La représentation ainsi que la résolution nous montre que l'effectif de la population ne croît pas de façon linéaire. Les intervalles de temps sont constants et la différence entre n_i et n_{i+1} est plus petite que celle entre n_j et n_{j+1} avec $i < j$. La loi d'évolution est linéaire par contre la série chronologique ne l'est pas.

Si $n_0 = 0$, la population reste nulle, quel que soit la valeur de k. Ce point particulier (singulier) qui reste invariant suite à l'application de la règle d'évolution est un point fixe et correspond à un état stationnaire du système dynamique.

Les propriétés de l'état stationnaire dépendent de celles de la règle d'évolution. Lorsque $k > 1$, la population augmente quel que soit l'effectif n_0 initial. Toute perturbation apportée au système provoque l'éloignement de ce point fixe. L'état stationnaire est instable, il est appelé répulseur (cf. fig. 8). Par contre,

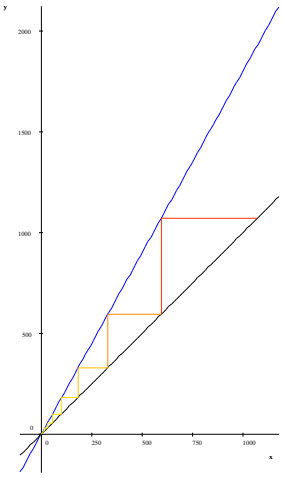


FIG. 8 – Equation de Malthus $k > 1$

lorsque $k < 1$, la population diminue de façon irrémédiable vers 0. L'état stationnaire est stable, il est appelé attracteur (cf. fig. 9).

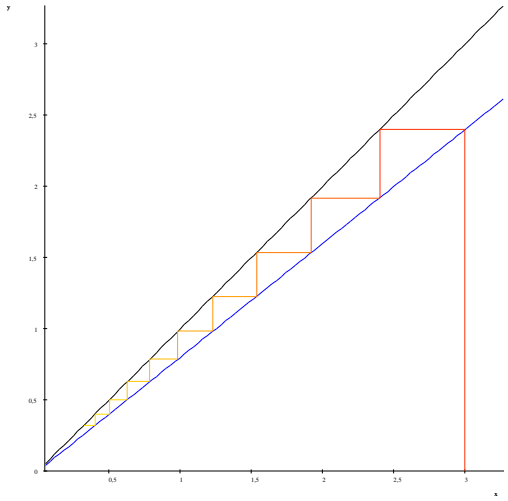


FIG. 9 – Equation de Malthus $k < 1$

Le modèle malthusien ne reflète pas réellement la réalité, puisque la croissance est sans fin. Un demi-siècle après, Darwin reprend le principe énoncé par Malthus et ajoute la notion de compétition entre espèces. Verhulst avait proposé (eq. 23) auparavant à partir du modèle Malthusien un modèle bornant la croissance exponentielle.

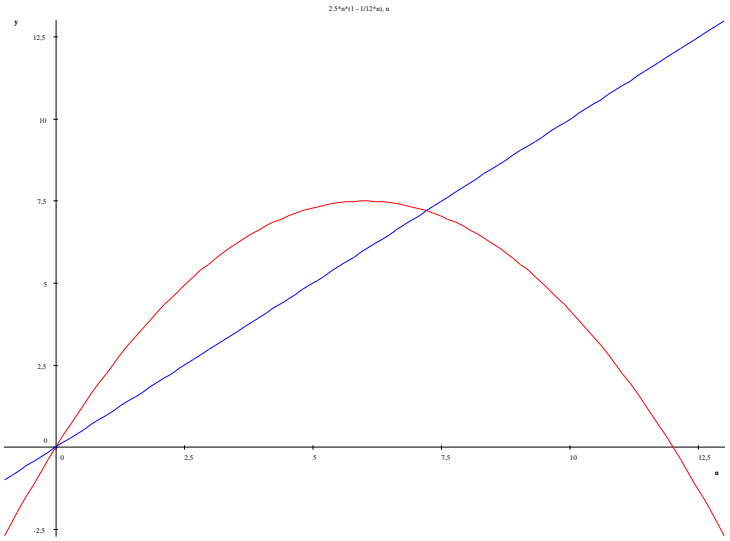
$$n_{(t+1)} = \lambda n_{(t)}(1 - \frac{n_{(t)}}{K}) \tag{23}$$

Ce modèle correspond à la loi logistique. On peut utiliser MuPAD pour étudier ce modèle.

```
Solution de l'équation logistique

>> equa_logistique := n -> lambda * n * (1 - n/K)

                          n -> lambda*n*(1 - n/K)
>> solve(equa_logistique(n), n)

piecewise({} if K = 0 and lambda = 0, C_ if K <> 0 and lambda = 0,

   {} if K = 0 and lambda <> 0, {K, 0} if K <> 0 and lambda <> 0)
```

L'effectif n varie entre 0 et K lorsque $\lambda \neq 0$ et $K \neq 0$.

```
Graphe de l'équation logistique

>> lambda := 2.5

                                    2.5
>> K := 12
>>  plotfunc2d(equa_logistique(n),n, n = -1.. 13)
```



L'équation logistique avec $0 \leq x \leq K$ présente un comportement intéressant en fonction de la valeur de λ. Pour des faibles valeurs de λ, la courbe se situe en dessous de la première bissectrice et quelle que soit la valeur initiale n_0 le système converge vers l'attracteur $(0, 0)$ (figure 10).

D'une façon génèrale, lorsque $\lambda < 3$, le système converge vers un état stationnaire qui révèle la présence d'un attracteur ponctuel (figure : 11).

Si $\lambda = 3.1$, le système oscille entre deux valeurs (figure 12).

et si l'on augmente encore $\lambda < 3.5699456718$ (figure 13) le système va osciller entre 4, 8, 16, 32 ... valeurs. On est en présence d'attracteurs cycliques, l'augmentation de λ provoquant des dédoublements de la période.

Lorsque $\lambda > 3.5699456718$, c'est le chaos (figure 14), le système se comporte de façon désordonnée et manifeste la présence d'un attracteur étrange.

La démarche pour l'établissement d'un modèle pour un système dynamique comporte donc les phases suivantes [4] :

- Identification des variables, de leur interaction et des boucles de régulation qui les relient ;
- Etablissement du système d'équations décrivant l'évolution temporelle des variables ;
- Identification des états stationnaires, leurs nombres et leurs conditions d'existence et leurs valeurs (analytique ou numérique) ;
- Etude de la stabilité des états stationnaires ;

[4] Remarque : les deux derniers points n'ont pas été traités dans ce qui précède car il sorte du contenu de ce cours.

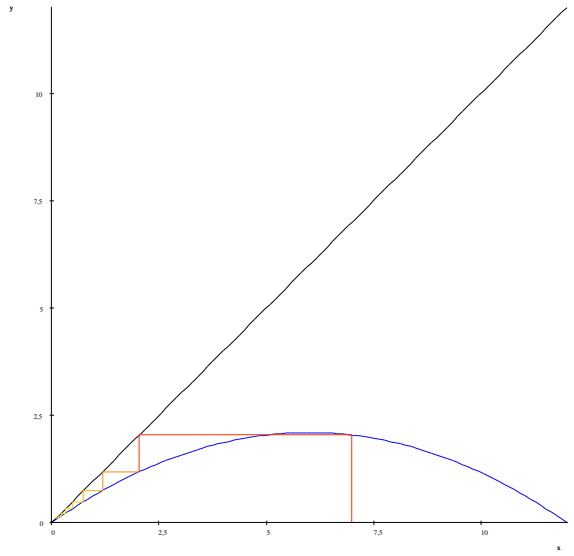


FIG. 10 – Loi logistique $\lambda = 0.7$

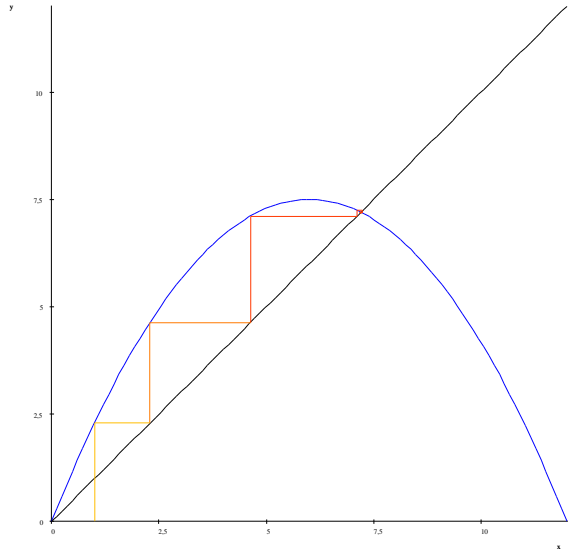


FIG. 11 – Loi logistique $\lambda = 2.5$

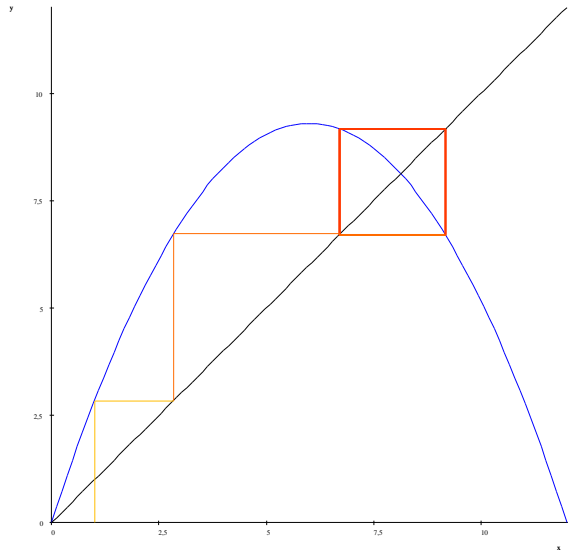


FIG. 12 – Loi logistique $\lambda = 2.5$

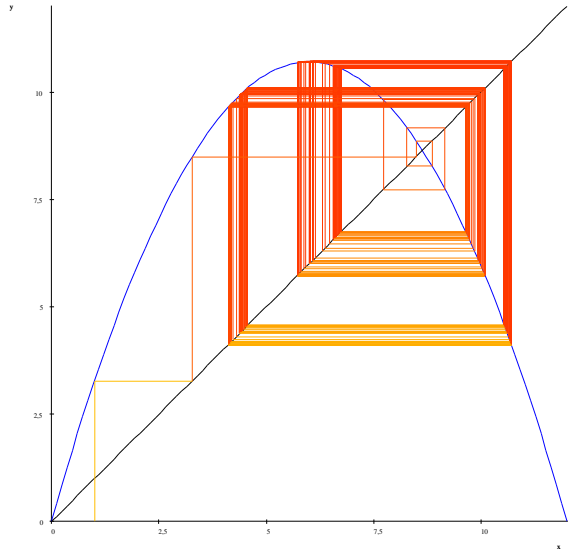


FIG. 13 – Loi logistique $\lambda = 3.569$

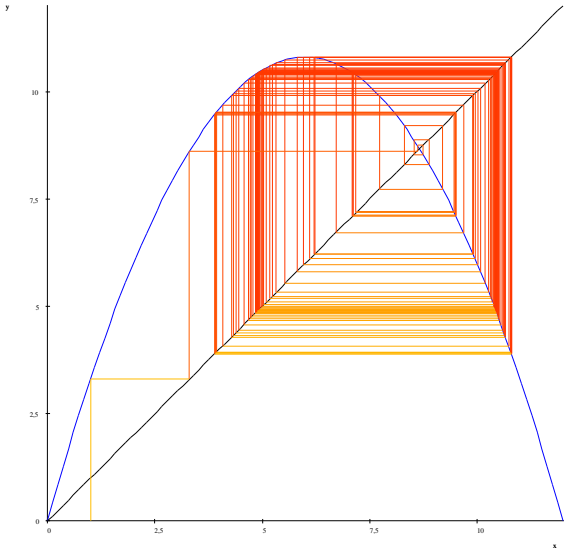


FIG. 14 – Loi logistique $\lambda = 3.6$

– Intégration (numérique) éventuelle du système d'équations différentielles ;
– Étude du chaos éventuel à l'aide du diagramme de bifurcation.

Solution

```
>> r := rec(v(k) = v(k - 1) + v(k - 2), v(k),  {v(0) = sqrt(5) - 1, v(1) = sqrt(5) - 3})

                                                                  1/2              1/2
                      rec(v(k) - v(k - 1) - v(k - 2), v(k), {v(0) = 5    - 1, v(1) = 5    - 3})
>> res := solve(r)

                                          {            /        1/2 \ k }
                                          {   1/2      |       5    |   }
                                          { (5    - 1) | 1/2 - ---- |   }
                                          {            \         2  /   }

>> u := fp::unapply(res[1], k)

                                    k -> (5^(1/2) - 1)*(1/2 - 1/2*5^(1/2))^k

>> u(100)

                                                    /        1/2 \ 100
                                              1/2   |       5    |
                                            (5    - 1) | 1/2 - ---- |
                                                       \         2  /
>> float(%)

                                                  1.560552308e-21
>> limit(u(k), k = +infinity)

                                                         0
```

5 Programmation en MuPAD

MuPAD permet de réaliser des programmes. Il utilise pour cela des objets, des instructions, des fonctions du logiciel et des fonctions définies par l'utilisateur.

5.1 Les variables

Tout d'abord commençons par la notion d'*identificateur*. Les identificateurs sont des noms qui représentent des variables, des inconnues, des fonctions ou des procédures. C'est une combinaison quelconque de lettres, de chiffres et du caractère _. Le premier caractère ne peut pas être un chiffre. Un identificateur a pour valeur l'objet qui lui a été affecté par l'intermédiaire de l'opérateur d'affectation :=. Si aucune valeur n'a été affectée alors le nom correspond à la valeur.

Dans la plupart des langages informatiques, on utilise des variables, c'est-à-dire que l'on associe à un identificateur (un nom) un contenu. On pourra ainsi appeler une variable `variable` et y stocker la valeur du nombre complexe $1 + 5i$.

```
Exemple de variables

>> variable

                                  variable
>> domtype(variable)

                                  DOM_IDENT
>> variable := 1 + 5*I

                                   1 + 5 I
>> domtype(variable)

                                 DOM_COMPLEX
>> x := 3: y:=4

                                       4
>> tmp := x : x := y : y := tmp

                                       3
>> x; y

                                       4

                                       3
>> delete tmp

>>
```

Toute variable a une durée de vie et une portée. En MuPAD cela définit deux types de variables :

– Les *variables locales* : ce sont toutes les variables qui sont déclarées à l'intérieur des procédures ou en tant que paramètres. La portée de ces variables est la procédure et la durée de vie est limitée à la durée d'exécution de la procédure. En dehors de la procédure, elles ne sont plus accessibles puisqu'elles n'existent plus.

– Les *variables globales* : cela doit être exceptionnel, mais vous pouvez avoir besoin d'une variable qui soit accessible depuis n'importe quelle procédure. Dans ce cas la variable existe tant que vous ne

l'avez pas détruite soit par `delete` ou par `reset()` ou encore en quittant votre session MuPAD . La définition de telles variables est précisée dans le paragraphe 5.3.1.

5.2 Objets structurés

Nous avons déjà rencontré de nombreux objets comme les nombres et les expressions symboliques, nous les présentons ici dans une optique de programmation. Tout objet MuPAD possède un type qui correspond à sa représentation interne, définit son domaine de valeur et les opérations possibles.

```
Type d'un objet MuPAD

>> domtype(1 - 2*I)
                                DOM_COMPLEX
>> domtype(x + y)

                                  DOM_EXPR
```

5.2.1 Expressions symboliques

Une expression est composée d'opérandes, d'opérateurs et d'appels de fonctions. Une expression peut elle-même contenir une sous-expression.

L'équation $\frac{f(x,y,z)}{g(x)} * PI * e^x = 0$ est une expression.

L'expression $a + b$ est composée de deux opérandes a et b et de l'opérateur $+$.

Il peut être nécessaire de décomposer une expression en composants pour pouvoir les manipuler. Les fonctions MuPAD qui permettent cela sont en particulier `nops()` et `op()`. La première fonction retourne le nombre d'opérandes contenues dans l'expression.

```
Utilisation de nops()

a := x + Y + sin(x)

                                                              Y + x + sin(x)
>> nops(a)

                                                                        3
```

La seconde fonction `op()` permet de récupérer un ou plusieurs opérandes donnés.

```
Utilisation de op()

>> op(a)

                                                                    Y, x, sin(x)
>> op(a,1)

                                                                          Y
>> op(a,2..3)

                                                                     x, sin(x)
>>
```

MuPAD permet l'utilisation d'opérateurs et les transforme en fonctions. Les différents opérateurs sont présentés dans le tableau 3.

La forme fonctionnelle des opérateurs est précisée dans l'aide de MuPAD pour chaque opérateur. Cette forme permet d'appliquer l'opérateur à plusieurs arguments.

```
Utilisation de la fonction à la place de l'opérateur

>>_plus(i^2 $ i = 0..100)

                                                                        338350
```

ou encore de l'appliquer à l'aide de la fonction `map()` aux éléments d'une liste.

```
Utilisation de la fonction à la place de l'opérateur

>> map([1, 2 , 3], fact)

                                                                    [1, 2, 6]
```

Des règles de priorité et d'associativité sont définies, elles sont précisées dans l'aide en consultant "Quick Reference" [5].

5.2.2 Expressions booléennes

Les expressions booléennes sont souvent importantes en programmation car elles permettent de tester si quelque chose est vrai ou de répéter un traitement jusqu'à ce qu'une condition devienne vraie. MuPAD manipule trois valeurs logiques `TRUE`, `FALSE`, et `UNKNOWN` et utilise les opérateurs `and`, `or` et `not`. La fonction `bool()` évalue les expressions booléennes.

Opérateur	Signification	Exemple
$+$	addition	$somme := a + b$
$-$	soustraction	$soustrac := a - b$
$*$	multiplication	$mult := a * b$
$/$	division	$division := a/b$
div	division entière	$quotient := a\ div\ b$
mod	reste division entière	$reste := a\ mod\ b$
^	puissance	$puissance := a\hat{}b$
!	factorielle	$fact := n!$
$	génération de séquence	$sequence := i\hat{}2\ \$i = 5..7$
,	concaténation de séquence	$seq := seq1, seq2$
$union$	union de deux ensembles	$s := s1\ union\ s2$
$intersect$	intersection de deux ensembles	$s := s1\ intersect\ s2$
$minus$	différence de deux ensembles	$s := s1\ minus\ s2$
$=$	équation	$equation = a * x + b = 0$
$<>$	inégalité	$condition := a <> b$
$<$	inférieur strictement	$condition := a < b$
$<=$	inférieur	$condition := a <= b$
$>$	supérieur strictement	$condition := a > b$
$>=$	supérieur	$condition := a >= b$
not	négation	$contraire := not\ condition$
and	et logique	$condition := a < b\ and\ b < c$
or	ou logique	$condition := a < b\ or\ b < c$
$\rightarrow$	application	$f := x \rightarrow a * x\hat{}2$
'	différentielle	$derivee := f'(x)$
@	composition de fonction	$h := f@g$
@@	itération	$g := f@@4$
..	intervalle	$intervalle := a..b$
.	concaténation	$nouveauNom := nouveau.Nom$
$identificateur()$	appel de fonction	$\sin(1), f(x)$

TAB. 3 – Principaux opérateurs

```
Exemple d'expressions booléennes

 >> a := PI; b := 3.14

                                                                                    PI
                                                                                  3.14
>> bool(a = b), bool(a <> b), bool(float(a) <= b) or not bool(float(a) > b)

                                                                  FALSE, TRUE, FALSE
```

5.2.3 Séquences

Une séquence (suite) est simplement une suite d'expressions séparées par des virgules.

```
Exemple de séquences

>> sequence := 1, a, sin(x), f(x)

                              1, a, sin(x), f(x)
>> domtype(sequence)

                                   DOM_EXPR
```

La séquence vide est créée par la function `null()` et il est possible de récupérer un élément d'une séquence en précisant sa place dans l'énumération. La concaténation de séquences est réalisée à l'aide de l'opérateur , (virgule).

```
Manipulation de séquences

>> sequence1 := a ,  b, f(x), PI, sin(x)

                            a, b, f(x), PI, sin(x)
>> sequence2 := 1, 2, null(), 8

                                     1, 2, 8
>> sequence := sequence1, sequence2

                        a, b, f(x), PI, sin(x), 1, 2, 8
>> sequence[4]

                                      PI
```

L'opérateur de génération de séquence $ permet la création de grandes séquences rapidement. Ainsi si m et n sont des entiers `objet(i) $ i = m..n` génère la séquence :
`objet(m), objet(m + 1), ...., objet(n).`

```
Génération de séquences

>> x$6

                                x, x, x, x, x, x
>> suite := sin(k*PI/5) $k=1..5

 1/2       1/2 1/2   1/2   1/2     1/2   1/2   1/2     1/2
2    (5 - 5   )     2    (5    + 5)     2    (5    + 5)
------------------, ------------------, ------------------,
         4                   4                   4

    1/2       1/2 1/2
   2    (5 - 5   )
   ------------------, 0
            4
>> map(suite, float)

        0.5877852523, 0.9510565163, 0.9510565163, 0.5877852523, 0.0
```

5.2.4 Les listes

Une liste est une suite ordonnée d'objets MuPAD quelconques séparés par des virgules et le tout placé entre crochets.

```
Exemple de liste

>> ex := [ceci, est, une, liste, 4, sin(x)^2, [sous, liste]]

                                                                  2
                                    [ceci, est, une, liste, 4, sin(x) , [sous, liste]]
```

La liste vide est représentée par [] et `nops()` permet de connaître le nombre d'opérandes. Le $i^{ème}$ élément de la liste peut être obtenu par la fonction `op()` ou avec l'opération d'indexation [*index*]. Une séquence entre crochets est donc une liste.

```
Manipulation d'une liste

>> nops(ex)

                                                                    7
>> op(ex,4)

                                                                 liste
>> ex[4]

                                                                 liste
```

Il existe plusieurs fonctions pour manipuler les listes (cf. tab. 4).

Fonction	Signification
$\_concat()$ ou .	Concaténation de listes
$append()$	Ajout d'éléments
$contains(liste, x)$	La $liste$ contient-elle des éléments x ?
$liste[i]$	Accès au $i^{ème}$ élément
$map(objet, f)$	Application d'une fonction f à $objet$
$nops()$	Nombre d'éléments de la liste
$op()$	Accès aux éléments
$select()$	Sélection d'éléments suivant une propriété
$sort()$	Tri de la liste
$split()$	Coupe la liste en fonction d'une propriété
$subsop()$	Remplace des éléments de la liste
$delete()$	Supprime des éléments
$zip()$	Combine deux listes

TAB. 4 – Fonctions sur les listes

```
Exemples sur une liste de différentes fonctions

>> a := [0, PI/2, PI, 3*PI/2]: map(a, sin)

                                                [0, 1, 0, -1]
>> select([i $ i = 1..20], isprime)

                                          [2, 3, 5, 7, 11, 13, 17, 19]
>>>> split([i $ i = 1..20], isprime)

              [[2, 3, 5, 7, 11, 13, 17, 19], [1, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20], []]
```

```
Exemple développé

>> l := [i $i=-7..7]

                                          [-7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]
>> //On cherche à enlever les éléments positifs
>> negatif := x -> is(x < 0)

                                                                  x -> is(x < 0)
>> select(l, negatif)

                                                        [-7, -6, -5, -4, -3, -2, -1]
>> >> // Ajout d'un élément à une liste
>> l1 := [i $i=0..9]

                                                     [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> l1 := [op(l1), dix]

                                                   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, dix]
>> l1 := l1.[onze]

                                                 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, dix, onze]
```

5.2.5 Les ensembles

Un ensemble est une suite d'objets MuPAD séparés par des virgules et encadrés par des accolades. Dans un ensemble, il n'y a pas d'ordre et de répétition.

```
Ensemble

>> ensemble := {a, b, c, a}

                                                                                {a, b, c}
>> op(ensemble)

                                                                                 c, b, a
```

L'ensemble vide est représenté par $\{\}$. Il existe différentes opérations possibles sur les ensembles (cf. tab. 5).

Opérateur	Fonction	Signification
	$contains(E, x)$	L'ensemble E contient-il l'élément x ?
$intersect$	$\_intersect()$	Intersection
$union$	$\_union()$	Union
$minus$	$\_minus()$	Différence
	$nops()$	Cardinal
	$map()$	Application d'une fonction aux éléments
$[indice]$		Accès à l'élément d'indice $indice$
	$select()$	Sélection en fonction de propriétés
	$split()$	Découpe l'ensemble en fonction de propriétés
	$combinat :: subsets :: list()$	Sous-ensembles de n éléments d'un ensemble
	$combinat :: cartesianProduct :: list()$	Produit cartésien

TAB. 5 – Opérations sur les ensembles

```
Exemple d'opérations sur les ensembles

>> {x, 1, 5} intersect {x, 1, 3, 4}, {x, 1, 5} union {x, 1, 3, 4}, {x, 1, 5} minus {x, 1, 3, 4}

                  {x, 1}, {x, 1, 3, 4, 5}, {5}
>> x := 8;
  {x, 1, 5} intersect {x, 1, 3, 4}, {x, 1, 5} union {x, 1, 3, 4}, {x, 1, 5} minus {x, 1, 3, 4}

                                8

                   {1, 8}, {1, 3, 4, 5, 8}, {5}
>> L := [{a, b}, {1, 2, a, c}, {3, a, b}, {a, c}]: _intersect(op(L))

                                a
>> map({1, 2, 3}, float@sin)

            {0.1411200081, 0.8414709848, 0.9092974268}
>> combinat::subsets::list({a,b,c,d,e},3)

        [{c, d, e}, {b, d, e}, {a, d, e}, {b, c, e}, {a, c, e},
         {a, b, e}, {b, c, d}, {a, c, d}, {a, b, d}, {a, b, c}]

>> combinat::cartesianProduct::list({Pique, Carreau, Coeur, Trefle},{7,8,9,10})

[[Trefle, 10], [Trefle, 9], [Trefle, 8], [Trefle, 7],
[Coeur, 10], [Coeur, 9], [Coeur, 8], [Coeur, 7],
[Carreau, 10],[Carreau, 9], [Carreau, 8], [Carreau, 7],
[Pique, 10], [Pique, 9], [Pique, 8], [Pique, 7]]
```

5.2.6 Tableaux et tables

Un tableau de taille $m_1 \times m_2 \times \ldots \times m_p$ est une structure qui contient $m_1 m_2 m_3 \ldots m_p$ entrées, numérotées par le p-uplet $(i_1, i_2, \ldots, i_p)$. Un tableau est donc une structure indexée, de plusieurs dimensions, dont les indices sont des entiers appartenant à un intervalle $a..b$. Un tableau sert à stocker des données et sa

taille est fixe. La fonction `array()` permet de créer un tableau. Un tableau peut donc être de dimension 1, ou 2 ou même n avec $n > 2$. On accède à un élément donné d'un tableau à l'aide de ses indices, lorsque le tableau est de dimension 1 il y a un unique indice pour repérer un élément, de dimension 2, il y a 2 indices, de dimension n il y a n indices.

```
Manipulation d'un tableau


 >> tab := array(5..7) /* Tableau indicé à partir de 5 jusqu'à 7*/

                                                                  +-                -+
                                                                  | ?[5], ?[6], ?[7] |
                                                                  +-                -+
>> tab[2] /* L'indice n'existe pas ! */
Error: Illegal argument [array]
> tab[5] := 1: tab[6] := "milieu" : tab

                                                                  +-                  -+
                                                                  | 1, "milieu", ?[7] |
                                                                  +-                  -+
```

Il est possible d'initialiser partiellement ou totalement un tableau lors de sa création.

```
Initialisation d'un tableau


>> tab2 := array(1..5, 1 = "initialisation", 2= "incomplete")

                                              +-                                                  -+
                                              | "initialisation", "incomplete", ?[3], ?[4], ?[5] |
                                              +-                                                  -+
>> tab3 := array(1..5, i = sqrt(i) $i=1..4)

                                                          +-    1/2   1/2         -+
                                                          | 1, 2   , 3   , 2, ?[5] |
                                                          +-                      -+
>> tab4 := array(1..5, ["ini","tia","lis", "ation", "complete"])

                                                 +-                                        -+
                                                 | "ini", "tia", "lis", "ation", "complete" |
                                                 +-                                        -+
```

```
Exemple d'un tableau de dimension 3

>> A := array(1..8, 1..8, 1..8,
           (1, 1, 1) = 111,
           (8, 8, 8) = 888)

                                                array(1..8, 1..8, 1..8,
                                                  (1, 1, 1) = 111,
                                                  (8, 8, 8) = 888
                                                )

>> A[1,2,3]

                                                      A[1, 2, 3]
>> A[8,8,8]

                                                          888
```

Une table est une structure indexée dont les indices (ou index) peuvent être de n'importe quel type. La fonction `table()` permet de créer une table.

```
Utilisation d'une table

>> newton := table(objet = "pomme", acceleration = "9.81 m/s^2", hauteur ="Dans l'arbre",
                   poids=0.12)

                                          table(
                                            poids = 0.12,
                                            hauteur = "Dans l'arbre",
                                            acceleration = "9.81 m/s^2",
                                            objet = "pomme"
                                          )
>> newton[objet], newton[poids]

                                                  "pomme", 0.12
```

5.2.7 Les chaînes de caractères

Une chaîne de caractères est une suite de caractères encadrés par des guillemets. `"Ceci est un exemple"`. La concaténation de deux chaînes de caractères se fait à l'aide de l'opérateur . et on peut également utiliser la fonction `_concat()`. L'opérateur d'indexation $[i]$ extrait le caractère d'indice i de la chaîne. Le premier indice est 0. Pour afficher une chaîne de caractères on peut utiliser la fonction `print()`.

```
Manipulation de chaînes

>> endroit:="esope reste ici et se repose"

                                          "esope reste ici et se repose"
>> envers :="esoper es te ici etser epose"

                                          "esoper es te ici etser epose"
>> stringlib::subs(endroit, " " = "")

                                             "esoperesteicietserepose"
>> stringlib::subs(envers, " " = "")

                                             "esoperesteicietserepose"
>> endroit.envers

                              "esope reste ici et se reposeesoper es te ici etser epose"
>> endroit[0]:="E"

                                                          "E"
>> endroit

                                          "Esope reste ici et se repose"
>> print(endroit)

                                          "Esope reste ici et se repose"
```

5.3 Programmation

MuPAD fournit les éléments indispensables d'un langage de programmation. Il y a, à la fois les instructions conditionnelles, les boucles et les procédures. L'aspect programmation que nous considérons ici, est celui qui consiste à définir des procédures en utilisant les instructions et les différentes fonctions prédéfinies ou définies par l'utilisateur.

5.3.1 Les procédures

Les procédures comme les fonctions MuPAD retournent des valeurs. Une procédure est appelée comme une fonction.

$$nom\_de\_la\_procedure(arguments)$$

Les procédures sont généralement définies dans un fichier que l'on charge à l'aide de la fonction `read()` par exemple. Une procédure est composée de :

- Une en-tête qui correspond à l'identificateur (le nom) de la procédure, sa déclaration à l'aide de `proc` et la liste de ses paramètres ;
- La déclaration des variables appartenant uniquement à la procédure (variables locales) ;
- Le corps de la procédure qui contient les instructions qui constituent la procédure. Ce corps est délimité par `begin` et `end_proc`.

```
Exemple de procédure


>> ExempleDeProcedure := proc(a, b) // En tête de la procédure
                                                  // a et b sont les paramètres de la procédure
   local variable1, variable2;         // Variables locales
   begin                               // Début du corps
     if a < b                             // Instruction
       then return(b)
       else return(a)
     end_if
   end_proc:                           // Fin du corps
```

Les variables peuvent être locales ou globales, pour qu'une variable soit locale elle doit avoir été déclarée dans la procédure précédée du mot clef `local`. Sinon toute variable non déclarée locale ou non déclarée est considérée comme globale.

Les instructions entre `begin` et `end_proc` peuvent être des commandes MuPAD quelconques, autrement dit cela peut être des fonctions MuPAD, des instructions de programmation, ou des appels de procédures. La commande `return` termine la procédure et transmet la valeur de son argument comme valeur de sortie qui correspond au résultat.

Une procédure peut retourner un objet MuPAD quelconque (expression, séquence, liste, ensemble, fonction, ou même une autre procédure).

```
Procédure qui retourne une fonction


>> creeFonctionPuissance := proc(puissance)
   option escape;
   begin
     x -> (x^puissance)
   end_proc:
>> f := creeFonctionPuissance(7)

                                                                        x -> x^puissance
>> f(a)

                                                                              7
                                                                             a
>> f(2)

                                                                            128
```

Lorsque la procédure ou la fonction retournée utilise une variable locale ou un paramère d'entrée on doit utiliser l'option `escape`.

5.3.2 Les instructions

Il existe deux catégories d'instructions :

– Les instructions conditionnelles ;
– Les boucles itératives et les boucles conditionnelles.

Les instructions conditionnelles
Il existe en MuPAD deux instructions conditionnelles : `if` et `case`. La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
if condition
   then
      instruction1;
      instruction2;
      ....
end_if
suiteinstruction;
```

Lorsque `condition` est vraie (`TRUE`), la suite d'instructions "`instruction1`, `instruction2`, ... " est exécutée, puis ensuite `suiteinstruction`. Si la `condition` est fausse on passe à `suiteinstruction`. `condition` est une expression booléenne.

```
if condition
   then
      instruction_alors1;
      instruction_alors2;
      ....
   else
      instruction_sinon1;
      instruction_sinon2;
      ....
end_if
suiteinstruction;
```

Lorsque `condition` est vraie (`TRUE`), la suite d'instructions "`instruction_alors1`, `instruction_alors2`, ... " est exécutée, puis ensuite `suiteinstruction`. Si la `condition` est fausse la suite d'instructions "`instruction_sinon1`, `instruction_sinon2`, ... " est exécutée puis ensuite `suiteinstruction`

Exemple de if

```
>> p := proc(i)
   begin
     if isprime(i)
       then print(Unquoted, NoNL, expr2text(i)." est ")
       else print(Unquoted, NoNL, expr2text(i)." n'est pas ")
     end_if:
     print(Unquoted, NoNL,"premier.");
   end_proc:
                                                        proc p(i) ... end
>> p(6)
6 n'est pas premier
```

Il existe une autre instruction conditionnelle qui permet des choix multiples. C'est l'instruction `case`.

```
case objet_MuPAD
   of valeur1 do instructions_1
   ....
   of valeurn do instructions_n
 end_case
```

ou son autre forme :

```
case objet_MuPAD
   of valeur1 do instructions_1
   ....
   of valeurn do instructions_n
   otherwise intructions_autrement
 end_case
```

Si la valeur de `objet_MuPAD` est égale à l'une des valeurs `valeur1, valeur2, ..., valeurn` **toutes** les instructions qui suivent sont exécutées jusqu'à rencontrer une instruction `break` ou `return`.

```
Utilisation du case


>> x := 2:
case x
  of 1 do print(1)
  of 2 do print(4)
  of 3 do print(9)
  otherwise print("autrement")
end_case:

                                                                        4

                                                                        9

                                                                  "autrement"
>> case x
  of 1 do print(1); break
  of 2 do print(4); break
  of 3 do print(9); break
  otherwise print("autrement")
end_case:

                                                                        4
```

Si aucune branche `of` ne correspond c'est la branche `otherwise` qui est exécutée.

```
Utilisation du case et otherwise


>> questCequeCest := proc(x)
   begin
     case domtype(x)
       of DOM_INT do
       of DOM_RAT do
       of DOM_FLOAT do print(x,"est un nombre réel"); break
       of DOM_COMPLEX do print(x, "est un nombre complexe"); break
       otherwise print(x," est ni un réel, ni un complexe");
     end_case
   end_proc:
>> questCequeCest(-6), questCequeCest(3/7), questCequeCest(sqrt(-1)),
   questCequeCest(questCequeCest)

                                              -6, "est un nombre réel"

                                              3/7, "est un nombre réel"

                                             I, "est un nombre complexe"

                           proc questCequeCest(x) ... end, " est ni un réel, ni un complexe"
```

Les boucles

Il existe deux sortes de boucles, les boucles itératives (`for`) et les boucles conditionnelles (`repeat` et `while`). Les boucles itératives sont effectuées un nombre de fois fixé à l'avance alors que les boucles conditionnelles sont effectuées en fonction d'une condition satisfaite ou non.

Commençons par décrire la boucle `for`. La syntaxe est la suivante :

```
for variableDeBoucle from valeurInitiale to valeurFinale do
   instructions_1                    /* Corps de la boucle */
   ....
   instructions_n
 end_for
```

`variableDeBoucle` est donc une variable (généralement locale) qui sert à contrôler la boucle. La valeur initiale est fixée par `valeurInitiale` et la valeur finale par `valeurFinale`. La variable de boucle `variableDeBoucle` va prendre successivement de façon croissante et de 1 en 1 les valeurs `valeurInitiale` jusqu'à `valeurFinale`.

Boucle for croissante

```
>> for i from 1 to 4 do
     x := i^2;
     print(Unquoted,expr2text(i)."^2=".expr2text(x))
   end_for:
```

```
1^2=1

2^2=4

3^2=9

4^2=16
```

Il est possible de spécifier le pas de variation (`pas`) de la boucle à l'aide de `step`.

```
for variableDeBoucle from valeurInitiale to valeurFinale step pas do
   instructions_1                    /* Corps de la boucle */
   ....
   instructions_n
 end_for
```

Boucle for croissante avec pas

```
>> for i from 1 to 4 step 2 do
     x := i^2;
     print(Unquoted,expr2text(i)."^2=".expr2text(x))
   end_for:
```

```
1^2=1

3^2=9
```

On pourra remarquer que la boucle s'arrête dès que `variableDeBoucle > valeurFinale`.
Les boucles `for` que nous avons présentées sont croissantes, elles peuvent être décroissantes.

```
for variableDeBoucle from valeurInitiale downto valeurFinale do
   instructions_1                    /* Corps de la boucle */
   ....
   instructions_n
 end_for
```

Boucle for décroissante

```
>> for i from 4 downto 1 do
     x := i^2;
     print(Unquoted,expr2text(i)."^2=".expr2text(x))
   end_for:
```

```
4^2=16

3^2=9

2^2=4

1^2=1
```

```
for variableDeBoucle from valeurInitiale downto valeurFinale step pas do
   instructions_1                  /* Corps de la boucle */
   ....
   instructions_n
 end_for
```

Boucle for décroissante avec pas

```
>> for i from 4 downto 1 step 2 do
     x := i^2;
     print(Unquoted,expr2text(i)."^2=".expr2text(x))
   end_for:
```

```
4^2=16

2^2=4
```

La variable de boucle peut également prendre les valeurs des objets contenus dans un objet structuré.

```
for variableDeBoucle in objetStructure do
   instructions_1                  /* Corps de la boucle */
   ....
   instructions_n
 end_for
```

```
Boucle for et objet structuré

>> liste :=[i $i=1..5, a*y^2]

                                                                    2
                                                  [1, 2, 3, 4, 5, a y ]
>> for i in liste do
    print(i)
   end_for:

                                                                  1

                                                                  2

                                                                  3

                                                                  4

                                                                  5

                                                                   2
                                                                a y
```

Les boucles `while` et `repeat` sont des boucles conditionnelles, la première est effectuée tant que la condition est vraie alors que la seconde jusqu'à ce que la condition devienne vraie. La syntaxe de la boucle `while` est la suivante :

```
while condition  do
   instruction 1          /* Corps de la boucle */
   ....
   instruction_n
end_while
```

Le corps de la boucle est effectué tant que `condition` est vraie.

```
Utilisation de while - algorithme d'Euclide -

>> n := 12: m:= 18

                                                                    18
>> b := n: r := m

                                                                    18

 >> while r <> 0 do
     a := b;
     b := r;
     r := a mod b;
   end_while:
   print(Unquoted,"PGCD(".expr2text(n,m).") = ".expr2text(b));

                                                          PGCD(12, 18) = 6
```

La syntaxe de la boucle `repeat` est la suivante :

```
repeat
   instruction 1        /* Corps de la boucle */
   ....
   instruction_n
 until condition end_repeat
```

```
Utilisation de repeat - algorithme d'Euclide -

>> n := 12: m:= 18

                                                                          18
>> b := n : r := m

                                                                          18
>> repeat
     a := b;
     b := r;
     r := a mod b;
   until r=0 end_repeat:
   print(Unquoted,"PGCD(".expr2text(n,m).") = ".expr2text(b));

                                                          PGCD(12, 18) = 6
```

Le corps de la boucle est effectué tant que la condition est fausse.

5.4 Exercices

X *Exercice n°10*

X.1 Utiliser une commande simple pour créer la somme double :

$$\sum_{i=1}^{10}\sum_{j=1}^{i}\frac{1}{i+j}$$

```
Solution

>> _plus((1/(i + j) $j= 1..i) $i= 1..10)

                          1464232069/232792560
>>
```

XI *Exercice n°11*

XI.1 Créez deux listes avec des éléments $un, deux, trois, quatre$ et $1, 2, 3, 4$. Concaténer les listes. Multiplier les listes deux à deux.

Solution

```
>> l1 := [un, deux, trois, quatre]: l2 := [1, 2, 3, 4]:
>> l1.l2, zip(l1, l2, _mult)

                          [un, deux, trois, quatre, 1, 2, 3, 4], [un, 2 deux, 3 trois, 4 quatre]
```

XII *Exercice n°12*

Soit $X = [x_1, \ldots, x_n]$ et $Y = [y_1, \ldots, y_n]$ deux listes de même longueur avec n fixé. Déterminer :

XII.1 Leur produit interne

$$x_1 y_1 + \ldots + x_n y_n$$

XII.2 Leur produit matriciel

$$[[x_1 y_1, x_1 y_2, \ldots, x_1 y_n], [x_2 y_1, x_2 y_2, \ldots, x_2 y_n] \ldots [x_n y_1, x_n y_2, \ldots, x_n y_n]$$

Solution

```
>> X := [x.i $i=1..7]: Y := [y.i $i=1..7]

                                        [y1, y2, y3, y4, y5, y6, y7]
>>  _plus(op(zip(X, Y, _mult)))

                                    x1 y1 + x2 y2 + x3 y3 + x4 y4 + x5 y5 + x6 y6 + x7 y7
>> f := x -> (map(Y, _mult, x)): map(X, f)

[[x1 y1, x1 y2, x1 y3, x1 y4, x1 y5, x1 y6, x1 y7],
 [x2 y1, x2 y2, x2 y3, x2 y4, x2 y5, x2 y6, x2 y7],
 [y1 x3, x3 y2, x3 y3, x3 y4, x3 y5, x3 y6, x3 y7],
 [y1 x4, y2 x4, x4 y3, x4 y4, x4 y5, x4 y6, x4 y7],
 [y1 x5, y2 x5, y3 x5, x5 y4, x5 y5, x5 y6, x5 y7],
 [y1 x6, y2 x6, y3 x6, y4 x6, x6 y5, x6 y6, x6 y7],
 [y1 x7, y2 x7, y3 x7, y4 x7, y5 x7, x7 y6, x7 y7]]
>>
```

XIII *Exercice n°13*

Liste et produit :

XIII.1 Calculer le produit des $x^{1/i}$ pour i variant de -5 à 7 avec $i \neq 0$

```
Solution


l := [x^(1/i) $i=-5..-1].[x^(1/i) $i=1..7]

  --   1     1     1     1   1      1/2   1/3   1/4   1/5   1/6   1/7 --
  |  ----, ----, ----, ----, -, x, x   , x   , x   , x   , x   , x     |
  |   1/5   1/4   1/3   1/2  x                                         |
  -- x     x     x     x                                             --
>> _mult(op(l))

                                   13/42
                                  x
```

XIV *Exercice n˚14*

Listes et ensembles :

XIV.1 Construisez une liste ordonnée des 100 premiers nombres premiers.

XIV.2 Diviser tous les éléments de la liste par 13 et déterminer la partie décimale à l'aide de `frac()`.

XIV.3 Multiplier tous les éléments par 13 et construisez une nouvelle liste qui sera le résultat ordonné de la liste.

XIV.4 Convertissez la liste en un ensemble et donnez le cardinal de l'ensemble.

Solution

```
>> liste := [ithprime(i) $i=1..100]

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,

   71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139,

   149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,

   227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293,

   307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383,

   389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463,

   467, 479, 487, 491, 499, 503, 509, 521, 523, 541]
>> map(liste, frac@_divide, 13)

[2/13, 3/13, 5/13, 7/13, 11/13, 0, 4/13, 6/13, 10/13, 3/13, 5/13, 11/13,

   2/13, 4/13, 8/13, 1/13, 7/13, 9/13, 2/13, 6/13, 8/13, 1/13, 5/13,

   11/13, 6/13, 10/13, 12/13, 3/13, 5/13, 9/13, 10/13, 1/13, 7/13, 9/13,

   6/13, 8/13, 1/13, 7/13, 11/13, 4/13, 10/13, 12/13, 9/13, 11/13, 2/13,

   4/13, 3/13, 2/13, 6/13, 8/13, 12/13, 5/13, 7/13, 4/13, 10/13, 3/13,

   9/13, 11/13, 4/13, 8/13, 10/13, 7/13, 8/13, 12/13, 1/13, 5/13, 6/13,

   12/13, 9/13, 11/13, 2/13, 8/13, 3/13, 9/13, 2/13, 6/13, 12/13, 7/13,

   11/13, 6/13, 3/13, 5/13, 2/13, 4/13, 10/13, 1/13, 7/13, 2/13, 6/13,

   8/13, 12/13, 11/13, 6/13, 10/13, 5/13, 9/13, 2/13, 1/13, 3/13, 8/13]
>> sort(map(%,_mult,13))

[0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3,

   3, 3, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6,

   6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9,

   9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11,

   11, 11, 12, 12, 12, 12, 12, 12, 12]
>> {op(%)}

                {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
>> nops(%)

                                    13
```

XV *Exercice n°15*

Listes et ensembles :

XV.1 Créer une liste `N2` constituée des carrés des nombres entiers compris entre 1000 et 10000, une liste `N3` contituée des cubes, une liste `N4` constituée des puissances $4^{ième}$ et une liste `N5`

constituée des puissances $5^{ième}$.

XV.2 Déterminer les élements en commun et leur nombre entre les listes prises deux à deux.

```
Solution


>> N2 := [i^2 $i = 1000..10000] : N3 := [i^3 $i = 1000..10000]:
   N4 := [i^4 $i = 1000..10000] : N5 := [i^5 $i = 1000..10000]:
>> //etc
>> N3etN4 := {op(N3)} intersect {op(N4)} :

                                {1000000000000}
>> nops(N3etN4)

                                       1
>> // etc
```

XVI *Exercice n˚16*

Nombres premiers racines d'un polynôme :

XVI.1 Mettre dans un ensemble les valeurs du polynôme $x^2 + x + 41$, $x \in \mathbb{N}$, et $-21 < x < 21$.

XVI.2 Combien y a t-il d'éléments dans l'ensemble ? Pourquoi ?

XVI.3 Sélectionner en utilisant une boucle les éléments de l'ensemble qui sont des nombres premiers. Que constatez-vous ?

XVI.4 Ne pouvait-on pas écrire plus simple ?

```
Solution


>> polynome := x^2 + x + 41

             2
            x + x  + 41

>> ensemble := {subs(polynome, x = i) $i = -20..20}

{41, 43, 47, 53, 61, 71, 83, 97, 113, 131, 151, 173, 197, 223, 251, 281,
 313, 347, 383, 421, 461}

 >> nops(ensemble)

  21

 >> premier  := {}:
   for p in ensemble do
     if isprime(p)
       then premier := {p} union premier
     end_if
   end_for:
   premier;
   if premier = ensemble
     then print("Ils sont identiques !!")
     else print("Ils sont différents")
   end_if;

{41, 43, 47, 53, 61, 71, 83, 97, 113, 131, 151, 173, 197, 223, 251, 281,
 313, 347, 383, 421, 461}

                             "Ils sont identiques !!"

>> if ensemble = select(ensemble, isprime)
     then print("Ils sont identiques !!")
   end_if

                             "Ils sont identiques !!"
```

XVII *Exercice n°17*

Nombres de Fermat :

XVII.1 Ecrire une procédure qui calcule les premiers nombres de Fermat :

$$2^{(2^k)} + 1, k \in \mathbb{N}$$

en s'arrêtant dès que le nombre n'est plus premier.

```
1  fermat := proc()
2  local k, premier, n;
3  begin
4    k := 0;
5    premier := TRUE;
6    while premier do
7      n := 2^(2^k)+1;
8      if isprime(n) = TRUE
```

```
      then print(n, "est premier")
      else
        print(n, "n'est pas premier", ifactor(n));
        premier := FALSE;
    end_if;
    k := k + 1;
  end_while;
  return(n);
end_proc:
```

XVIII *Exercice n°18*

Il y a 4000 ans, les Sumériens connaissaient déjà un algorithme qui permettait de calculer $\sqrt{a}$ à partir de a.

$$\text{Pour } x_0 > 0,\ x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n}) \text{ et } \lim_{n\to\infty} x_n = \sqrt{a}$$

XVIII.1 Soit $f(x) = \frac{x+a/x}{2}$, résoudre l'équation $f(x) = x$.

XVIII.2 Ecrire une procédure `racineCarree` qui prend en entrée a et n et calcule $\sqrt{a}$.

```
>> f := x -> (x + a/x)/2;

                                    x -> (x + a/x)/2
>> solve(x=f(x),x)

                                                 1/2      1/2
                    piecewise({} if a = 0, {a    , - a    } if a <> 0)

racineCarree := proc(a, n)
local x, i;
begin
  x := (1 + a)/ 2;
  for i from 1 to n do
    x := (x + a/x)/2;
  end_for;
  return(x);
end_proc:
```

XIX *Exercice n°19*

Calcul de Π par la méthode de Monte-Carlo.

XIX.1 n gouttes de pluie tombent sur un carré de coté égal à 1 et on compte le nombre r de gouttes qui tombent à l'intérieur du quart de cercle inscrit dans le carré. Alors

$$\frac{r}{n} \simeq \frac{\Pi}{4}$$

En utilisant `frandom()` écrire une procédure qui connaissant n approche la valeur de Π.

```
calculPi := proc(n)
local r, i, goutte;
begin
```

```
  r := 0;
  for i from 1 to n do
    goutte := [frandom(), frandom()];
    if goutte[1]^2 + goutte[2]^2 < 1
      then r := r + 1;
    end_if;
  end_for;
  return(float(4*r/n));
end_proc:
```

XX *Exercice n°20*

Nombres premiers jumeaux

XX.1 Deux nombres premiers p et q sont jumeaux si $q = p + 2$. Ecrire une procédure qui détermine tous les entiers naturels premiers jumeaux dans un intervalle donné $[a, b]$.

XX.2 Que peut-on dire de la répartition des nombres premiers jumeaux ? On s'aidera d'un graphe.

```
jumeaux := proc(a, b)
local i, twin;
begin
  twin := [];
  for i from a to b do
    if isprime(i) and isprime(i+2)
      then twin := twin.[[i, i +2]]; // append(twin,[i, i +2])
    end_if;
  end_for;
  return(twin);
end_proc:

f := x -> nops(jumeaux(1,x));
plotfunc2d(f,x=1..1000);
```

XXI *Exercice n°21*

Fonction de tracé graphique pour étudier les suites récurrentes.

XXI.1 Ecrire une procédure `escargot()` qui trace les termes de la suite reccurente $u_{n+1} = f(u_n)$. La procédure doit avoir comme argument : la fonction f, le nombre de termes à calculer n et le premier terme de la suite u_0.

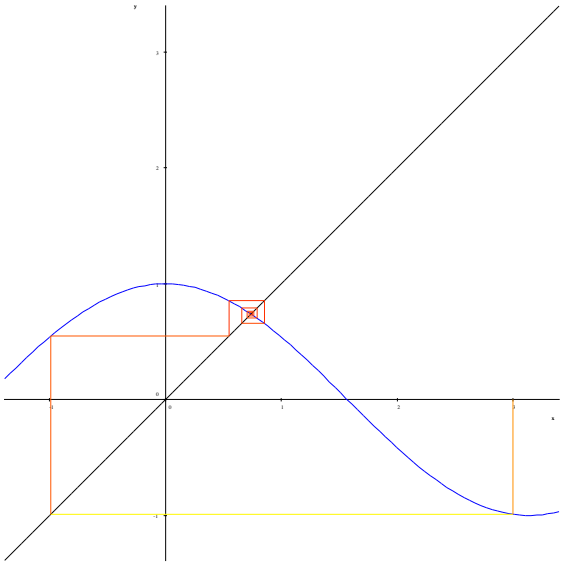
XXI.2 Tester votre procédure pour la fonction `cos(x)`.

XXI.3 Tester votre procédure pour la fonction $g : x \mapsto \sqrt{x+2}$ et $n = 8$.

```
escargot := proc(f, u0, n)
local liste, i, u, v, minimum, maximum, intervallex;
begin
  liste := [[u0, 0]];
  u       := float(u0);
  minimum := u;
  maximum := u;
  for i from 1 to n do
    v      := f(u);
```

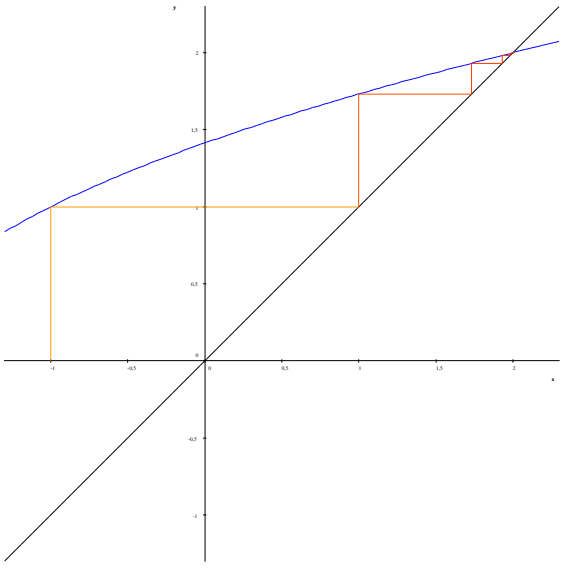
```
    liste := [op(liste), [u, v], [v, v]];
    u     := v;
    minimum := min(minimum, u);
    maximum := max(maximum, u);
  end_for;
  intervallex := minimum - 0.1 * (maximum - minimum)
                 .. maximum + 0.1 * (maximum - minimum);
  plot(plot::Function2d(f(x), x = intervallex, Color = RGB::Blue),
       plot::Function2d(x, x = intervallex, Color = RGB::Black),
       plot::Polygon(op(liste)),Scaling=Constrained);
end_proc:

escargotBorne := proc(f, u0, n, intervallex)
local liste, i, u, v;
begin
  liste := [[u0, 0]];
  u     := float(u0);
  for i from 1 to n do
    v     := f(u);
    liste := [op(liste), [u, v], [v, v]];
    u     := v;
  end_for;
  plot(plot::Function2d(f(x), x = intervallex, Color = RGB::Blue),
       plot::Function2d(x, x = intervallex, Color = RGB::Black),
       plot::Polygon(op(liste)),Scaling=Constrained);
end_proc:
```

```
>> escargot(cos, 3, 10)
```



```
>> g := x -> sqrt(x + 2)
```

```
                                                                    x -> sqrt(x + 2)
>> escargot(g, -1, 8)
```



6 MuPAD emacs LaTeX and co

Emacs peut servir d'environnement d'utilisation de MuPAD, en utilisant le copier/coller ou en lisant un fichier.

```
génération de code Latex

>>  print(Unquoted,generate::TeX(solve( b*x + c = 0, x))
```

$$\begin{cases} \left\{-\frac{c}{b}\right\} & \text{if } b \neq 0 \\ C & \text{if } b = 0 \wedge c = 0 \\ \{\} & \text{if } b = 0 \wedge c \neq 0 \end{cases}$$

7 Récréation

Les L-systems, abréviation de Lindenmayer-systems, du nom du biologiste Aristid Lindenmayer ont été créés en 1968 pour étudier la croissance des plantes. Przemyslaw Prusinkiewicz les a étendus à trois dimensions et a développé un système informatique pour visualiser des L-systèmes. Pour cela, il s'est inspiré du langage LOGO créé dans les années 1960 par Seymour Papert, pour apprendre la programmation aux enfants.

Imaginons une tortue (turtle) qui regarde vers le nord ; pour la faire tourner vers la droite, il suffit de lui donner l'ordre +, pour la faire tourner vers la gauche, l'ordre -, et l'ordre F pour faire un pas. Ainsi, F+F+F+F permet de tracer un rectangle et F-F+F+F-F permet de tracer la courbe 15.

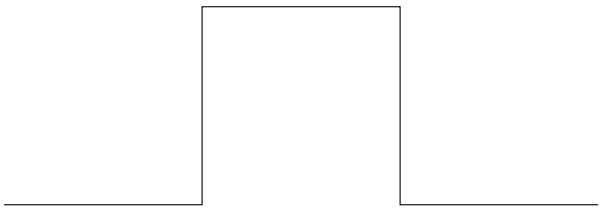


FIG. 15 – Exemple de L-system

Chaque L-système a une règle d'axiome et de production, qui décrit comment la dessiner. Ainsi le flocon de Von Koch peut être décrit de la façon suivante :

$$\begin{cases} \text{Axiome} & F \\ \text{Régle de production} & F \rightarrow F + F - - F + F \\ \text{Angle} & 60^{\circ} \end{cases}$$

Chaque F est donc remplacé par la partie droite de la régle de production. En MuPAD on peut écrire :

```
Flocon de Von Koch


>> for i from 2 to 7 do
     L := plot::Lsys(60, "F", "F"= "F+F--F+F"):
     L::generations := i:
     plot(L, Axes = None);
     input();
   end_for:
```

Pour permettre des représentations proches des plantes, on introduit deux opérateurs [et]. [permet de garder en mémoire la position de la tortue (empile la position) et de commencer une nouvelle branche qui se termine par] (dépile la position au sommet de la pile). Le tracé reprend reprend alors à partir de la position dépilée.

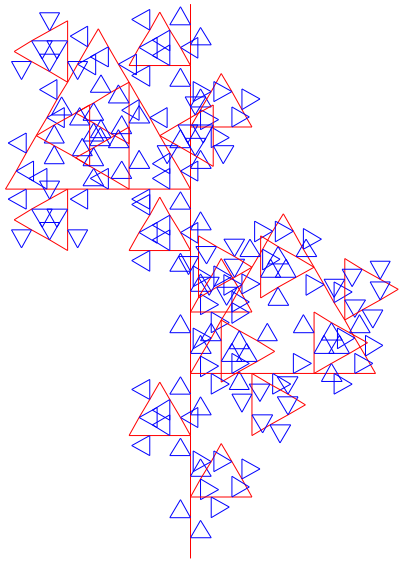
L-system

```
>> L := plot::Lsys(22.5, "X", "F"= "FF", "X"="F-[[X]+X]+F[+FX]-X"):
   L::generations := 7:
   plot(L, Axes = None)
```

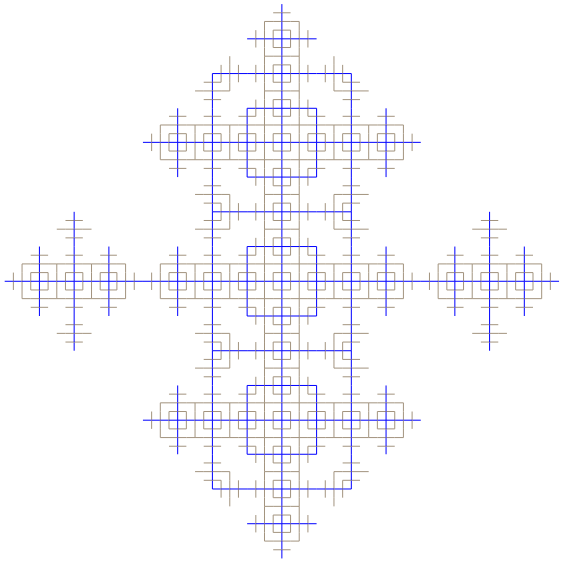
On l'aura constaté on utilise `plot::Lsys()`. Le premier argument correspond à l'angle de développement, le deuxième à l'axiome de départ et les suivants aux règles de production.

L-system

```
>> ornament4 := plot::Lsys(30, "F",
                             "F"="AF[---BF++++BF++++BF]AF[+++BF----BF----BF]AF",
                             "A"=RGB::Red,
                             "B"=RGB::Blue):
   ornament4::generations := 4:
   plot(ornament4, Axes = None, Scaling=Constrained)
```



```
L-system


>> ornament5 := plot::Lsys(90, "F",
                              "F"="BF[-AF][+AF]BF[+AFF][-AFF]BF[-AF][+AF]BF",
                              "A"=RGB::Beige,
                              "B"=RGB::Blue):
   ornament5::generations := 4:
   plot(ornament5, Axes = None, Scaling=Constrained)
```



Références

[1] Sylvain Damour. Page maple de sylvain damour. http ://www.cmi.univ-mrs.fr/ damour/maple/, 2001.

[2] Miguel de Guzmán. *Aventures mathématiques*. Presses polytechniques et universitaires romandes, 1990.

[3] J. Gerhard, W. Oevel, F Postel, and S Wehmeir. *Introduction à MuPAD*. Springer, 2000. Une version est disponible à partir de l'aide MuPAD sous MuPAD Tutorial.

[4] The MuPAD Group. *MuPAD User's Manual*. Wiley-Teubner.

[5] W. Oevel and J Gerhard. *Quick Reference - MuPAD The Open Computer Algebra System*. Disponible dans le menu aide de MuPAD.