

# **FAIRE DES MATHÉMATIQUES AU LYCÉE EN PROGRAMMANT**

***Un catalogue libre de 400 programmes  
avec XCAS, CAML, SAGE et PYTHON***

**Licence Creative Commons** 

**Guillaume Connan - IREM de Nantes**

**Courriel : [gconnan - at - free.fr](mailto:gconnan-at-free.fr)**

**VERSION PROVISOIRE DU 16 décembre 2010**

<b>1</b>	<b>Présentation du stage PAF de janvier/mars 2010</b>	<b>2</b>
<b>2</b>	<b>Algorithme breton</b>	<b>4</b>
<b>3</b>	<b>XCAS et CAML</b>	<b>5</b>
A	CAML pour l' impatient . . . . .	5
B	XCAS pour l' impatient . . . . .	10
<b>4</b>	<b>Dis monsieur, dessine-moi un icosagone..</b>	<b>14</b>
A	Algorithme . . . . .	14
B	Traduction XCAS . . . . .	14
<b>5</b>	<b>Algorithmes récursifs / algorithmes impératifs</b>	<b>15</b>
A	Programmation impérative et affectations . . . . .	15
B	Deux manières d' aborder un même problème . . . . .	21
C	Algorithme récursif ne signifie pas suite récurrente . . . . .	33
D	Alors : avec ou sans affectation ? . . . . .	34
<b>6</b>	<b>Outils mathématiques basiques</b>	<b>36</b>
A	Valeur absolue . . . . .	36
B	Partie entière . . . . .	37
C	Arrondis . . . . .	39
D	Variante . . . . .	40
E	Calculs de sommes . . . . .	40
F	Mini et maxi . . . . .	42
<b>7</b>	<b>Algorithmes de calculs « à la STG »</b>	<b>45</b>
A	Taux de remise variable . . . . .	45
<b>8</b>	<b>Algorithmes en arithmétique</b>	<b>47</b>
A	Division euclidienne . . . . .	47
B	Nombre de diviseurs d' un entier . . . . .	51
C	Nombres parfaits . . . . .	54
D	PGCD . . . . .	58
E	Algorithme d'Euclide étendu . . . . .	60
F	Fractions continues dans $Q$ . . . . .	63
G	Fractions continues dans $R$ . . . . .	65
H	Tests de primalité . . . . .	67
I	Crible d'Ératosthène . . . . .	71
J	Décomposition en facteurs premiers . . . . .	73
K	Avec la bibliothèque arithmétique de XCAS . . . . .	77
L	En base 10 . . . . .	77
M	Décomposition en base 2 . . . . .	78
N	Problèmes de calendrier . . . . .	80

<b>9 Construction d'ensembles de nombres</b>	<b>82</b>
A Somme et multiplication d'entiers	82
B Opérations avec des fractions	82
C Fractions continues : le retour	83
<b>10 Algorithmes de calcul plus complexes</b>	<b>85</b>
A Algorithme de HÖRNER	85
B Exponentiation rapide	87
C Calculatrice en légo	91
D Tableaux de signes	93
<b>11 Algorithmes en analyse</b>	<b>97</b>
A Les réels et le processeur	97
B Tracé de représentations graphiques	98
C Fonctions définies par morceaux	101
D Recherche de minimum	103
E Courbe du blanc-manger	105
F Escalier de CANTOR	107
G Dichotomie	112
H Méthode des parties proportionnelles	114
I Méthode de NEWTON	115
J Fractale de Newton	119
K Méthode d'EULER	121
L Intégration numérique	128
<b>12 Résolutions de systèmes</b>	<b>134</b>
A Systèmes de CRAMER	134
B Pivot de GAUSS	135
<b>13 Approximations d'irrationnels</b>	<b>138</b>
A Algorithme de HÉRON	138
B Influence de la première approximation	139
C Calcul de 300 décimales de e	139
D Méthode d'EULER pour approcher la racine carrée d'un entier	140
<b>14 Algorithmes géométriques</b>	<b>143</b>
A Approximations de $\pi$	143
B Approximation de $\pi$ par dénombrement des points de coordonnées entières d'un disque	148
C Flocons fractales	148
D Polygones de SIERPINSKI	150
E Végétation récursive	156
F Ensembles de JULIA et de MANDELBROT	158
G Les vecteurs, le renard et le lapin	163
<b>15 Algorithmes en probabilité</b>	<b>166</b>
A Pile ou face	166
B Tirage de boules	167
C Tirage de boules avec remise	168
D Le Monty Hall	169
E Problème du Duc de Toscane	171
F Lancers d'une pièce et résultats égaux	173
G Le lièvre et la tortue	174
H Générateurs de nombres aléatoires	177
I Calcul de $\pi$ avec une pluie aléatoire	177
J Loi normale	178
<b>16 Statistiques</b>	<b>182</b>
A Lissage par moyennes mobiles	182
B Intervalle de fluctuation en 2 <sup>nde</sup>	185

<b>17 Un peu de logique</b>	<b>191</b>
A Fonction mystérieuse . . . . .	191
<b>18 Problèmes d'optimisation</b>	<b>192</b>
A Algorithme glouton et rendu de monnaie . . . . .	192
<b>19 Manipulation de chaînes de caractères</b>	<b>194</b>
A Qu'est-ce qu'une chaîne de caractères? . . . . .	194
B Quelques exercices basiques sur les chaînes . . . . .	195
<b>20 Algorithmes de tri</b>	<b>197</b>
A Test de croissance . . . . .	197
B Tri à bulles . . . . .	199
C Tri par insertion . . . . .	201
D Tri rapide . . . . .	202
<b>21 Cryptographie</b>	<b>204</b>
A Codage de César . . . . .	204
B RSA . . . . .	205
C Cryptosystème du sac à dos . . . . .	207
<b>22 Graphes</b>	<b>208</b>
<b>23 Suites définies par une relation de récurrence</b>	<b>209</b>
A Suite de Fibonacci . . . . .	209
B Suite de Syracuse . . . . .	211
C Encore des approximations de $\pi$ . . . . .	212
D Suites logistiques . . . . .	214
E Trop de Zrājdzs nuit aux Zrājdzs . . . . .	214
F Exposant de Lyapounov . . . . .	218
<b>24 Quelques particularités de CAML...</b>	<b>220</b>
A Toutes sortes de fonctions . . . . .	220
B Fonctions d'ordre supérieur . . . . .	221
C Dérivée formelle . . . . .	222
<b>25 Exemples de TP d'algorithmique</b>	<b>224</b>
A Fractions et PGCD en 2 <sup>nde</sup> . . . . .	224
B Suite de Fibonacci en 2 <sup>nde</sup> . . . . .	227
C L'approximation de $\pi$ d'Archimède en 2 <sup>nde</sup> . . . . .	232

# 1 - Présentation du stage PAF de janvier/mars 2010

Un aspect qui nous intéresse particulièrement dans la programmation est que la réalisation d'un programme informatique de façon traditionnelle passe par l'écriture de son code source. Il s'agit bien d'une véritable activité de rédaction et cela nous semble important de convaincre les élèves que là encore, l'obtention d'un résultat désiré passe par la phase, peut-être laborieuse, d'écriture.

Il est bien question d'analyser un problème et de décrire un moyen d'obtenir une solution, mais le « robot » – interpréteur ou compilateur – reste imperturbablement extrêmement strict, et n'infère en rien les détails pensés mais non explicités par l'auteur du programme. La rigueur est ici strictement nécessaire à cette activité. Convoquer la rigueur par le biais de la programmation, activité pour l'instant inhabituelle chez les collégiens ou lycéens, ne peut qu'être un bénéfice à plus ou moins court terme pour les autres apprentissages.

Certes, les environnements modernes de développement logiciel (le « Rapid Application Development » dans la terminologie anglo-saxonne) foisonnent de dispositifs d'assistance au programmeur, dans les outils employés mais aussi et surtout depuis quelques années, la programmation modulaire et la conception objet qui permettent de segmenter de gigantesques projets pour une réalisation commune et partagée par des centaines d'individus. Il n'en demeure pas moins que les phases d'écriture perdurent, qui plus est avec des langages de programmation qui sont aujourd'hui au nombre de plus de 2000 en ce début de troisième millénaire, pour une activité qui a réellement pris son essor un peu avant la Seconde guerre mondiale.

La programmation utilise l'algorithmique pour être efficace mais doit fournir en plus des programmes lisibles, facilement utilisables et modifiables par d'autres utilisateurs. Depuis *FORTRAN* en 1956, langage qui laissait l'utilisateur très proche de la machine, les langages évoluent vers un plus « haut niveau », c'est-à-dire qu'ils s'éloignent du langage machine pour se rapprocher du langage humain et gagnent ainsi en simplicité de lecture et d'utilisation.

Par exemple, voici deux programmes calculant la factorielle d'un entier naturel  $n$  :

– en langage C :

```
long Factorielle(int n) {
    unsigned i;
    long resultat = 1;
    if (n==0)
        return 1;
    if (n>0) {
        for (i=2; i<n+1; ++i)
            resultat *=i;
        return resultat;
    }
    return -1;
}
```

– en langage CAML :

```
let rec factorielle = function
| 0 -> 1
| n -> n*factorielle(n-1);;
```

Quel est le langage de plus haut niveau?...

En mathématiques, les langages de haut niveau basés sur un formalisme logique nous intéressent au plus haut point car ils ont cette rigueur qui sied à notre discipline et sont en même temps moins parasités par les détails technologiques des langages trop près de la machine ou trop lâches logiquement parlant.

Cependant, les langages comme le C sont largement employés (car ils sont liés aux systèmes *UNIX* pour le C) tout comme la programmation objet si précisée pour la réalisation des interfaces homme/machine des grands projets industriels.

C'est pourquoi nous avons choisi trois langages qui ont fait leur preuve :

- le langage utilisé par *XCAS* car il est proche du *C++* et qu'en même temps *XCAS* est un outil pour faire des mathématiques au lycée et possède de nombreuses fonctionnalités intéressantes en calcul formel et géométrie ;
- *Python* qui est orienté objet et permet également une programmation impérative et récursive. *Python* présente une syntaxe claire et reste particulièrement abordable pour les débutants, tout en offrant des constructions de haut niveau ;
- *(O)CAML* qui est essentiellement fonctionnel (et donc très mathématique) mais permet également une programmation impérative et orientée objet.

Les deux derniers langages sont largement répandus et sont développés par des équipes à la pointe de la recherche informatique : ils ont été pensés, améliorés depuis des années et évitent les écueils de certaines interfaces prétendument simples d'utilisation mais qui peuvent cacher de nombreux vices et ne déboucheront sur aucune utilisation en dehors du lycée.

Comme nous continuons à travailler les mathématiques « avec un papier et un crayon » ou bien « calculer à la main », nous préférons un contexte de programmation le plus sobre possible, dépouillé de toutes les choses fortement inutiles et qui nuisent à la concentration sur le sujet de réflexion. Un éditeur de texte simple (et non un logiciel de traitement de texte) fait amplement l'affaire, accompagné d'une console (*\*N\*X* de préférence) pour l'interprétation ou la compilation, puis l'exécution d'un programme. Il s'agit vraiment de se concentrer sur l'essentiel, et nous voyons les élèves accepter ce travail, dans la mesure où la récompense est immédiate : le programme fonctionne ou ne fonctionne pas, ou bien effectue autre chose que ce que son auteur prévoyait. Il y a ici une question de satisfaction intellectuelle, sans aucune autre récompense qu'un apprentissage (!), ce qui reste une question centrale de l'école.

Le 20 janvier 2010,

Pascal CHAUVIN

Guillaume CONNAN

## 2 - Algorithme breton

Ce n'est pas très mathématique mais c'est un algorithme et pour affronter tous ceux qui nous attendent, il va nous falloir des forces....

**Entrées** : masse m totale

**Initialisation** :

beurre ← m/4

sucre ← m/4

farine ← m/4

œuf ← m/4

**début**

Couper le beurre en petits morceaux et le mettre à fondre doucement au bain-marie de préférence. Dès qu'il est fondu arrêter. Laisser refroidir mais attention : le beurre doit être encore liquide ! Il ne doit pas redevenir solide

Mettre le four à préchauffer à 160° (th 5)

Mettre les œufs entiers avec le sucre dans un saladier. Battre longuement le mélange pour qu'il blanchisse et devienne bien mousseux

Y ajouter le beurre fondu ET FROID

Rajouter progressivement à l'appareil la farine en l'incorporant bien. Cela doit donner une pâte élastique et un peu épaisse

Verser la préparation dans un moule à manqué ou à cake bien beurré

Laisser cuire environ une heure. Il faut surveiller le gâteau régulièrement.

**si** *il semble brunir trop vite* **alors**

└ il faut baisser un peu le four et mettre une feuille d'aluminium sur le dessus.

Il faut que le dessus du gâteau soit blond foncé, mais pas trop coloré.

**si** *lorsqu'une pique plantée en son milieu ressort sèche* **alors**

└ le gâteau est cuit


**fin**

**Algorithme 1** : algorithme breton pur beurre

## A CAML pour l' impatient

### A1 Installation et utilisation

Pour tous les O.S. il existe des distributions de CAML *clé-en-main* sur le page de [Jean Mouric](#).  
Pour les utilisateurs de Emacs, vous pouvez charger le [mode tuareg](#).  
CAML sous Emacs avec le mode tuareg :



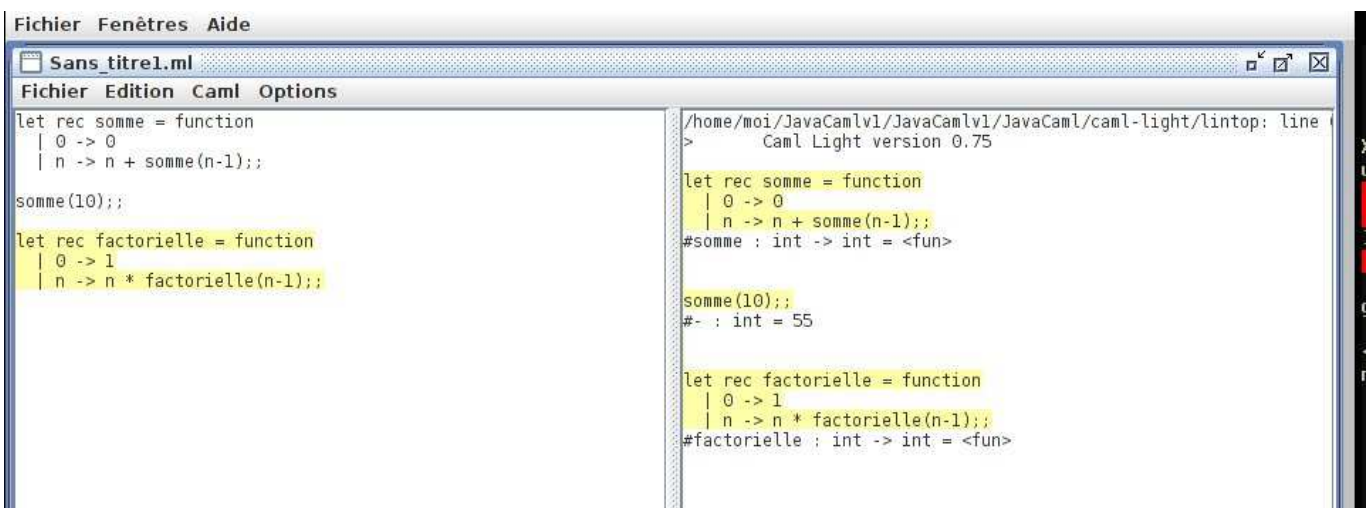
```
File Edit Options Buffers Tools Tuareg Complete In/Out Signals Help
Objective Caml version 3.11.1

# let rec somme = fonction
| 0 -> 0
| n -> n + somme(n-1);;
  val somme : int -> int = <fun>

# somme(10);;
- : int = 55

# let rec factorielle = fonction
| 0 -> 1
| n -> n * factorielle(n-1);;
```

CAML sous JavaCaml :



```
Fichier Fenêtres Aide
Sans_titre1.ml
Fichier Edition Caml Options
let rec somme = fonction
| 0 -> 0
| n -> n + somme(n-1);;

somme(10);;

let rec factorielle = fonction
| 0 -> 1
| n -> n * factorielle(n-1);;

/home/moi/JavaCamlv1/JavaCamlv1/JavaCaml/caml-light/lintop: line 1:
> Caml Light version 0.75

let rec somme = fonction
| 0 -> 0
| n -> n + somme(n-1);;
#somme : int -> int = <fun>

somme(10);;
#- : int = 55

let rec factorielle = fonction
| 0 -> 1
| n -> n * factorielle(n-1);;
#factorielle : int -> int = <fun>
```



## A2 Quelques ressources électroniques

Une passionnante introduction à OCAML par un de ses papas...

Le manuel de référence de CAML par Xavier LEROY et Pierre WEIS

Une introduction à CAML

Ouvrage de référence CAML

Ressources CAML

Un tutoriel en français

Des ouvrages en papier sont référencés dans la bibliographie à la fin de ce document.

## A3 Les nombres

Nous utiliserons dans tout le document la version basique de CAML (en fait OCAML (c'est-à-dire CAML avec des modules pour faire de la programmation orientée objet) ou CAML Light (c'est la version de base que nous utiliserons)) sans charger de modules complémentaires sauf éventuellement pour créer des graphiques ou travailler sur certaines listes.

Nous travaillerons en mode *oplevel* c'est-à-dire que nous compilerons automatiquement de manière interactive. Pour se repérer, ce que nous taperons sera précédé d'un # et ce que renverra CAML sera précédé le plus souvent d'un -.

On peut travailler avec des entiers :

```
# 1+2;;
- : int = 3
```

Vous remarquerez que CAML répond que le résultat de 1+2 est un entier (int) égal à 3. En effet, CAML est adepte de l'*inférence de type*, c'est-à-dire qu'il devine quel type de variable vous utilisez selon ce que vous avez tapé. Nous en reparlerons plus bas. Les priorités des opérations sont respectées. En cas d'ambiguïté, un parenthésage implicite à gauche est adopté :

```
# 1+2*3;;
- : int = 7
# 1-2+3;;
- : int = 2
# 1-(2+3);;
- : int = -4
# (1+2)*3;;
- : int = 9
```

La division renvoie le quotient entier bien sûr :

```
# 11/3;;
- : int = 3
# 11/3*2;;
- : int = 6
# 11/(3*2);;
- : int = 1
```

On peut utiliser mod pour le reste entier :

```
# 7 mod 2;;
- : int = 1
# 7/2;;
- : int = 3
# 7-7/2*2;;
- : int = 1
```

Enfin, les entiers sont de base compris entre  $-2^{30}$  et  $2^{30} - 1$ .

Les nombres non entiers sont de type *float*. Ils sont représentés en interne par deux entiers : une *mantisse*  $m$  et un exposant  $n$  tel que le nombre flottant soit  $m \times 10^n$ . En externe, il apparaît sous forme décimale, le séparateur étant le point.

```
# 1.;;
- : float = 1.
```

Attention alors aux opérations :

```
# 3.14 + 2;;
Characters 0-4:
 3.14 + 2;;
  ^^^^
Error: This expression has type float but an expression was expected of type int
```

Cela indique que vous avez utilisé l'addition des entiers pour ajouter un entier à un flottant. CAML en effet n'effectue pas de conversion implicite et demande que celle-ci soit explicite. Cela peut apparaître comme contraignant mais permet l'inférence de type qui évite nombre de « bugs ».

Les opérations arithmétiques sur les entiers doivent donc être suivies d'un point :

```
# 1. +. 2.1 ;;
- : float = 3.1
# 1.2 +. 2.1 ;;
- : float = 3.3
# 1./ .2.;;
- : float = 0.5
# 1.5e-5 *. 100. ;;
- : float = 0.0015
# sqrt(2.);;
- : float = 1.41421356237309515
# 3.**2.;;
- : float = 9.
# log(2.);;
- : float = 0.693147180559945286
# exp(1.);;
- : float = 2.71828182845904509
# cos(0.);;
- : float = 1.
# cos(2.*.atan(1.));;
- : float = 6.12303176911188629e-17
# sin(2.*.atan(1.));;
- : float = 1.
```

Il existe des moyens de convertir un entier en flottant :

```
# float(1) +. 3.1;;
- : float = 4.1
# float 1 +. 3.1;;
- : float = 4.1
```

et inversement :

```
# int_of_float(sqrt(2.));;
- : int = 1
```

Il ne faut pas confondre `int_of_float` et `floor`

```
# floor(2.1);;
- : float = 2.
# int_of_float(2.1);;
- : int = 2
```

## A4 Les autres types de base

### A 4 a Les booléens

Ils sont bien sûr au nombre de deux : `true` et `false`. Nous en aurons besoin pour les tests. Les fonctions de comparaison renvoient un booléen. On peut combiner des booléens avec `not`, `&` et `or` :

```
# 3>2;;
- : bool = true
# 3=2;;
- : bool = false
# (3>2) & (3=2);;
- : bool = false
# (3>2) or (3=2);;
- : bool = true
# (3>2) & not(3=2);;
- : bool = true
# (3>2) & not(3=2) & (0<1 or 1>0);;
- : bool = true
```

### A 4 b Les chaînes de caractères

En anglais : *string*. On les entoure de guillemets `vb+"` et on les concatène avec `^` :

```
# "Tralala" ^ "pouet pouet";
- : string = "Tralala pouet pouet"
```

### A 4 c Les caractères

En CAML : *char*. On les utilisera surtout pour la cryptographie. Il sont entrés entre accents aigus :

```
# 'a';;
- : char = 'a'
```

## A 5 Les définitions

C'est comme en maths...mais en anglais! Donc *Soit* se dit *Let* :

```
# let x=3*2;;
val x : int = 6
# 2+x;;
- : int = 8
# let pi=acos(-1.);;
val pi : float = 3.14159265358979312
# sin(pi/.2.);;
- : float = 1.
```

Le nom des identificateurs doit commencer par une lettre minuscule.

On peut définir *localement* une variable, c'est-à-dire que sa portée ne dépassera pas l'expression où elle a été définie :

```
# let x=99 in x+1;;
- : int = 100
# x;;
- : int = 6
# let a=3 and b=2 in a*a+b*b;;
- : int = 13
```

## A6 Fonctions

On peut créer des fonctions avec `function` :

```
# let delta = function
| (a,b,c) -> b*b-4*a*c;;
val delta : int * int * int -> int = <fun>

# delta(1,1,1);;
- : int = -3
```

Notez le `val delta : int * int * int -> int = <fun>` qui indique que la fonction construite...est une fonction (`<fun>`) qui va de  $\mathbb{Z}^3$  dans  $\mathbb{Z}$ .

On peut de manière plus standard (informatiquement parlant!) définir la fonction par son expression générale :

```
# let discriminant(a,b,c)=b*b-4*a*c;;
val discriminant : int * int * int -> int = <fun>
# discriminant(1,1,1);;
- : int = -3
```

On peut disjoindre des cas :

```
# let sina = function
| 0. -> 1.
| x -> sin(x)/.x;;
val sina : float -> float = <fun>

# sina(0.);;
- : float = 1.

# sina(0.1);;
- : float = 0.998334166468281548
```

On peut travailler avec autre chose que des nombres :

```
# let mystere = function
| (true,true) -> true
| _ -> false;;
val mystere : bool * bool -> bool = <fun>
# mystere(3>2,0=1-1);;
- : bool = true
# mystere(3>2,0>1);;
- : bool = false
```

Le tiret `_` indique « dans tous les autres cas ».

On peut aussi créer des fonctions *polymorphes*, c'est-à-dire qui ne travaillent pas sur des types de variables particuliers.

```
# let composee(f,g) = function
| x ->f(g(x));;
val composee : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
```

Ici,  $f$  est toute fonction transformant un type de variable  $a$  en un type de variable  $b$  et  $g$  une fonction transformant un type de variable  $c$  en un type de variable  $a$ . La fonction composée transforme bien un type de variable  $c$  en un type de variable  $b$ . Si nous reprenons la fonction définie précédemment :

```
# let carre = function
| x -> x*.x;;
val carre : float -> float = <fun>

# composee(sina,carre);;
- : float -> float = <fun>
```

```
# composee(sina,carre)(3.);
- : float = 0.04579094280463962
```

## A7 If...then...else

On peut utiliser une structure conditionnelle pour définir une fonction :

```
# let val_abs = function
| x-> if x>=0 then x else -x;;
val val_abs : int -> int = <fun>
```

ou bien

```
# let valeur_abs(x)=
  if x>=0 then x
  else -x;;
val valeur_abs : int -> int = <fun>
```

## A8 Fonctions récursives

C'est une fonction construite à partir d'elle-même. On utilise la même syntaxe que pour une fonction simple mais on fait suivre le let d'un rec :

```
# let rec factorielle = function
| 0 -> 1
| n -> n*factorielle(n-1);;
val factorielle : int -> int = <fun>

# let rec fact(n)=
  if n=0 then 1
  else n*fact(n);;
val fact : int -> int = <fun>
```

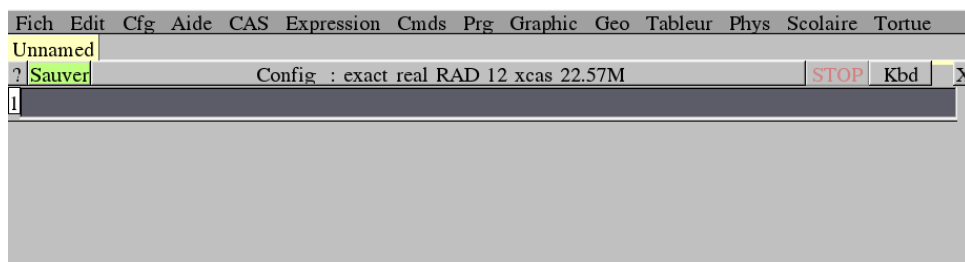
Pour plus de détails sur les fonctions récursives, voir le chapitre 5 page 15.

## B XCAS pour l' impatient

### B1 Installation et utilisation

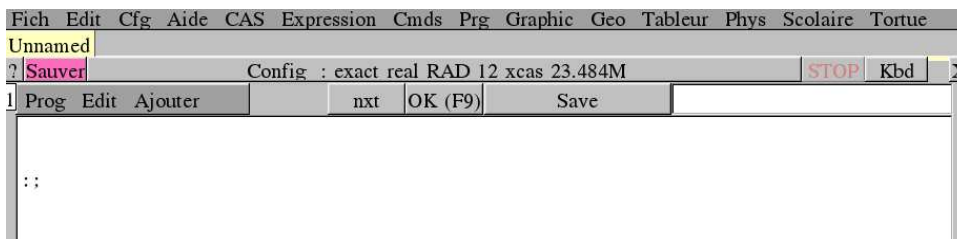
Vous récupérez la version pour votre O.S. sur le [Site XCAS](#).

Vous suivez les instructions et voilà...



Il y a beaucoup à dire sur XCAS mais nous nous contenterons d'explorer les outils de programmation. Nous allons donc ouvrir une fenêtre de programmation en tapant **Alt + P**.

Une fenêtre de programmation apparaît :

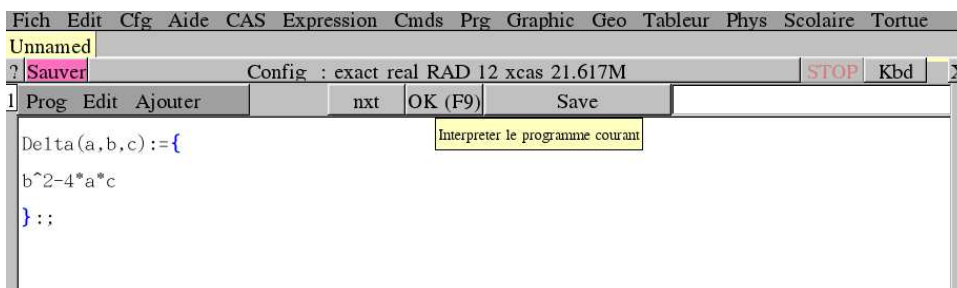


Il ne reste plus qu'à programmer...

## B2 Les procédures

XCAS est écrit en C++ et a donc une syntaxe proche de ce langage...mais en bien plus simple. Cependant, comme le C++, l'outil de base est la procédure qui *ressemble* aux fonctions de CAML.

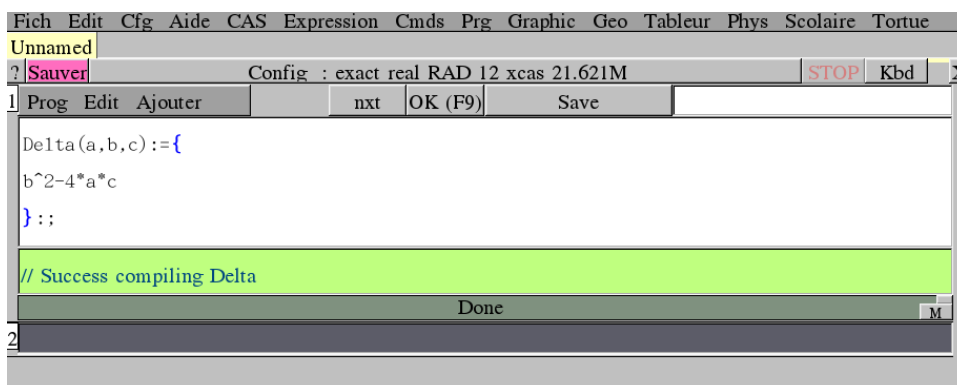
Par exemple, nous voudrions créer une procédure qui prend trois nombres  $a$ ,  $b$  et  $c$  en argument et renvoie  $b^2 - 4ac$ . Appelons-la au hasard...Delta :



Par la suite, nous représenterons ce script par :

```
Delta(a,b,c):={
  b^2-4*a*c
};;
```

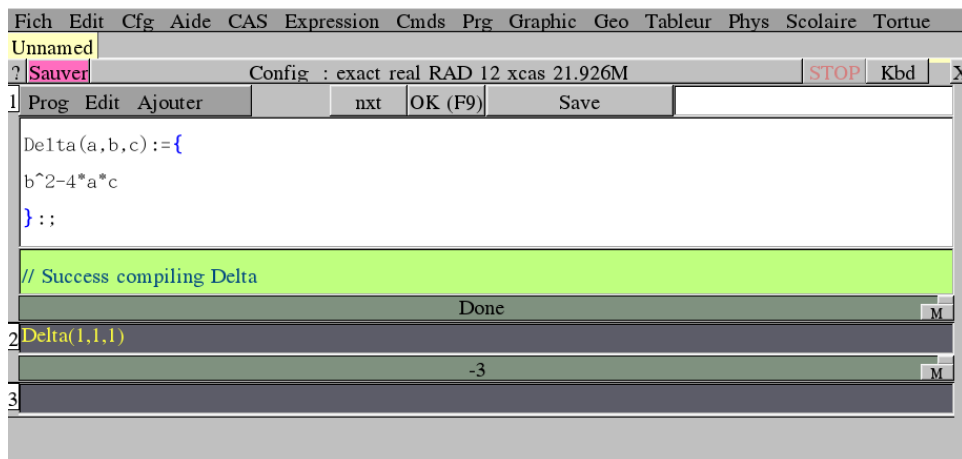
Il suffit ensuite de cliquer sur OK ou de taper sur F9



Pour évaluer cette procédure en un certain triplet, nous entrons dans une cellule de calcul :

```
Delta(1,1,1)
```

ce qui apparaît sous la forme :



### B3 Tests

Avec XCAS, on peut programmer en français ou en anglais.

```
val_abs(x) := {
  si x>0 alors x sinon -x fsi;
};;
```

ou

```
val_abs(x) := {
  if(x>0) then {x} else {-x}
};;
```

ou même

```
val_abs(x) := {
  ifte(x>0, x, -x)
};;
```

### B4 Boucles

Le bilinguisme est ici aussi de rigueur. Calculons par exemple la somme des premiers entiers :

```
Somme(n) := {
  local S, k;
  S := 0;
  pour k de 1 jusque n faire
    S := S+k;
  fpour;
  retourne(S);
};;
```

```

Fich Edit Cfg Aide CAS Expression Cmds Prg Graphic Geo Tableur Phys Scolaire Tortue
Unnamed
?sauve Config : exact real RAD 12 xcas 22.094M STOP Kbd X
1 Prog Edit Ajouter      nxt OK (F9)      Save
Somme(n) := {
  local S,k;
  S:=0;
  pour k de 1 jusque n faire
    S:=S+k;
  fpour;
  return(S);
};;

// Parsing Somme
// Success compiling Somme
Done
2 Somme(10)
55
3

```

ou

```

Somme(n) := {
  local S,k;
  S:=0;
  for(k:=1;k<=n;k:=k+1){
    S:=S+k
  }
  return(S);
};;

```

ou

```

Somme(n) := {
  local S,k;
  S:=0;
  k:=1;
  tantque k<=n faire
    S:=S+k;
    k:=k+1;
  ftantque;
  retourne(S);
};;

```

ou

```

Somme(n) := {
  local S,k;
  S:=0;
  k:=1;
  while(k<=n){
    S:=S+k;
    k:=k+1;
  };
  return(S);
};;

```

Pour plus de détails sur la programmation impérative voir le chapitre 5 page 15.



# 4 - Dis monsieur, dessine-moi un icosagone..

Certains connaissent peut-être le langage Logo qui avait permis aux petits écoliers d'il y a une vingtaine d'années de s'initier à la programmation.

Voyons par exemple un petit algorithme permettant de tracer un polygone fermé régulier quelconque.

## A Algorithme

**Entrées :** nombre  $N$  de côtés et longueur  $a$  d'un côté

**début**

```
| pour  $k$  de 1 jusqu'à  $N$  faire  
|   avance de  $a$  pas  
|   tourne de  $360/a$  degrés  
fin
```

Algorithme 2 : polygone

## B Traduction XCAS

XCAS comprend un module *Tortue* très complet :

```
polylogo(N,a):={  
local k;  
pour k de 1 jusque N faire  
avance(a);  
tourne_droite(360/N);  
fpour;  
};;
```

Programme 1 – Totue LOGO

# 5 - Algorithmes récursifs / algorithmes impératifs

## A Programmation impérative et affectations

### A1 Aspect technique

La programmation impérative est la plus répandue (Fortran, Pascal, C,...) et la plus proche de la machine réelle. Son principe est de modifier à chaque instant l'état de la mémoire via les affectations. Historiquement, son principe a été initié par John VON NEUMANN et Alan TURING.

Par exemple, lorsqu'on entre avec XCAS :

```
a:=2
```

cela signifie que la case mémoire portant le petit « fanion » a contient à cet instant la valeur 2.

On peut changer le contenu de cette case mémoire un peu plus tard :

```
a:=4
```

et cette fois la case a contient la valeur 4.

Le caractère chronologique de ce type de programmation est donc primordial. Regardons la suite d'instructions suivante :

```
a:=1;  
b:=2;  
a:=a+b; // maintenant a contient 3  
b:=a-b; // maintenant b contient 3-2=1  
a:=a-b; // maintenant a contient 3-1=2
```

Programme 2 – attention aux affectations

Cela permet d'échanger les contenus des cases mémoire a et b.

Si l'on change l'ordre d'une quelconque des trois dernières lignes, les valeurs présentes dans les cases mémoires nommées a et b vont être différentes.

Si l'on regarde les deux dernières lignes, elles affectent apparemment la même valeur (a-b) à a et à b et pourtant c'est faux car les variables a et b ne contiennent pas les mêmes valeurs selon le moment où elles sont appelées.

Il faut donc être extrêmement attentif à la chronologie des instructions données. C'est un exercice intéressant en soi mais pas sans danger (c'est d'ailleurs la source de la plupart des « bugs » en informatique).

Question : est-ce que

```
a:=3;  
a:=a+1;
```

dit la même chose que

```
a:=a+1;  
a:=3;
```

La motivation première de ce type de fonctionnement a été pratique : les ordinateurs étant peu puissants, le souci principal du concepteur de programme a longtemps été d'économiser au maximum la mémoire. Au lieu d'accumuler des valeurs

dans différents endroits de la mémoire (répartition spatiale), on les stocke au même endroit mais à différents moments (répartition temporelle). C'est le modèle des machines à états (machine de TURING et architecture de VON NEUMANN) qui impose au programmeur de connaître à tout instant l'état de la mémoire centrale.

## A2 Exploitation mathématique possible

En classe de mathématique, cet aspect technique nous importe peu. Il nous permet cependant d'illustrer dynamiquement la notion de fonction :

```
y:=x^2+1;
x:=1; y; // ici y=2
x:=-5; y; // ici y=26
x:=1/2; y; // ici y=5/4
```

Programme 3 – affectation et image par une fonction

La fonction  $f : x \mapsto x^2 + 1$  dépend de la variable  $x$ . On affecte une certaine valeur à la variable ce qui détermine la valeur de son image. Il est toutefois à noter qu'en fait on a travaillé ici sur des expressions numériques et non pas des fonctions.

On peut enchaîner des affectations :

```
y:=x^2+1;
z:=y+3;
x:=1; y; z; /* ici y:=2 et z=5 */
x:=-5 y; z; // ici y=26 et z=29;
x:=1/2; y; z; // ici y=5/4 et z=17/4;
```

Programme 4 – affectation et composition de fonctions

et illustrer ainsi la composition de fonctions.

Avec un peu de mauvaise foi (car le cas est simple et artificiel) mais pas tant que ça puisque de tels problèmes sont source de milliers de bugs sur des milliers de programmes plus compliqués, nous allons mettre en évidence un problème majeur de ce genre de programmation quand on n'est pas assez attentif.

On crée à un certain moment une variable à laquelle on affecte une valeur :

```
a:=2;
```

On introduit plus tard une procédure (une sorte de fonction informatique des langages impératifs) qu'on nomme  $f$  et qui effectue certains calculs en utilisant  $a$  :

```
f(x):={
  a:=a+1;
  retourne(x+a)
}
```

Programme 5 – exemple de procédure

Alors  $f(5)$  renvoie 8 mais si on redemande  $f(5)$  on obtient à présent 9 et si on recommence on obtient 10, etc. car à chaque appel de la procédure  $f$ , la variable  $a$  est modifiée : ici  $2 \times f(5) \neq f(5) + f(5)$  !

On peut donc construire un objet qui **ressemble** à une fonction d'une variable (mais qui n'en est évidemment pas) et qui à une même variable renvoie plusieurs valeurs. En effet, une procédure<sup>a</sup> ne dépend pas uniquement des arguments entrés et donc peut changer de comportement au cours du programme : c'est ce qu'on appelle en informatique des *effets de bord*.

On a en fait plutôt construit ici une fonction de deux variables :

$$g : (a, x) \rightarrow x + a$$

Le premier appel de  $f(5)$  est en fait  $g(3, 5)$  et le deuxième  $g(4, 5)$ , etc. mais on a pu le faire en utilisant une forme qui **laisse à penser** que  $f$  est une fonction qui renvoie une infinité d'images différentes d'un même nombre.

Informatiquement,  $a$  et  $x$  jouent des rôles différents puisque  $x$  est un argument de la procédure  $f$  et  $a$  est une variable globale donc « extérieure » à  $f$  (on dirait un *paramètre*). D'ailleurs, XCAS envoie un avertissement pour nous le rappeler :

```
// Warning: a, declared as global variable(s) compiling f
```

a. Certains langages utilisent le terme fonction mais nous préférons « procédure » pour ne pas confondre avec les fonctions mathématiques.

### A3 Programmation fonctionnelle

Les langages fonctionnels bannissent totalement les affectations. Ils utilisent des fonctions *au sens mathématique du terme*, c'est-à-dire qu'une fonction associe une unique valeur de sortie à une valeur donnée en entrée. Les fonctions peuvent être testées une par une car *elles ne changent pas selon le moment où elles sont appelées dans le programme*. On parle de *transparence référentielle*. Le programmeur n'a donc pas besoin de savoir dans quel état se trouve la mémoire. C'est le logiciel qui s'occupe de la gestion de la mémoire.

Historiquement, ce style de programmation est né du  $\lambda$ -calcul développé par Alonzo CHURCH juste avant la publication des travaux d'Alan TURING<sup>b</sup>.

Ce principe d'utilisation de fonctions au sens mathématique est perturbant pour des programmeurs formés aux langages impératifs mais ne devraient pas troubler des mathématiciens...

La **récurtivité** est le mode habituel de programmation avec de tels langages : il s'agit de fonctions qui s'appellent elles-mêmes. Pendant longtemps, la qualité des ordinateurs et des langages fonctionnels n'était pas suffisante et limitaient les domaines d'application de ces langages.

On utilise également en programmation fonctionnelle des *fonctions d'ordre supérieur*, c'est-à-dire des fonctions ayant comme arguments d'autres fonctions : cela permet par exemple de créer des fonctions qui à une fonction dérivable associe sa fonction dérivée.

### A4 Variable, affectation, déclaration, valeur

Nous allons préciser maintenant comment est traitée une variable dans les deux modes de programmation étudiés.

En programmation fonctionnelle, une variable est un nom pour une valeur alors qu'en programmation impérative une variable est un nom pour une case mémoire. En programmation fonctionnelle, on ne peut pas changer la valeur d'une variable. Nous parlerons dans ce cas de déclaration de variable pour la distinguer de l'affectation de variable des langages impératifs. Une variable en programmation fonctionnelle correspond à une constante en programmation impérative.

Par exemple, si nous écrivons dans CAML :

```
# let x=3;;
```

puis

```
# let x=4;;
```

on ne modifie pas la variable x mais on en crée une nouvelle qui porte le même nom. L'ancienne variable x est maintenant masquée et inaccessible. « Différence subtile pour informaticien coupant les cheveux en quatre » me direz-vous...et bien non car cela peut créer des surprises si l'on n'en est pas conscient.

Considérons les déclarations suivantes sur CAML :

```
# let x=3;; (* le # est créé automatiquement par CAML pour signifier qu'il attend nos instructions
. Les ;; indiquent que nous avons fini de lui « parler » *)

# let f = fonction
  y -> y+x;;
```

Si l'on veut la valeur de l'image de 2 par la fonction f :

```
# f(2);; (* ce que nous entrons *)
- : int = 5 (* la réponse de CAML *)
```

en effet,  $2 + 3 = 5$ .

Observez maintenant ce qui se passe ensuite :

```
# let x=0;;
x : int = 0

# f(2) ;;
- : int = 5
```

b. qui a d'ailleurs été un étudiant de Church à Princeton...

On a modifié la valeur de  $x$  et pourtant  $f(2)$  ne change pas. En effet, on a défini  $f$  en utilisant le nom  $x$  qui vaut 3 lors de la définition de  $f$ . Plus tard, on redéclare la variable  $x$  avec la valeur 0. On n'a pas modifié la valeur de l'ancien identificateur  $x$  et donc  $f(2)$  vaut toujours 5. Ainsi  $x$  est un nom pour la valeur 3 et  $f$  un nom pour l'objet fonction  $y \rightarrow y+x$  c'est-à-dire fonction  $y \rightarrow y+3$ .

Observons maintenant ce que cela donne en programmation impérative avec XCAS.

```
x:=3;
f:=y->y+x;
```

Les objets sont affectés. XCAS nous renvoie un message précisant qu'il comprend que  $x$  est une variable globale dans la définition de la procédure  $f$ .

```
// Warning: x, declared as global variable(s)
```

```
f(2);
```

5

Maintenant, ré-affectons  $x$  :

```
x:=0;
f(2);
```

2

Cette fois on a ré-affecté la case mémoire  $x$  qui contient maintenant 0 : son ancien contenu est écrasé et c'est comme s'il n'avait jamais existé. La procédure  $f$  utilise donc ce qu'elle trouve dans la case mémoire  $x$  au moment où on l'appelle et  $f(2)$  peut donc prendre des valeurs différentes au cours du temps.

Dans toute la suite de ce document, on distinguera donc l'affectation de variable (en tant que modification du contenu d'une case mémoire étiquetée) de la programmation impérative et la déclaration de variable de la programmation fonctionnelle.

#### A 4 a Une récursion concrète

On place un certain nombre d'élèves en les classant par ordre décroissant de taille. Le plus grand voudrait connaître sa propre taille mais chaque élève ne peut mesurer que son écart de taille avec son voisin. Seul le plus petit connaît sa taille.

Le plus grand mesure son écart avec son voisin et garde le nombre en mémoire. Le voisin fait de même avec son propre voisin et ainsi de suite. Enfin l'avant dernier fait de même puis le plus petit lui transmet sa taille : il peut donc calculer sa propre taille et transmet la au suivant et ainsi de suite : chacun va ainsi pouvoir calculer sa taille de proche en proche....

#### A 4 b Un premier exemple d'algorithme récursif

Si nous reprenons notre exemple précédant, nous pouvons faire exprimer aux élèves que l'entier suivant  $n$  est égal à  $1+$  l'entier suivant  $n-1$  : on ne sort pas des limites du cours de mathématiques et on n'a pas besoin d'introduire des notions d'informatique en travaillant sur cette notion.

On peut écrire un algorithme ainsi :

```
si n = 0 alors
  1
sinon
  1+ successeur de n-1
```

La récursivité n'est pas l'apanage des langages fonctionnels. Par exemple XCAS permet d'écrire des programmes récursifs.

En français :

```
successeur(k) := {
  si k==0 alors 1
  sinon 1+successeur(k-1)
  fsi
};;
```

Programme 6 – exemple de programme récursif

En anglais :

```

successeur (k) := {
  if (k==0) then {1}
  else {1+successeur (k-1)}
} ;;

```

Programme 7 – exemple de programme récursif en anglais

Que se passe-t-il dans le cœur de l'ordinateur lorsqu'on entre successeur(3) ?

- On entre comme argument 3 ;
- Comme 3 n'est pas égal à 0, alors successeur(3) est stocké en mémoire et vaut 1+successeur(2) ;
- Comme 2 n'est pas égal à 0, alors successeur(2) est stocké en mémoire et vaut 1+successeur(1) ;
- Comme 1 n'est pas égal à 0, alors successeur(1) est stocké en mémoire et vaut 1+successeur(0) ;
- Cette fois, successeur(0) est connu et vaut 1 ;
- successeur(1) est maintenant connu et vaut 1+successeur(0) c'est-à-dire 2 ;
- successeur(2) est maintenant connu et vaut 1+successeur(1) c'est-à-dire 3 ;
- successeur(3) est maintenant connu et vaut 1+successeur(2) c'est-à-dire 4 : c'est fini !

C'est assez long comme cheminement mais ce n'est pas grave car *c'est l'ordinateur qui effectue le « sale boulot »* ! Il stocke les résultats intermédiaires dans une *pile* et n'affiche finalement que le résultat.

Comme il calcule vite, ce n'est pas grave. L'élève ou le professeur ne s'est occupé que de la définition récursive (mathématique !) du problème.

XCAS calcule facilement successeur(700) mais successeur(7000) dépasse les possibilités de calcul du logiciel.

Et oui, un langage impératif ne traite pas efficacement le problème de pile, c'est pourquoi pendant longtemps seuls les algorithmes impératifs ont prévalu.

**Remarque :** par défaut, XCAS s'arrête au bout de 50 niveaux de récursion. Pour aller plus loin, il faut cliquer sur config à côté de save. Régler alors la fenêtre recurs à 1000 par exemple

Utilisons à présent le langage fonctionnel CAML, développé par l'INRIA<sup>c</sup>. Même s'il intègre des aspects impératifs pour faciliter l'écriture de certains algorithmes, il est en premier lieu un langage fonctionnel qui en particulier gère très efficacement la mémoire de l'ordinateur pour éviter sa saturation lors d'appels récursifs.

Le programme s'écrit comme en mathématique (mais en anglais...) :

```

# let rec successeur (k)=
  if k=0 then 1
  else 1+successeur (k-1) ;;

```

Programme 8 – successeur d'un entier

Alors par exemple :

```

# successeur(36000) ;;
- : int = 36001

```

Mais

```

# successeur(3600000) ;;
Stack overflow during evaluation (looping recursion?).

```

La *pile* où sont stockés les résultats intermédiaires créés par la récursion est en effet saturée.

Aujourd'hui, les langages fonctionnels sont très efficaces et gèrent de manière intelligente la *pile*. Ils le font soit automatiquement, soit si la fonction utilise une *récursion terminale*, c'est-à-dire si l'appel récursif à la fonction n'est pas *enrobé* dans une autre fonction.

Ici, ce n'est pas le cas car l'appel récursif successeur(k-1) est enrobé dans la fonction  $x \mapsto 1 + x$ .

On peut y remédier en introduisant une fonction intermédiaire qui sera récursive terminale :

```

# let rec successeur_temp(k,resultat)=
  if k=0 then resultat
  else successeur_temp(k-1,resultat+1) ;;

```

Programme 9 – successeur version terminale (étape 1)

puis on appelle cette fonction en prenant comme résultat de départ 1 :

c. Institut National de Recherche en Informatique et Automatique

```
# let successeur_bis(k)=
  successeur_temp(k,1);;
```

Programme 10 – successeur version terminale (étape 2)

Alors :

```
# successeur_bis(360000000);;
- : int = 360000001
```

Il n'y a donc aucun problème pour traiter 360 millions d'appels récursifs !

**Dans un premier temps, on peut laisser de côté ce problème de récursion terminale au lycée et se contenter de travailler sur de petits entiers. Ainsi, on peut choisir de travailler avec CAML ou XCAS par exemple, même si ce dernier n'est pas un langage fonctionnel.**

## A5 Un premier exemple de fonction d'ordre supérieur

Définissons une fonction qui à deux fonctions  $f$  et  $g$  associe sa composée  $f \circ g$  :

```
# let compose=function
  (f,g) -> function x->f(g(x));;
```

Programme 11 – composition de fonction

CAML devine le type des objets introduits en répondant :

```
val compose : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
```

Ceci signifie que `compose` est une fonction (le `<fun>` final) qui prend comme arguments deux fonctions (la première qui transforme une variable de type `a` en une variable de type `b` et la deuxième qui transforme une variable de type `c` en une variable de type `a`) et qui renvoie une fonction qui transforme une variable de type `c` en une variable de type `b`. En résumé, on peut schématiser ce qu'a compris CAML :

$$f : 'a \mapsto 'b \quad g : 'c \mapsto 'a \quad f \circ g : 'c \mapsto 'b$$

Créons deux fonctions numériques :

```
# let carre = function
  x -> x*x;;

# let double = function
  x -> 2*x;;
```

Programme 12 – création de fonctions simples

Composons-les :

```
# compose(double,carre)(2);;
- : int = 8
# compose(carre,double)(2);;
- : int = 16
```

En effet, le double du carré de 2 est 8 alors que le carré du double de 2 vaut 16...

On peut faire la même chose avec XCAS :

```
compose(f,g):={
  x->f(g(x))
}
```

Programme 13 – composée de fonctions avec XCAS

Puis par exemple :

```
compose(x->x^2,x->2*x)(2)
```

## B Deux manières d'aborder un même problème

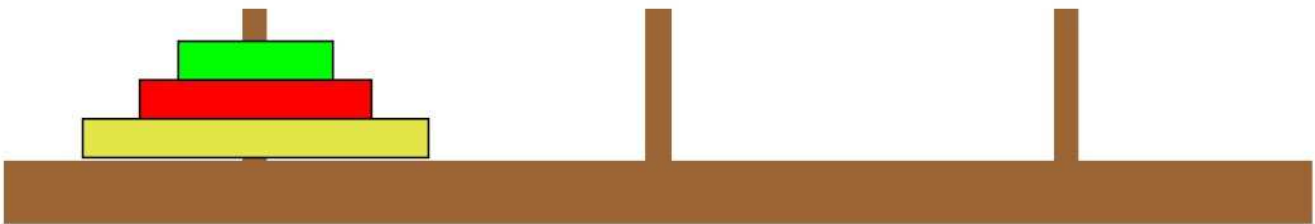
### B 1 Les tours de Hanoï

#### B 1 a Le principe

Ce casse-tête a été posé par le mathématicien français Édouard LUCAS en 1883.

Le jeu consiste en une plaquette de bois où sont plantés trois piquets. Au début du jeu,  $n$  disques de diamètres croissant de bas en haut sont placés sur le piquet de gauche. Le but du jeu est de mettre ces disques dans le même ordre sur le piquet de droite en respectant les règles suivantes :

- on ne déplace qu'un disque à la fois ;
- on ne peut poser un disque que sur un disque de diamètre supérieur.



Essayez avec 2 puis 3 disques... Nous n'osons pas vous demander de résoudre le problème avec 4 disques !

#### B 1 b Non pas comment mais pourquoi : version récursive

En réfléchissant récursivement, c'est enfantin!... Pour définir un algorithme récursif, il nous faut régler un cas simple et pouvoir simplifier un cas compliqué.

- *cas simple* : s'il y a zéro disque, il n'y a rien à faire !
- *simplification d'un cas compliqué* : nous avons  $n$  disques au départ sur le premier disque et nous savons résoudre le problème pour  $n-1$  disques. Appelons A, B et C les piquets de gauche à droite. Il suffit de déplacer les  $n-1$  disques supérieurs de A vers B (hypothèse de récurrence) puis on déplace le plus gros disque resté sur A en C. Il ne reste plus qu'à déplacer vers C les  $n-1$  disques qui étaient en B. (hypothèse de récurrence).

On voit bien la récursion : le problème à  $n$  disques est résolu si on sait résoudre le cas à  $n-1$  disques et on sait quoi faire quand il n'y a plus de disques.

On sait pourquoi cet algorithme va réussir mais on ne sait pas comment s'y prendre étape par étape.

On va donc appeler à l'aide l'ordinateur en lui demandant d'écrire les mouvements effectués à chaque étape.

On commence par créer une fonction qui affichera le mouvement effectué :

```
let mvt depart arrivee=
print_string
("Déplace un disque de la tige " ^ depart ^ " vers la tige " ^ arrivee);
print_newline();;
```

Programme 14 – mouvement élémentaire

Le programme proprement dit :

```
let rec hanoi a b c= function
| 0 -> () (*0 disque : on ne fait rien*)
| n -> hanoi a c b (n-1); (*n-1 disques sont placés dans l'ordre de a vers b*)
      mvt a c; (*on déplace le disque restant en a vers c*)
      hanoi b a c (n-1) (*n-1 disques sont placés dans l'ordre de b vers c*);;
```

Programme 15 – résolution récursive du problème des tours de Hanoï



Les phrases entre (\* et\*) sont des commentaires.

Par exemple, dans le cas de 3 disques :

```
# hanoi "A" "B" "C" 3;;
Deplace un disque de la tige A vers la tige C
Deplace un disque de la tige A vers la tige B
Deplace un disque de la tige C vers la tige B
Deplace un disque de la tige A vers la tige C
Deplace un disque de la tige B vers la tige A
Deplace un disque de la tige B vers la tige C
Deplace un disque de la tige A vers la tige C
- : unit = ()
```

### B 1 c Non pas pourquoi mais comment : version impérative

Prenez un papier et un crayon. Dessinez trois piquets A, C et B et les trois disques dans leur position initiale sur le plot A.

Déplacez alors les disques selon la méthode suivante :

- déplacez le plus petit disque vers le plot suivant selon une permutation circulaire A-C-B-A ;
- un seul des deux autres disques est déplaçable vers un seul piquet possible.

Réitérez (en informatique à l'aide de boucles...) ce mécanisme jusqu'à ce que tous les disques soient sur C dans la bonne position.

On voit donc bien ici comment ça marche ; il est plus délicat de savoir pourquoi on arrivera au résultat.

Le programme est lui-même assez compliqué à écrire : vous trouverez une version en C (235 lignes...) à cette adresse :

<http://files.codes-sources.com/fichier.aspx?id=38936&f=HANOI%5cHANOI.C>

## B 2 Suites définies par une relation de récurrence

### B 2 a Sans affectations

La relation  $u_{n+1} = 3u_n + 2$  valable pour tout entier naturel  $n$  et la donnée de  $u_0 = 5$  constituent un algorithme de calcul récursif de tout terme de la suite  $(u_n)$ .

```
si n=0 alors
  | u0
sinon
  | 3×u(n-1,u0)+2
```

Fonction  $u(n, u_0)$

Sa traduction dans des langages sachant plus ou moins traiter de tels algorithmes est directe.

Par exemple, en XCAS, cela devient en français :

```
u(n,uo) := {
  si n==0 alors uo
  sinon 3*u(n-1,uo)+2
}
```

Programme 16 – suite récurrente

et en anglais :

```
u(n,uo) := {
  if(n==0){return(uo)}
  else{3*u(n-1,uo)+2}
}
```

Programme 17 – suite récurrente (VO)

En CAML :

```
# let rec u(n,uo)=
  if n=0 then uo
  else 3*u(n-1,uo)+2;;
```

Programme 18 – suite récurrente avec CAML

En Sage :

```
sage: def u(n,uo):
....: if n==0:
....: return uo
....: else:
....: return 3*u(n-1,uo)+2
```

Programme 19 – suite récurrente avec Sage

Ici encore, l'accent est mis sur le « pourquoi » et non pas le « comment » même s'il est beaucoup plus simple de le savoir que dans le cas des tours de Hanoï. On illustre le cours de 1ère sur les suites.

## B 2 b Avec affectations

Au lieu de stocker de nombreuses valeurs dans une pile, on affecte une seule case mémoire aux valeurs de la suite qui sera *ré-affectée*, au fur et à mesure du déroulement du programme.

```
Entrées : n,uo
Initialisation : temp ← uo
début
  | pour k de 1 jusque n faire
  |   | temp ← 3 × temp + 2
fin
```

Algorithme 4 : Suite récurrente : version impérative

Cela donne en français avec XCAS :

```
u_imp(n,uo) := {
local k, temp;
temp := uo;
pour k de 1 jusque n faire
  temp := 3 * temp + 2;
fpour
retourne(temp);
}
```

Programme 20 – suite récurrente version impérative

En anglais :

```
u_imp(n,uo) := {
local k, temp;
temp := uo;
for(k:=1; k<=n; k:=k+1){
temp := 3 * temp + 2;
}
return(temp);
}
```

Programme 21 – suite récurrente version impérative (VO)

Avec Sage/Python :

```
sage: def u_imp(n,uo):
....:     temp=uo
....:     for k in [1..n] : temp=3*temp+2
....:     return(temp)
```

## Programme 22 – suite récurrente version impérative en Sage

Dans ce cas, on voit comment faire pour calculer  $u_n$  à partir de  $u_0$ . La méthode peut paraître moins « élégante » mais elle est sûrement plus « concrètement » assimilable.

**B3** Calculs de sommes

On veut calculer la somme des entiers de 1 à  $n$ .

**B3 a** Sans affectation

Il suffit de remarquer que cela revient à ajouter  $n$  à la somme des entiers de 1 à  $n-1$  sachant que la somme des entiers de 1 à 1 vaut... 1.

L'écriture est donc simple et mathématique :

```

si n=1 alors
  | 1
sinon
  | 1+som_ent(n-1)

```

En français avec XCAS :

```

som_ent(n):={
  si n==1 alors 1
  sinon n+som_ent(n-1)
  fsi
};

```

## Programme 23 – somme des entiers avec XCAS

En anglais :

```

som_ent(n):={
  if(n==1){1}
  else{n+som_ent(n-1)}
};

```

## Programme 24 – somme des entiers avec XCAS (VO)

En CAML :

```

# let rec som_ent(n)=
  if n=1 then 1
  else n+som_ent(n-1);;

```

## Programme 25 – somme des entiers avec CAML

Par exemple :

```

# som_ent(100000);;
- : int = 705082704

```

On peut même généraliser cette procédure à n'importe quelle somme du type  $\sum_{k=k_0}^n f(k)$  :

```

# let rec som_rec(f,ko,n)=
  if n=ko then f(ko)
  else f(n)+som_rec(f,ko,n-1);;

```

## Programme 26 – somme des images des entiers par une fonction avec CAML

Avec SAGE :

```
sage: def som_ent(n):
....:   if n==1 : return 1
....:   else : return n+som_ent(n-1)
```

Programme 27 – somme des entiers avec Sage

```
sage: def som_rec(f,ko,n):
....:   if n==ko : return f(ko)
....:   else : return f(n)+som_rec(f,ko,n-1)
```

Programme 28 – somme des images des entiers par une fonction avec Sage

### B 3 b Avec affectation

**Entrées :**  $n$ (entier naturel)  
**Initialisation :**  $S \leftarrow 0$   
**début**  
 | **pour**  $k$  de 1 à  $n$  **faire**  
 | |  $S \leftarrow S+k$   
 | **retourner**  $S$   
**fin**

Traduction XCAS en français :

```
som(n):={
local S;
S:=0;
pour k de 1 jusque n faire
  S:=S+k;
fpour;
retourne(S)
}
```

Programme 29 – somme des entiers version impérative avec XCAS

En anglais :

```
som(n):={
local S;
S:=0;
for(k:=1;k<=n;k++){S:=S+k};
return(S)
}
```

Programme 30 – somme des entiers version impérative (VO)

Avec Sage/Python :

```
sage: def som(n):
....:   S=0
....:   for k in [1..n] : S+=k
....:   return S
```

Programme 31 – somme des entiers version impérative avec Sage

On a une approche physique avec l'algorithme impératif :  $S$  vaut 0 au départ et on demande  $S$  en sortie ;  $S$  ne vaut donc plus 0 en général... Cela peut être troublant pour des élèves qui ont du mal à résoudre des équations simples et oublient ce qui se cache derrière le symbole « = ». Il y aurait une égalité en mathématiques et une autre en informatique !

Cependant, cette manière de voir peut être plus intuitive : si on considère  $S$  comme le solde de son compte en banque, on surveille son compte régulièrement en ajoutant  $k$  à chaque visite. Le compte porte le même nom,  $S$ , mais son contenu change régulièrement.

La version impérative peut de plus être directement liée à l'écriture :

$$\sum_{k=1}^n k$$

qu'on lit « somme des  $k$  pour  $k$  variant de 1 à  $n$  » où la boucle « pour » apparaît de manière naturelle.

## B 4 Partie entière

On définit la partie entière d'un réel  $x$  comme étant le plus grand entier inférieur à  $x$ .

### B 4 a Sans affectation

On part du fait que la partie entière d'un nombre appartenant à  $[0; 1[$  est nulle.

Ensuite, on « descend » de  $x$  vers 0 par pas de 1 si le nombre est positif en montrant que  $\lfloor x \rfloor = 1 + \lfloor x - 1 \rfloor$ .

Si le nombre est négatif, on « monte » vers 0 en montrant que  $\lfloor x \rfloor = -1 + \lfloor x + 1 \rfloor$ .

```

si  $x$  est positif et  $x < 1$  alors
  | 0
sinon
  | si  $x$  est positif alors
  | | 1+partie_entiere(x-1)
  | sinon
  | | -1+partie_entiere(x+1)

```

En français avec XCAS :

```

partie_entiere(x):={
  si ((x>=0) et (x<1)) alors 0
  sinon si (x>0)
    alors 1+partie_entiere(x-1)
    sinon -1+partie_entiere(x+1)
  fsi
fsi
}

```

Programme 32 – partie entière récursive

En anglais :

```

partie_entiere(x):={
  if((x>=0) and (x<1)){0}
  else{ if(x>0){1+partie_entiere(x-1)}
        else{-1+partie_entiere(x+1)}
  }
};;

```

Programme 33 – partie entière récursive (VO)

En CAML :

```

# let rec partie_entiere (x)=
  if x >= 0. && x < 1.
  then 0.
  else if x > 0.
    then 1. +. partie_entiere (x -. 1.)
    else -.1. +. partie_entiere (x +. 1.);;

```

Programme 34 – partie entière récursive avec CAML

*Remarque* : la double esperluette correspond à l'opérateur logique « ET ». CAML travaille avec deux types de nombres : les entiers et les flottants. Les opérations avec les entiers suivent les notations habituelles. Lorsqu'on travaille avec des flottants (c'est le cas ici car  $x$  est un réel quelconque) le symbole habituel de l'opération est suivi d'un point.

Avec Sage/Python :

```
sage: def partie_entiere(x):
.....:     if x>=0 and x<1 : return 0
.....:     elif x>0 : return 1+partie_entiere(x-1)
.....:     else : return -1+partie_entiere(x+1)
```

Programme 35 – partie entière récursive avec Sage

On notera que l'algorithme récursif s'applique ici à des **réels** et non plus à des **entiers** : on dépasse ainsi le cadre des suites. Pour que l'algorithme fonctionne, on raisonne sur des intervalles et non plus sur une valeur initiale.

Ici, on réfléchit à la notion *mathématique* de partie entière. On met en place des résultats *mathématiques* intéressants.

#### B 4 b Avec affectation

Pour les nombres positifs, on part de 0. Tant qu'on n'a pas dépassé  $x$ , on avance d'un pas. La boucle s'arrête dès que  $k$  est strictement supérieur à  $x$ . La partie entière vaut donc  $k - 1$ .

Dans le cas d'un réel négatif, il faut cette fois reculer d'un pas à chaque tour de boucle et la partie entière est la première valeur de  $k$  à être inférieure à  $x$  :

```
Entrées : x(réel)
Initialisation : k ← 0
début
|   si x>=0 alors
|   |   tant que k<=x faire
|   |   |   k ← k+1
|   |   |   retourner k-1
|   sinon
|   |   tant que k>x faire
|   |   |   k ← k-1
|   |   |   retourner k
fin
```

Avec XCAS en Français :

```
pe(x):={
local k;
k:=0;
si x>=0 alors
    tantque k<=x faire
        k:=k+1;
    ftantque;
    retourne(k-1);
sinon
    tantque k>x faire
        k:=k-1;
    ftantque;
    retourne(k);
fsi;
};;
```

Programme 36 – partie entière impérative (XCAS)

Avec XCAS en anglais :

```

pe(x) := {
  local k;
  k := 0;
  if (x >= 0) {
    while (k <= x) { k := k + 1 };
    return (k - 1);
  }
  else {
    while (k > x) { k := k - 1 };
    return (k);
  }
};

```

Programme 37 – partie entière impérative (VO)

Avec Sage/Python :

```

>>> def pe(x):
    k=0
    if x>=0:
        while k<=x : k+=1
        return k-1
    else :
        while k>x : k+=-1
        return k

```

Programme 38 – partie entière impérative (Sage/Python)

On utilise ici une affectation ( $k$  devient  $k + 1$  ou  $k - 1$ ) et une « boucle while » avec un test et une action d'affectation tant que le test est vrai. Il s'agit d'un traitement physique de la mémoire. On perd un peu de vue la notion mathématique mais on donne une vision dynamique de la recherche de la partie entière : on peut imaginer un petit « personnage » se promenant sur la droite des réels en faisant des pas de longueur 1 et qui continue à avancer *tant qu'* il n'a pas dépassé le réel.

## B5 Développement en fractions continues

### B5 a Pratique du développement

– Vous connaissez l'algorithme suivant :

$$172 = 3 \times 51 + 19 \quad (5.1)$$

$$51 = 2 \times 19 + 13 \quad (5.2)$$

$$19 = 1 \times 13 + 6 \quad (5.3)$$

$$13 = 2 \times 6 + 1 \quad (5.4)$$

$$6 = 6 \times 1 + 0 \quad (5.5)$$

– On peut donc facilement compléter la suite d'égalité suivante :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = \dots$$

– Quand tous les numérateurs sont égaux à 1, on dit qu'on a développé  $\frac{172}{51}$  en fraction continue et pour simplifier l'écriture on note :

$$\frac{172}{51} = [3; 2; \dots]$$

– Par exemple, on peut développer  $\frac{453}{54}$  en fraction continue.

– Dans l'autre sens on peut écrire  $[2; 5; 4]$  sous la forme d'une fraction irréductible.

Nous allons construire les algorithmes correspondant.

### B5 b Sans affectation

On suppose connus les algorithmes donnant le reste et le quotient d'une division euclidienne.

```

fc(a,b)
si b=0 alors
  | retourner liste vide
sinon
  | retourner [quotient(a,b),fc(b,reste(a,b))]

```

En français en XCAS :

```

fc(a,b):={
  si b==0 alors []
  sinon concat(iquo(a,b),fc(b,irem(a,b)))
  fsi
};;

```

Programme 39 – fractions continues en récursif

En anglais :

```

fc(a,b):={
  if(b==0)then{[]}
  else{concat(iquo(a,b),fc(b,irem(a,b)))}
};;

```

Programme 40 – partie entière en récursif (VO)

La commande `concat(élément, liste)` ajoute `élément` au début de `liste`.

Alors `fc(172, 51)` renvoie `[3, 2, 1, 2, 6]`

En CAML

```

# let rec fc (a,b) =
  if b = 0
  then []
  else a/b :: fc(b, a-a/b*b);;

```

Programme 41 – partie entière en récursif avec CAML

Pour ajouter un élément à une liste sur CAML, la syntaxe est `élément :: liste`.

Avec Sage/Python :

```

sage: def fc(a,b):
  ....: if b==0 : return []
  ....: else : return [a//b]+fc(b, a%b)

```

Programme 42 – partie entière en récursif (Sage)

La concaténation des listes sur Sage/Python se fait avec `+`; le quotient entier s'obtient avec `//` et le reste entier avec `%`. L'algorithme est très simple à écrire et à comprendre.



**B 5 c Avec affectation****Entrées** : 2 entiers a et b**Initialisation** :

num ← a

den ← b

res ← reste(num,den)

Liste ← vide

**début**  **tant que** res ≠ 0 **faire**

Liste ← Liste, quotient(num,den)

num ← den

den ← res

res ← reste(num,den)

**fin****retourner** [Liste, quotient(num,den)]

En français avec XCAS :

```

frac_cont(a,b):={
local num,den,res,Liste;
num:=a;
den:=b;
res:=irem(num,den);
Liste:=NULL;
  tantque res>0 faire
    Liste:=Liste,iquo(num,den);
    num:=den;
    den:=res;
    res:=irem(num,den);
  ftantque
retourne([Liste,iquo(num,den)]);
};

```

Programme 43 – partie entière en impératif

En anglais :

```

frac_cont(a,b):={
local num,den,res,Liste;
num:=a;
den:=b;
res:=irem(num,den);
Liste:=NULL;
while(res>0){
  Liste:=Liste,iquo(num,den);
  num:=den;
  den:=res;
  res:=irem(num,den);
}
return([Liste,iquo(num,den)]);
};

```

Programme 44 – partie entière en impératif (VO)

Avec Sage/Python :

```

sage: def fc(a,b):
....:     num=a
....:     den=b
....:     res=num%den
....:     Liste=[]

```

```

.....: while res>0:
.....:     Liste+=[num//den];
.....:     num=den;
.....:     den=res;
.....:     res=num%den;
.....:     return Liste+[num//den]

```

Programme 45 – partie entière en impératif (Sage)

Ici, la manipulation de plusieurs niveaux d'affectation et d'une boucle while est assez délicate. La chronologie des affectations est primordiale : on ne peut proposer cette version à des débutants...

## B6 Dichotomie

Pour résoudre une équation du type  $f(x) = 0$ , on recherche graphiquement un intervalle  $[a, b]$  où la fonction semble changer de signe.

On note ensuite  $m$  le milieu du segment  $[a, b]$ . On évalue le signe de  $f(m)$ .

Si c'est le même que celui de  $f(a)$  on remplace  $a$  par  $m$  et on recommence. Sinon, c'est  $b$  qu'on remplace et on recommence jusqu'à obtenir la précision voulue.

### B6a Sans affectation

On appellera  $\text{eps}$  la précision voulue.

```

si b-a est plus petit que la précision eps alors
| (b+a)/2
sinon
| si f(a) et f((b+a)/2) sont de même signe alors
| | dico_rec(f,(b+a)/2,b,eps)
| sinon
| | dico_rec(f,a,(b+a)/2,eps)

```

On remarquera ici que la récursion se fait sans faire intervenir d'entier : le test d'arrêt est effectué sur la précision du calcul. On va rajouter justement un compteur pour savoir combien de calculs ont été effectués.

En français avec XCAS :

```

dicho_rec(f,a,b,eps,compteur):={
si evalf(b-a)<eps alors 0.5*(b+a),compteur+1
sinon si f(a)*f(0.5*(b+a))>0
alors dico(f,0.5*(b+a),b,eps,compteur+1)
sinon dico(f,a,0.5*(b+a),eps,compteur+1)
fsi
};

```

Programme 46 – dichotomie en récursif

```

dicho_rec(f,a,b,eps,compteur):={
if(evalf(b-a)<eps){0.5*(b+a),compteur+1};
if(f(a)*f(0.5*(b+a))>0){dicho(f,0.5*(b+a),b,eps,compteur+1)}
else{dicho(f,a,0.5*(b+a),eps,compteur+1)}
};

```

Programme 47 – dichotomie en récursif (VO)

En CAML :

```

# let rec dico_rec(f,a,b,eps,compteur)=
if abs_float(b-.a)<eps then 0.5*(b+.a),compteur
else if f(a)*.f(0.5*(b+.a))>0. then dico_rec(f,0.5*(b+.a),b,eps,compteur+1)
else dico_rec(f,a,0.5*(b+.a),eps,compteur+1);;

```

## Programme 48 – dichotomie en récursif avec CAML

ce qui donne :

```
# dico_rec( (fun x->(x*.x-.2.)),1.,2.,0.00001,0) ;;
- : float * int = (1.41421127319335938,17)
```

Avec Sage/Python :

```
sage: def dico(f,a,b,p,compteur):
....:     if abs(b-a).n(digits=p)<10**(-p) : return [0.5*(b+a).n(digits=p+len(str(floor(a)))),
compteur]
....:     elif f(a)*f(0.5*(b+a))>0 : return dico(f,0.5*(b+a).n(digits=p+len(str(floor(a)))),b,p,
compteur+1)
....:     else : return dico(f,a,0.5*(b+a).n(digits=p+len(str(floor(a)))),p,compteur+1)
....:
```

## Programme 49 – dichotomie en récursif (Sage/Python)

Alors :

```
sage: def f(x): return x**2-2
....:
sage: dico(f,1,2,5,0)
[1.41421, 17]

sage: def f(x): return x**2-153
....:
sage: dico(f,10,20,5,0)
[12.36931, 20]
```

Le rajout de `len(str(floor(a)))` permet d'adapter au nombre de chiffres de la partie entière.

## B 6 b Avec affectation

**Entrées :** une fonction  $f$ , les bornes  $a$  et  $b$ , une précision  $p$

**Initialisation :**  $aa \leftarrow a$ ,  $bb \leftarrow b$

**début**

**tant que**  $bb-aa > p$  **faire**

**si** *le signe de  $f((aa+bb)/2)$  est le même que celui de  $f(bb)$*  **alors**

$bb \leftarrow (aa+bb)/2$

**sinon**

$aa \leftarrow (aa+bb)/2$

**retourner**  $(aa+bb)/2$

**fin**

Avec XCAS, il faut juste ne pas oublier de régler quelques problèmes de précision. On rajoute aussi pour le plaisir un compteur qui nous dira combien de « tours de boucles » le programme aura effectué.

```
dico(F,p,a,b):={
local aa,bb,k,f,c;
aa:=a;
bb:=b;
epsilon:=1e-100;
f:=unapply(F,x);
compteur:=0;
while(evalf(bb-aa,p)>10^(-p)){
if( f(0.5*(bb+aa))*f(bb)>0 )
then{bb:=evalf((aa+bb)/2,p+1)}
```

```

else{aa:=evalf((aa+bb)/2,p+1)}
k:=k+1;
}
return evalf((bb+aa)/2,p+1)+" est la solution trouvee apres " +(k+1)+ "iterations";
};;

```

Programme 50 – dichotomie en impératif (XCAS)

En reprenant le même exemple :

```
dicho(x^2-2,5,1,2)
```

On obtient la même réponse :

1.414211273 est la solution trouvee apres 18 iterations

Avec Sage/Python :

```

sage: def dichotomie(f,a,b,p):
....:     [aa,bb,compteur,c]=[a,b,0,len(str(floor(a)))]
....:     while (bb-aa).n(digits=p)>10**(-p):
....:         if f(0.5*(bb+aa))*f(bb)>0 :
....:             bb=(0.5*(aa+bb)).n(digits=p+c)
....:         else :
....:             aa=(0.5*(aa+bb)).n(digits=p+c)
....:             compteur=compteur+1
....:     return [(0.5*(aa+bb)).n(digits=p+c),compteur]

```

Programme 51 – dichotomie en impératif (Sage)

Les deux versions sont ici assez similaires avec toujours peut-être une plus grande simplicité d'écriture dans le cas récursif.

## C Algorithme récursif ne signifie pas suite récurrente

Si les suites définies par une relation  $u_{n+1} = f(u_n)$  sont facilement décrites par un algorithme récursif, ce n'est pas le seul champ d'action de la récursion.

Dès qu'on connaît une situation simple et qu'on est capable de simplifier une situation compliquée, on peut utiliser la récursion.

### C1 Longueur d'une liste

Par exemple, si on veut calculer la longueur d'une liste :

- la liste vide est de longueur nulle (cas simple) ;
- la longueur d'une liste est égale à 1+la longueur de la liste sans son premier élément.

Cela donne en traduction directe avec CAML :

```

# let rec longueur = function
| [] -> 0
| tete::corps -> 1+longueur(corps);;

```

Programme 52 – longueur d'une liste en récursif (CAML)

La barre verticale permet de définir une fonction par morceaux en distinguant les cas.

L'écriture `x::liste` signifie qu'on rajoute `x` en tête de la liste `liste`.

CAML trouvant lui-même le type des variables utilisées sait qu'on va lui entrer une liste de n'importe quoi (puisqu'on peut l'écrire sous la forme `tete` rajoutée à `corps`) et que `longueur` sera un entier (puisqu'on lui ajoute 1).

C'est mathématique...

Avec Sage :

```
sage: def longueur(L):
.....:     if L==[]: return 0
.....:     else: del L[0]; return(1+longueur(L))
```

Programme 53 – longueur d’une liste en récursif (Sage)

## C2 Somme d’entiers

Supposons que nous ne sachions que trouver le successeur ou le prédécesseur d’un entier. Comment additionner deux entiers quelconques ?

Avec CAML :

```
# let rec add = function
| (0,x) -> x
| (x,y) -> add((x-1),y)+1;;
```

Programme 54 – somme d’entiers en récursif (CAML)

Additionner  $x$  et  $y$  c’est prendre le successeur de l’addition du prédécesseur de  $x$  et de  $y$ ...c’est-à-dire :  $x + y = ((x - 1) + y) + 1$  sachant que  $0 + x = x$ .

Avec Sage :

```
sage: def add(x,y):
.....:     if x==0 : return y
.....:     else : return add((x-1),y)+1
.....:
sage: add(3,4)
7
```

Programme 55 – somme d’entiers en récursif (Sage)

## D Alors : avec ou sans affectation ?

Beaucoup de professeurs (et donc d’élèves...) sont habitués à traiter certains problèmes à l’aide d’un tableur...qui fonctionne récursivement !

En effet, pour étudier par exemple les termes de la suite  $u_{n+1} = 3u_n + 2$  de  $n = 1$  à  $n = 30$  avec  $u_1 = 7$ , on entre = 7 dans la cellule A1 et = 3 \* A29 + 2 dans la cellule A30 et on « fait glisser » de la cellule A30 jusqu’à la cellule A2. On a plutôt l’habitude de faire glisser vers le bas mais pourquoi pas vers le haut !

Passer d’un « faire glisser » à l’écriture de l’algorithme correspondant est une évolution naturelle ce que l’introduction d’un langage impératif n’est pas.

La notion de fonction est naturellement illustrée par l’utilisation d’un langage fonctionnel alors que les procédures des langages impératifs peuvent prêter à confusion.

Comme de nombreux collègues, j’assure des « colles MAPLE » en classes préparatoires scientifiques depuis plusieurs années. Les élèves de ces classes ont habituellement un niveau en mathématiques supérieur au niveau moyen rencontré en Seconde et pourtant la plupart des étudiants éprouve de réelles difficultés à maîtriser un langage de programmation impératif comme MAPLE et certains n’y arrivent pas après une année d’initiation !

Évidemment, nous n’aborderons pas les mêmes problèmes en Seconde mais pouvoir se concentrer sur les mathématiques et simplement traduire nos idées grâce au clavier de manière assez directe nous permettra de préserver nos élèves de nombreux pièges informatiques sans parler des notions trompeuses que l’affectation peut introduire dans leurs esprits.

Cependant, utiliser des algorithmes récursifs nécessite de maîtriser un tant soit peu... la notion de récurrence. Or cette étude a été réservée jusqu’à maintenant aux seuls élèves de Terminale S. L’étude des suites débute depuis de nombreuses années en Première et chaque professeur a fait l’expérience des difficultés des élèves à maîtriser cette notion.

Ainsi, même si (ou plutôt parce que) les algorithmes récursifs peuvent apparaître proches des notions mathématiques qu’ils permettent d’étudier, ils demandent de comprendre une notion mathématique délicate, la récurrence, même si l’étude des phénomènes évolutifs modélisés par une relation  $u_{n+1} = f(u_n)$  a failli entrer au programme de Seconde !...

Il faut surtout noter que le champ d’action des algorithmes récursifs dépasse largement la notion de suite numérique. On peut raisonner récursivement dès qu’on connaît un état simple et qu’on peut simplifier un état compliqué.

D'un autre côté, les algorithmes utilisant les affectations contournent la difficulté de la récurrence, facilitent l'étude de certains mécanismes comme le calcul matriciel, permettent souvent une illustration « physique » d'un phénomène mathématique et en tous cas complémentaire des versions sans affectation.

Ils doivent de plus être étudiés car ils sont largement répandus dans le monde informatique.

Il ne faut enfin pas oublier les difficultés que pourraient rencontrer des professeurs de mathématiques n'ayant pas réfléchi à ces problèmes d'algorithmique et qui ne pourront pas être formé(e)s avant de devoir enseigner une notion méconnue, sans manuels et sans le recul nécessaire.

Mais il ne faudrait pas pour autant abandonner l'idée d'initier les élèves à une « algorithmique raisonnée ». La notion de test par exemple (SI...ALORS...SINON) est tout à fait abordable par le plus grand nombre... et dans tout style de programmation. La notion de fonction peut être étudiée sans danger avec un langage fonctionnel et avec prudence avec un langage impératif. D'un point de vue plus informatique, faire découvrir aux élèves que le « dialogue » avec la machine ne se réduit pas à des « clics » mais que chacun peut « parler » presque directement à la machine en lui tapant des instructions est également un apprentissage enrichissant.

Quant au match avec ou sans affectation, il ne peut se terminer par l'élimination d'un des participants : il faut savoir que l'un ou l'autre sera plus adapté à la résolution d'un problème et l'emploi **combiné** des deux types de programmation abordés dans cet article peut surtout permettre d'illustrer sous des angles différents une même notion mathématique et donc illustrer de manière enrichissante nos cours, ce qui induit la nécessité de connaître différentes manières de procéder pour choisir la plus efficace. Cela accroît d'autant plus la difficulté d'une introduction improvisée de l'algorithmique au lycée...

Mais les difficultés rencontrées par les étudiants débutants de l'enseignement supérieur doivent nous inciter à la prudence et à nous demander si un enseignement informatique optionnel en seconde et plus poussé dans les classes scientifiques supérieures ne serait pas plus souhaitable. On pourra lire à ce sujet l'opinion de Bernard PARISSE, développeur de XCAS<sup>d</sup>.

---

d. <http://www-fourier.ujf-grenoble.fr/parisse/irem.html#algo>

# 6 - Outils mathématiques basiques

## A Valeur absolue

### A1 Prérequis

Test « si...alors...sinon ».

### A2 Algorithme

On peut définir la valeur absolue d'un réel  $x$  de la manière suivante, même si ce n'est pas l'usage en seconde :

```
Entrées :  $x$  (un réel)
début
  si  $x$  positif alors
    retourner  $x$ 
  sinon
    retourner  $(-x)$ 
fin
```

Algorithme 5 : valeur absolue

### A3 Traduction XCAS

```
va(x):={
  si x>0 alors
    retourne(x) sinon
    retourne(-x)
  fsi;
}
```

Programme 56 – valeur absolue

en anglais :

```
va(x):={
  if(x>0){return(x)}
  else{return(-x)}
}
```

Programme 57 – valeur absolue (VO)

en plus compact :

```
va(x):={
  ifte(x>0, x, -x)
}
```

## A4 En CAML

```
# let va(x)=
  if x>0 then x else -x;;
```

Programme 58 – valeur absolue (CAML)

## A5 Avec Sage/Python

```
sage: def va(x):
....:   if x>0 : return x
....:   else : return -x
```

Programme 59 – valeur absolue (Sage)

## B Partie entière

### B1 Intérêt

La partie entière n'est pas explicitement au programme de seconde mais sa détermination algorithmique enrichit l'étude des ensembles de nombres.

On peut définir la partie entière d'un réel  $x$  comme étant le plus grand entier inférieur à  $x$ .

### B2 Algorithme récursif

On part du fait que la partie entière d'un nombre appartenant à  $[0; 1[$  est nulle.

Ensuite, on « descend » de  $x$  vers 0 par pas de 1 si le nombre est positif en montrant que  $\lfloor x \rfloor = 1 + \lfloor x - 1 \rfloor$ .

Si le nombre est négatif, on « monte » vers 0 en montrant que  $\lfloor x \rfloor = -1 + \lfloor x + 1 \rfloor$ .

#### B2 a Avec XCAS

```
partie_entiere_rec(r):={
  si(r>=0) et (r<1)
  alors 0
  sinon si r>0
    alors 1+partie_entiere_rec(r-1)
    sinon -1+partie_entiere_rec(r+1)
  fsi
};;
```

Programme 60 – partie entier en récursif

En anglais :

```
partie_entiere_rec(r):={
  if((r>=0) and (r<1)){0}
  else{if(r>0){1+partie_entiere_rec(r-1)}
  else{-1+partie_entiere_rec(r+1)}
}
};;
```

Programme 61 – partie entière en récursif (VO)



**B 2 b Avec CAML**

```
# let rec partie_entiere (r)
=
  if r >= 0. && r < 1.
  then 0.
  else if r > 0.
    then 1. +. partie_entiere (r -. 1.)
    else -.1. +. partie_entiere (r +. 1.)
```

Programme 62 – partie entière en récursif (CAML)

**B 2 c Avec Sage/Python**

Avec Sage/Python :

```
sage: def partie_entiere(x):
....:   if x>=0 and x<1 : return 0
....:   elif x>0 : return 1+partie_entiere(x-1)
....:   else : return -1+partie_entiere(x+1)
```

Programme 63 – partie entière récursive avec Sage

**B 3 Algorithme impératif**

On peut commencer par se restreindre aux nombres positifs. On part de 0. Tant qu'on n'a pas dépassé  $x$ , on avance d'un pas. La boucle s'arrête dès que  $k$  est strictement supérieur à  $x$ . La partie entière vaut donc  $k - 1$ .

**Entrées :**  $x$ (réel positif)  
**Initialisation :**  $k \leftarrow 0$   
**début**  
 | **tant que**  $k \leq x$  **faire**  
 |    $k \leftarrow k+1$   
 | **retourner**  $k-1$   
**fin**

**Algorithme 6 :** partie entière d'un nombre positif

Dans le cas d'un réel négatif, il faut cette fois reculer d'un pas à chaque tour de boucle et la partie entière est la première valeur de  $k$  à être inférieure à  $x$  :

**Entrées :**  $x$ (réel)  
**Initialisation :**  $k \leftarrow 0$   
**début**  
 | eSi **tant que**  $k \leq x$  **faire**  
 |    $k \leftarrow k+1$   
 | **retourner**  $k-1$  **tant que**  $k > x$  **faire**  
 |    $k \leftarrow k-1$   
 | **retourner**  $k$   
**fin**

**Algorithme 7 :** partie entière d'un réel quelconque**B 3 a Traduction XCAS**

```
pe(x):={
local k;
k:=0;
si x>=0 alors
  tantque k<=x faire
```

```

    k:=k+1;
  ftantque;
  retourne(k-1);
sinon
  tantque k>x faire
    k:=k-1;
  ftantque;
  retourne(k);
fsi;
};

```

Programme 64 – partie entière en impératif

En anglais :

```

pe(x) := {
  local k;
  k:=0;
  if(x>=0){
    while(k<=x){k:=k+1};
    return(k-1);
  }
  else{
    while(k>x){k:=k-1};
    return(k);
  }
};

```

Programme 65 – partie entière en impératif (VO)

### B 3 b Avec Sage/Python

```

>>> def pe(x):
    k=0
    if x>=0:
        while k<=x : k+=1
        return k-1
    else :
        while k>x : k+=-1
        return k

```

Programme 66 – partie entière impérative (Sage/Python)

## C Arrondis

On veut arrondir un nombre réel à  $10^{-2}$  près par défaut.

### C1 Algorithme

On va prendre la partie entière (avec l'algorithme créé précédemment) du nombre multiplié par  $10^2$  puis le rediviser par  $10^2$  :

```

Entrées : x(réel)
début
  | retourner (partie entière de 100x) divisée par 100
fin

```

Algorithme 8 : arrondi

## C2 Traduction XCAS

```
arrondi(x) := {
  retourne (pe(100*x)/100.0)
}
```

Programme 67 – arrondi au centième

## C3 Traduction CAML

```
let arrondi_au_100_eme_par_defaut (r)
=
partie_entiere (100. *. r) /. 100.
```

Programme 68 – arrondi au centième avec CAML

## D Variante

On veut arrondir à  $10^{-2}$  près par défaut si la partie décimale est strictement inférieure à 0,5 et à  $10^{-2}$  près par excès si la partie décimale est supérieure à 0,5.

La recherche de la partie décimale peut poser des problèmes. On commence donc par traiter le cas des nombres positifs :

```
Entrées : x (réel positif)
Initialisation : deci ← (x - partie entière de x)
début
  si deci < 0,5 alors
    | retourner (partie entière de 100x) divisée par 100
  sinon
    | retourner 0,01 + (partie entière de 100x) divisée par 100
fin
```

Algorithme 9 : arrondi au plus proche

On peut ensuite chercher à généraliser à un réel quelconque : beaucoup d'élèves devraient se tromper ici...ce qui est formateur!

## E Calculs de sommes

On veut par exemple calculer la somme des entiers de 1 à n.

### E1 Algorithme récursif

#### E1a Avec XCAS

```
som_ent(n) := {
  if(n==0){0}
  else{n+som_ent(n-1)}
};;
```

Programme 69 – somme des entiers en récursif

**E 1 b Avec CAML**

```
# let rec som_ent(n)=
  if n=0 then 0
  else n+som_ent(n-1);;
```

Programme 70 – somme des entiers en récursif avec CAML

Par exemple :

```
# som_ent(100000);;
- : int = 705082704
```

On pense bien sûr à généraliser cette procédure à n'importe quelle somme du type  $\sum_{k=k_0}^n f(k)$  :

```
# let rec som_rec(f,ko,n)=
  if n=ko then f(ko)
  else f(n)+som_rec(f,ko,n-1);;
```

Programme 71 – somme des images des entiers par une fonction (CAML)

**E 1 c Avec SAGE**

```
sage: def som_ent(n):
.....:   if n==1 : return 1
.....:   else : return n+som_ent(n-1)
```

Programme 72 – somme des entiers avec Sage

```
sage: def som_rec(f,ko,n):
.....:   if n==ko : return f(ko)
.....:   else : return f(n)+som_rec(f,ko,n-1)
```

Programme 73 – somme des images des entiers par une fonction avec Sage

**E 2 Algorithme impératif**

```
Entrées : n(entier naturel)
Initialisation : S ← 0 // la somme est nulle au départ
début
  | pour k de 1 à n faire
  |   | S ← S+k
  |   retourner S
fin
```

Algorithme 10 : somme des entiers de 1 à n

**E 2 a Avec XCAS**

```
som(n):={
local S;
S:=0;
pour k de 1 jusque n faire
  S:=S+k;
fpour;
```

```
retourne(S)
}
```

Programme 74 – somme des entiers en impératif

en anglais :

```
som(n) := {
  local S;
  S := 0;
  for (k := 1; k <= n; k := k + 1) { S := S + k };
  return(S)
}
```

Programme 75 – somme des entiers en impératif (VO)

## E 2 b Avec Sage

```
sage: def som(n):
.....: S=0
.....: for k in [1..n] : S+=k
.....: return S
```

Programme 76 – somme des entiers version impérative avec Sage

## F Mini et maxi

### F 1 Version récursive

Un petit travail sur les valeurs absolues définies au paragraphe A page 36 permet de montrer que :

$$\max(x, y) = \frac{x + y + |x - y|}{2} \quad \min(x, y) = \frac{x + y - |x - y|}{2}$$

Maintenant, on voudrait trouver les extrema d'une liste de nombres.

```
si liste n'a que deux éléments
alors
  | max(les deux éléments)
sinon
  | max(tête de liste, maxi(liste sans sa tête))
```

Algorithme 11 : maxi(liste)

Avec CAML, on utilise `List.hd` qui donne la tête, c'est-à-dire le premier élément d'une liste, et `List.tl` qui donne sa queue, c'est-à-dire la liste privée de sa tête :

```
# let absi(x)=
  if x>0 then x
  else -x;;

val absi : int -> int = <fun>

# let maxi(x,y)=
  (x+y+absi(x-y))/2;;

val maxi : int * int -> int = <fun>

# let rec maxlist(lx)=
  if List.tl(List.tl(lx))=[]
```

```

then maxi(List.hd(lx),List.hd(List.tl(lx)))
else maxi(List.hd(lx),maxlist(List.tl(lx)));;

    val maxlist : int list -> int = <fun>

# maxlist([1;2;3;54;5;3;8;7;98;7;4;2;9]);;

- : int = 98

```

Programme 77 – maximum d'une liste en récursif (1)

Voici une variante évitant le recours au module « List » :

```

# let rec maxilist = function
  | x::y::[] -> maxi(x,y)
  | x::queue -> maxi(x,maxilist(queue));;

```

Programme 78 – maximum d'une liste en récursif (2)

Quand on écrit `x::y::[]`, CAML comprend qu'on a rajouté deux éléments à la liste vide : il s'agit donc d'une liste de deux éléments...

Voici une variante sans utiliser `maxi` et en créant un « garde-fou » :

```

# exception Liste_vide;;

# let rec maxbis = function
  | [] -> raise Liste_vide
  | x::[] -> x
  | x::y::queue -> if x>y then maxbis(x::queue)
                  else maxbis(y::queue);;

```

Programme 79 – maximum d'une liste en récursif (3)

Avec XCAS :

```

maxlist(lx):={
  if(size(lx)==2)
  then{maxi(lx[0],lx[1])}
  else{maxi(lx[0],maxlist(tail(lx)))}
};;

```

Programme 80 – maximum d'une liste en récursif (XCAS)

Avec Sage/Python :

```

sage: def maxbis(lx):
....:   if len(lx)==1 : return lx[0]
....:   elif lx[0]>lx[1] : del lx[1]; return maxbis(lx);
....:   else : del lx[0]; return maxbis(lx);
....:
sage: maxbis([1,2,-1,6,87,2,-10])
87

```

Programme 81 – maximum d'une liste en récursif (Sage)

## F2 Version impérative

**Entrées** : liste

**Initialisation** :  $\text{maxloc} \leftarrow$  tête de la liste

**début**

**pour**  $k$  de 1 jusqu'à la taille de la liste **faire**

$\text{maxloc} \leftarrow$  le plus grand entre  $\text{maxloc}$  et  $k$ -eme élément de la liste

**fin**

**retourner**  $\text{maxloc}$

**Algorithme 12** : maximum : version impérative

Avec XCAS :

```
maxlist_imp(lx):={
local maxloc,k;
maxloc:=lx[0];
  for(k:=1;k<size(lx);k:=k+1){
    maxloc:=maxi(maxloc,lx[k])
  }
return(maxloc)
};;
```

Programme 82 – maximum d'une liste en impératif (XCAS)

```
sage: def max_imp(lx):
....:   maxloc=lx[0]
....:   for k in [1..len(lx)-1]:
....:     if lx[k]>maxloc : maxloc=lx[k]
....:   return(maxloc)
```

Programme 83 – maximum d'une liste en impératif (Sage)

# 7 - Algorithmes de calculs « à la STG »

Les sections STG sont initiées à l'algorithmique en travaillant sur des problèmes simples de gestion.

## A Taux de remise variable

On entre un montant HT  $ht$ . On effectue une remise sur  $ht$  selon la règle suivante :

- si  $ht < 2500\text{€}$  alors il n'y a pas de remise ;
- si  $2500 \leq ht < 4000\text{€}$  alors la remise est de 5% ;
- dans les autres cas la remise est de 8%.

La TVA est de 19,6%. On demande le prix TTC en fonction du prix HT.

### A1 Algorithme

```
Entrées : HT(réel positif)
Initialisation : ht ← HT
début
  si ht ≥ 2500 et ht < 4000 alors
    ht ← ht × 0,95
  si ht ≥ 4000 alors
    ht ← ht × 0,92
  retourner ht × 1.196
fin
```

Algorithme 13 : prix TTC selon remise

### A2 Avec XCAS

```
remise(HT) := {
  local ht;
  ht := HT;
  si (ht ≥ 2500) et (ht < 4000) alors ht := ht * 0.95; fsi;
  si ht ≥ 4000 alors ht := ht * 0.92; fsi;
  retourne(ht * 1.196)
}
```

Programme 84 – taux de remise variable en impératif

En anglais :

```
remise(HT) := {
  local ht;
  ht := HT;
  if (ht ≥ 2500) and (ht < 4000) {ht := ht * 0.95};
  if(ht ≥ 4000) {ht := ht * 0.92};
  return(ht * 1.196)
}
```



Programme 85 – taux de remise variable en impératif (VO)

### A3 Avec CAML

```
let prix_ttc_apres_remise (prix_hors_taxe)
=
  let
    taux_remise = if prix_hors_taxe < 2500.
                  then 0.
                  else if prix_hors_taxe < 4000.
                  then 0.05
                  else 0.08
  in
    (1. -. taux_remise) *. prix_hors_taxe *. 1.196
```

Programme 86 – taux de remise variable en récursif (CAML)

On remarque au passage comment définir une fonction locale imbriquée dans une autre fonction.

### A4 Avec Sage

```
sage: def remise(HT):
.....:   ht=HT
.....:   if ht>=2500 and ht<4000 : ht=ht*0.95
.....:   if ht>=4000 : ht=ht*0.92
.....:   return ht*1.196
```

Programme 87 – taux de remise variable (Sage)

# 8 - Algorithmes en arithmétique

## A Division euclidienne

### A1 Quotient de la division euclidienne

#### A1a Version récursive

```
reste(a,b)
début
  | si a<b alors
  |   L retourner 0
fin
retourner 1+quotient(a-b,b)
```

Algorithme 14 : Quotient de la division euclidienne : version récursive

#### A1b Avec XCAS

```
quotient(a,b):={
if(a<b){return(0)};
return(1+quotient(a-b,b));
}
```

Programme 88 – quotient euclidien en récursif (XCAS)

#### A1c Avec Sage

```
sage: def quotient(a,b):
....:   if a<b : return 0
....:   return 1+quotient(a-b,b)
```

Programme 89 – quotient euclidien en récursif (Sage)

#### A1d Avec CAML

```
# let rec quotient (a,b)
=
  if a<b
  then 0
  else 1+quotient(a-b,b);;
```

Programme 90 – quotient euclidien en récursif (CAML)

### ♪ Remarque 1 : reste et quotient en CAML

CAML étant intelligemment typé, si on entre :

```
# 37/5;;
```

On obtient

```
- : int = 7
```

c'est-à-dire par défaut le quotient entier car on travaille avec des entiers.

Pour avoir une approximation décimale, il faut entrer :

```
# 37./5.;;
```

et on obtient :

```
- : float = 7.4
```

Pour le reste, on peut donc entrer :

```
# 37-37/5*5;;
```

ou utiliser la commande infixée `mod`

```
# 37 mod 2 ;;
```

## A 1 e Version impérative

**Entrées :** 2 entiers  $a$  et  $b$

**début**

**Initialisation :**  $k \leftarrow 0$

**tant que**  $k \times b \leq a$  **faire**

$k \leftarrow k+1$

**fin**

**retourner**  $k-1$

Algorithme 15 : Quotient de la division euclidienne

## A 1 f Avec XCAS

```
quotient(a,b):={
local k;
k:=0;
while(k*b<=a){k:=k+1}
return(k-1)
}
```

Programme 91 – quotient euclidien en impéartif (XCAS)

## A 1 g Avec Sage

```
sage: def quotient(a,b):
.....:     k=0
.....:     while k*b<=a : k+=1
.....:     return(k-1)
```

Programme 92 – quotient euclidien en impéartif (Sage)

## A 2 Reste de la division euclidienne

**A 2 a** Version récursive

```

reste(a,b)
début
  | si a<b alors
  |   L retourner a
fin
retourner reste(a-b,b)

```

Algorithme 16 : Reste de la division euclidienne : version récursive

**A 2 b** Version récursive avec XCAS

```

reste(a,b) := {
  if(a<b){return(a)};
  return(reste(a-b,b));
}

```

Programme 93 – reste euclidien en récursif

**A 2 c** Version récursive avec CAML

```

# let rec reste (a,b)
=
  if a<b
  then a
  else reste(a-b,b);;

```

Programme 94 – reste euclidien en récursif (CAML)

**A 2 d** Version récursive avec Sage/Python

```

sage: def reste(a,b):
.....:   if a<b : return a
.....:   return reste(a-b,b)

```

Programme 95 – reste euclidien en récursif (Sage)

**A 2 e** Version impérative

```

Entrées : 2 entiers a et b
début
  | Initialisation : k ← 0
  | tant que k × b ≤ a faire
  |   L k ← k+1
fin
retourner a - b(k-1)

```

Algorithme 17 : Reste de la division euclidienne : version impérative

**A 2 f** Version impérative avec XCAS

```
reste(a,b) := {
  local k;
  k:=0;
  while (k*b <= a) {k:=k+1}
  return (a-b*(k-1))
}
```

Programme 96 – reste euclidien en impératif (XCAS)

### A 2 g Version impérative avec Sage/Python

```
sage: def reste(a,b):
.....:     k=0
.....:     while k*b <= a : k+=1
.....:     return a-b*(k-1)
```

Programme 97 – reste euclidien en impératif (Sage)

### A 3 Divisibilité

On peut créer une fonction qui testera si un nombre divise un autre.

```
Entrées : 2 entiers n et p
début
| (quotient entier n/p) × p == n
fin
```

Algorithme 18 : Test de divisibilité

On effectue un test booléen en utilisant l'opérateur infix `==` qui désigne la relation d'égalité. La réponse du programme sera donc « vrai » ou « faux » (« 1 » ou « 0 » sur XCAS et « true » ou « false » sur CAML)

#### A 3 a Avec XCAS

```
divise(p,n) := {iquo(n,p)*p=n};;
```

Programme 98 – divisibilité (XCAS)

#### A 3 b Avec CAML

```
# let divise p n = ((n/p)*p=n);;
```

Programme 99 – divisibilité (CAML)

Qui s'utilise ainsi :

```
# divise 4 13;;
- : bool = false
# divise 4 12;;
- : bool = true
```

**A 3 c Avec Sage/Python**

```
sage: def divide(p,n):
....:     return (n//p)*p==n
....:
sage: divide(3,37)
False
sage: divide(3,36)
True
```

Programme 100 – divisibilité (Sage)

**B Nombre de diviseurs d'un entier****B 1 Version récursive avec CAML**

Une méthode naïve consisterait à tester tous les entiers jusqu'à  $n$  pour compter les diviseurs de  $n$  :

```
# let nombre_div(a)=
  let rec nombre_div_local(a,n)=
    if n>a/2
      then 1
      else if a mod n = 0
        then 1+nombre_div_local(a,n+1)
        else nombre_div_local(a,n+1)
  in nombre_div_local(a,1);;
```

Programme 101 – nombre de diviseurs en récursif (1)

Par exemple, 123456789 admet 12 diviseurs :

```
# nombre_div(123456789);;
- : int = 12
```

On peut facilement être plus efficace en remarquant que les diviseurs marchent par deux!

```
# let nombre_div(a)=
  let rec nombre_div_local(a,n)=
    if n*n>a
      then 1
      else if n=1 or (a mod n=0 & n=a/n)
        then 1+nombre_div_local(a,n+1)
      else if a mod n = 0
        then 2+nombre_div_local(a,n+1)
      else nombre_div_local(a,n+1)
  in nombre_div_local(a,1);;
```

Programme 102 – nombre de diviseurs en récursif (2)

Pour 999999999 :

```
# nombre_div(999999999);;
- : int = 20
```

## B 2 Version impérative

### B 2 a Avec XCAS

Méthode naïve :

```
NbDivNaif(x):={
local t,k;
t:=0;
pour k de 1 jusque x pas 1 faire
    si irem(x,k)==0 alors t:=t+1
    fsi
fpour
retourne(t)
};
```

Programme 103 – nombre de diviseurs en impératif (1) XCAS

Il faut alors plus de 7 minutes pour compter les diviseurs de 100010010 (mille cent dix :-)

Plus efficacement !

```
NbDiv(n):={
local t,k,fin;
t:=2;
fin:=n;
pour k de 2 jusque fin pas 1 faire
    si irem(n,k)==0 alors
        fin:=n/k;
        si k==fin alors t:=t+1 sinon t:=t+2
    fsi
    fsi
fpour
retourne(t)
};
```

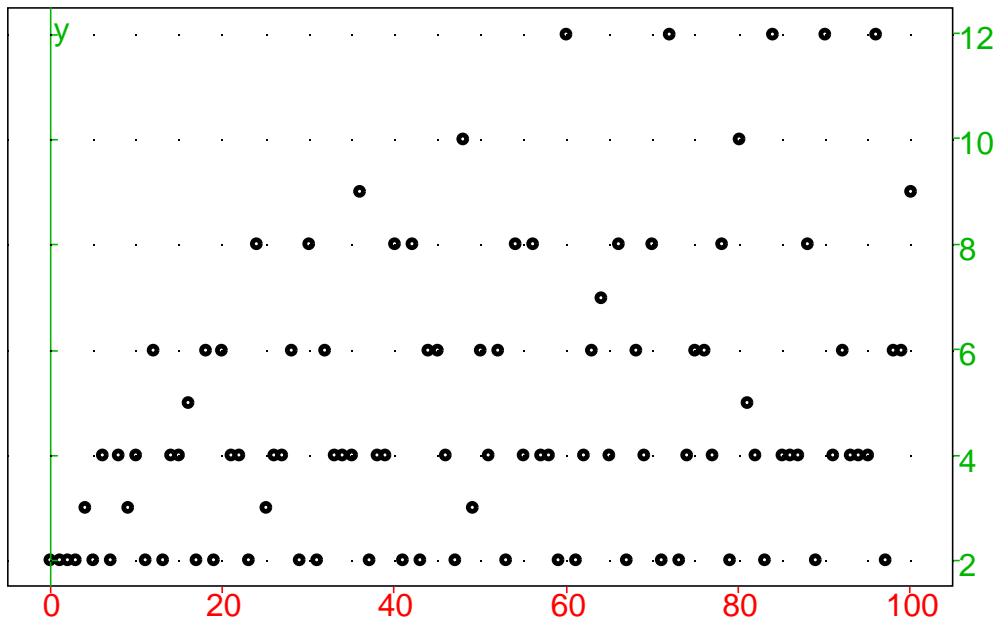
Programme 104 – nombre de diviseurs en impératif (2) XCAS

Ensuite, on peut avoir une représentation graphique :

```
Graphe_NbDiv(a,b):={
local P,k;
P:=NULL;
pour k de a jusque b pas 1 faire
    P:=P,[k,NbDiv(k)]
fpour
affichage(nuage_points([P]),point_point+epaisseur_point_3)
};
```

Programme 105 – représentation graphique du nombre de diviseurs (XCAS)

Entre 0 et 100 cela donne :



## B 2 b Avec Sage

Méthode naïve :

```
sage: def NbDivNaif(n):
....:     t=0
....:     for k in [1..n] :
....:         if n%k==0 : t+=1
....:     return t
....:
sage: NbDivNaif(1000100)
36
```

Programme 106 – nombre de diviseurs en impératif (1) Sage

Méthode plus rapide :

```
sage: def NbDiv(n):
....:     t=2
....:     fin=n
....:     k=2
....:     while k<fin :
....:         if n%k==0 :
....:             fin=n//k;
....:             if k==fin : t+=1
....:             else : t+=2
....:         k+=1
....:     return t
```

Programme 107 – nombre de diviseurs en impératif (2) Sage

Et on obtient instantanément :

```
sage: NbDiv(100010010)
32
```

Pour le tracé :

```
sage: def plot_NbDiv(a,b):
....:     P=[]
```

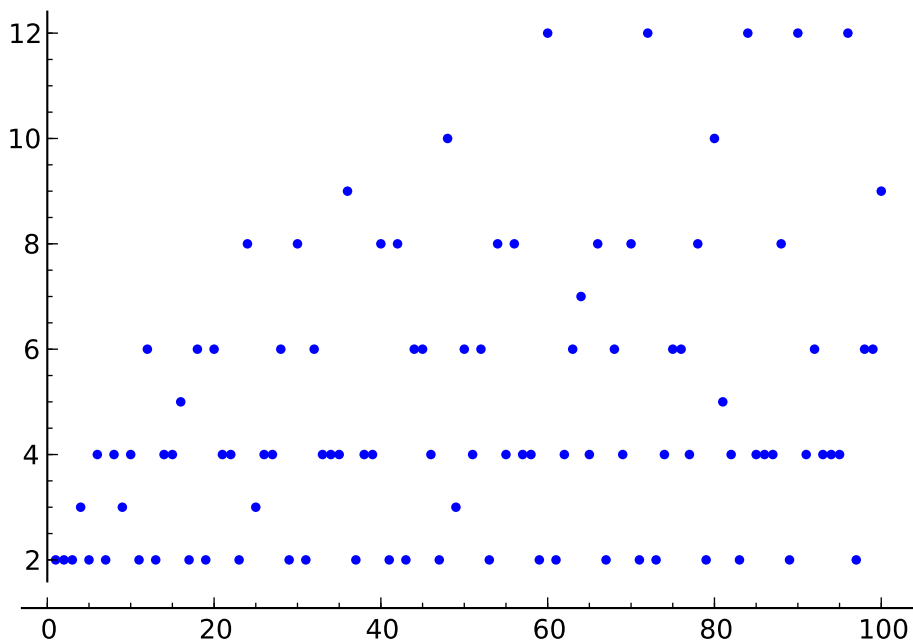


```

.....: for k in [a..b]:
.....:     P.append([k,NbDiv(k)])
.....: p=point(P)
.....: p.show()

```

Programme 108 – représentation graphique du nombre de diviseurs(Sage)



## C Nombres parfaits

Un nombre parfait est un nombre entier  $n$  strictement supérieur à 1 qui est égal à la somme de ses diviseurs (sauf  $n$  bien sûr!)

On peut adapter les programmes précédents.

### C1 En récursif avec CAML

```

# let liste_div(a)=
  let rec liste_div_n(a,n)=
    if n>a/2
    then []
    else if a mod n = 0
    then n::liste_div_n(a,n+1)
    else liste_div_n(a,n+1)
  in liste_div_n(a,1);;

```

Programme 109 – liste des diviseurs en récursif (1)

donne la liste des diviseurs.

```

# liste_div(240);;
- : int list =
[1; 2; 3; 4; 5; 6; 8; 10; 12; 15; 16; 20; 24; 30; 40; 48; 60; 80; 120]

```

On crée ensuite une fonction qui calcule la somme des termes d'une liste :

```
# let rec somme=function
| []-> 0
| tete::queue -> tete + somme(queue);;
```

Programme 110 – somme des termes d'une liste en récursif (1)

Puis on teste si un nombre est parfait :

```
# let parfait(a)=
  somme(liste_div(a))=a;;
```

Programme 111 – nombres parfaits

Ocaml répond « vrai » ou « faux » :

```
# parfait(8128);;
- : bool = true
```

On aurait pu être plus efficace en utilisant des algorithmes terminaux et en remarquant que les diviseurs vont souvent 2 par 2 et qu'il suffit de s'arrêter après la partie entière de  $\sqrt{n}$  :

```
# let liste_div(a)=
  let rec liste_div_n(a,n,res)=
    if n*n>a
    then res
    else if n=1 or (a mod n=0 & n=a/n)
    then liste_div_n(a,n+1,n::res)
    else if a mod n = 0
    then liste_div_n(a,n+1,n::a/n::res)
    else liste_div_n(a,n+1,res)
  in liste_div_n(a,1,[]);;
```

Programme 112 – liste des diviseurs d'un entier (2)

On rend somme terminal :

```
# let somme(liste)=
  let rec som_temp=function
  | ([],res)-> res
  | (tete::queue,res) -> som_temp(queue,res+tete)
  in som_temp(liste,0);;
```

Programme 113 – somme des termes d'une liste en récursif (2)

On ne change pas parfait :

```
# let parfait(a)=
  somme(liste_div(a))=a;;
```

Par exemple :

```
# parfait(33550336);;
- : bool = true
```

On peut pousser le vice jusqu'à demander de créer la liste des nombres parfaits entre 0 et un nombre  $n$  donné :  
Version non terminale :

```
# let rec liste_parfait(n)=
  if n=1 then []
  else if parfait(n) then n::liste_parfait(n-1)
  else liste_parfait(n-1);;
```

Programme 114 – liste des nombre parfait en récursif

Version terminale :

```
# let liste_parfait(n)=
  let rec liste_parfait_local(n,res)=
    if n=1 then res
    else if parfait(n)
      then liste_parfait_local(n-1,n::res)
      else liste_parfait_local(n-1,res)
  in liste_parfait_local(n,[]);;
```

Programme 115 – liste des nombres parfaits en récursif (version terminale)

Alors on trouve 4 nombres parfaits entre 0 et 100000 :

```
# liste_parfait(100000);;
- : int list = [6; 28; 496; 8128]
```

## C2 En impératif avec XCAS

On adapte un peu NbDiv pour avoir la liste des diviseurs :

```
ListeDiv(n):={
  local L,k,fin;
  L:=1,n;
  fin:=n;
  for(k:=2;k<fin;k:=k+1){
    if(irem(n,k)==0)then{
      fin:=n/k;
      if(k==fin) then{L:=L,k} else{L:=L,k,n/k};}
  }
  return([L])
};;
```

Programme 116 – Liste des diviseurs en impératif

Une petite retouche pour savoir si un entier est parfait :

```
Parfait(n):={
  local S,k,fin;
  S:=1;
  fin:=n;
  for(k:=2;k<fin;k:=k+1){
    if(irem(n,k)==0)then{
      fin:=n/k;
      if(k==fin) then{S:=S+k} else{S:=S+k+n/k};}
  }
  if(S==n)then{return("Parfait !")}
  else{return("Pas parfait...")}
};;
```

Programme 117 – nombres parfaits en impératif

Par exemple :

```
Parfait(33550336)
```

Parfait !

Il ne reste plus qu'à obtenir la liste des nombres parfaits de 0 à un  $n$  donné :

```
Liste_parfait(n):={
local k,L;
L:=NULL;
for(k:=2;k<=n;k:=k+1){
    if(Parfait(k)=="Parfait !")then{L:=L,k}
}
return([L])
};;
```

Programme 118 – liste des nombres parfaits en impératif

### C3 En impératif avec Sage

La liste des diviseurs :

```
sage: def ListeDiv(n):
....: L=[1,n]
....: fin=n
....: k=2
....: while k<fin :
....:     if n%k==0 :
....:         fin=n//k;
....:         if k==fin : L+=[k]
....:         else : L+=[k,n//k]
....:         k+=1
....:     return L
....:
sage: ListeDiv(10)
[1, 10, 2, 5]
```

Programme 119 – liste des diviseurs d'un entier (Sage)

Test « de perfection » :

```
sage: def parfait(n):
....:     if sum(ListeDiv(n))==2*n : return True
....:     else : return False
....:
sage: parfait(10)
False
sage: parfait(8128)
True
```

Programme 120 – test entier parfait (Sage)

Liste des nombres parfaits :

```
sage: def Liste_parfait(n):
....:     L=[]
....:     for k in [2..n]:
....:         if parfait(k) : L+=[k]
....:     return L
....:
sage: Liste_parfait(10000)
[6, 28, 496, 8128]
```

Programme 121 – Liste des entiers parfaits (Sage)

## D PGCD

Nous allons voir plusieurs algorithmes pour calculer le PGCD de deux entiers.

### D1 Version récursive

Le plus naturel est la version récursive de l'algorithme d'Euclide :

```
PGCD1(a,b)
Entrées : 2 entiers a et b
début
  si b=0 alors
    retourner a
  sinon
    retourner PGCD1(b,reste(a,b))
fin
```

Algorithme 19 : PGCD : version récursive de l'algorithme d'Euclide

#### D1a Avec XCAS

```
PGCD1(a,b):={
if(b==0){return(a)}
else{return(PGCD1(b,irem(a,b)))}
}
```

Programme 122 – PGCD en récursif (1)

#### D1b Avec CAML

```
# let rec pgcd (a,b)
=
if b=0
then a
else pgcd(b,a-a/b*b);;
```

Programme 123 – PGCD en récursif (1) avec CAML

#### D1c Avec Sage

```
sage: def pgcd(a,b):
.....: if b==0 : return a
.....: else : return pgcd(b,a%b)
```

Programme 124 – PGCD en récursif (1) avec Sage

## D2 Version impérative

En version impérative, c'est un peu plus long et moins naturel...

```

Entrées : 2 entiers a et b
début
  tant que b ≠ 0 faire
    r ← reste(a,b)
    a ← b
    b ← r
fin
retourner a

```

Algorithme 20 : PGCD : version impérative de l'algorithme d'Euclide

### D2 a Avec XCAS

```

PGCD2(a,b) := {
  local r;
  while (b != 0) {
    r := irem(a,b);
    a := b;
    b := r;
  }
  return(a)
};

```

Programme 125 – PGCD en impératif (1) (XCAS)

### D2 b Avec Sage

```

sage: def pgcd2(a,b):
....:   while b != 0 :
....:       r=a%b
....:       a=b
....:       b=r
....:   return a

```

Programme 126 – PGCD en impératif (1) (Sage)

## D3 Algorithme des différences

On peut également utiliser le fait que  $a \wedge b = a \wedge (b - a) = a \wedge (a - b)$ , mais c'est plus (beaucoup trop) long... Une idée de recherche peut être de comparer les deux algorithmes.

```

PGCD3(a,b)
Entrées : 2 entiers a et b
début
  si b=0 alors
    retourner a
  si a>=b alors
    retourner PGCD3(b,a-b)
  sinon
    retourner PGCD3(a,b-a)
fin

```

Algorithme 21 : PGCD par l'algorithme des différences en version récursive

**D 3 a Avec XCAS**

```
PGCD3(a,b) := {
  if(b==0){return(a)}
  if(a>=b){return(PGCD3(b,a-b))}
  else{return(PGCD3(a,b-a))}
}
```

Programme 127 – PGCD en récursif (2)

**D 3 b Avec CAML**

```
let rec pgcdif (a,b)
=
  if b=0
  then a
  else if a>b
        then pgcdif(b,a-b)
  else pgcdif(a,b-a);;
```

Programme 128 – PGCD en récursif (2) avec CAML

**Remarque 2 : Problèmes de pile**

CAML comme LISP, HASKEL et quelques autres sont de vrais langages fonctionnels donc traitent efficacement les récursions. L’algorithme précédent est très peu efficace et la pile de XCAS souffre énormément avec PGCD3(45678,3) alors que ça passe comme une fleur sur CAML...

```
sage: def pgcd3(a,b):
....:   if b==0 : return a
....:   if a>=b : return pgcd3(b,a-b)
....:   else : return pgcd3(a,b-a)
```

Programme 129 – PGCD en récursif (2) avec Sage

**E Algorithme d’Euclide étendu**

Petit rappel sur cet algorithme qui permet de prouver le théorème de Bézout et d’obtenir des coefficients vérifiant  $au + bv = a \wedge b$  :

$k$	$u_k$	$v_k$	$r_k$	$q_k$
0	1	0	$r_0 = a$	/
1	0	1	$r_1 = b$	$q_1$
2	$u_0 - u_1 q_1$	$v_0 - v_1 q_1$	$r_2 = r_0 - r_1 q_1$	$q_2$
3	$u_1 - u_2 q_2$	$v_1 - v_2 q_2$	$r_3 = r_1 - r_2 q_2$	$q_3$
$\vdots$			$\vdots$	$\vdots$
$p-1$			$r_{p-1} = a \wedge b$	$q_{p-1}$
$p$			$r_p = 0$	

Et le secret tient dans le schéma

$$u_k - (u_{k+1} \times q_{k+1}) = u_{k+2}$$

et pareil pour les  $v_k$  et les  $r_k$ .  
Par exemple, pour 19 et 15 :

$k$	$u_k$	$v_k$	$r_k$	$q_k$	
0	1	0	19	/	$L_0$
1	0	1	15	1	$L_1$
2	1	-1	4	3	$L_2 \leftarrow L_0 - 1 \times L_1$
3	-3	4	3	1	$L_3 \leftarrow L_1 - 3 \times L_2$
4	4	-5	1	3	$L_4 \leftarrow L_2 - 1 \times L_3$
5			0		$L_5 \leftarrow L_3 - 3 \times L_4$

C'est-à-dire  $4 \times 19 - 5 \times 15 = 1$  et  $19 \wedge 15 = 1$ .

En version récursive, on peut procéder en deux étapes :

```

aee(u,v,r,u',v',r')
Entrées : 6 entiers u, v, r, u', v', r'
début
  | si r'=0 alors
  | | retourner ([u,v,r])
  | sinon
  | | aee(u',v',r',u-quotient(r,r')*u',v-quotient(r,r')*v',r-quotient(r,r')*r')
fin
AEE(a,b) Entrées : 2 entiers a et b
début
  | aee(1,0,a,0,1,b)
fin
    
```

Algorithme 22 : algorithme d'Euclide étendu : version récursive

En XCAS :

```

aee(u,v,r,u',v',r') := {
if (r'==0) {return ([u,v,r])};
aee(u',v',r',u-iquo(r,r')*u',v-iquo(r,r')*v',r-iquo(r,r')*r')
};;

AEE(a,b) := aee(1,0,a,0,1,b);;
    
```

Programme 130 – algorithme d'Euclide étendu en récursif

Par exemple, AEE(19,15) renvoie [4, -5, 1] ce qui signifie que  $4 \times 19 - 5 \times 15 = 1$  et que  $19 \wedge 15 = 1$ .

En CAML :

```

# let rec aee(u,v,r,u',v',r')
=
if r'=0
then [u;v;r]
else aee(u',v',r',u-r/r'*u',v-r/r'*v',r-r/r'*r');;

# let aet(a,b)
= aee(1,0,a,0,1,b);;
    
```

Programme 131 – algorithme d'Euclide étendu en récursif (CAML)



Avec Sage :

```
sage: def aee(u,v,r,U,V,R):
....:   if R==0 : return [u,v,r]
....:   else : return aee(U,V,R,u-r//R*U,v-(r//R)*V,r-(r//R)*R)
....:
sage: def AEE(a,b):
....:   return aee(1,0,a,0,1,b)
....:
sage: AEE(19,15)
[4, -5, 1]
```

Programme 132 – algorithme d’Euclide étendu en récursif (Sage)

La version impérative est comme d’habitude plus longue à écrire mais suit malgré tout l’algorithme présenté dans le tableau :

```
Entrées : 2 entiers a et b
Initialisation : liste a ← (1,0,a)
liste b ← (0,1,b)
reste ← b
début
  tant que reste ≠ 0 faire
    q ← quotient(3eopérande de liste a, reste)
    liste temp ← liste a - q × liste b
    liste a ← liste b
    liste b ← liste temp
    reste ← 3eopérande de liste b
  fin
retourner liste a
```

Algorithme 23 : algorithme d’Euclide étendu : version impérative

En XCAS :

```
bezout(a,b) := {
local la,lb,lr,q,reste;
la := [1,0,a];
lb := [0,1,b];
reste := b;
while (reste != 0) {
q := quotient(la[2],reste);
lr := la + (-q)*lb;
la := lb;
lb := lr;
reste := lb[2];
}
return(la);
}
```

Programme 133 – algorithme d’Euclide étendu en impératif (XCAS)

Avec Sage :

Nous aurons besoin de définir des opérations sur les listes :

- somme terme à terme;
- multiplication par un scalaire.

Nous utiliserons la commande map :

```
sage: def suml(L1,L2):
....:   return map(lambda x,y:x+y,L1,L2)
```

Programme 134 – somme de listes terme à terme (Sage)

```
sage: def scal(L1,q):
.....:     return map(lambda x:q*x,L1)
```

Programme 135 – multiplication d'une liste par un scalaire (Sage)

Revenons à ce cher Bezout :

```
sage: def bezout(a,b):
.....:     la=[1,0,a]
.....:     lb=[0,1,b]
.....:     rem=b
.....:     while rem!=0 :
.....:         q=la[2]//rem
.....:         lr=suml(la,scal(lb,-q))
.....:         la=lb
.....:         lb=lr
.....:         rem=lb[2]
.....:     return(la)
sage: bezout(19,15)
[4, -5, 1]
```

Programme 136 – algorithme d'Euclide étendu en impératif (Sage)

## F Fractions continues dans $\mathbb{Q}$

– Vous connaissez l'algorithme suivant :

$$172 = 3 \times 51 + 19 \quad (8.1)$$

$$51 = 2 \times 19 + 13 \quad (8.2)$$

$$19 = 1 \times 13 + 6 \quad (8.3)$$

$$13 = 2 \times 6 + 1 \quad (8.4)$$

$$6 = 6 \times 1 + 0 \quad (8.5)$$

– On peut donc facilement compléter la suite d'égalité suivante :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = \dots$$

### Pratique du développement

– Quand tous les numérateurs sont égaux à 1, on dit qu'on a développé  $\frac{172}{51}$  en fraction continue, et pour simplifier l'écriture on note :

$$\frac{172}{51} = [3; 2; \dots]$$

– Par exemple, on peut développer  $\frac{453}{54}$  en fraction continue.

– Dans l'autre sens on peut écrire  $[2; 5; 4]$  sous la forme d'une fraction irréductible.

Nous allons construire les algorithmes correspondant.

On suppose connus les algorithmes donnant le reste et le quotient d'une division euclidienne.

```
fc(a,b)
début
  | si b=0 alors
  |   L retourner liste vide
fin
Retour[quotient(a,b),fc(b,reste(a,b))]
```

Algorithme 24 : fractions continues : version récursive

En XCAS :

```

fc(a,b):={
  if(b==0){return([])};
  return(concat(iquo(a,b),fc(b,irem(a,b))))
};

```

Programme 137 – fractions continues en récursif (1)

Alors  $fc(172, 51)$  renvoie  $[3, 2, 1, 2, 6]$

En CAML

```

# let rec fc (a,b)
  =
  if b = 0
  then []
  else a/b :: fc(b, a-a/b*b);

```

Programme 138 – fractions continues en récursif (1) avec CAML

Pour l'opération inverse, on pourra se reporter au paragraphe [C page 84](#).

En Sage/Python :

```

sage: def fc(a,b):
.....:   if b==0 : return []
.....:   else : return [a//b]+fc(b,a%b)
.....:
sage: fc(172,51)
[3, 2, 1, 2, 6]

```

Programme 139 – fractions continues en récursif (1) avec Sage

La version impérative :

```

Entrées : 2 entiers a et b
Initialisation :
num ← a
den ← b
res ← reste(num,den)
Liste ← vide
début
  tant que res ≠ 0 faire
    Liste ← Liste, quotient(num,den)
    num ← den
    den ← res
    res ← reste(num,den)
fin
retourner [Liste, quotient(num,den)]

```

Algorithme 25 : Fractions continues : version impérative

Avec XCAS :

```

frac_cont(a,b):={
  local num,den,res,Liste;
  num:=a;
  den:=b;
  res:=remain(num,den);
  Liste:=NULL;
  while(res>0){
    Liste:=Liste,iquo(num,den);
    num:=den;
    den:=res;
    res:=remain(num,den);
  }
}

```

```
[Liste, iquo(num, den)];
};;
```

Programme 140 – fractions continues en impératif (1) avec XCAS

Avec Sage/Python :

```
sage: def fc(a,b):
.....:     num=a
.....:     den=b
.....:     res=num%den
.....:     Liste=[]
.....:     while res>0 :
.....:         Liste+= [num//den]
.....:         num=den
.....:         den=res
.....:         res=num%den
.....:     return Liste+[num//den]
.....:
sage: fc(172,51)
[3, 2, 1, 2, 6]
```

Programme 141 – fractions continues en impératif (1) avec Sage

## G Fractions continues dans R

En fait, l'étude précédente correspond à un cas particulier d'une définition plus générale. Soit  $x$  un réel non entier. On construit une suite entière  $(z_n)$  de la manière suivante :

$$x = z_0 + \frac{1}{x_1} \quad z_0 = \lfloor x \rfloor \quad x_1 = \frac{1}{x - z_0}$$

puis, tant que  $x_k$  n'est pas entier :

$$x_k = z_k + \frac{1}{x_{k+1}} \quad z_k = \lfloor x_k \rfloor \quad x_{k+1} = \frac{1}{x_k - z_k}$$

Le développement de  $x$  en fractions continues est alors donné par  $[z_0, z_1, z_2, \dots]$ , cette liste pouvant être infinie.

### G1 En impératif

#### G1a Avec XCAS

```
fcr(x,n) := {
X:=evalf(x,n);
L:=floor(X);
for(k:=0;k<n;k:=k+1){
if(X!=floor(X))
then{X:=1/(X-floor(X))
L:=L, floor(X);}
}
return([L])
};;
```

Programme 142 – fractions continues en impératif (2) avec XCAS

Alors, par exemple :

```
fcr(pi,12)
```

[3,7,15,1,292,1,1,1,2,1,3,1,14]

```
fcr(sqrt(2),20)
```

[1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]

### G 1 b Avec Sage

```
sage: def fcr(x,n):
.....:     X=x.n(digits=n)
.....:     L=[floor(X)]
.....:     for k in [0..n-1]:
.....:         if X!=floor(X) :
.....:             X=1/(X-floor(X))
.....:             L+=[floor(X)]
.....:     return L
.....:
sage: fcr(sqrt(2),20)
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

Programme 143 – fractions continues en impératif (2) avec Sage

## G 2 En récursif

### G 2 a Avec XCAS

```
fcrec(x,n) := {
if(n==0)
then{return([floor(evalf(x))])}
else{
if(x==floor(x)) then {[x]}
else{
concat(floor(evalf(x,n)), fcrec(1/(x-floor(evalf(x,n))), n-1))
}
}
};;
```

Programme 144 – fractions continues en récursif (2)

### G 2 b Avec CAML

```
# let rec fcrec(x,n)=
  if n=0 then [floor(x)]
  else if x=floor(x) then [x]
  else floor(x)::fcrec(1./.(x-.floor(x)),n-1);;
```

Programme 145 – fractions continues en récursif (2) avec CAML

Alors on obtient pour  $\sqrt{2}$  :

```
# fcrec(sqrt(2.),10);;
- : float list = [1.; 2.; 2.; 2.; 2.; 2.; 2.; 2.; 2.; 2.]
```

**G 2 c Avec Sage**

```
sage: def fcr(x,n):
....:     if n==0 : return [floor(x)]
....:     elif x==floor(x) : return [x]
....:     else : return [floor(x)] + fcr(1/(x-floor(x)),n-1)
....: sage: fcr(pi,12)
[3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14]
```

Programme 146 – fractions continues en récursif (2) avec Sage

**H Tests de primalité****H 1 Première version**

Un premier test naïf consiste à diviser le nombre successivement par les entiers de 2 à la partie entière de  $n$  et de sortir de la boucle dès que le reste est nul :

```
Entrées : un entier n
début
  pour k de 2 jusque partie entière de  $\sqrt{n}$  faire
    si reste de la division de n par k est nul alors
      retourner n n'est pas premier
  retourner n est premier
fin
```

Algorithme 26 : test de primalité 1

**H 1 a En XCAS**

```
testPrem1(n):={
local k;
pour k de 2 jusque floor(sqrt(n)) faire
  si irem(n,k)==0 alors retourne(n+" n'est pas premier");
  fsi;
fpour;
retourne(n+" est premier");
};;
```

Programme 147 – test de primalité en impératif (1)

En anglais :

```
testPrem1(n):={
local k;
for(k:=2;k<=floor(sqrt(n));k:=k+1){
  if(irem(n,k)==0){return(n+" n'est pas premier");}
};
return(n+" est premier");
};;
```

Programme 148 – test de primalité en impératif (1) en VO

Cependant, il faut parfois être patient : `testPrem1(2^(10000)+1)` a mis près de 10 minutes avant de répondre que ce nombre n'était pas premier.

**H 1 b** En CAML

```
# let test_naif n =
  let rec test_local(m,k)=
    if m*m>k then "premier"
    else if k/m*m =k then "compose"
    else test_local(m+1,k)
  in test_local(2,n);;
```

Programme 149 – test de primalité en récursif (1) avec CAML

On note au passage comment utiliser une fonction localement à l'intérieur d'une autre.  
On aurait pu tout aussi bien les écrire séparément :

```
# let rec test_local(m,k)=
  if m*m>k then "premier"
  else if k/m*m =k then "compose"
  else test_local(m+1,k);;

# let test_naif n = test_local(2,n);;
```

Programme 150 – test de primalité en récursif (1bis) avec CAML

**H 1 c** Avec Sage

```
sage: def test_naif(n):
....:   for k in [2..floor(sqrt(n))] :
....:     if n%k==0 : return False
....:   return True
....:
sage: test_naif(97)
True
sage: test_naif(99)
False
```

Programme 151 – test de primalité en impératif (1) avec Sage

```
sage: def test_local(m,k):
....:   if m*m>k : return True
....:   elif k%m==0 : return False
....:   else : return test_local(m+1,k)
....:
sage: def test_naif(n):
....:   return test_local(2,n)
....:
sage: test_naif(97)
True
sage: test_naif(99)
False
```

Programme 152 – test de primalité en récursif (1) avec Sage

## H2 Deuxième version

Une première possibilité pour diviser le nombre de tests par 2 est de vérifier au départ si le nombre est pair. Dans le cas contraire, on peut faire des sauts de 2 à partir de 3 :

```

Entrées : un entier n
début
  si n est pair alors
    | retourner n n'est pas premier
  pour k de 2 jusqu'à partie entière de  $\sqrt{n}$  avec un pas de 2 faire
    | si reste de la division de n par k est nul alors
    | | retourner n n'est pas premier
  retourner n est premier
fin

```

Algorithme 27 : test de primalité 2

### H2 a Avec XCAS

```

testPrem1(n):={
local k;
si irem(n,2)==0 alors retourne(n+ " n'est pas premier"); fsi;
pour k de 3 jusqu'à floor(sqrt(n)) pas 2 faire
  si irem(n,k)==0 alors retourne(n+" n'est pas premier");
  fsi;
fpour;
retourne(n+" est premier");
};

```

Programme 153 – test de primalité en impératif (2) avec XCAS

En anglais :

```

testPrem1(n):={
local k;
if(irem(n,2)==0){return(n+ " n'est pas premier")};
for(k:=3; k<=floor(sqrt(n));k:=k+2){
  if(irem(n,k)==0){return(n+" n'est pas premier")};
};
return(n+" est premier");
};

```

Programme 154 – test de primalité en impératif (2) en VO

Plus que 291 secondes... On a bien divisé le temps de compilation par deux.

### H2 b En CAML

```

# let test_naif_2 n =
  let rec test_local(m,k)=
    if k/2*2=k then "pair"
    else
      if m*m>k then "premier"
      else if k/m*m =k then "compose"
      else test_local(m+2,k)
  in test_local(3,n);;

```

Programme 155 – test de primalité en récursif (2) avec CAML



**H2 c Avec Sage**

```
sage: def test2(n):
....:     if n%2==0 : return False
....:     for k in srange(3,floor(sqrt(n)),2):
....:         if n%k==0 : return False
....:     return True
```

Programme 156 – test de primalité en impératif (2) avec Sage

```
sage: def test2local(m,k):
....:     if k%2==0 : return False
....:     elif m*m>k : return True
....:     elif k%m==0 : return False
....:     else : return test2local(m+2,k)
....:
sage: def test2(n):
....:     return test2local(3,n)
....:
sage: test2(2**100+1)
False
```

Programme 157 – test de primalité en récursif (2) avec Sage

**H3 Troisième version**

On peut également remarquer qu'un nombre entier strictement supérieur à 3 et multiple de 3 ne peut pas être premier. On doit donc diviser par 5, 5 + 2, 5 + 2 + 4, 5 + 2 + 4 + 2, 5 + 2 + 4 + 2 + 4, 5 + 2 + 4 + 2 + 4 + 2,...

**H3 a Avec XCAS**

On introduit une fonction intermédiaire :

```
reste_sym(n)={
if(iirem(n,6)==5){return(2)};
if(iirem(n,6)==1){return(4)};
};;
```

Programme 158 – reste symétrique

Et on l'utilise :

```
testPrem3(n)={
local k;
if((iirem(n,2)==0)or (iirem(n,3)==0)){return(n+ " n'est pas premier")};
for(k:=5; k<=floor(sqrt(n));k:=k+reste_sym(k)){
if(iirem(n,k)==0){return(n+ " n'est pas premier")};
};
return(n+ " est premier");
};;
```

Programme 159 – test de primalité en impératif (3)

Plus que 188 secondes... Les outils plus efficaces dépassent les connaissances d'un élève de 2<sup>nde</sup>.

**H3b En CAML**

```
# let test_naif_3 n =
  let rec test_local(m,k)=
    if k/2*2=k or k/3*3=k then "compose"
    else
      if m*m>k then "premier"
      else if k/m*m =k then "compose"
      else test_local(m+2,k)
  in test_local(5,n);;
```

Programme 160 – test de primalité en récursif (3)

**I Crible d'Ératosthène****I1 En impératif****I1a Avec XCAS**

```
erato(n):={
  local j,k,P;
  P:=[seq(k,k=1..n)];
  P[0]:=0;
  pour j de 2 jusque floor(sqrt(n)) faire
    si P[j-1]>=1 alors
      pour k de 2 jusque floor(n/j) faire
        P[j*k-1]:=0;
      fpour;
    fsi;
  fpour;
  retourne(select(x->(x>=1),P));
};;
```

Programme 161 – crible d'Ératosthène en impératif

En anglais :

```
erato(n):={
  local j,k,P;
  P:=[seq(k,k=1..n)];
  P[0]:=0;
  for(j:=2;j<=floor(sqrt(n));j:=j+1){
    if(P[j-1]>=1){
      for(k:=2;k<=floor(n/j);k:=k+1){
        P[j*k-1]:=0;
      };
    };
  };
  return(select(x->(x>=1),P));
};;
```

Programme 162 – crible d'Ératosthène en impératif (VO)

**11b Avec Sage**

```

sage: def erato(n):
....:     P=[1..n]
....:     P[0]=0
....:     for j in [2..floor(sqrt(n))]:
....:         if P[j-1]>=1 :
....:             for k in [2..floor(n/j)]:
....:                 P[j*k-1]=0
....:     return filter(lambda x : x>=1,P)
....:
sage: erato(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]

```

Programme 163 – crible d’Ératosthène en impératif (Sage)

**12 En récursif****12a Avec CAML**

En CAML, la récursion peut se faire en étapes pour plus de lisibilité.  
On fabrique une fonction qui renvoie un intervalle entier dont on a entré les bornes :

```
# let rec inter(a,b)= if a>b then [] else a::inter(a+1,b);;
```

Programme 164 – entiers compris entre deux valeurs

Ainsi :

```
# inter(2,12);;
- : int list = [2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12]
```

Ensuite, on crée une fonction qui retire les multiples d’un entier dans une liste donnée :

```
# let rec retmultiples(l,n)=
  if l=[] then [] else
    if List.hd(l) mod n = 0
      then (retmultiples(List.tl(l),n))
      else List.hd(l)::retmultiples(List.tl(l),n);;
```

Programme 165 – pour retirer les multiples d’un entier dans une liste donnée (1)

Ainsi on peut enlever les multiples de 3 compris entre 2 et 12 :

```
# retmultiples(inter(2,12),3);;
- : int list = [2; 4; 5; 7; 8; 10; 11]
```

Voici une variante sans utiliser le module « List » :

```
# let rec retmultiples = function
| [],n -> []
| x::l,n -> if x mod n=0 then retmultiples(l,n)
            else x::retmultiples(l,n);;
```

Programme 166 – pour retirer les multiples d’un entier dans une liste donnée (2)

Enfin, voici le cœur de la procédure :

```
# let crible(m)=
  let rec crible_rec(l,n)=
    if n>m then []
    else
      if n*n>m then n::(crible_rec(l,n+1))
      else n::retmultiples(crible_rec(l,n+1),n)
  in crible_rec(inter(2,m),2);;
```

Programme 167 – crible d’Ératosthène en récursif

L’utilisation est simple :

```
# crible(100);;
- : int list =
[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]
```

Comparons : erato(10000) répond en 7,83 secondes et crible(10000) en 1,70 secondes.  
On peut évidemment améliorer chacun des algorithmes avec du savoir-faire.

## J Décomposition en facteurs premiers

On utilise la fonction ou la procédure du crible d’Ératosthène précédemment définie...ce qui va rendre nos algorithmes peu efficaces!

### J1 En récursif

#### J1 a Avec XCAS

```
decompotemp(n,liste):={
  if(n==1)then{[]}
  else{
    if(irem(n,head(liste))==0)
      then{append(decompotemp(n/head(liste),liste),head(liste))}
      else{decompotemp(n,tail(liste))}
  }
};;
```

Programme 168 – décomposition en facteurs premiers en récursif (XCAS)

puis

```
decompo(n):={
  decompotemp(n,erato(n))
};;
```

alors

```
decompo(8004)
```

[29,23,3,2,2]

#### J1 b Avec CAML

```
# let decompo(n)=
  let rec decompotemp(n,liste)=
    if n=1 then []
    else if n mod List.hd(liste) = 0
      then List.hd(liste)::(decompotemp(n/List.hd(liste),liste))
      else decompotemp(n,List.tl(liste))
  in
  decompotemp(n,crible(n));;
```

Programme 169 – décomposition en facteurs premiers en récursif avec CAML (1)

ou bien, sans utiliser le module « List » :

```
# let decompo(n)=
  let rec decompotemp = function
    | 1, l -> []
    | n, tete::queue -> if n mod tete = 0
      then tete::(decompotemp(n/tete, tete::queue))
      else decompotemp(n, queue)
  in
  decompotemp(n,crible(n));;
```

Programme 170 – décomposition en facteurs premiers en récursif avec CAML (2)

Par exemple :

```
# decompo(8004);;
- : int list = [2; 2; 3; 23; 29]
```

### J1c Avec Sage

```
def tete(L):
    return L[0]

def queue(L):
    del L[0];
    return L

def decompotemp(n,liste):
    if n==1 : return []
    elif n%tete(liste)==0 :
        return [tete(liste)]+ decompotemp(n//tete(liste),liste)
    else : return decompotemp(n,queue(liste))

def decompo(n):
    return decompotemp(n,erato(n))

sage: decompo(8004)
[2, 2, 3, 23, 29]
```

Programme 171 – décomposition en facteurs premiers en récursif (Sage)

### J1d Variante indiquant la valuation

On crée une fonction qui compte le nombre d'occurrences d'un élément donné dans une liste :

```
# let rec ocu = function
  | (n,[]) -> 0
  | (n,t::q) -> if n=t then ocu(n,q) + 1
                 else ocu(n,q);;
```

Programme 172 – nombre d’occurrences d’un nombre dans une liste en récursif

Par exemple :

```
# ocu(2, [1;2;2;2;3;4;5;5;5;5;6;6;7;8;9]);;
- : int = 3
```

Ensuite, on crée une fonction qui transforme une liste en une liste contenant des couples formés de chaque élément différent de la liste et de leur occurrence.

On utilise la commande `List.nth liste rang` qui donne l’élément de rang `rang` de la liste `liste` et la commande `List.length liste` qui renvoie le nombre d’éléments de la liste `liste`.

```
# let valuation(liste)=
  let rec valu( n) =
    if n=List.length(liste) then [(List.nth liste (n-1),ocu(List.nth liste (n-1),liste))]
    else if List.nth liste n > List.nth liste (n-1)
          then (List.nth liste (n-1),ocu(List.nth liste (n-1),liste))::valu(n+1)
          else valu(n+1)
  in valu(1);;
```

Programme 173 – liste des éléments d’une liste avec leur valuation

Par exemple :

```
# valuation([1;2;2;2;3;4;5;5;5;5;6;6;7;8;9;9;9]);;
- : (int * int) list =
[(1, 1); (2, 3); (3, 1); (4, 1); (5, 4); (6, 2); (7, 1); (8, 1); (9, 3)]
```

On modifie alors `decompo`

```
# let decompo(n)=
  let rec decompotemp = function
    | n,[] -> []
    | 1,1 -> []
    | n,tete::queue -> if n mod tete = 0
                       then tete::(decompotemp(n/tete,tete::queue))
                       else decompotemp(n,queue)
  in
  valuation(decompotemp(n,crible(n)));;
```

Programme 174 – décomposition en facteurs premiers en récursif avec CAML (2)

Par exemple :

```
# decompo(35640);;
- : (int * int) list = [(2, 3); (3, 4); (5, 1); (11, 1)]
```

C’est-à-dire  $35640 = 2^3 \times 3^4 \times 5 \times 11$ .

## J2 En impératif

### J2α Sans la valuation

```

decompo(n) := {
  local D, L, N, k;
  D := NULL;
  L := erato(n);
  N := n;
  for(k:=0; k < size(L); k:=k+1) {
    while(member(N, L) == 0) {
      if(irem(N, L[k]) == 0)
        then { D := D, L[k];
              N := iquo(N, L[k]); }
      else { break }
    }
  }
  return(D, N);
};

```

Programme 175 – décomposition en facteurs premiers en impératif (1)

Vous noterez l'utilisation de la commande break qui permet de sortir de la boucle while  
Par exemple :

```
decompo(2800)
```

donne :

2,2,2,2,5,5,7

## J2b Avec la valuation

On rajoute une petite procédure :

```

Valuation(L) := {
  local LV, val, k, s;
  s := size(L);
  LV := "";
  k := 0;
  val := [seq(1, k=1..s)]; /* une liste de n 1 */
  while(k < s-1) {
    if(L[k] == L[k+1])
      then { val[k+1] := val[k]+1; k:=k+1 }
      else { LV := LV+L[k]+"^"+val[k]+"*"; k:=k+1 }
    }
  LV := LV+L[s-1]+"^"+val[s-1]
  return(expr(LV)+"="+LV)
};

```

Programme 176 – décomposition en facteurs premiers en impératif (2)

Elle contient une petite touche esthétique...

On modifie ensuite decompo en remplaçant return(D,N); par return(Valuation(D,N));

Alors

```
decompo(2800)
```

renvoie maintenant :

$2800 = 2^4 * 5^2 * 7^1$

## K Avec la bibliothèque arithmétique de XCAS

### K1 Nombres premiers jumeaux

Des nombres premiers dont la différence vaut 2 sont appelés nombres premiers jumeaux.

Il a été conjecturé qu'il en existe une infinité. On utilise les commandes `isprime` et `nextprime` disponible sur XCAS.

```
twin(A,n):={
  if(isprime(A))then{a:=A}else{a:=nextprime(A)}
  L:=NULL;
  for(k:=a;k<=A+n-2;k:=k+2){
    if((isprime(k) and (isprime(k+2)))
      then{L:=L,[k,k+2]}
  }
  return("Il y a "+size(L)+" couple(s) de jumeaux entre "+A+" et "+(A+n)+" :
  "+L)
};;
```

Programme 177 – nombres jumeaux en impératif

## L En base 10

### L1 Nombre de chiffres d'un entier

On pense au logarithme ou à la rigueur à une ruse informatique du style : « je transforme mon nombre en chaîne de caractères et je demande au logiciel de compter le nombre de caractères de la chaîne (ce qu'ils savent tous faire) ». La première solution est peu utilisable en 2<sup>nd</sup>e et la deuxième peu intéressante mathématiquement.

Alors on « pense base 10 » :

```
NbChiffres(n) Entrées : un entier naturel n
début
  | si n<10 alors
  | | 1
  | sinon
  | | 1+NbChiffres(quotient_entier(n,10))
fin
```

Algorithme 28 : nombre de chiffres d'un entier naturel

En XCAS :

```
NbChiffres(n):={
  if(n<10){1}else{1+NbChiffres(iquo(n,10))}
};;
```

Programme 178 – nombre de chiffres d'un entier (XCAS)

En CAML :

```
# let rec NbChiffres n = if n<10 then 1 else 1+NbChiffres(n/10);;
```

Programme 179 – nombre de chiffres d'un entier (CAML)

En Sage/Python :

```
def NbChiffres(n):
  if n<10 : return 1
  else : return 1+NbChiffres(n//10)
```

Programme 180 – nombre de chiffres d'un entier (Sage)



## M Décomposition en base 2

### M 1 Expérimentation

Une méthode pour obtenir l'écriture en base 2 d'un nombre est d'effectuer des divisions successives. Par exemple pour 11 :

$$\begin{array}{r} 11 \mid 2 \\ 1 \mid 5 \end{array} \quad \begin{array}{r} 5 \mid 2 \\ 1 \mid 2 \end{array} \quad \begin{array}{r} 2 \mid 2 \\ 0 \mid 1 \end{array} \quad \begin{array}{r} 1 \mid 2 \\ 1 \mid 0 \end{array}$$

$$\begin{aligned} 11 &= (2 \times 5 + 1) \\ &= (2 \times (2 \times 2 + 1) + 1) \\ &= (2 \times (2 \times (2 \times 1) + 1) + 1) \\ &= (2 \times (2^2 + 1) + 1) \\ &= 2^3 + 2 + 1 \\ &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \end{aligned}$$

L'écriture de 11 en base 2 est donc 1011 : c'est la liste des restes obtenus mais dans l'ordre inverse. La méthode est facilement généralisable.

### M 2 Algorithme

**Entrées :** un entier  $n$

**Initialisation :**

$r \leftarrow$  le reste de la division de  $n$  par 2

$q \leftarrow$  le quotient de la division de  $n$  par 2

R une liste contenant  $r$  au départ.

**début**

**tant que**  $q > 0$  **faire**

$r \leftarrow$  le reste de la division de  $q$  par 2

$q \leftarrow$  le quotient de la division de  $q$  par 2

    On ajoute  $r$  au début de la liste R.

**retourner** R

**fin**

**Algorithme 29 :** décomposition en base 2

### M 3 En impératif

#### M 3 a Avec XCAS

Comme dans la plupart des langages de calcul, le reste et le quotient de la division euclidienne de  $a$  par  $b$  sont obtenus respectivement avec `irem(a,b)` et `iquo(a,b)`.

On concatène deux listes avec `concat(L1,L2)`.

```
base_deux(n):={
local r,R,q;
r:=irem(n,2);
q:=iquo(n,2);
R:=[r];
tantque q>0 faire
r:=irem(q,2);
q:=iquo(q,2);
R:=concat([r],R);
```

```

ftantque;
retourne(R);
};;

```

Programme 181 – décomposition en base 2 en impératif (XCAS)

### M 3 b Avec Sage/Python

```

def base2(n):
    r=n%2
    q=n//2
    R=[r]
    while q>0 :
        r=q%2
        q=q//2
        R=[r]+R
    return R

```

```

sage: base2(4)
[1, 0, 0]

```

Programme 182 – décomposition en base 2 en impératif (Sage)

On peut bien sûr généraliser ceci à n'importe quelle base.

### M 4 Version récursive

Notons  $q_n$  et  $r_n$  le quotient et le reste après  $n$  divisions.

$q_{n+1}$  est le quotient entier de  $q_n$  par 2 et on rajoute  $r_{n+1}$  à notre liste de restes.

Après un certain nombre de divisions, le quotient sera strictement inférieur à 2.

```

Binaire(n)
Entrées : un entier n
début
    si n<2 alors
        retourner n
    sinon
        retourner Binaire(quotient(n,2),reste(n,2))
fin

```

Algorithme 30 : Conversion binaire : version récursive

### M 4 a Avec XCAS

```

Binaire(n):={
    if (n<2){return n}
    else{
        return(Binaire(iquo(n,2),irem(n,2)));
    }
};;

```

Programme 183 – décomposition en base 2 en récursif (XCAS)

### M 4 b Avec CAML

```
# let rec binaire n =
  if n < 2
  then [n]
  else binaire(n/2) @ [n mod 2];;
```

Programme 184 – décomposition en base 2 en récursif (CAML)

### Remarque 3 : Concaténation de listes en CAML

On utilise `liste 1 @ liste 2` pour concaténer deux listes et `élément :: liste` pour rajouter un élément en début de liste.

### M4 c avec Sage

```
def base2(n):
  if n < 2 : return [n]
  else : return base2(n//2)+[n%2]
```

Programme 185 – décomposition en base 2 en récursif (Sage)

## N Problèmes de calendrier

### N1 Formule de Zeller

Connaissant une date, à quel jour de la semaine correspond-elle ? L'Allemand ZELLER a proposé une formule en 1885. On note  $m$  le numéro du mois à partir de janvier,  $s$  le numéro du siècle,  $a$  le numéro de l'année dans le siècle,  $j$  le numéro du jour.

En fait,  $m$  doit être modifié selon le tableau suivant :

Mois	Jan.	Fév.	Mars	Avril	Mai	Juin	Juil.	Août	Sep.	Oct.	Nov.	Déc.
Rang	13*	14*	3	4	5	6	7	8	9	10	11	12

Les mois marqués d'une astérisque comptent pour l'année précédente.

Le nom du jour (0 correspondant à samedi) est alors déterminé en prenant le reste dans la division par 7 de  $nj$  donné par la formule :

$$nj = j + a + \text{quotient entier}(a, 4) + (26m - 2)[10] + \text{quotient entier}(s, 4) - 2s$$

Ceci est valable à partir du 15 octobre 1582, date à laquelle le pape Grégoire XIII a modifié le calendrier : on est passé du jeudi 4 octobre 1582 au vendredi 15 octobre 1582 (en Grande-Bretagne il a fallu attendre 1752, au Japon en 1873, en Russie 1918, en Grèce 1924).

L'année est alors de 365 jours, sauf quand elle est bissextile, i.e., divisible par 4, sauf les années séculaires (divisibles par 100), qui ne sont bissextiles que si divisibles par 400.

Avant le changement, il faut remplacer  $\text{quotient entier}(s, 4) - 2s$  par  $5 - s$ .

### N1 a Avec XCAS

```
jour(j, m, a) := {
  local mm, aa, s, na, nj, J, greg;
  (mm, aa) := if(m < 3) then {m+12, a-1} else {m, a};
  s := iquo(aa, 100);
  na := irem(aa, 100);
```

```

greg:=if((a<1582)or((a==1582)and((m<10)or((m==10)and(j<5))))))then{5-s}else{iquo(s,4)-2*s};
nj:=j+na+iquo(na,4)+iquo(26*(mm+1),10)+greg;
J:=[ "Samedi", "Dimanche", "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi"];

return(J[irem(nj,7)])
};;

```

Programme 186 – calcul du jour de la semaine d'une date donnée (XCAS)

Pour les démonstrations, voir l'article original de ZELLER disponible (en allemand...) à l'adresse :

<http://www.merlyn.demon.co.uk/zell-86px.htm>

Par exemple, le 14 juillet 1789 était un...

```
jour(14,7,1789)
```

Mardi

## N 1 b Avec Sage/Python

```

def jour(j,m,a):
    if m<3 : [mm,aa]=[m+12,a-1]
    else : [mm,aa]=[m,a]
    s=aa//100
    na=aa%100
    if ((a<1582)or((a==1582)and((m<10)or((m==10)and(j<5)))))) :
        greg=5-s
    else :
        greg=s//4-2*s
    nj=j+na+na//4+(26*(mm+1))//10+greg
    J=['Samedi','Dimanche','Lundi','Mardi','Mercredi','Jeudi','Vendredi']
    return J[nj%7]

```

Programme 187 – calcul du jour de la semaine d'une date donnée (Sage)

Par exemple :

```
sage: jour(25,12,1)
'Dimanche'
```

# 9 - Construction d'ensembles de nombres

## A Somme et multiplication d'entiers

Supposons que nous ne sachions que trouver le successeur ou le prédécesseur d'un entier. Comment additionner deux entiers quelconques ?

```
# let rec add = function
  (0,x) -> x
| (x,y) -> add((x-1),y)+1;;
```

Programme 188 – addition de deux entiers en récursif

Additionner  $x$  et  $y$  c'est prendre le successeur de l'addition du prédécesseur de  $x$  et de  $y$ ...c'est-à-dire :  $x + y = ((x - 1) + y) + 1$  sachant que  $0 + x = x$ .

Pour la multiplication :

```
# let rec mult = function
  (0,x)->0
| (x,y)->add(mult(x-1,y),y);;
```

Programme 189 – multiplication de deux entiers en récursif

C'est-à-dire :  $0 \times x = 0$  et  $x \times y = ((x - 1) \times y) + y$   
C'est mathématique...

## B Opérations avec des fractions

On aura besoin d'un algorithme de calcul du PGCD :

```
# let rec pgcd (a,b) =
  if b=0
  then a
  else pgcd(b,a-a/b*b);;
```

Programme 190 – calcul du PGCD en récursif

On va représenter une fraction par une liste du type [numérateur;dénominateur].

Pour prévenir toute division par zéro on va créer un opérateur traitant les exceptions qu'on nommera `Division_par_zero` et qui nous servira de garde-fou :

```
# exception Division_par_zero;;
```

On invoquera cette exception avec la commande `raise` : étant donné le niveau d'anglais des élèves, cette commande est naturelle!...

On peut créer un opérateur qui simplifie les fractions :

```
# let simp([a;b]) =
  if b=0 then raise Division_par_zero
  else [a/pgcd(a,b);b/pgcd(a,b)];;
```

Programme 191 – simplification de fraction

On crée ensuite une somme simplifiée de fractions :

```
# let som([a;b],[c;d]) =
  if b=0 or d=0 then raise Division_par_zero else
  simp([a*d+b*c;b*d]);;
```

Programme 192 – somme de fractions

Par exemple :

```
# som([2;3],[1;6]);;
- : int list = [5; 6]
```

Une multiplication :

```
# let mul([a;b],[c;d]) =
  if b=0 or d=0 then raise Division_par_zero else
  simp([a*c;b*d]);;
```

Programme 193 – produit de fractions

Par exemple :

```
# mul([3;2],[2;9]);;
- : int list = [1; 3]
```

Un inverse :

```
# let inv([a;b]) =
  if b=0 or a=0 then raise Division_par_zero else
  simp([b;a]);;
```

Programme 194 – inverse d'une fraction

Une division :

```
# let div([a;b],[c;d])=mul([a;b],inv([c;d]));;
```

Programme 195 – division de fractions

Par exemple, comment entrer

$$1 + \frac{2 + \frac{3}{4}}{1 - \frac{5}{6}}$$

```
# som([1;1],div(som([2;1],[3;4]),som([1;1],[-5;6])));;
- : int list = [35; 2]
```

Évidemment, c'est moins pratique que de taper le résultat sur une machine mais ça permet de réfléchir aux différentes opérations, à la notion de fonction,...

## C Fractions continues : le retour

Nous avons vu à l'algorithme 24 page 63 comment, à partir d'un nombre écrit sous forme fractionnaire, on pouvait obtenir son écriture sous forme de fraction continue.

Nous allons à présent voir comment effectuer l'opération inverse, c'est-à-dire comment obtenir une fraction irréductible à partir de la liste du développement en fraction continue.

Souvenez-vous :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = 3 + \frac{1}{2 + \frac{1}{1 + \frac{1}{2 + \frac{1}{6}}}} = [3; 2; 1; 2; 6]$$

On en déduit l'algorithme suivant :

```
frac_inv(lx)
Entrées : La liste lx du développement en fraction continue
début
  | si lx ne contient qu'un seul élément alors
  |   | retourner cet élément
  | sinon
  |   | retourner le premier élément de lx + l'inverse de frac_inv(lx privé de son premier élément)
fin
```

**Algorithme 31** : Fractions continues : le retour récursif

```
# let rec frac_inv(lx)=
  if List.length lx=1 then [List.hd(lx);1]
  else som([List.hd(lx);1],inv(frac_inv(List.tl(lx))));;
```

Programme 196 – développement de fractions continues (1)

ou bien sans « List » :

```
# let rec frac_inv = function
  | x::[] -> [x;1]
  | x::l -> som([x;1],inv(frac_inv(l)));;
```

Programme 197 – développement de fractions continues (2)

Et voici comment l'utiliser :

```
# frac_inv([3;2;1;2;6]);;
- : int list = [172; 51]
```

Une petite curiosité au passage :

```
# frac_inv([0;3;2;1;2;6]);;
- : int list = [51; 172]
```

# 10 - Algorithmes de calcul plus complexes

## A Algorithme de Hörner

### A1 Conventions

Dans la suite, nous confondrons polynôme et fonction polynôme, ce qui n'est pas gênant si on travaille sur  $\mathbb{R}[X]$ . Pour aller vite, nous utiliserons un vocabulaire non connu des élèves de 2<sup>nde</sup> mais tout à fait adaptable car les notions sont simples et font travailler les expressions algébriques et réfléchir sur la complexité d'un calcul.

### A2 Principe

Prenons l'exemple de  $P(x) = 3x^5 - 2x^4 + 7x^3 + 2x^2 + 5x - 3$ . Le calcul classique nécessite 5 additions et 15 multiplications. On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \dots + a_2 x^2 + a_1 x + a_0}_{\text{on met } x \text{ en facteur}} \\ &= \left( \underbrace{a_n x^{n-1} + \dots + a_2 x + a_1}_{\text{on met } x \text{ en facteur}} \right) x + a_0 \\ &= \dots \\ &= (\dots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \dots)x + a_0 \end{aligned}$$

Ici cela donne  $P(x) = (((((3x - 2)x + 7)x + 2)x + 5)x - 3$  c'est-à-dire 5 multiplications et 5 additions. En fait il y a au maximum  $2 \times$  degré de P opérations (voire moins avec les zéros).

### A3 Algorithme récursif

Il faut connaître les coefficients de P.

```
horner(P,x) Entrées : les coefficients  $a_i$  dans l'ordre décroissant des exposants des monômes, le degré  $n$  de P et le
                nombre  $x$ 
début
  | si P est vide alors
  | | retourner 0
  | sinon
  | | retourner Tête de P +  $x \times$  horner(P privé de sa tête, x)
fin
```

Algorithme 32 : algorithme de HÖRNER : version récursive

### A3a En XCAS

```
Horner(P,x) := {
  if(P=[]) {return(0)}
```



```
else{return(P[0]+x*Horner(tail(P),x))}
};;
```

Programme 198 – algorithme de Hörner en récursif (XCAS)

### A 3 b En CAML

```
# let rec horner = function
|([],x) -> 0
|(tete::queue,x) -> tete + x*horner(queue,x);;
```

Programme 199 – algorithme de Hörner en récursif (CAML)

Ainsi, pour calculer  $P(3)$  avec  $P(x) = 3x^5 - 2x^4 + 7x^3 + 2x^2 + 5x - 3$  :

```
# horner([3;-2;7;2;5;-3],3);;
- : int = -210
```

### A 3 c En Sage/Python

```
def queue(L):
    del L[0];
    return L

def horner(P,x):
    if P==[] : return 0
    else :
        return P[0]+x*horner(queue(P),x)
```

Programme 200 – algorithme de Hörner en récursif (Sage)

Par exemple :

```
sage: horner([3,-2,7,2,5,-3],x)
-(((3*x - 5)*x - 2)*x - 7)*x + 2)*x + 3
sage: horner([3,-2,7,2,5,-3],3)
-210
```

## A 4 Algorithme impératif

Pour la version impérative :

**Entrées** : les coefficients  $a_i$  dans l'ordre décroissant des exposants des monômes, le degré  $n$  de  $P$  et le monôme  $x$   
**Initialisation** :  $Q \leftarrow a_n$   
**début**  
 | **pour**  $k$  de 1 à  $n$  **faire**  
 | |  $Q \leftarrow Q \times x + a_k$   
 | **retourner**  $Q$   
**fin**

Algorithme 33 : algorithme de HÖRNER : version impérative

### A 4 a traduction XCAS

```

Horner(C,d,x):={
  local Q,k;
  Q:=C[0];
  pour k de 1 jusque d faire
    Q:=(Q*x)+C[k];
  fpour;
  retourne(Q)
}

```

Programme 201 – algorithme de Hörner version impérative

Pour notre exemple, on entre :

```
(Horner([3,-2,7,2,5,-3],5,X))
```

et on obtient bien :

$$(((3 * X - 2) * X + 7) * X + 2) * X + 5) * X - 3$$

## B Exponentiation rapide

### B 1 Méthode naïve

Calculer  $a^n$  nécessite a priori  $n - 1$  multiplications selon l'algorithme suivant :

#### B 1 a En impératif

```

Entrées : un réel  $a$  et une puissance  $n$ 
Initialisation : Résultat ←  $a$ 
début
  | pour  $k$  de 2 à  $n$  faire
  |   | Résultat ← Résultat ×  $a$ 
fin
retourner Résultat

```

Algorithme 34 : exponentiation naïve

#### B 1 b Avec XCAS

```

puissance_naive(a,n):={
  local res,k;
  res:=a;
  pour k de 2 jusque n faire
    res:=res*a;
  fpour;
  retourne(res);
};

```

Programme 202 – exponentiation rapide version impérative (XCAS)

#### B 1 c Avec Sage/Python

```

def puissance_naive(a,n):
    res=a
    for k in [2..n]:

```

```

    res*=a
    return res

sage: puissance_naive(2,10)
1024

```

Programme 203 – exponentiation rapide version impérative (Sage)

**B 1 d** Version récursive

**Procédure** : puissance\_naive\_rec  
**Entrées** : un réel  $a$  et une puissance  $n$   
**début**  
    | **si**  $n$  est nul **alors**  
    | | retourner 1  
    | **sinon**  
    | |  $a$  multiplié par puissance\_naive\_rec( $a,n-1$ )  
**fin**

**Algorithme 35** : exponentiation naïve récursive**B 1 e** Avec XCAS

```

puissance_naive_rec(a,n):={
si n==0 alors retourne(1);
sinon puissance_naive_rec(a,n-1)*a;
fsi;
};

```

Programme 204 – exponentiation rapide version récursive (XCAS)

**B 1 f** Avec CAML

```

let rec puissance_naive_rec(a,n)=
if n=0 then 1
else puissance_naive_rec(a,n-1)*a;;

```

Programme 205 – exponentiation rapide version récursive (CAML)

**B 1 g** Avec Sage/Python

```

def puissance_naive_rec(a,n):
    if n==0 : return 1
    else : return puissance_naive_rec(a,n-1)*a

```

Programme 206 – exponentiation rapide version récursive (Sage)

**B 2** Avec l'algorithme de Hörner

On peut essayer de faire mieux, surtout si on pense aux problèmes de cryptographie du style RSA. Voyons une première méthode qui utilise l'algorithme de HÖRNER étudié précédemment et la décomposition en base 2 de l'exposant.

Par exemple,  $11 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ . On peut donc associer à cette écriture un polynôme  $P$  tel que 11 soit égal à  $P(2)$  :

$$P(X) = 1 \times X^3 + 0 \times X^2 + 1 \times X^1 + 1 \times X^0 = (X+0)X+1)X+1$$

Donc  $a^{P(2)} = a^{1+2(1+2(2 \times 1+0))} = a \cdot a^{2(1+2(2 \times 1+0))} = a \cdot (a^{1+2(2 \times 1+0)})^2 = a^1 \cdot (a^1 \cdot (a^0 \cdot (a^1)^2))^2 = \left( \left( (a^1)^2 a^0 \right)^2 a^1 \right)^2 a^1$

On en déduit l'algorithme :

```

Entrées : un réel a et un entier n
Initialisation : C ← décomposition de n en base 2
res ← a
début
  pour j de 1 jusque taille de C - 1 faire
    res ← res × res
    si le j-ème chiffre de la décomposition en base 2 est 1 alors
      res ← a × res
  retourner res
fin

```

**Algorithme 36** : exponentiation « à la HÖRNER »

## B 2 a Avec XCAS en impératif

```

puissance_horner(a,n):={
local C,j,res;
C:=base_deux(n);
res:=a;
pour j de 1 jusque size(C)-1 faire
  res:=res*res;
  si C[j]==1 alors res:=a*res;
  fsi;
fpour;
retourne(res);
};

```

Programme 207 – exponentiation rapide en récursif « à la Hörner »

Pour calculer  $a^{11}$  nous n'avons donc plus à effectuer que 6 multiplications ou plutôt 5 si on ne compte pas la multiplication par 1.

## B 2 b Avec Sage/Python en impératif

```

def puissance_horner(a,n):
C=base2(n)
res=a
for j in [1..len(C)-1]:
  res*=res
  if C[j]==1: res*=a
return(res)

```

Programme 208 – exponentiation rapide en récursif « à la Hörner »

## B 2 c Avec CAML en version récursive

```

# let pow_horn(a,n)=
let rec pow_horn_rec(a,n,k)=
  if k=0 then a
  else if nth(binaire n) k=0 then pow_horn_rec(a,n,k-1)*pow_horn_rec(a,n,k-1)

```

```

else a*pow_horn_rec(a,n,k-1)*pow_horn_rec(a,n,k-1)
in pow_horn_rec(a,n,length(binaire n)-1);

```

Programme 209 – exponentiation rapide en récursif « à la Hörner »

### B3 Variante sans utiliser l'écriture en base 2

Voyons comment procéder sur notre exemple habituel :

$$a^{11} = a \cdot a^{10} = a \cdot (a^5)^2 = a \cdot (a \cdot a^4)^2 = a \cdot (a \cdot (a^2)^2)^2$$

Ainsi, on réduit l'exposant  $k$  selon sa parité :

- si  $k$  est pair, on écrit  $a^k = \left(a^{\frac{k}{2}}\right)^2$  ;
- sinon,  $a^k = a \cdot \left(a^{\frac{k-1}{2}}\right)^2$

```

Entrées : un réel a et un entier n
Initialisation :
res ← 1
puissance ← n
temp ← a
début
  tant que puissance non nulle faire
    si puissance est impaire alors
      res ← temp × res
      puissance ← puissance - 1
    puissance ← puissance ÷ 2
    temp ← temp × temp
fin

```

Algorithme 37 : exponentiation rapide sans utiliser la base 2

### B3a Avec XCAS en impératif

```

expo_rapido(a,n):={
local res,temp,Puissance;
res:=1;
Puissance:=n;
temp:=a;
tantque Puissance >0 faire
  si irem(Puissance,2)==1 alors
    res:=temp*res;
    Puissance:=Puissance-1;
  fsi;
  Puissance:=Puissance/2;
  temp:=temp*temp;
ftantque;
retourne(res);
};

```

Programme 210 – exponentiation rapide en impératif (3) (XCAS)

### B3b Avec Sage/Python en impératif

```

def expo_rapido(a,n):
  res=1
  Puissance=n
  temp=a

```

```

while Puissance>0:
    if Puissance%2==1 :
        res*=temp
        Puissance+/-1
    Puissance=Puissance/2
    temp*=temp
return(res)

```

Programme 211 – exponentiation rapide en impératif (3) (Sage)

### B 3 c Avec CAML en récursif

Avec CAML en version récursive c'est immédiat :

```

# let rec pow_rap(a,n)=
  if n=0 then a
  else if n/2*2=n then pow_rap(a,n/2)*pow_rap(a,n/2)
  else a*pow_rap(a,n/2);;

```

Programme 212 – exponentiation rapide en récursif (3)

### B 3 d Avec Sage/Python en récursif

```

def pow_rap(a,n):
    if n==0 : return a
    elif n%2==0 :
        return pow_rap(a,n//2)**2
    else :
        return a*pow_rap(a,n//2)

```

Programme 213 – exponentiation rapide en récursif (3) (Sage)

## B 4 Comparaison des performances

Grâce à la commande `time` de XCAS, nous pouvons comparer les temps de calcul des différents algorithmes pour calculer  $2^{1000000}$  :

```
time(puissance_naive(2,1000000))
```

donne 70.23 secondes sur notre ordinateur.

Avec `expo_rapido` le temps tombe à 0,21 seconde et avec `puissance_horner` 0,09 seconde!

Mais réduire le nombre d'itérations n'est pas toujours efficace si les opérations engendrées à chaque itération sont plus compliquées. Ce serait par exemple le cas si nous travaillons avec des polynômes à plusieurs variables... ce qui est loin du programme du lycée!

Cependant, il peut être intéressant (en spécialité de terminale S ou un futur clone quelconque...) d'adapter cet algorithme à la puissance modulaire pour travailler par exemple sur le « petit » théorème de FERMAT.

## C Calculatrice en légo

Andrew CAROL a fabriqué une machine à calculer entièrement constituée de légos.

Par exemple, considérons l'expression :

$$P(x) = 2x^2 + 3x + 5$$

Alors la machine peut calculer cette expression en remplaçant  $x$  par des nombres entiers.

## C1 Le principe

Remplissons le tableau suivant jusqu'à la 4<sup>e</sup> ligne :

x	P(x)	première différence	deuxième différence
1	10	9	4
2	19	13	
3			
4			
5			
6			
7			
8			
9			

sachant que 9 est obtenu en faisant  $19 - 10$  et 4 en faisant  $13 - 9$ .

Qu'observez-vous dans la dernière colonne ?

Finissez alors de remplir le tableau en n'effectuant que des additions : c'est tout ce que peut faire la machine en légos grâce à ses roues dentées.

## C2 Un exemple

Soit  $Q(x) = 4x^2 + 5x + 1$

Remplissez un tableau similaire au précédent en n'effectuant aucune multiplication...

## C3 Généralisation

Pourquoi ça marche ? Essayez de remplir un tableau avec  $R(x) = ax^2 + bx + c$ ,  $a$ ,  $b$  et  $c$  étant des nombres quelconques.

## C4 Un petit brin d'informatique

Faisons calculer l'ordinateur avec uniquement des additions.

### C4 a Version récursive avec CAML

```
# let lego(n, a, b, c)=
  let rec legotemp(n, p, d1, d2)=
    if n=1 then p
    else legotemp(n-1, p+d1, d1+d2, d2)
  in legotemp(n, a+b+c, a+a+a+b, a+a);;
```

Programme 214 – la machine légo en récursif (CAML)

### C4 b Version récursive avec XCAS

```
legotemp(n, p, d1, d2) := {
  if(n==1) then {p}
  else {legotemp(n-1, p+d1, d1+d2, d2)}
};;
```

Programme 215 – la machine légo en récursif 1ère étape (XCAS)

puis

```
lego(n, a, b, c) := {
  legotemp(n, a+b+c, a+a+a+b, a+a)
} ;
```

Programme 216 – la machine légo en récursif 2ème étape (XCAS)

Par exemple, avec  $P(x) = 2x^2 + 3x + 5$ , on peut calculer  $P(20)$  avec uniquement des sommes :

```
lego(20, 2, 3, 5)
```

865

**C4c** Version impérative avec XCAS

```
lego(n, a, b, c) := {
  local p, d1, d2, k;

  p := a+b+c;
  d1 := a+a+a+b;
  d2 := a+a;

  for(k:=1; k<n; k:=k+1) {
    p := p+d1;
    d1 := d1+d2;
  }

  return(p)
} ;
```

Programme 217 – la machine légo en impératif

**D** Tableaux de signes

On veut étudier le signe d'un produit de 2 facteurs affines. Par exemple, en tapant `signeproduit(-2*x+3, 4*x+5)` on veut obtenir :

valeurs de x		$-\frac{5}{4}$	$\frac{3}{2}$	
signe de $-2x+3$	+		+	0 -
signe de $4x+5$	-	0	+	+
signe du produit	-	0	+	0 -

Pour cela, on divise le tableau en plusieurs zones :

Valeurs de x		1		2	
Signe de $ax + b$	3	4	5	6	7
Signe de $cx + d$	8	9	10	11	12
Signe du produit	13	14	15	16	17

Reste à savoir comment remplir les 17 cases numérotées...

Nous aurons besoin de résoudre des équations affines pour déterminer les zéros.



Voici un petit algorithme qui va nous rendre service :

```
Entrées : expression affine d'inconnue x sous la forme ax+b  
début  
| si  $a=0$  alors  
| | retourner il n'y a pas de solution unique  
| sinon  
| | retourner  $-b/a$   
fin
```

**Algorithme 38** : résolution d'équations affines

XCAS sait résoudre ce genre d'équations grâce à la commande `resoudre`.

Il est également simple de déterminer le minimum et le maximum entre deux réels. Par exemple :

```
Entrées : une liste L de deux réels  
début  
| si  $L[0]<L[1]$  alors  
| | retourner  $L[0]$   
| sinon  
| | retourner  $L[1]$   
fin
```

**Algorithme 39** : minimum d'une liste de deux réels

Ensuite, il ne reste plus qu'à distinguer les cas...

**Entrées** : deux expressions affines  $ax+b$  et  $cx+d$

**Initialisation** :

$z_A \leftarrow$  solution de  $ax+b=0$ ;  $z_B \leftarrow$  solution de  $cx+d=0$

$z_{\min} \leftarrow$  minimum de  $(z_A, z_B)$ ;  $z_{\max} \leftarrow$  maximum de  $(z_A, z_B)$

**début**

On construit un tableau

1ère ligne : valeurs de x

un vide puis on affiche  $z_{\min}$  puis un vide puis on affiche  $z_{\max}$  puis un vide puis

2ème ligne : signe de  $f(x)=ax+b$

**si**  $f(z_{\min}-1) > 0$  **alors**

└ on affiche +

**sinon**

└ on affiche -

**si**  $f(z_{\min}) = 0$  **alors**

└ on affiche 0

**sinon**

└ on n'affiche rien

**si** l'image par  $f$  du milieu de  $z_{\min}$  et  $z_{\max}$  est positive **alors**

└ on affiche +

**sinon**

└ on affiche -

**si**  $f(z_{\max}) = 0$  **alors**

└ on affiche 0

**sinon**

└ on n'affiche rien

**si**  $f(z_{\max}-1) > 0$  **alors**

└ on affiche +

**sinon**

└ on affiche -

3ème ligne : signe de  $g(x)=cx+d$

idem en remplaçant  $f$  par  $g$

4ème ligne : signe du produit

**si** le signe du produit des images de  $z_{\min}-1$  par  $f$  et  $g$  est positif **alors**

└ on affiche +

**sinon**

└ on affiche -

on affiche 0

**si** le signe du produit des images du milieu de  $z_{\min}$  et  $z_{\max}$  par  $f$  et  $g$  est positif **alors**

└ on affiche +

**sinon**

└ on affiche -

on affiche 0

**si** le signe du produit des images de  $z_{\max}+1$  par  $f$  et  $g$  est positif **alors**

└ on affiche +

**sinon**

└ on affiche -

**fin**

**Algorithme 40** : étude du signe d'un produit de facteurs affines

```
signeproduit2(F,G) := {
f := unapply(F,x);
g := unapply(G,x); \ \ on transforme les expressions F et G en fonctions
zA := resoudre(f(x)=0,x)[0];
zB := resoudre(g(x)=0,x)[0];
zmin := min(zA,zB);
zmax := max(zA,zB);
```

```

[ ["valeurs de x", " ", zmin, " ", zmax, " "],

  ["signe de "+f(x),   si f(zmin-1)>0 alors " + "; sinon " - "; fsi,
                           si f(zmin)==0 alors 0; sinon " "; fsi,
                           si f((zmin+zmax)*.5)>0 alors " + "; sinon " - "; fsi,
                           si f(zmax)==0 alors 0; sinon " "; fsi,
                           si f(zmax+1)>0 alors " + "; sinon " - "; fsi

  ],

  ["signe de "+g(x),   si g(zmin-1)>0 alors " + "; sinon " - "; fsi,
                           si g(zmin)==0 alors 0; sinon " "; fsi,
                           si g((zmin+zmax)*.5)>0 alors " + "; sinon " - "; fsi,
                           si g(zmax)==0 alors 0; sinon " "; fsi,
                           si g(zmax+1)>0 alors " + "; sinon " - "; fsi

  ],

  ["signe du produit", si g(zmin-1)*f(zmin-1)>0 alors " + "; sinon " - "; fsi,
                           0,
                           si g((zmin+zmax)*.5)*f((zmin+zmax)*.5)>0 alors " + "; sinon " - "; fsi,
                           0,
                           si g(zmax+1)*f(zmax+1)>0 alors " + "; sinon " - "; fsi

  ]

]
};

```

Programme 218 – signe d'un produit de facteurs affines

On peut généraliser à un nombre quelconque de facteurs et même à des cas plus compliqués comme vous pouvez le voir ici à partir de la ligne 1802 :

[http://download.tuxfamily.org/tehessinmath/les\\_sources/tablor.html](http://download.tuxfamily.org/tehessinmath/les_sources/tablor.html)

# 11 - Algorithmes en analyse

## A Les réels et le processeur

Ayez toujours en tête qu'un ordinateur ne peut pas travailler avec des réels! Par essence, il a un nombre fini de bits sur lesquels il peut coder des nombres...

Les nombres *flottants* qu'il manipule sont donc des classes qui représentent chacune une infinité de réels (mais tout en constituant un intervalle cohérent).

Si on n'y fait pas attention, cela peut créer des surprises...

Entrons trois nombres sur XCAS :

```
x:=10;  
y:=-5;  
z:=5.00000001;
```

Alors

```
(x*y)+(x*z)
```

renvoie :

1.000000012e-07

Mais

```
x*(y+z)
```

renvoie :

9.999999939e-08

Le problème est le même avec CAML :

```
# let x,y,z=10.,-5.,5.00000001;;  
val x : float = 10.  
val y : float = -5.  
val z : float = 5.00000001  
  
# (x*.y)+(x*.z);;  
- : float = 1.00000001168609742e-07  
  
# x*.(y+.z);;  
- : float = 9.99999993922529e-08
```

ou avec Sage :

```
sage: [x,y,z]=[10,-5,5.00000001]  
sage: (x*y)+(x*z)  
1.00000001168610e-7  
sage: x*(y+z)  
9.99999993922529e-8
```

Ainsi l'addition n'est pas toujours distributive sur la multiplication quand on travaille sur les flottants!

Ces petites erreurs qui s'accumulent peuvent en créer de grandes!

Créons une fonction qui renvoie l'approximation de la dérivée d'une fonction en un nombre donné avec un « dx » valant  $10^{-10}$  :

```
# let dernum(f)=function
  x->(f(x+.0.0000000001)-.f(x))/ .0.0000000001;;
```

Programme 219 – dérivation numérique

Dérivons le sinus cinq fois de suite et évaluons la fonction obtenue en 0 :

```
# dernum(dernum(dernum(dernum(dernum(sin)))))(0.);
- : float = -1.33226762955018773e+25
```

Oups! On obtient  $\cos(0) \approx -10^{25}$  ...Ce n'est plus de l'ordre de l'erreur négligeable!

## B Tracé de représentations graphiques

### B 1 Version récursive à pas constant

#### B 1 a Avec XCAS

```
liste_points(f,a,b,h):={
  if(a>b)then{[b,f(b)]}
  else{[a,f(a)],liste_points(f,a+h,b,h)}
}::;
```

Programme 220 – liste de coordonnées de points d'un graphe de fonction en récursif (XCAS)

On a créé une liste de coordonnées du type  $(x, f(x))$  avec un pas de  $h$  entre  $a$  et  $b$ .

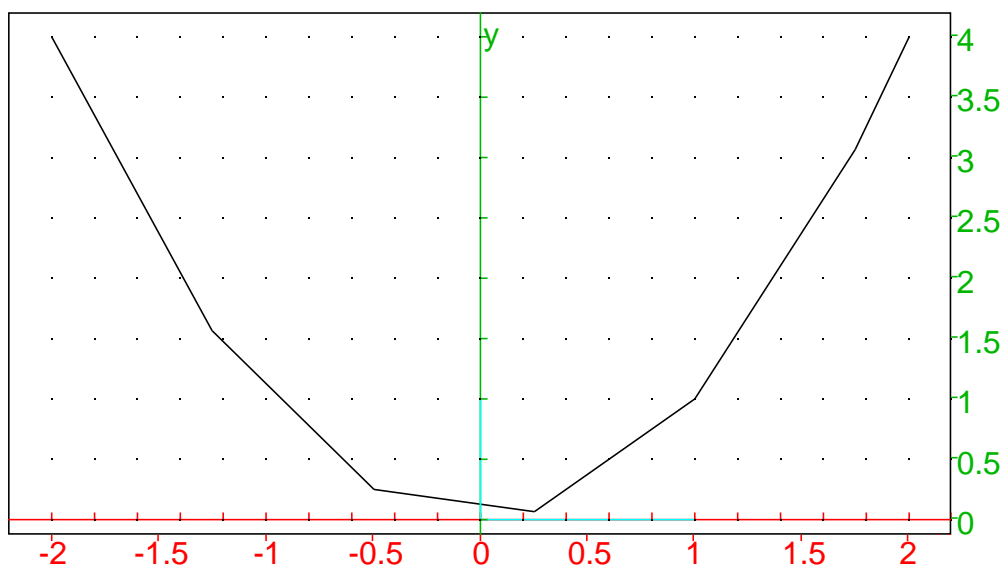
On relie ces points par des segments avec `polygonplot(liste de coordonnées)` :

```
graph(f,a,b,h):={
  polygonplot([liste_points(f,a,b,h)])
}::;
```

Par exemple, on trace la fonction carrée entre  $-2$  et  $2$  avec un pas de  $0,75$  :

```
graph(x->x^2,-2,2,0.75)
```

ce qui donne :



### B1b Avec Sage

```
def liste_points(f,a,b,h):
    if a>b : return [[b,f(b)]]
    else : return [[a,f(a)]]+liste_points(f,a+h,b,h)
```

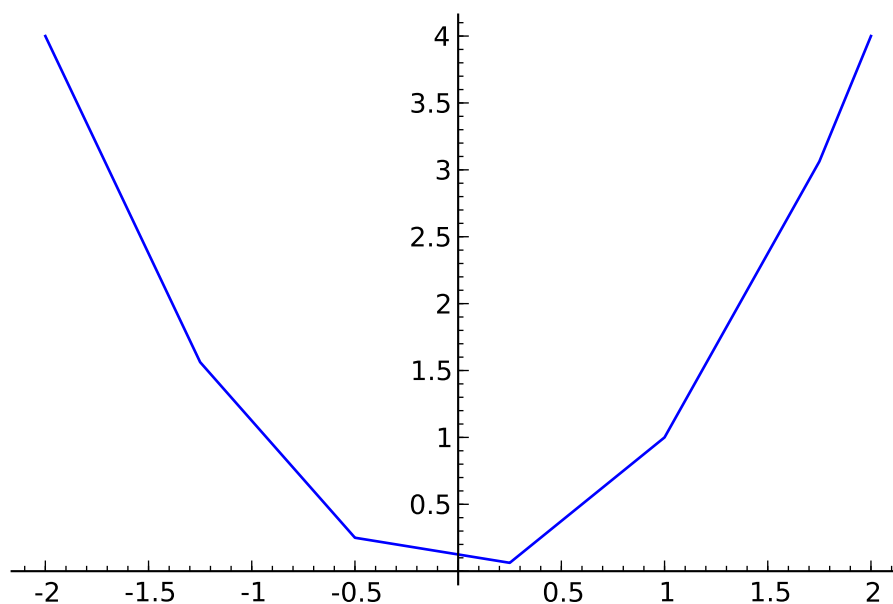
Programme 221 – liste de coordonnées de points d'un graphe de fonction en récursif (Sage)

Par exemple, on trace la fonction carrée entre  $-2$  et  $2$  avec un pas de  $0,75$  :

```
def f(x):
    return x**2

def graph(f,a,b,h):
    l=line(liste_points(f,a,b,h))
    l.show()
```

Ce qui donne :



## B2 Version récursive par sondage aléatoire

On interroge au hasard un réel et on lui demande son image par une certaine fonction : cela nous permettra-t-il d'avoir une bonne idée de l'allure de la courbe ?

### B2a Avec XCAS

```
liste_points_hasard(f,a,a0,b,n):={
  if(n==0)then{[b,f(b)]}
  else{[a0,f(a0)],liste_points_hasard(f,a,a+rand(0,1)*(b-a),b,n-1)}
};;
```

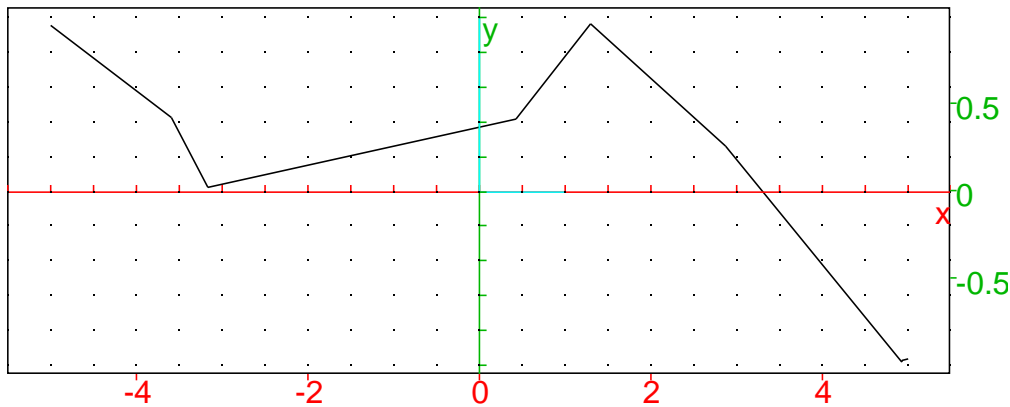
Programme 222 – liste aléatoire de point du graphe d'une fonction (XCAS)

```
graph_hasard(f,a,b,n):={
  polygonplot([liste_points_hasard(f,a,a,b,n)])
};;
```

Alors on obtient pour la fonction sinus entre  $-5$  et  $5$  avec 8 « sondés » :

```
graph_hasard(x->sin(x),-5,5,8)
```

donne :



### B2b Avec Sage

```
def liste_points_hasard(f,a,b,n):
  if n==0 : return [[b,f(b)]]
  else : return [[a,f(a)]]+liste_points_hasard(f,a+random()*(b-a)/n,b,n-1)

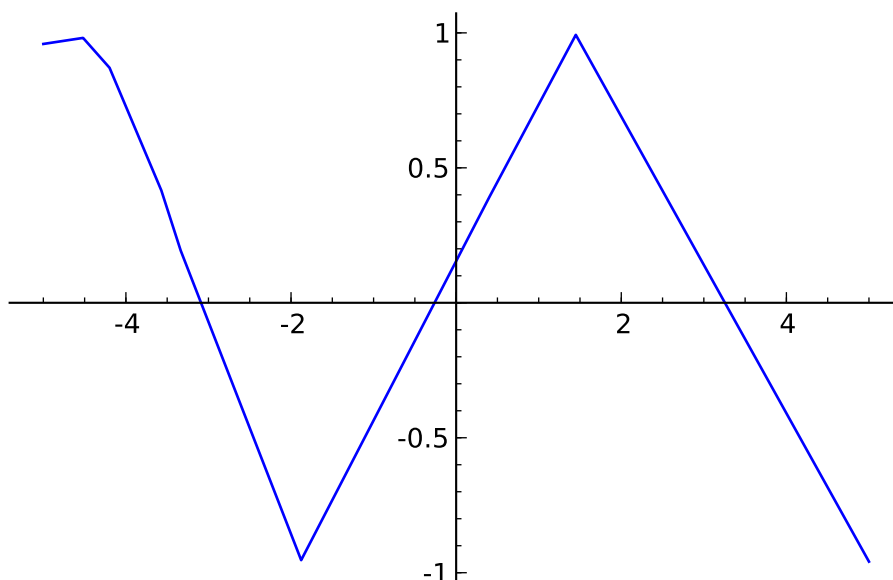
def graph_hasard(f,a,b,n):
  l=line(liste_points_hasard(f,a,b,n))
  l.show()
```

Programme 223 – liste aléatoire de point du graphe d'une fonction (Sage)

Ce qui donne pour le sinus entre  $-5$  et  $5$  :

```
def f(x):
  return sin(x)

sage: graph_hasard(f,-5,5,8)
```



## C Fonctions définies par morceaux

Comment faire dessiner une courbe en dents de scie, chaque dent ayant une largeur de 2 et une hauteur de 1 ? Peut-on avoir une formule explicite qui permette de calculer l'image de n'importe quel nombre ?

Les coefficients directeurs valent alternativement 1 et  $-1$ .

Les ordonnées s'annulent aux points d'abscisses paires...

### C1 Avec XCAS

```
scie_rec(n):={
  if(n==0)then{[0,0]}
  else{[n,(1-(-1)^n)/2],scie_rec(n-1)}
};;
```

Programme 224 – fonction « dent de scie »

Puis :

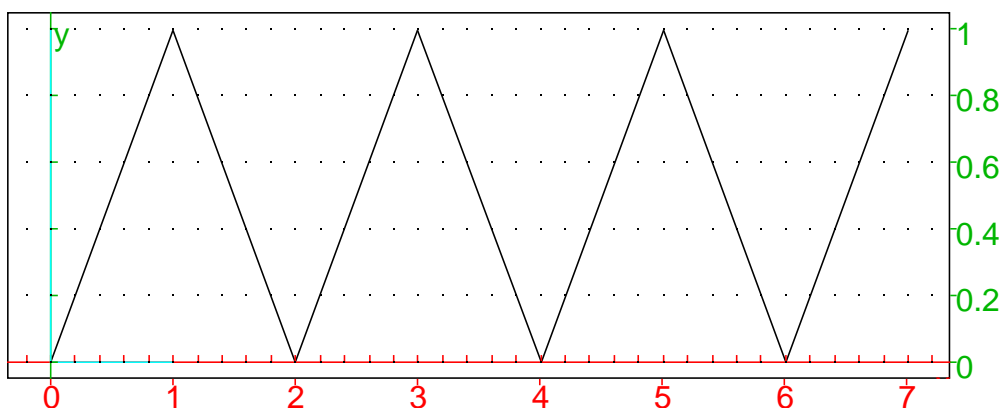
```
scie(n):={
  polygonplot([scie_rec(n)])
};;
```

Alors

```
scie(7)
```

donne :





Le tracé a été obtenu sans formule explicite pour chaque réel. Cherchons une telle formulation :

```
f(x):={
  if(irem(floor(x),2)==0)
    then{x-floor(x)}
    else{1-x+floor(x)}
};;
```

Programme 225 – fonction dent de scie (2)

sachant que `floor` donne la partie entière et `irem` le reste de la division euclidienne.

## C2 Avec Sage

```
def scie_rec(n):
  if n==0 : return [[0,0]]
  else : return [[n,(1-(-1)**n)/2]]+scie_rec(n-1)

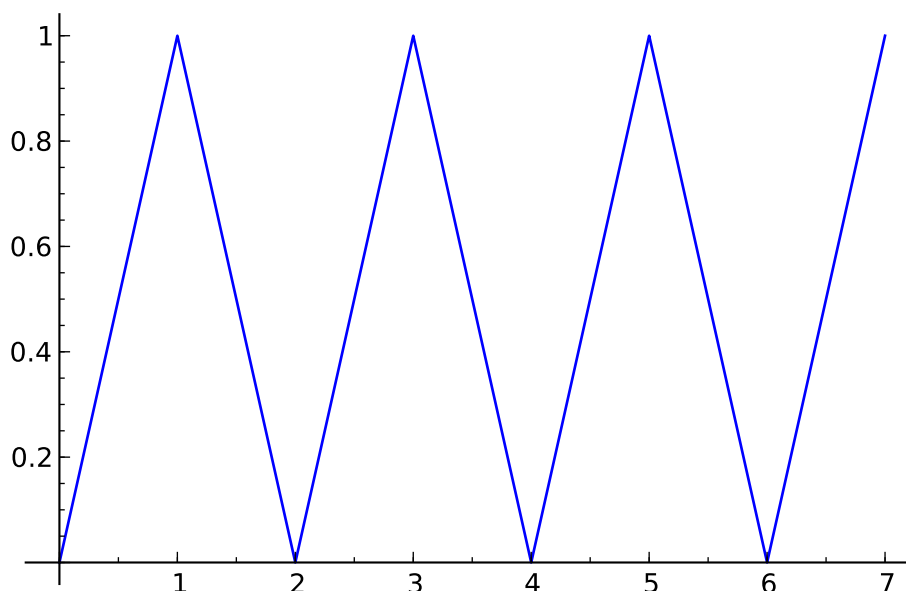
def scie(n):
  l=line(scie_rec(n))
  l.show()
```

Programme 226 – fonction « dent de scie »

Alors :

```
scie(7)
```

donne :



## D Recherche de minimum

On repère graphiquement qu'une fonction semble atteindre par exemple un minimum. Pour obtenir une approximation de ce minimum, on fournit un point de départ et une précision.

### D1 Version récursive

#### D1a En XCAS

```
min_rec(f,xo,h):={
  if(f(xo+h)>f(xo)){return(xo)}
  else{min_rec(f,xo+h,h)}
}
```

Programme 227 – minimum d'une fonction en récursif 1 (XCAS)

#### D1b En CAML

```
# let rec min_rec(f,xo,h)=
  if f(xo+.h)>f(xo) then xo
  else min_rec(f,xo+.h,h);;
```

Programme 228 – minimum d'une fonction en récursif 1 (CAML)

#### D1c En Sage

```
def min_rec(f,xo,h):
  if f(xo+h)>f(xo): return xo
  else : return min_rec(f,xo+h,h)
```

Programme 229 – minimum d'une fonction en récursif 1 (Sage)

Par exemple, pour la fonction  $x \mapsto \sqrt{x^2 + 25} + \sqrt{(x - 18)^2 + 49}$ , il semble que le minimum soit vers 7. Demandons une approximation du minimum à  $10^{-1}$  près :

```
def f(x):
    return sqrt((x)**2+25)+sqrt((x-18)**2+49)

sage: min_rec(f,7,0.1)
7.500000000000000
```

## D2 Version impérative

**Entrées :** fonction  $f$ , point de départ  $x_0$  et incrément  $i$

**Initialisation :**  $Y \leftarrow f(x_0)$   $X \leftarrow x_0 + i$

**début**

**tant que**  $t(X) < Y$  **faire**

$Y \leftarrow \text{evalf}(t(X))$   $X \leftarrow X + i$

**retourner**  $X - i$

**fin**

Algorithme 41 : approximation d'un minimum

### D2a Avec XCAS

```
minimum(f,xo,p):={
local X,Y,compteur;
Y:=f(xo);
X:=xo+10^(-p);
compteur:=0 /* on rajoute un compteur pour que la compilation ne dure pas trop longtemps */
tantque evalf(f(X))<Y et compteur<10^p faire
    Y:=evalf(f(X));
    X:=X+10^(-p);
    compteur:=compteur+1;
ftantque;
X-10^(-p);
};;
```

Programme 230 – minimum d'une fonction en impératif (XCAS)

Par exemple, pour la fonction  $x \mapsto \sqrt{x^2 + 25} + \sqrt{(x - 18)^2 + 49}$ , il semble que le minimum soit vers 7. Demandons une approximation du minimum à  $10^{-5}$  près :

```
minimum(x->sqrt((x)^2+25)+sqrt((x-18)^2+49),7.0,5)
```

et on obtient 7.50000

### D2b Avec Sage

```
def minimum(f,xo,p):
Y=float(f(xo))
X=float(xo+10**(-p))
compteur=0
while f(X)<Y and compteur<10**p:
    Y=f(X)
    X+=10**(-p)
    compteur+=1
return X-10**(-p)
```

Programme 231 – minimum d'une fonction en impératif (Sage)

Par exemple :

```
def f(x):
    return sqrt((x)**2+25)+sqrt((x-18)**2+49)

sage: minimum(f,7,5)
7.4999999999810711
```

### D3 Recherche de minimum avec affinage progressif du pas

On peut améliorer la situation en affinant petit à petit le balayage : d'abord  $h$  puis  $h/10$ , etc.

#### D3 a Avec XCAS

```
min_bis(f,xo,h,p)={
if(abs(h)<p)then{xo}
else{if(f(xo+h)>f(xo))
then{min_bis(f,xo,-0.1*h,p)}
else{min_bis(f,xo+h,h,p)}
}
};;
```

Programme 232 – minimum d'une fonction en récursif 2 (XCAS)

#### D3 b Avec CAML

```
# let rec min_bis(f,xo,h,p)=
if abs_float(h)<p then xo
else if f(xo+h)>f(xo)
then min_bis(f,xo,-0.1*.h,p)
else min_bis(f,xo+h,h,p);;
```

Programme 233 – minimum d'une fonction en récursif 2 (CAML)

#### D3 c Avec Sage

```
def min_bis(f,xo,h,p):
if abs(h)<p : return xo
elif f(xo+h)>f(xo): return min_bis(f,xo,-0.1*h,p)
else : return min_bis(f,xo+h,h,p)
```

Programme 234 – minimum d'une fonction en récursif 2 (Sage)

## E Courbe du blanc-manger

On note  $d(x)$  la distance de  $x$  à l'entier le plus proche puis  $d_k(x) = \frac{d(2^k x)}{2^k}$  et enfin  $bm(x) = \sum_{k=0}^{+\infty} d_k(x)$ .

#### E0 d Avec XCAS

On a déjà défini la partie entière d'un réel page 37 :

```

pe(r):={
  si(r>=0) et (r<1)
    alors 0
    sinon si r>0
      alors 1+pe(r-1)
      sinon -1+pe(r+1)
    fsi
  fsi
};

```

On va l'utiliser pour définir  $d$  :

```

d(x):={
  si x-pe(x)<=0,5
    alors x-pe(x)
    sinon 1+pe(x)-x
  fsi
};

```

Programme 235 – distance d'un réel à l'entier le plus proche

Puis  $d_k$  :

```

D(k,x):=d(2.0^k*x)/2.0^k;

```

Pour  $bm$ , on utilisera bien sûr une somme partielle :

```

bm(n,x):={
  si n==0
    alors D(0,x)
    sinon D(n,x)+bm(n-1,x)
  fsi
};

```

Programme 236 – fonction « blanc-manger »

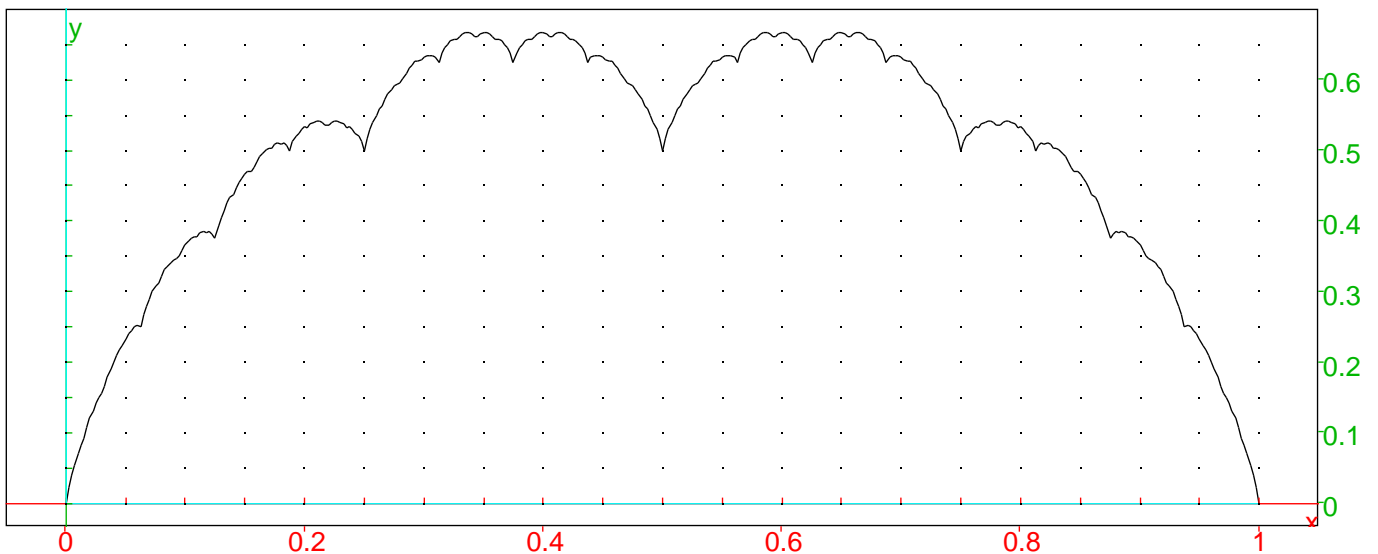
On demande le tracé de la somme partielle jusqu'à  $n = 10$  :

```

graphe(bm(10,x),x=0..1)

```

et on obtient :



On aurait pu utiliser les outils formels de XCAS :

```
d(k,x):=min(2^k*x-floor(2^k*x),ceil(2^k*x)-2^k*x)/2.0^k
plot(sum(d(k,x),k=0..10),x=0..1)
```

Il s'agit d'un exemple de fonction continue et nulle part dérivable.

## E 0 e Avec Sage

```
def d(x):
    if x-floor(x)<=0.5 : return x-floor(x)
    else : return 1+floor(x)-x

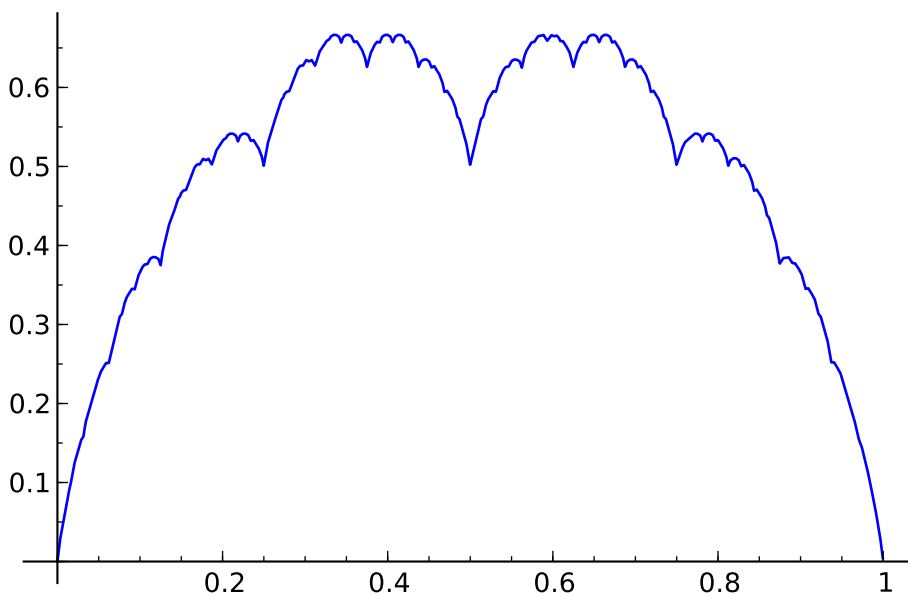
def D(k,x):
    return d(x*2.**k)/2.**k

def bm(n,x):
    if n==0 : return D(0,x)
    else : return D(n,x)+bm(n-1,x)

def gbm(n):
    l=line([[x,bm(n,x)] for x in srange(0,1,0.0001)])
    l.show()
```

Programme 237 – courbe du blanc-manger avec Sage

```
sage: gbm(10)
```



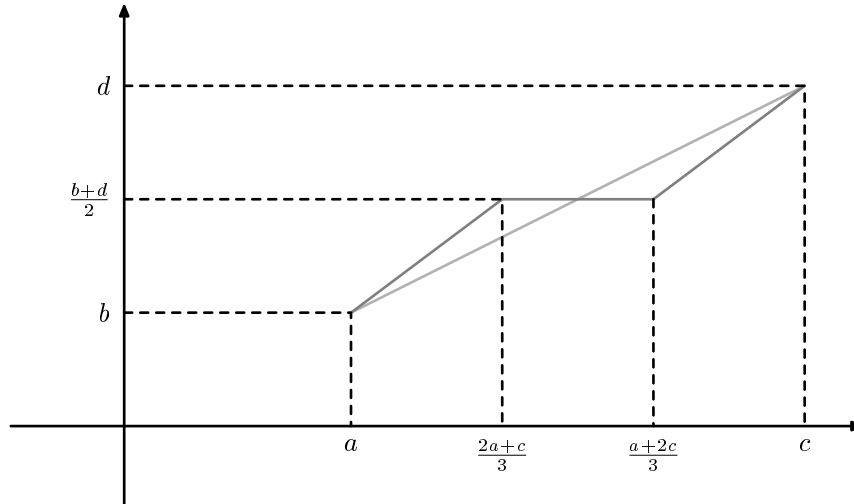
## F Escalier de Cantor

### F 1 Cas particulier

On part d'une certaine fonction  $f_0$  définie sur  $[0; 1]$  puis on définit  $f_1$  à partir de  $f_0$  :

$$f_1(x) = \begin{cases} \frac{1}{2}f_0(3x) & \text{si } x \in [0; \frac{1}{3}] \\ \frac{1}{2} & \text{si } x \in [\frac{1}{3}; \frac{2}{3}] \\ \frac{1}{2} + \frac{1}{2}f_0(3x-2) & \text{si } x \in [\frac{2}{3}; 1] \end{cases}$$

On prend par exemple  $f_0(x) = x$  :



On réitère sur chaque segment « oblique ». On peut faire travailler les élèves sur les coefficients directeurs.

### F1 a Avec XCAS

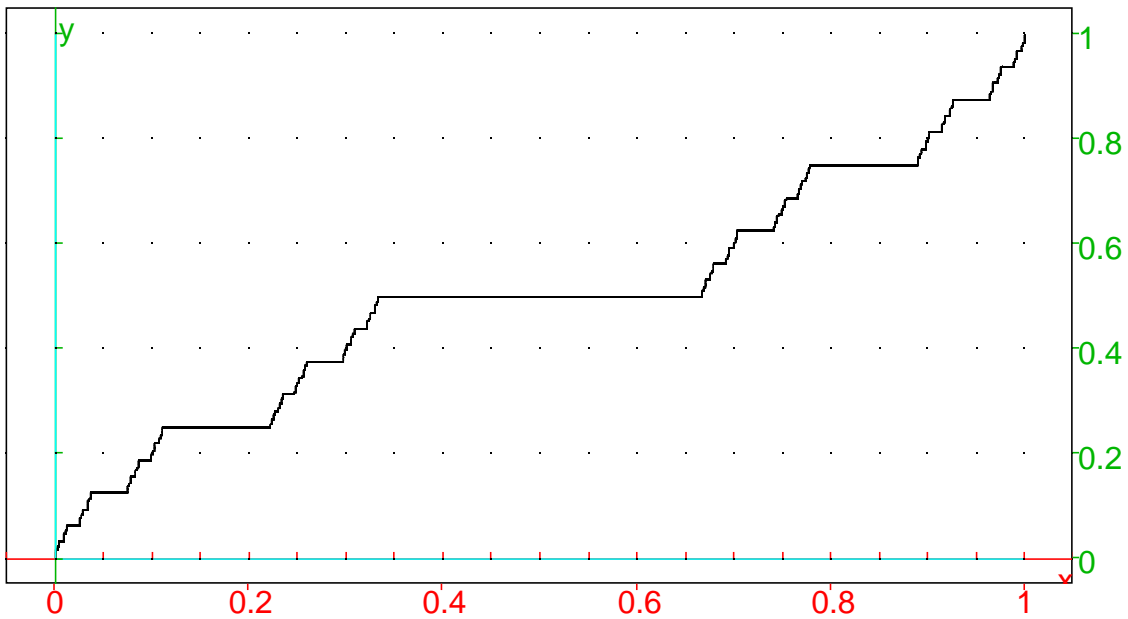
```
cantor(a,b,c,d,n) := {
  if(n==0)
    then{[a,b],[c,d]}
  else{cantor(a,b,(2*a+c)/3,(b+d)/2,n-1),
        cantor((2*a+c)/3,(b+d)/2,(a+2*c)/3,(b+d)/2,n-1),
        cantor((a+2*c)/3,(b+d)/2,c,d,n-1)}
};;
```

Programme 238 – escalier de Cantor en récursif (XAS)

Alors

```
polyplot([cantor(0,0,1,1,10)])
```

donne :



### F1 b Avec CAML

```
# let rec cantor(n,a,b,c,d)=
  if n=0 then (Graphics.moveto a b;Graphics.lineto c d)
  else (cantor(n-1,a,b,((2*a+c)/3),(b+d)/2);
        cantor(n-1,((2*a+c)/3),((b+d)/2),((a+2*c)/3),((b+d)/2));
        cantor(n-1,((a+2*c)/3),((b+d)/2),c,d););;
```

Programme 239 – escalier de Cantor en récursif (CAML)

Puis on trace :

```
# Graphics.open_graph ""; cantor(10,0,0,500,500);;
```

L'escalier de CANTOR est continu, a presque partout une tangente horizontale, et pourtant il monte!

### F1 c Avec Sage

```
def cantor(a,b,c,d,n):
  if n==0 : return [[a,b],[c,d]]
  else : return cantor(a,b,(2*a+c)/3,(b+d)/2,n-1)+cantor((2*a+c)/3,(b+d)/2,(a+2*c)/3,(b+d)/2, n
            -1)+cantor((a+2*c)/3,(b+d)/2,c,d,n-1)

def gcantor(a,b,c,d,n):
  l=line(cantor(a,b,c,d,n))
  l.show()
```

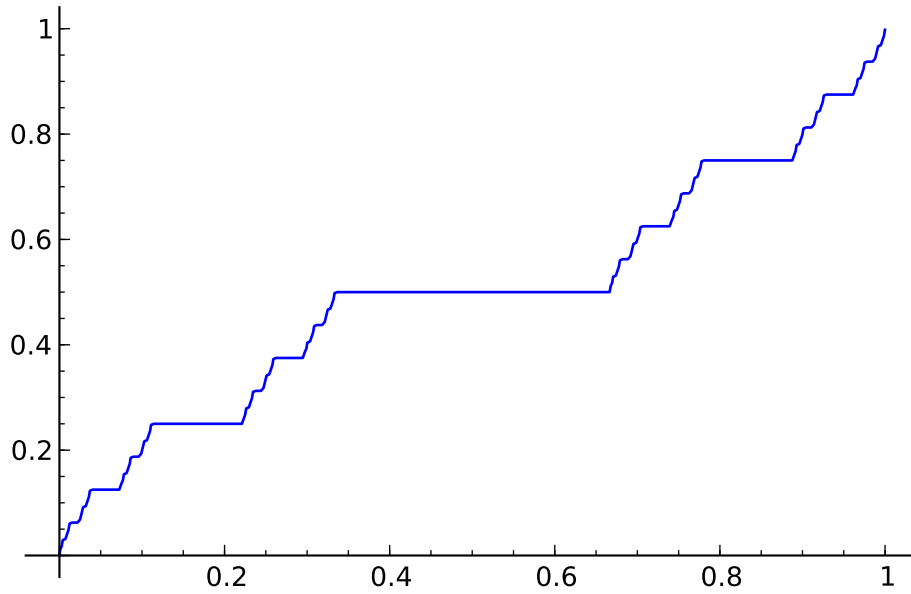
Programme 240 – escalier de Cantor en récursif (Sage)

Par exemple :

```
sage: gcantor(0,0,1,1,10)
```

donne :



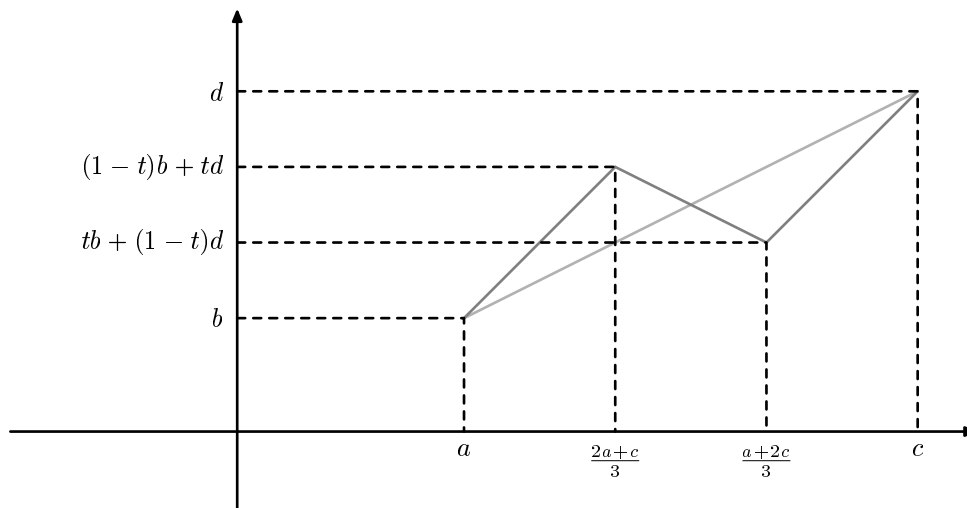


## F2 Généralisation

On peut prendre un autre coefficient que  $\frac{1}{2}$ .

$$f_1(x) = \begin{cases} t f_0(3x) & \text{si } x \in [0; \frac{1}{3}] \\ t + (1-2t) f_0(3x-1) & \text{si } x \in [\frac{1}{3}; \frac{2}{3}] \\ 1-t + t f_0(3x-2) & \text{si } x \in [\frac{2}{3}; 1] \end{cases}$$

Par exemple, pour  $f_0(x) = x$  et  $t = \frac{2}{3}$ , on a au départ :



### F2a Avec XCAS

```
zigzag(a,b,c,d,t,n):={
  if(n==0)
    then{[a,b],[c,d]}
  else{zigzag(a,b,(2*a+c)/3,(1-t)*b+t*d,t,n-1),
    zigzag((2*a+c)/3,(1-t)*b+t*d,(a+2*c)/3,t*b+(1-t)*d,t,n-1),
    zigzag((a+2*c)/3,t*b+(1-t)*d,c,d,t,n-1)}
```

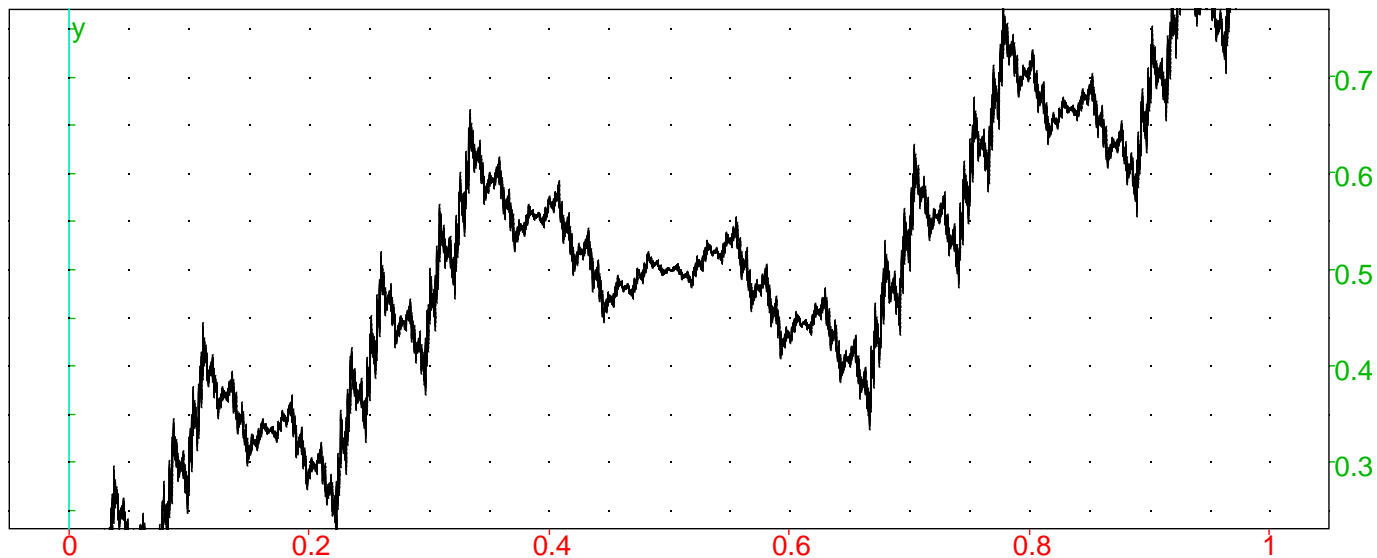
```
};;
```

Programme 241 – zig-zag de Cantor en récursif)

et

```
polygonplot([zigzag(0,0,1,1,2/3.,10)])
```

donne :



On obtient quelque chose qui s'approche de la courbe de BOLZANO-LEBESGUE qui est continue partout et dérivable nulle part...

## F2b Avec Sage

```
def zigzag(a,b,c,d,t,n):
    if n==0 : return [[a,b],[c,d]]
    else : return zigzag(a,b,(2*a+c)/3,(1-t)*b+t*d,t,n-1)+zigzag((2*a+c)/3,(1-t)*b+t*d,(a+2*c)/3,t
        *b+(1-t)*d,t, n-1)+zigzag((a+2*c)/3,t*b+(1-t)*d, c, d, t,n-1)

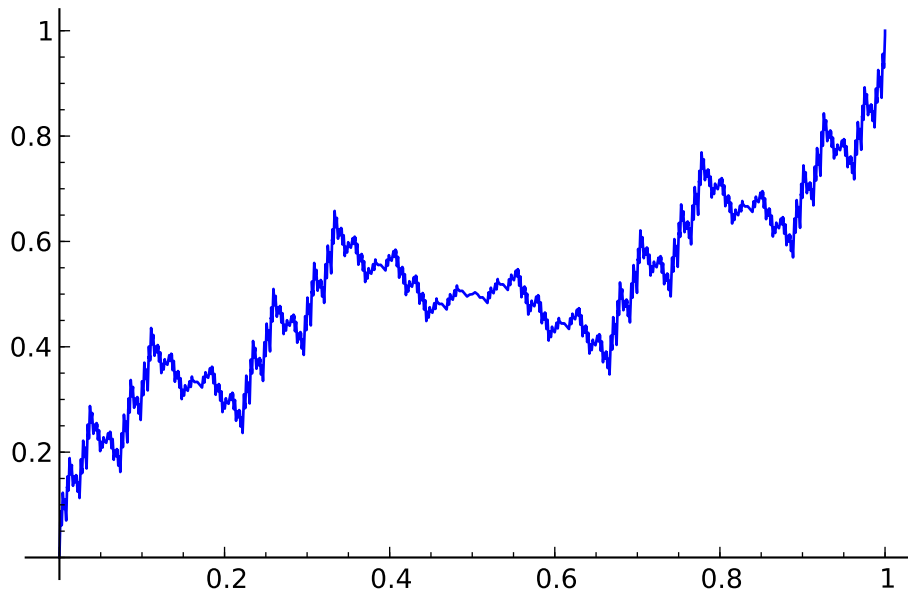
def gzigzag(a,b,c,d,t,n):
    l=line(zigzag(a,b,c,d,t,n))
    l.show()
```

Programme 242 – zigzag de Cantor en récursif( Sage)

Par exemple :

```
sage: gzigzag(0,0,1,1,2/3.,10)
```

donne :



On adapte facilement avec CAML.

## G Dichotomie

Pour résoudre une équation du type  $f(x) = 0$ , on recherche graphiquement un intervalle  $[a, b]$  où la fonction semble changer de signe.

On note ensuite  $m$  le milieu du segment  $[a, b]$ . On évalue le signe de  $f(m)$ .

Si c'est le même que celui de  $f(a)$  on remplace  $a$  par  $m$  et on recommence. Sinon, c'est  $b$  qu'on remplace et on recommence jusqu'à obtenir la précision voulue.

### G1 Version récursive

#### G1a En XCAS

```
dicho_rec(f,a,b,eps):={
  if(evalf(b-a)<eps){return(0.5*(b+a))};
  if(f(a)*f(0.5*(b+a))>0){return(dicho(f,0.5*(b+a),b,eps))}
  else{return(dicho(f,a,0.5*(b+a),eps))}
};;
```

Programme 243 – dichotomie récursive (XCAS)

### G2 En CAML

```
# let rec dicho_rec(f,a,b,eps)=
  if abs_float(b-.a)<eps then 0.5*. (b+.a)
  else if f(a)*.f(0.5*. (b+.a))>0. then dicho_rec(f,0.5*. (b+.a),b,eps)
  else dicho_rec(f,a,0.5*. (b+.a),eps);;
```

Programme 244 – dichotomie récursive (CAML)

ce qui donne :

```
# dicho_rec( (fun x->(x*.x-.2.)),1.,2.,0.00001);;
- : float = 1.41421127319335938
```

### G3 Avec Sage

```
def dichotomie(f,a,b,p,compteur):
    if abs(b-a).n(digits=p)<10**(-p) : return [0.5*(b+a).n(digits=p+len(str(floor(a)))), compteur]
    elif f(a)*f(0.5*(b+a))>0 : return dichotomie(f,0.5*(b+a).n(digits=p+len(str(floor(a)))),b,p,compteur
        +1)
    else : return dichotomie(f,a,0.5*(b+a).n(digits=p+len(str(floor(a)))),p,compteur+1)
```

Programme 245 – dichotomie en récursif (Sage/Python)

Alors :

```
sage: def f(x): return x**2-2
.....:
sage: dichotomie(f,1,2,5,0)
[1.41421, 17]

sage: def f(x): return x**2-153
.....:
sage: dichotomie(f,10,20,5,0)
[12.36931, 20]
```

Le rajout de `len(str(floor(a)))` permet d'adapter au nombre de chiffres de la partie entière.

### G4 Version impérative

**Entrées** : une fonction  $f$ , les bornes  $a$  et  $b$ , une précision  $p$

**Initialisation** :  $aa \leftarrow a$ ,  $bb \leftarrow b$

**début**

**tant que**  $bb - aa > p$  **faire**

**si** *le signe de  $f((aa+bb)/2)$  est le même que celui de  $f(bb)$*  **alors**

$bb \leftarrow (aa+bb)/2$

**sinon**

$aa \leftarrow (aa+bb)/2$

**retourner**  $(aa+bb)/2$

**fin**

Algorithme 42 : dichotomie

### G4a Avec XCAS

Avec XCAS, il faut juste ne pas oublier de régler quelques problèmes de précision :

```
dichotomie(F,p,a,b):={
local aa,bb,k,f;
aa:=a;
bb:=b;
epsilon:=1e-100; /* on règle le "zéro" de XCAS a une valeur suffisamment petite */
f:=unapply(F,x); /* on transforme l'expression f en fonction */
k:=0; /* on place un compteur */
tantque evalf(bb-aa,p)>10^(-p) faire
si sign(evalf(f((bb+aa)/2),p))==sign(evalf(f(bb),p))
alors bb:=evalf((aa+bb)/2,p);
sinon aa:=evalf((aa+bb)/2,p);
k:=k+1; /* un tour de plus au compteur */
fsi;
ftantque;
retourne evalf((bb+aa)/2,p)+" est la solution trouvee apres "+k+" iterations";
};;
```

Programme 246 – dichotomie impérative

**G 4 b Avec Sage/Python**

```
def dichotomie(f, a, b, p):
    [aa, bb, compteur, c] = [a, b, 0, len(str(floor(a)))]
    while (bb - aa).n(digits=p) > 10 ** (-p):
        if f(0.5 * (bb + aa)) * f(bb) > 0:
            bb = (0.5 * (aa + bb)).n(digits=p + c)
        else:
            aa = (0.5 * (aa + bb)).n(digits=p + c)
        compteur = compteur + 1
    return [(0.5 * (aa + bb)).n(digits=p + c), compteur]
```

Programme 247 – dichotomie en impératif (Sage)

**H Méthode des parties proportionnelles**

Il peut être parfois plus efficace de ne plus considérer le milieu de  $[a; b]$  mais l'abscisse de l'intersection de la droite passant par les points  $(a, f(a))$  et  $(b, f(b))$  avec l'axe des abscisses. Alors :

$$\frac{x - a}{0 - f(a)} = \frac{b - x}{f(b) - 0}$$

avec  $x$  vérifiant  $f(x) = 0$ .

On obtient :

$$x = \frac{af(a) - bf(b)}{f(b) - f(a)}$$

Il suffit de remplacer  $\frac{a+b}{2}$  par le résultat précédent.

**H 1 Avec CAML en récursif**

```
# let rec propor_rec(f, a, b, eps) =
  if abs_float(b -. a) < eps
  then (a *. f(b) -. b *. f(a)) /. (f(b) -. f(a))
  else
    if f(a) *. f((a *. f(b) -. b *. f(a)) /. (f(b) -. f(a))) > 0.
    then propor_rec(f, (a *. f(b) -. b *. f(a)) /. (f(b) -. f(a)), b, eps)
    else propor_rec(f, a, (a *. f(b) -. b *. f(a)) /. (f(b) -. f(a)), eps);;
```

Programme 248 – partie proportionnelles en récursif

**H 2 Avec XCAS en impératif**

```
propor(F, p, a, b) := {
  local aa, bb, k, f, xo;
  aa := a;
  bb := b;
  epsilon := 1e-100; /* on règle le "zéro" de XCAS a une valeur suffisamment petite */
  f := unapply(F, x); /* on transforme l'expression f en fonction */
  k := 0; /* on place un compteur */
```

```

tantque evalf(bb-aa,p)>10^(-p) faire
  xo:=(aa*f(bb)-bb*f(aa))/(f(bb)-f(aa));
  si evalf(f(xo),p)*evalf(f(bb),p)>0
  alors bb:=evalf(xo,p);
  sinon aa:=evalf(xo,p);
  k:=k+1; /* un tour de plus au compteur */
  fsi;
ftantque;
retourne evalf(xo,p)+" est la solution trouvée après "+k+ " itération(s)";
};

```

Programme 249 – parties proportionnelles en impératif (XCAS)

On remarque que

```
propor(x^2-2,10,1,2)
```

donne :

1.414213562 est la solution trouvée après 20 itération(s)

mais que :

```
propor(x^2-2,10,-2,-1)
```

donne :

-1.414213562 est la solution trouvée après 1 itération(s)

La concavité joue un rôle important pour cette méthode (faites un dessin...) alors que les résultats ne changent pas pour la dichotomie.

## I Méthode de Newton

Les prérequis sont plus nombreux! Il faut avoir étudié les suites définies par une relation du type  $u_{n+1} = f(u_n)$  et la dérivation... et ne pas être trop perdu en Analyse...

Voici un exemple de TP possible en terminale.

### I1 Historique

La méthode de résolution des équations numériques que nous allons voir aujourd'hui a été initiée par Isaac NEWTON vers 1669 sur des exemples numériques mais la formulation est fastidieuse. Dix ans plus tard, Joseph RAPHSON met en évidence une formule de récurrence. Un siècle plus tard, MOURAILLE et LAGRANGE étudient la convergence des approximations successives en fonction des conditions initiales par une approche géométrique. Cinquante ans plus tard, FOURIER et CAUCHY s'occupe de la rapidité de la convergence.

### I2 Principe

NEWTON présenta sa méthode en traitant l'équation  $x^3 - 2x - 5 = 0$ .

Soit  $f$  la fonction  $x \mapsto x^3 - 2x - 5$ . On peut rapidement tracer son graphe à l'aide d'un outil quelconque.

Il coupe l'axe des abscisses pour une valeur comprise entre 2 et 3.

On assimile la courbe à sa tangente au point d'abscisse 2.

Celle-ci a pour équation  $y = (x - 2) \times f'(2) + f(2) = 10(x - 2) - 1$

Posons  $x = 2 + \varepsilon$  alors  $f(2 + \varepsilon) = 0 \iff \varepsilon^3 + 6\varepsilon^2 + 10\varepsilon - 1 = 0$ .

Si on assimile la courbe à sa tangente en 2, alors  $\varepsilon$  vérifie aussi  $0 = 10(2 + \varepsilon - 2) - 1$  c'est-à-dire  $\varepsilon = 0,1$ .

En fait, assimiler la courbe à sa tangente, c'est négliger les termes en  $\varepsilon$  d'ordre supérieur à 1 (ici  $\varepsilon^3 + 6\varepsilon^2$ ).

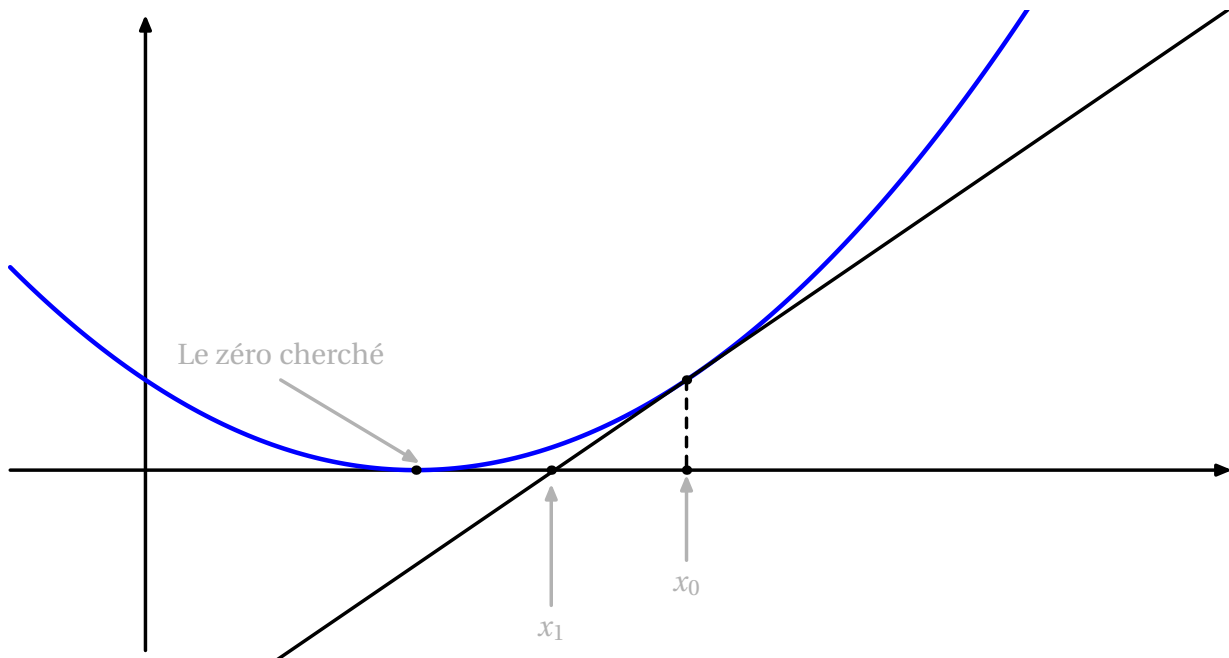
Un approximation de la solution est donc 2,1.

On recommence ensuite en partant de 2,1 au lieu de 2 puis on reprendra la nouvelle valeur trouvée comme valeur de départ, etc.

### 13 La formule de récurrence

On sent la procédure algorithmique poindre son nez. Pour finir de la mettre en évidence, nous allons formaliser la méthode précédente en introduisant la suite des approximations successives.

- On part d'un nombre quelconque  $x_0$  ;
- à partir de  $x_0$ , on calcule un nouveau nombre  $x_1$  de la manière suivante (voir figure) : on trace la tangente au graphe de  $f$  au point d'abscisse  $x_0$ , et on détermine le point d'intersection de cette tangente avec l'axe des abscisses. On appelle  $x_1$  l'abscisse de ce point d'intersection ;
- et on recommence : on calcule un nouveau nombre  $x_2$  en appliquant le procédé décrit au point 2 où l'on remplace  $x_0$  par  $x_1$  ;
- etc.



À partir de cette description graphique de la méthode de NEWTON, trouver la formule, notée (1), donnant  $x_1$  en fonction de  $x_0$ , puis  $x_{n+1}$  en fonction de  $x_n$ .

Quelles hypothèses doit-on faire sur  $f$  et les  $x_n$  pour que la formule ait un sens ?

Nous n'irons pas plus loin pour l'instant concernant la convergence de ces suites. Traitons l'exemple de NEWTON.

### 14 Étude de la suite associée à l'équation $x^3 - 2x - 5 = 0$ .

On veut résoudre l'équation  $x^3 - 2x - 5 = 0$  par la méthode de NEWTON-RAPHSON appelée aussi méthode de la tangente. On note  $f$  la fonction  $x \mapsto x^3 - 2x - 5$ .

1. Montrez rapidement que l'équation  $f(x) = 0$  admet une unique solution  $\alpha$  sur  $\mathbb{R}$ . Montrez que  $2 < \alpha < 3$ .
2. Déterminez la fonction  $\varphi$  telle que  $x_{n+1} = \varphi(x_n)$ , la suite  $(x_n)$  étant celle décrite au paragraphe précédent en prenant  $x_0 = 3$ .
3. Étudiez le sens de variation de la fonction  $\varphi$  puis celui de  $\varphi'$  et déduisez-en que  $[\alpha, 3]$  est stable par  $\varphi$  et que  $\varphi$  est strictement croissante sur  $[\alpha, 3]$ .
4. Que pouvez-vous en déduire sur la convergence de la suite  $x_n$  ?

### 15 Test d'arrêt

Afin de construire un algorithme donnant une approximation d'une solution d'une équation numérique par la méthode de NEWTON-RAPHSON, il faudrait déterminer un test d'arrêt c'est-à-dire savoir à partir de quel rang  $n$   $|x_n - \alpha|$  restera inférieur à une valeur donnée.

Il suffit de remarquer que  $f(x_n) = f(x_n) - f(\alpha)$ . Nous supposons  $f$  de classe  $\mathcal{C}^2$  sur un « bon » voisinage  $I$  de  $\alpha$  (ce critère nous échappe encore à notre niveau). Alors  $f$  est en particulier dérivable en  $\alpha$  donc

$$f(x_n) = f(x_n) - f(\alpha) \sim f'(\alpha) \times (x_n - \alpha)$$

C'est-à-dire, puisque  $f'(\alpha)$  est supposé non nul :

$$x_n - \alpha \sim \frac{f(x_n)}{f'(\alpha)}$$

Or  $f$  étant de classe  $\mathcal{C}^2$ , on a  $f'$  continue et non nulle en  $\alpha$  donc  $f'(\alpha) \sim f'(x_n)$ .  
Finalement

$$x_n - \alpha \sim \frac{f(x_n)}{f'(x_n)} = x_n - x_{n+1}$$

Nous choisirons donc comme test d'arrêt  $\left| \frac{f(x_n)}{f'(x_n)} \right| < p$  avec  $p$  la précision choisie.  
Il ne reste plus qu'à écrire l'algorithme.

## 16 Algorithme récursif

D'abord l'algorithme sous forme récursive. En fait, tout a déjà été dit alors traduisons directement.

### 16 a Avec XCAS

On notera que `function_diff(f)` renvoie la *fonction* dérivée donc `function_diff(f)(x0)` désigne le *nombre* dérivé de  $f$  en  $x_0$ .

```
newton_rec(f,x0,eps):={
  if(evalf(abs(f(x0)/function_diff(f)(x0)))<eps){evalf(x0)}
  else{newton_rec(f,evalf(x0-f(x0)/function_diff(f)(x0)),eps)}
};;
```

Programme 250 – méthode de Newton-Raphson en récursif (XCAS)

Alors, après avoir fixé par exemple la précision à 100 :

```
DIGITS:=100;;
newton_rec(x->x^2-2,1.0,evalf(10^(-99)))
```

On obtient immédiatement :

1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276415728

### 16 b Avec CAML

Il y a un petit plus car il faut définir une dérivée approchée.

```
# let der(f,x,dx)=(f(x+.dx)-.f(x))/.dx;;
```

Maintenant la partie Newton :

```
# let rec newton_rec(f,xo,dx,eps)=
  if abs_float(f(xo)/.der(f,xo,dx))<eps then xo
  else newton_rec(f,xo-.f(xo)/.der(f,xo,dx),dx,eps);;
```

Programme 251 – méthode de Newton-Raphson en récursif (CAML)

Par exemple :

```
# let k(x)=x*.x-.2.;;

# newton_rec(k,1.,0.0001,0.000000001);;
- : float = 1.41421356245305962
```



**16 c** Avec Sage

```

def der(f,x):
    t=var('t')
    return diff(f(t),t).subs(t=x)

def newton_rec(f,xo,p):
    if abs(f(xo)/der(f,xo)).n(digits=p)<10**(-p) : return xo.n(digits=p)
    else : return newton_rec(f,xo-f(xo)/der(f,xo),p)

sage: newton_rec(f,1.,15)
1.41421356237310

```

Programme 252 – méthode de Newton-Raphson en récursif (Sage)

**17** Algorithme impératif

Ça se complique un peu : évidemment, car il faut faire de l'informatique au lieu de se concentrer sur les mathématiques...

**Entrées :**fonction  $f$ précision  $p$ premier terme  $a_0$ 

nombre maximum d'itération // Pour éviter de « planter » si on tombe sur un cas pathologique

**Initialisation :**  $fp \leftarrow$  dérivée de  $f$ compteur  $\leftarrow 0$  // le compteur d'itérations $un \leftarrow u_0 - \frac{f(u_0)}{fp(u_0)}$ **début**

tant que	$\left  \frac{f(x_n)}{fp(x_n)} \right $ est plus grand que la précision $p$ et que $k < N$ faire
----------	--

si	$fp(un) = 0$ alors
----	--------------------

└	On sort de la boucle avant de diviser par zéro et on explique pourquoi
---	--

$un \leftarrow un - \frac{f(un)}{fp(un)}$
---

compteur $\leftarrow$ compteur+1
----------------------------------

**fin****retourner** L'approximation et la valeur du compteur**Algorithme 43 :** NEWTON-RAPHSON : version impérative

Comparez ensuite avec les résultats trouvés avec la dichotomie.

Regardez également ce qui se passe avec l'équation  $(x-1)^4 = 0$  : quelle précaution supplémentaire faut-il prendre ? (La preuve n'est évidemment pas envisageable au Lycée).

**17 a** Avec Sage

```

def newton_imp(f,xo,p,Nmax):
    def fp(x):
        return der(f,x)
    c=len(str(floor(xo)))
    xn = xo-f(xo)/fp(xo).n(digits=p+c)
    k = 0
    while abs(f(xn)/fp(xn))>=10**(-p) and k<Nmax :
        if fp(xn)==0:
            return 'la dérivée est nulle'
        else:
            xn = (xn - f(xn)/fp(xn)).n(digits=p+c)
            k = k+1

```

```
return [xn,k]
```

Programme 253 – NEWTON-RAPHSON : version impérative (Sage)

Ce qui donne pour  $x \mapsto x^2 - 2$  :

```
sage: def f(x):
....: return x**2-2
....:
sage: newton_imp(f,1.,70,100)
[1.4142135623730950488016887242096980785696718753769480731766797379907325, 6]
```

## J Fractale de Newton

Considérons l'équation  $z^5 - 1 = 0$ . Elle admet cinq solutions dans  $\mathbb{C}$ .

### J1 Avec XCAS

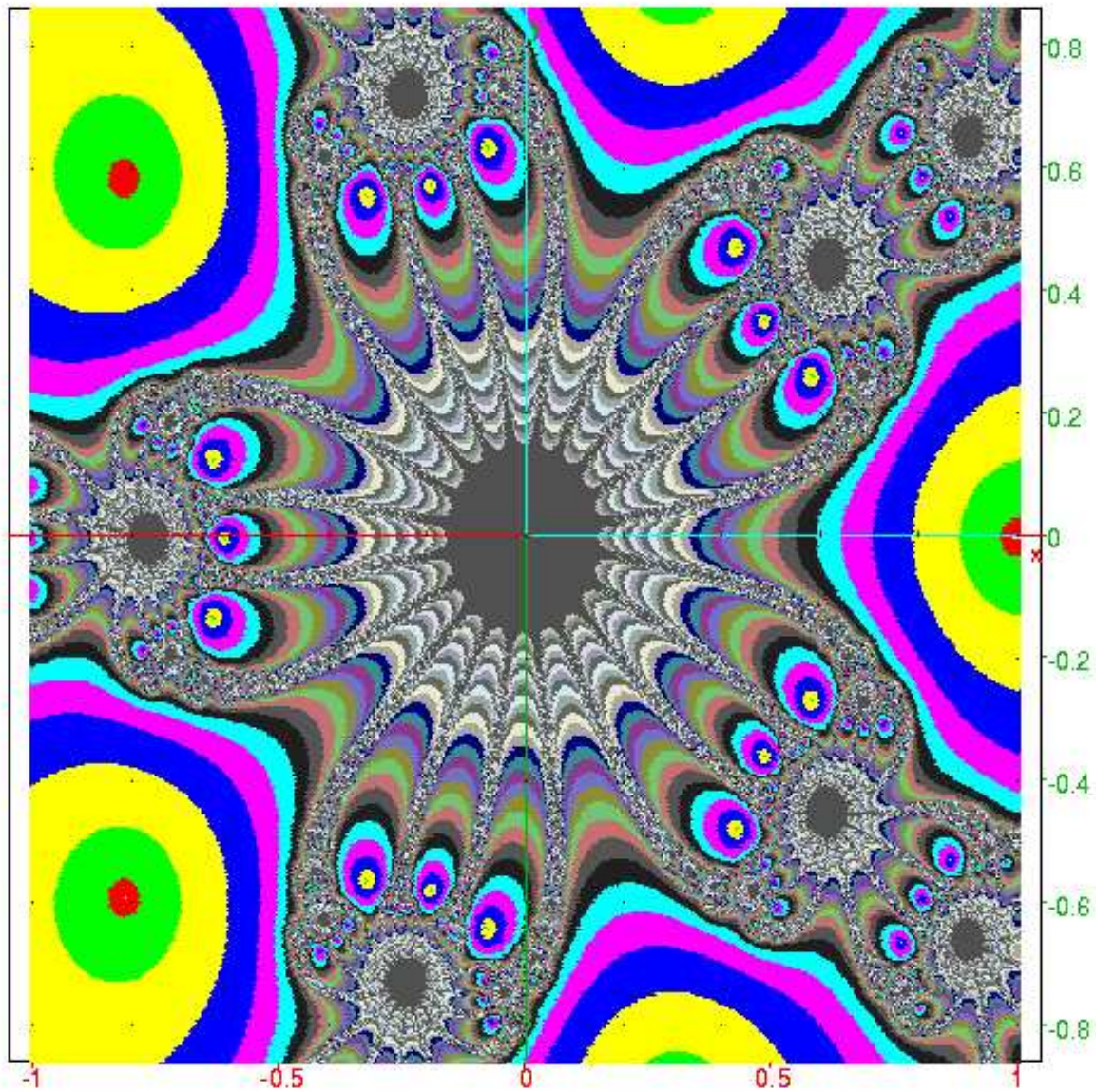
Étudiez le programme suivant :

```
coul(a,b)={
local z,j,c;
z:=a+i*b;
c:=0;
while((abs(z^n-1)>=0.01) and (c<20))
{z:=z-(z^n-1)/(n*z^(n-1));
c:=c+1};
return(c)
};;
```

Ensuite, on trace dans le carré  $[-1;1] \times [-1;1]$  des points de coordonnées  $(a,b)$  colorés avec la couleur  $\text{coul}(a,b)$  : en effet, chaque couleur peut être représentée par un nombre.

```
seq(seq(couleur(point([a*0.002,b*0.002])),coul(a*0.002,b*0.002)+point_point+point_width_2),a=-500..500),b=-500..500)
```

Cela nécessite pas mal de calculs et donc prend BEAUCOUP de mémoire.  
Voici ce que cela donne : c'est beau mais comment interpréter ce graphique ?

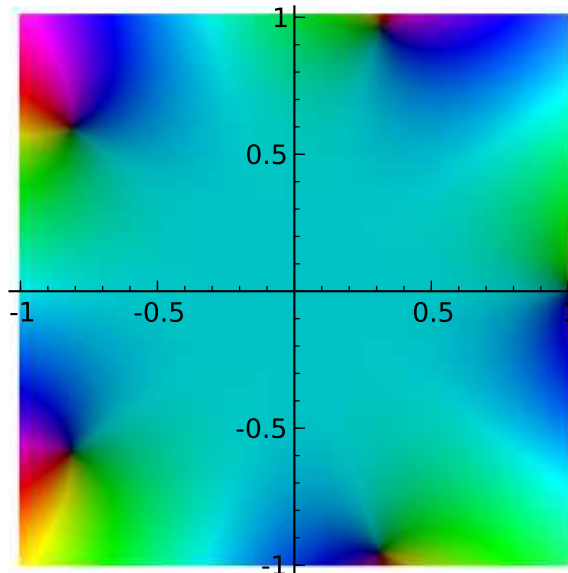


## J2 Avec Sage

On utilise la commande `complex_plot`(équation, (xmin,xmax), (ymin,ymax), plot\_points) :

```
sage: sage: z=var('z'); p=complex_plot(z^5-1, (-1,1), (-1,1), plot_points=3000)
sage: p.show()
```

Ce qui donne :



## K Méthode d'Euler

Supposons qu'on connaisse  $f'(x)$  en fonction de  $x$  et de  $f(x)$  sous la forme  $f'(x) = u(f(x), x)$ . On utilise le fait que  $f'(x) \approx \frac{f(x+h) - f(x)}{h}$ . Alors

$$f(x+h) \approx h \cdot u(f(x), x) + f(x)$$

La traduction algorithmique est alors directe.

### K1 Version récursive

#### K1a Avec XCAS

Pour la liste des coordonnées de points :

```
Euler(u, x, xmax, y, h, liste) := {
  if(x >= xmax)
    then{liste}
    else{ Euler(u, x+h, xmax, y+u(y, x)*h, h, [op(liste), [x, y]]) }
};
```

Programme 254 – méthode d'Euler générale en récursif (XCAS)

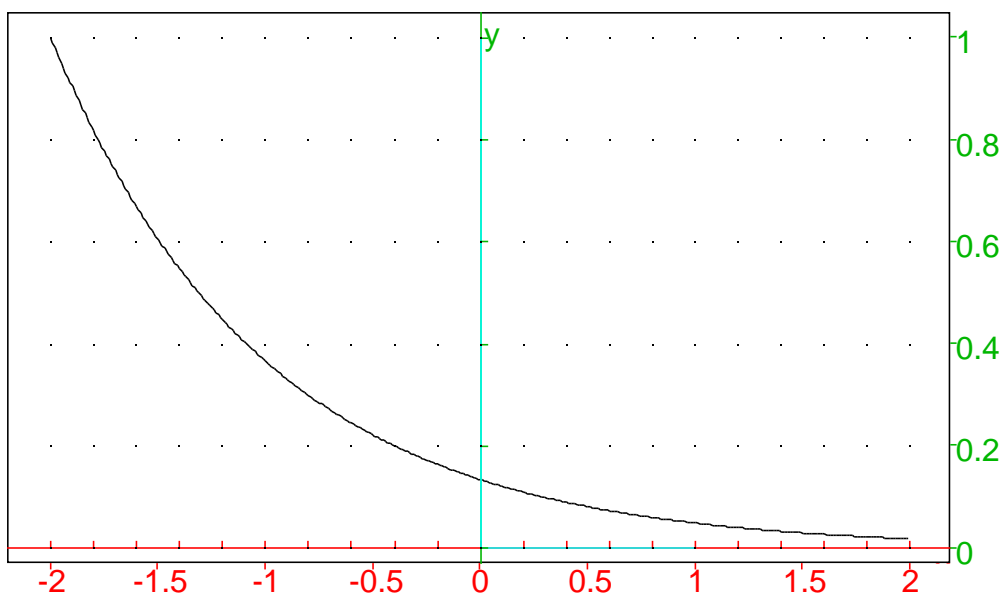
Puis pour le tracé :

```
trace_euler(u, xo, xmax, yo, h) := {
  polygonplot([Euler(u, xo, xmax, yo, h, [])])
};
```

Alors, pour résoudre graphiquement  $y' = -y$  avec  $y(-2) = 1$ , sur  $[-2; 2]$  avec un pas de 0,01 on entre :

```
trace_euler((y, x) -> -y, -2, 2, 1, 0.01)
```

et on obtient :



### K1b Avec Sage

```
def euler(u,x,xmax,y,h,liste):
    if x>=xmax : return liste
    else : return euler(u,x+h,xmax,y+u(y,x)*h,h,liste+[[x,y]])

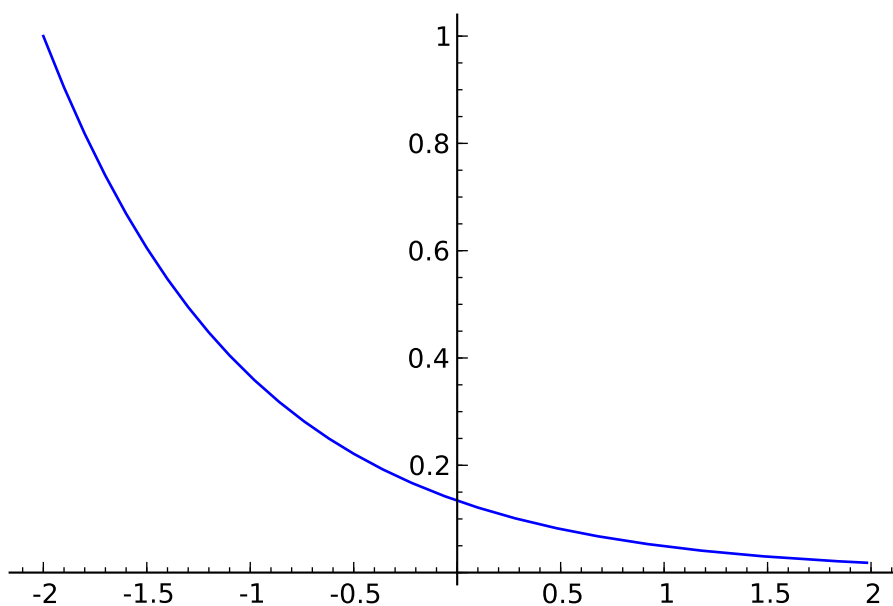
def trace_euler(u,xo,xmax,yo,h):
    p=line(euler(u,xo,xmax,yo,h,[]))
    p.show()
```

Programme 255 – méthode d’Euler générale en récursif (Sage)

Alors :

```
sage: def e(y,x):
....: return -y
....:
sage: trace_euler(e,-2,2,1,0.01)
```

donne :



## K2 Version impérative

### K2a Avec XCAS

On effectue une division régulière en découpant notre intervalle d'étude  $[a; b]$  en  $N$  points :

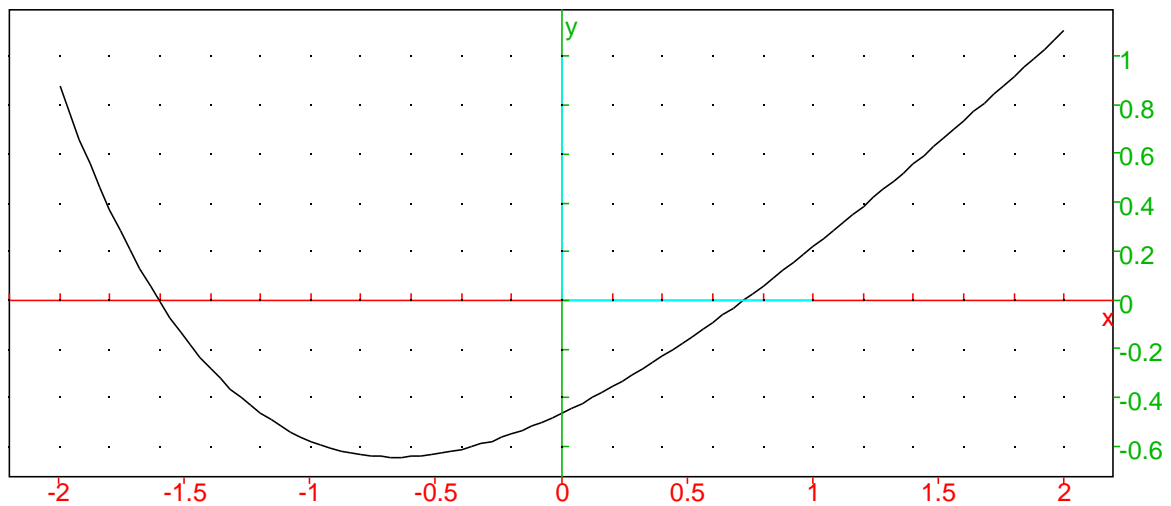
```
EulerImp(u,N,a,b,yo):={
S:=NULL;
X:=a;
Y:=yo;
h:=(b-a)/N
pour k de 0 jusque N pas 1 faire
X:=a+k*h;
Y:=Y+u(Y,X)*h;
S:=S,[X,Y];
fpour
polyplot([S]);
};;
```

Programme 256 – méthode d'Euler générale en impératif

Ensuite, pour avoir la solution de  $y' = -y + x$  sur  $[-2;2]$  avec  $y(-2) = 1$  on entre :

```
EulerExpo((y,x)->-y+x,100,-2,2,1)
```

et on obtient :



### K 2 b Avec Sage

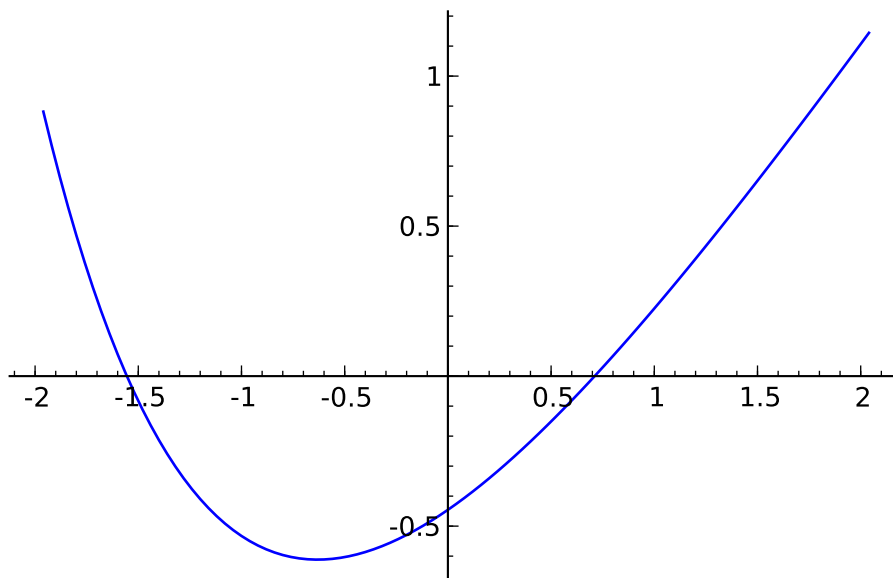
```
def eulerimp(u,N,a,b,yo):
    S=[]
    X=a
    Y=yo
    h=(b-a)/N
    for k in [0..N]:
        X+=h
        Y+=u(Y,X)*h
        S+=[[X,Y]]
    P=line(S)
    P.show()
```

Programme 257 – méthode d'Euler générale en impératif (Sage)

Alors :

```
sage: def u(y,x):
.....: return -y+x
.....:
sage: eulerimp(u,100,-2,2,1)
```

donne :



### K3 En Terminale : introduction de l'exponentielle

#### K3a Avec XCAS

Voici un petit programme XCAS qui construit la courbe approchée par la méthode d'Euler, c'est-à-dire que l'on va tracer la courbe correspondant à la fonction vérifiant :

- $f' = f$ ;
- $f(x_0) = y_0$ ;
- $f$  est définie sur  $[x_0; x_{final}]$ ;
- On va faire des pas de  $h$ .

Sur XCAS, les coordonnées d'un point se notent entre crochets et pour tracer la ligne polygonale passant par des points dont on a la liste des coordonnées, on utilise la commande `polygone_ouvert([liste des coordonnées])`

```
EulerExpo(xo,yo,xfinal,h):={
X:=xo; /* au départ X vaut a */
Y:=yo; /* au départ Y vaut yo */
ListePoints:=[X,Y] /* il y a un point au départ dans notre liste */

tantque abs(X) < abs(xfinal) faire /* pour tester même si h est négatif */
  X:=X+h; /* on avance de h */
  Y:=(1+h)*Y; /* f(X+h)=(1+h).f(X) */
  ListePoints:=append(ListePoints,[X,Y]); /* on rajoute à notre liste [X,Y] */
ftantque /* fin de la boucle */

polygone_ouvert(ListePoints); /* on relie les points de la liste "à la règle" */
};;
```

Programme 258 – méthode d'Euler pour l'exponentielle en impératif (XCAS)

Par exemple, pour avoir le tracé entre  $-2$  et  $2$  sachant que  $f(0) = 1$  et en prenant un pas de  $0,01$  on entre :

```
EulerExpo(0,1,-2,-0.01),EulerExpo(0,1,2,0.01)
```

Analysez ce programme plus rapide à écrire :

```
Euler(x,y,xfinal,h,ListePoints):={
si abs(x)>=abs(xfinal)
alors polygone_ouvert(ListePoints)
```



```

sinon Euler(x+h,(1+h)*y,xfinal,h,append(ListePoints,[x,y]))
fsi
};;

```

Programme 259 – méthode d'Euler pour l'exponentielle en récursif (XCAS)

que vous pouvez tester en tapant :

```
Euler(0,-2,1,-0.01,[]),Euler(0,2,1,0.01,[])
```

### K3b Avec Sage

La même avec Sage :

```

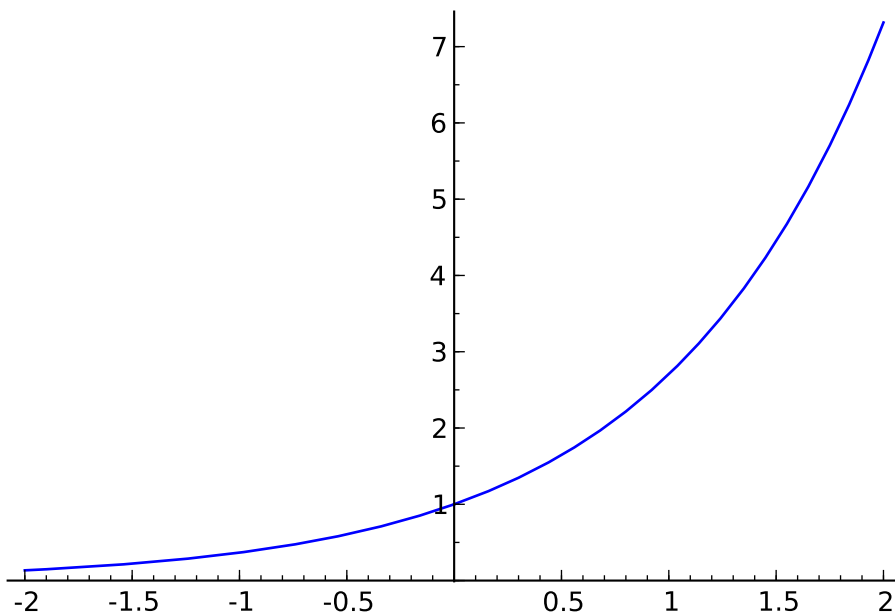
def eulerexpo(xo,yo,xfinal,h):
    X=xo # au départ X vaut a
    Y=yo # au départ Y vaut yo
    ListePoints=[[X,Y]] # il y a un point au départ dans notre liste
    while abs(X)<abs(xfinal): # pour tester même si h est négatif
        X+=h # on avance de h
        Y*=(1+h) # f(X+h)=(1+h).f(X)
        ListePoints+=[[X,Y]] # on rajoute à notre liste [X,Y]
    return line(ListePoints) # on relie les points de la liste "à la règle"

```

Programme 260 – méthode d'Euler pour l'exponentielle en impératif (Sage)

Par exemple, pour avoir le tracé entre -2 et 2 sachant que  $f(0) = 1$  et en prenant un pas de 0,01 on entre :

```
sage: (eulerexpo(0,1,2,0.01)+eulerexpo(0,1,-2,-0.01)).show()
```



En récursif :

```

def eulerexporec(x,y,xfinal,h,ListePoints):
    if abs(x)>=abs(xfinal):
        return line(ListePoints)
    else :
        return eulerexporec(x+h,(1+h)*y,xfinal,h,ListePoints+[[x,y]])

```

Programme 261 – méthode d'Euler pour l'exponentielle en récursif (Sage)

On l'invoque ainsi :

```
sage: (eulerexporec(0,1,2,0.01,[]) + eulerexporec(0,1,-2,-0.01,[])).show()
```

## K4 En Terminale : introduction du logarithme

### K4 a Avec XCAS

Construisons une approximation du graphe de  $\ln$  sachant que la dérivée de  $\ln$  est la fonction inverse et que  $\ln 1 = 0$ . Rappelons brièvement le principe de la méthode.

On utilise le fait que  $\frac{1}{x} = f'(x) \approx \frac{f(x+h)-f(x)}{h}$ . Alors

$$f(x+h) \approx h \cdot \frac{1}{x} + f(x)$$

La traduction algorithmique est alors directe.

```
EulerLn(x,y,xlimite,h,liste_points):={
if(h>=0)then{condition:=x>=xlimite}else{condition:=x<=xlimite}
if(condition)
then{liste_points}
else{ EulerLn(x+h,y+h/x,xlimite,h,[op(liste_points)],[x,y]]}
};;
```

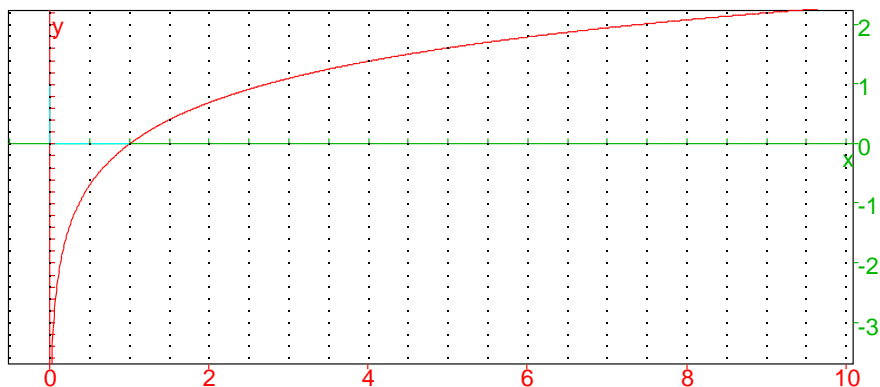
Programme 262 – construction  $\ln$  (XCAS)

Puis pour le tracé :

```
trace_EulerLn(xo,yo,xlimite,h):={
polygonplot([EulerLn(xo,yo,xlimite,h,[])])
};;
```

Ce qui donne pour le tracé sur  $[0,01; 10]$  :

```
trace_EulerLn(1,0,10,0.01), trace_EulerLn(1,0,0.01,-0.01)
```



### K4 b Avec Sage

```
def eulerln(x,y,xlimite,h,liste_points):
if h>=0: condition = x>=xlimite
else : condition = x<=xlimite
if condition :
return line(liste_points)
else :
```

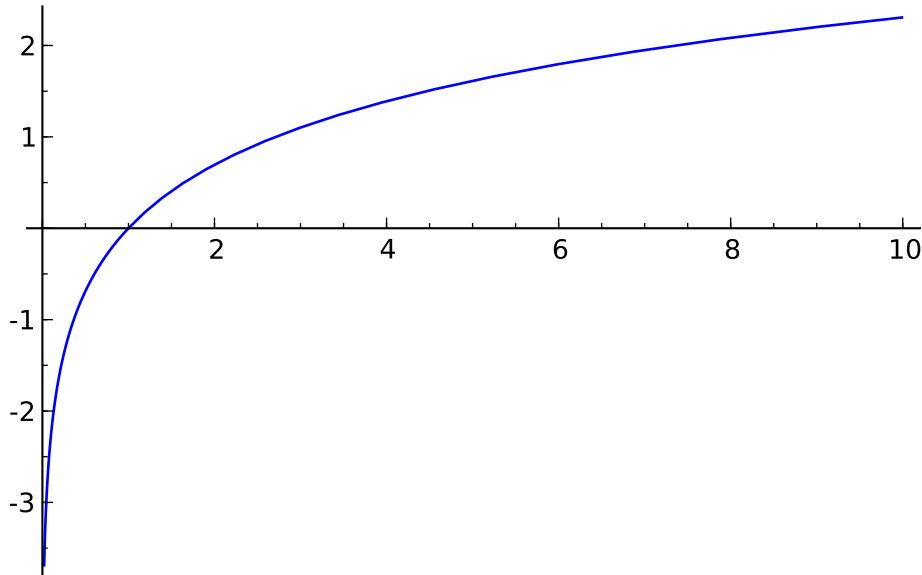
```
return eulerln(x+h,y+h/x,xlimite,h,liste_points+[[x,y]])
```

Programme 263 – construction ln (Sage)

Par exemple :

```
sage: (eulerln(1,0,10,0.01,[])+eulerln(1,0,0.01,-0.01,[])).show()
```

donne :



## L Intégration numérique

### L1 Méthode des rectangles

On veut calculer une intégrale d'une fonction par la méthode des rectangles.

#### L1a Version récursive avec CAML

La définition récursive est assez naturelle :

```
# let rec integ(f,a,b,dx)=
  if a>b then 0.
  else f(a)*.dx+.integ(f,a+.dx,b,dx);;
```

Programme 264 – méthode des rectangles en récursif (CAML)

Par exemple :

```
# let g(x)=x;;
# integ(g,0.,1.,0.0001);;
```

donne :

```
- : float = 0.500049999999960249
```

C'est-à-dire  $\int_0^1 x dx \approx 0.5$

Mais la récursion n'est pas terminale. Introduisons une fonction intermédiaire :

```
# let rec integ_temp(f,a,b,dx,res)=
  if a>b then res
  else integ_temp(f,a+.dx,b,dx,res+.f(a)*.dx);;
```

Programme 265 – méthode des rectangle en récursif terminal étape 1

Puis :

```
# let integrale(f,a,b,dx)=integ_temp(f,a,b,dx,0.);;
```

Programme 266 – méthode des rectangles en récursif terminal étape 2

Alors on obtient :

```
# integrale(g,0.,1.,0.0000001);;
- : float = 0.500000049944005598
```

Pour améliorer l'approximation, il faudrait en fait améliorer la méthode mathématique...

### L1b Version récursive avec Sage

```
def integ(f,a,b,dx):
  if a>b : return 0
  else : return f(a)*dx+integ(f,a+dx,b,dx)
```

Programme 267 – méthode des rectangles en récursif (Sage)

Alors :

```
sage: def g(x):
....: return x
....:
sage: integ(g,0,1,0.01)
0.4950000000000000
```

### Remarque 4 : nombre limite de récursions avec Python

On arrive vite aux limites du nombre standard de récursions de Python. Pour l'augmenter :

```
import sys
sys.setrecursionlimit(100000)
```

```
import sys
sys.setrecursionlimit(100000)

sage: integ(g,0,1,0.0001)
0.5000499999999960
```

### L1c Version impérative avec XCAS

```
integ_rectangle(f,a,b,dx):={
  local aa,I;
  aa:=a;
  I:=0;
  while(aa<b){
```

```
I:=I+f(aa)*dx;
aa:=aa+dx;
}
return(I)
};;
```

Programme 268 – méthode des rectangles en impératif (XCAS)

### L1d Version impérative avec Sage

```
def integ_rectangle(f,a,b,dx):
    aa=a
    I=0
    while aa<b:
        I+=f(aa)*dx
        aa+=dx
    return I
```

Programme 269 – méthode des rectangles en impératif (Sage)

Alors par exemple :

```
sage: def g(x):
....: return x
....:
sage: integ_rectangle(g,0,1,0.000001)
0.499999499999612
```

## L2 Méthode des trapèzes

Il faut juste adapter un peu...

### L2a Version récursive avec CAML

```
# let rec integ_trap(f,a,b,dx)=
  if a>b then 0.
  else 0.5*.dx*.(f(a)+f(a+.dx))+.integ_trap(f,a+.dx,b,dx);;
```

Programme 270 – méthode des trapèzes en récursif (CAML)

```
# integ_trap(g,0.,1.,0.0001);;
- : float = 0.500100004999960213
```

### L2b Version récursive avec Sage

```
def integ_trap(f,a,b,dx):
    if a>b : return 0
    else : return 0.5*dx*(f(a)+f(a+dx))+integ_trap(f,a+dx,b,dx)
```

Programme 271 – méthode des trapèzes en récursif (Sage)

Par exemple :

```
sage: def g(x):
.....: return x
.....:
sage: integ_trap(g,0,1,0.0001)
0.500100004999960
```

### L2c Version impérative avec XCAS

```
integ_trap(f,a,b,dx):={
local aa,I;
aa:=a;
I:=0;
while(aa<b){
I:=I+(f(aa)+f(aa+dx))*dx*0.5;
aa:=aa+dx;
}
return(I)
};;
```

Programme 272 – méthode des trapèzes en impératif (XCAS)

### L2d Version impérative avec Sage

```
def integ_trap(f,a,b,dx):
aa=a
I=0
while aa<b:
I+=(f(aa)+f(aa+dx))*dx*0.5
aa+=dx
return I
```

Programme 273 – méthode des trapèzes en impératif (Sage)

Ce qui donne :

```
sage: def g(x):
.....: return x
.....:
sage: integ_trap(g,0,1,0.00001)
0.500010000049675
```

## L3 Méthode de Simpson

On cherche une approximation de la courbe par un arc de parabole. Pour cela, on va déterminer les  $c_i$  tels que

$$\int_a^b f(x)dx = c_0 f(a) + c_1 f\left(\frac{a+b}{2}\right) + c_2 f(b)$$

soit exacte pour  $f(x)$  successivement égale à 1,  $x$  et  $x^2$ .

Posons  $h = b - a$  et ramenons-nous au cas  $a = 0$ . On obtient le système suivant :

$$\begin{cases} c_0 + c_1 + c_2 = h \\ c_1 + 2c_2 = h \\ c_1 + 4c_3 = \frac{4}{3}h \end{cases}$$

alors  $c_0 = c_2 = \frac{h}{6}$  et  $c_1 = \frac{4}{6}h$

$$\int_a^b f(x)dx = \frac{b-a}{6} \left( f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right)$$

### L3a Version récursive avec CAML

```
# let rec integ_sims(f,a,b,dx)=
  if a>b then 0.
  else (f(a)+.4* .f(a+.dx/.2)+.f(a+.dx))* .dx/.6+.integ_sims(f,a+.dx,b,dx);;
```

Programme 274 – méthode de Simpson en récursif (CAML)

### L3b Version récursive avec Sage

```
def integ_sims(f,a,b,dx):
  if a>b : return 0
  else : return (f(a)+4*f(a+dx/2)+f(a+dx))*dx/6+integ_sims(f,a+dx,b,dx)
```

Programme 275 – méthode de Simpson en récursif (Sage)

Par exemple :

```
sage: def g(x):
....: return 4*sqrt(1-x*x)
....:
sage: integ_sims(g,0,1,0.0001)
3.14159249127822 + 3.60953260836123e-6*I
```

### L3c Version impérative avec XCAS

```
integ_sims(f,a,b,dx):={
local aa,I;
aa:=a;
I:=0;
while(aa<b){
I:=I+(f(aa)+4*f(aa+0.5*dx)+f(aa+dx))*dx/6.;
aa:=aa+dx;
}
return(I)
}::;
```

Programme 276 – méthode de Simson en impératif (XCAS)

### L3d Version impérative avec Sage

```
def integ_sims(f,a,b,dx):
  aa=a
  I=0
  while aa<b:
    I+=(f(aa)+4*f(aa+0.5*dx)+f(aa+dx))*dx/6
    aa+=dx
  return I
```

Programme 277 – méthode de Simpson en impératif (Sage)

Par exemple :

```
sage: def g(x):
....: return 4*sqrt(1-x*x)
....:
sage: integ_simps(g,0,1,0.00001)
3.14159264848556 + 1.14141804900498e-7*I
```



# 12 - Résolutions de systèmes

## A Systèmes de Cramer

$$\text{Soit un système } \begin{cases} a_{11}x + a_{12}y = b_1 \\ a_{21}x + a_{22}y = b_2 \end{cases}.$$

Soit  $D$  le déterminant du système :  $D = a_{11}a_{22} - a_{21}a_{12}$ . S'il est non nul, il est assez aisé de montrer en Seconde que les solutions sont :

$$x = \frac{b_1a_{22} - b_2a_{12}}{D} \quad y = \frac{a_{11}b_2 - a_{21}b_1}{D}$$

Cela peut mettre en évidence les rôles des inconnues (qui n'apparaissent pas en entrée dans l'algorithme) et des coefficients.

**Entrées :** Les coefficients

**début**

**si**  $D=0$  **alors**

        | pas de solution unique

**sinon**

        |  $x = \frac{b_1a_{22} - b_2a_{12}}{D}$       $y = \frac{a_{11}b_2 - a_{21}b_1}{D}$

**fin**

Algorithme 44 : système de CRAMER

## A1 Avec XCAS

```
cramer(a11,a12,b1,a21,a22,b2):={
  si a11*a22-a21*a12==0
  alors retourne("Pas de solution unique")
  sinon retourne((b1*a22-b2*a12)/(a11*a22-a21*a12), (a11*b2-a21*b1)/(a11*a22-a21*a12))
  fsi
};;
```

Programme 278 – système de Cramer en impératif (XCAS)

## A2 Avec Sage

```
def cramer(a11,a12,b1,a21,a22,b2):
  if a11*a22-a21*a12==0 :
    return 'Pas de solution unique'
  else:
    return [(b1*a22-b2*a12)/(a11*a22-a21*a12), (a11*b2-a21*b1)/(a11*a22-a21*a12)]
```

Programme 279 – système de Cramer en impératif (Sage)

Par exemple :

```
sage: cramer(1,2,3,4,5,6)
[-1, 2]
sage: cramer(1,2,3,2,4,6)
'Pas de solution unique'
```

## B Pivot de Gauss

### B1 Cas d'un système 3x3

Nous traiterons le cas d'un système  $3 \times 3$  mais rien n'empêche de généraliser.

Comme nous sommes entre nous, nous « parlerons matrice ».

Nous disposons donc d'une matrice carrée  $A$  de dimension 3 et d'une matrice colonne  $B$  de longueur 3. Nous voulons résoudre dans  $\mathcal{M}_{13}(\mathbb{R})$  le système  $A \cdot X = B$ .

Nous noterons  $T$  le tableau :

$$\left[ \begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array} \right]$$

Si la matrice est inversible, alors, en effectuant des opérations élémentaires sur les lignes, on peut se ramener à la matrice identité dans la partie gauche du tableau et on obtient  $X$  dans la partie droite.

Pour y arriver, l'idée est de balayer le tableau par colonne.

Étant donné une colonne, on cherche un élément non nul. S'il n'y en a pas, la matrice n'est pas inversible et le système n'admet pas une unique solution (on ne s'occupe pas trop du rang au lycée...); sinon, on permute éventuellement deux lignes pour placer l'élément non nul de la colonne  $k$  sur la ligne  $k$  et on divise tous les éléments de la ligne par le nouveau  $a_{kk}$  pour obtenir 1.

Il reste ensuite à remplacer chaque ligne (autre que  $L_k$ ) dont l'élément de la colonne  $k$  est non nul par  $L_i - a_{ik} \times L_k$ .

Cela donne :

```
Entrées : Le tableau T
Initialisation : S ← T
début
  pour col de 1 jusqu'à 3 faire
    line ← col
    tant que S[line][col]=0 faire
      line ← line + 1
      si line=4 alors
        retourner Le système n'admet pas de solution unique
    // On échange les lignes "line" et "col"
    temp ← S[line]
    S[line] ← S[col]
    S[colonne] ← 1/temp[colonne]*temp
    pour k dans {1,2,3} \ {col} faire
      S[k] ← S[k]-S[k][col]*S[col]
  fin
retourner la dernière colonne
```

Algorithme 45 : pivot de GAUSS pour un système  $3 \times 3$

### B2 Avec XCAS

```
Gauss(T) := {
  local S, NoColonne, NoLigne, temp, E, k, j;
  S := T;
```

```

pour NoColonne de 0 jusque 2 faire
  NoLigne:=NoColonne;
  tantque S[NoLigne][NoColonne]==0 faire
    NoLigne:=NoLigne+1;
    si NoLigne==3 alors retourne("pas de solution unique");
  fsi;
ftantque;

temp:=S[NoLigne];
S[NoLigne]:=S[NoColonne];
S[NoColonne]:=1/temp[NoColonne]*temp;

E:={0,1,2} minus {NoColonne};

pour k in E faire
  S[k]:=S[k]-S[k][NoColonne]*S[NoColonne];
fpour;

retourne([seq(S[j][3],j=0..2)])
};

```

Programme 280 – pivot de Gauss (XCAS)

```
Gauss([[1,2,3,4],[5,6,7,8],[9,10,9,8]])
```

```
[0,-1,2]
```

### B3 Avec Sage

```

def gauss(T):
  S=T
  for NoColonne in [0..2]:
    NoLigne=NoColonne
    while S[NoLigne][NoColonne]==0 :
      NoLigne+=1
      if NoLigne==3:
        return 'Pas de solution unique'
    temp=S[NoLigne]
    S[NoLigne]=S[NoColonne]
    S[NoColonne]=map(lambda x:x/temp[NoColonne],temp)
    E=Set([0,1,2])-Set([NoColonne])
    for k in E:
      S[k]=map(lambda x,y:x-S[k][NoColonne]*y,S[k],S[NoColonne])
  return [S[j][3] for j in [0..2]]

```

Programme 281 – pivot de Gauss (Sage)

Par exemple résolvons le système :

$$\begin{cases} x + 2y + 3z = 4 \\ 5x + 6y + 7z = 8 \\ 9x + 10y + 9z = 8 \end{cases}$$

```
sage: gauss([[1,2,3,4],[5,6,7,8],[9,10,9,8]])
[0, -1, 2]
```

et celui-ci :

$$\begin{cases} x + 2y + 3z = 4 \\ 5x + 6y + 7z = 8 \\ 9x + 10y + 11z = 12 \end{cases}$$

```
sage: gauss([[1,2,3,4],[5,6,7,8],[9,10,11,12]])  
'Pas de solution unique'
```

# 13 - Approximations d'irrationnels

## A Algorithme de Héron

HÉRON d'Alexandrie n'avait pas attendu NEWTON et le calcul différentiel pour trouver une méthode permettant de déterminer une approximation de la racine carrée d'un nombre positif puisqu'il a vécu seize siècles avant Sir Isaac.

Si  $x_n$  est une approximation strictement positive par défaut de  $\sqrt{a}$ , alors  $a/x_n$  est une approximation par excès de  $\sqrt{a}$  (pourquoi?) et vice-versa.

La moyenne arithmétique de ces deux approximations est  $\frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$  et constitue une meilleure approximation que les deux précédentes.

On peut montrer c'est une approximation par excès (en développant  $(x_n - \sqrt{a})^2$  par exemple).

On obtient naturellement un algorithme :

```
Entrées : le réel positif  $a$ , une première approximation par défaut strictement positive  $x_0$  et une précision  $\epsilon$   
Initialisation :  $X \leftarrow 0.5(x_0 + a/x_0)$   
début  
  | tant que  $|x_0 - a/x_0| < \epsilon$  faire  
  |   |  $X \leftarrow 0.5(X + a/X)$   
  | retourner  $X$   
fin
```

Algorithme 46 : approximation de  $\sqrt{a}$

On notera que cette méthode est un cas particulier de la méthode de NEWTON-RAPHSON appliquée à une équation particulière :  $x^2 - a = 0$ .

## A1 Avec XCAS en impératif

```
heron(a, xo, eps) := {  
  local X;  
  X := 0.5 * (xo + a/xo);  
  while (abs(evalf(X - a/X)) >= eps) {  
    X := 0.5 * (X + a/X);  
  }  
  return(X)  
};;
```

Programme 282 – algorithme de Héron en impératif (XCAS)

## A2 Avec CAML en version récursive

```
# let rec heron(a, xo, eps) =  
  if abs_float(xo - .a/.xo) < eps then xo  
  else heron(a, 0.5 * (xo + .a/.xo), eps);;
```

Programme 283 – algorithme de Héron en récursif

## B Influence de la première approximation

Notons  $\varepsilon_n$  l'erreur relative après  $n$  itérations par la méthode précédente.

Alors  $\varepsilon_n = \frac{x_n - \sqrt{a}}{\sqrt{a}}$  et donc  $x_n = \frac{1 + \varepsilon_n}{\sqrt{a}}$ . Or  $x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$ .

On peut montrer qu'alors

$$\varepsilon_{n+1} = \frac{\varepsilon_n^2}{2(1 + \varepsilon_n)}$$

Si on est trop loin de la solution au départ,  $\varepsilon_n$  n'est pas négligeable devant 1 et alors  $\varepsilon_{n+1}$  est de l'ordre de  $\varepsilon_n/2$ .

Sinon,  $\varepsilon_{n+1}$  est de l'ordre de  $(\varepsilon_n)^2/2$ .

Ainsi, si on est loin de la solution, la convergence est lente au départ (puisqu'elle est linéaire) puis elle devient d'ordre 2 (on dit qu'elle est quadratique).

Le problème, c'est de pouvoir rapidement calculer un inverse...

## C Calcul de 300 décimales de e

Posons  $e_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \dots + \frac{1}{n!}$

Alors on a aussi

$$e_n = 1 + 1 + \frac{1}{2} \left( 1 + \frac{1}{3} \left( 1 + \frac{1}{4} \left( \dots \left( \frac{1}{n-1} \left( 1 + \frac{1}{n} \right) \right) \right) \right) \right)$$

Un exercice classique montre que  $e - e_n < \frac{n+2}{n+1} \frac{1}{(n+1)!}$ .

Ainsi, pour  $n = 167$ , on obtiendra 300 bonnes décimales au moins.

### C1 Version récursive

```
expo_rec(n) := {
  if(n==167) then {1}
  else {1+expo_rec(n+1)/n}
};
```

Programme 284 – décimales de e en récursif

On obtient la réponse avec :

```
evalf(expo_rec(1), 300)
```

### C2 Version impérative

#### C2 a Avec XCAS

```
expo() := {
  local s, k;
  s := 1;
  for(k := 167; k > 0; k := k - 1) {
    s := s/k + 1;
  }
  return(s);
};
```

Programme 285 – décimale de e en impératif (XCAS)

On obtient la réponse avec :

```
evalf(expo(), 300)
```

**C2b Avec Sage**

```
def expo():
    s=1
    for k in xrange(167,0,-1):
        s=s/k+1
    return s.n(digits=300)
```

Programme 286 – 300 décimales de e en impératif (Sage)

On obtient :

```
sage: expo()
2.
7182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664274274
```

On peut créer une fonction qui fixe le seuil :

```
def seuil_expo(n):
    def fonction_seuil(j):
        return (j+2)/((j+1)*factorial(j+1))
    k=1
    while fonction_seuil(k)>=10**(-n):
        k+=1
    return k
```

Programme 287 – seuil pour obtenir n décimales de e

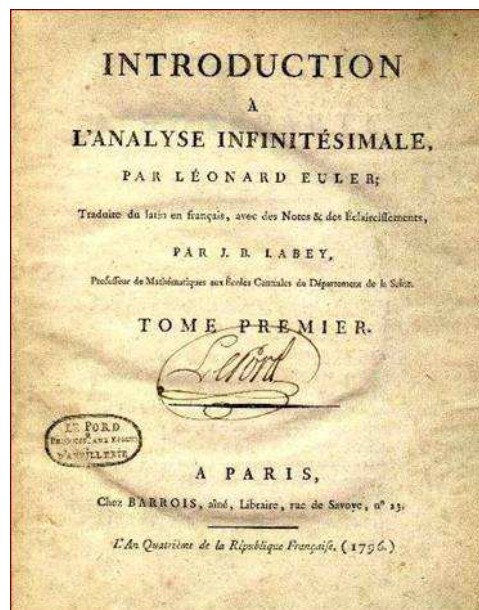
et l'utiliser pour avoir une précision quelconque :

```
def expo(n):
    s=1
    for k in xrange(seuil_expo(n),0,-1):
        s=s/k+1
    return s.n(digits=n)
```

Programme 288 – n décimales de e en impératif (Sage)

**D Méthode d'Euler pour approcher la racine carrée d'un entier****D1 Le texte d'Euler**

Voici, retranscrit en français, le texte d'Euler datant de 1748, extrait de « Introduction à l'Analyse des infiniments petits », chapitre XVI.



Le premier moyen dont nous parlerons, suppose qu'on ait déjà déterminé assez exactement la valeur d'une racine(\*) ; qu'on sache, par exemple, qu'une telle valeur surpasse 4, et qu'elle est plus petite que 5.

Dans ce cas, si l'on suppose cette valeur =  $4 + p$ , on est sûr que  $p$  exprime une fraction. Or si  $p$  est une fraction, et par conséquent moindre que l'unité, le carré de  $p$ , son cube, et en général toutes les puissances plus hautes de  $p$ , seront encore beaucoup plus petites à l'égard de l'unité, et cela fait que, puisqu'il ne s'agit que d'une approximation, on peut les omettre dans le calcul. Quand on aura déterminé à peu près la fraction  $p$ , on connaîtra déjà plus exactement la racine  $4 + p$  ; on partira de là pour déterminer une valeur encore plus exacte, et on continuera de la même manière, jusqu'à ce qu'on ait approché de la vérité autant qu'on le souhaitait.

Nous éclaircirons cette méthode d'abord par un exemple facile, en cherchant par approximation la racine de l'équation  $x^2 = 20$ .

On voit ici que  $x$  est plus grand que 4 et plus petit que 5 ; en conséquence de cela, on fera  $x = 4 + p$ , et on aura :  $x^2 = 16 + 8p + p^2 = 20$  ; mais comme  $p^2$  est très petit, on négligera ce terme pour avoir finalement l'équation :  $16 + 8p = 20$  ce qui donne  $p = 1/2$  et  $x = 4 + 1/2$  ce qui approche déjà beaucoup plus de la vérité.

Si donc on suppose à présent  $x = 9/2 + p$ , on est sûr que  $p$  signifie une fraction encore beaucoup plus petite qu'auparavant, et qu'on pourra négliger  $p^2$  à bien plus forte raison.

On aura donc  $x^2 = 81/4 + 9p = 20$  d'où  $p = -1/36$  donc  $x = 4 + 1/2 - 1/36$ .

Que si l'on voulait approcher encore davantage de la vraie valeur, on ferait  $x = 4$  et  $17/36 + p$ , et on aurait  $x^2 = 10$  et  $-1/1296 + 8$  et  $34/36p = 20$  ainsi  $p = -1/11592$ , donc  $x = 4$  et  $17/36$  et  $-1/11592$  d'où  $x = 4$  et  $5473/11592$ , valeur qui approche si fort de la vérité, qu'on peut avec confiance regarder l'erreur comme nulle.

(\*) cette méthode est celle que Newton a donnée au commencement de « la méthode des fluxions ». En l'approfondissant, on la trouve sujette à différentes imperfections ; c'est pourquoi on y sublimera avec avantage la méthode que M. de la Grange a donnée dans les Mémoires de Berlin, pour les années 1767 et 1768.

Ainsi, en partant de la partie entière  $a_0$  d'une des solutions de  $x^2 = a$ , on a  $(a_0 + p)^2 = a$  et donc

$$p \approx \frac{a - a_0^2}{2a_0}$$

On en déduit qu'une meilleure approximation est

$$a_0 + p \approx \frac{a_0^2 + a}{2a_0}$$

## D2 Version récursive avec CAML

Cela donne :

```
# let rec euleracine(ao, a, n)=
  if n=0 then ao
  else euleracine((ao**2.+a)/(2.*.ao), a, n-1);;
```



Programme 289 – approximation de  $\sqrt{a}$  par la méthode d'Euler en récursif

Par exemple, 10 itérations de cette méthode donnent pour  $\sqrt{2}$  :

```
# euleracine(1.,2.,10);;
- : float = 1.41421356237309492
```

### D3 Version impérative avec XCAS

```
euleracine(ao,a,n):={
  local k,xo;
  xo:=ao;
  for(k:=1;k<n;k:=k+1){
    xo:=xo+(a-xo^2)/(2*xo)
  }
  return(xo)
}::;
```

Programme 290 – approximation de  $\sqrt{a}$  par la méthode d'Euler en impératif

### D4 Version impérative avec Sage

```
def euleracine(ao,a,n):
  xo=ao
  for k in [1..n-1]:
    xo+=(a-xo^2)/(2*xo)
  return float(xo)
```

Programme 291 – approximation de  $\sqrt{a}$  par la méthode d'Euler en impératif (Sage)

Alors :

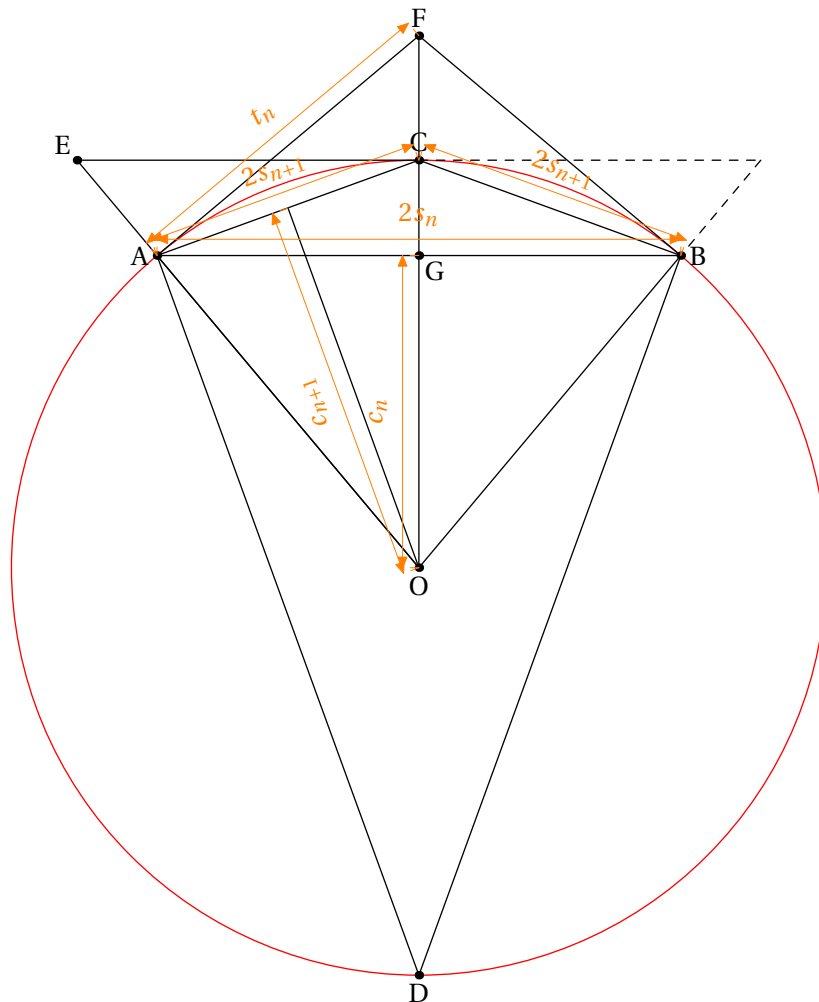
```
sage: euleracine(1,2,10)
1.4142135623730949
sage: sqrt(2.).n(digits=20)
1.4142135623730951455
```

# 14 - Algorithmes géométriques

## A Approximations de $\pi$

### A1 Méthode d'Archimède

#### A1a La méthode



On encadre un cercle entre des polygones tels que le cercle soit respectivement le cercle inscrit et le cercle circonscrit.

On représente le passage de l'étape  $n$  à l'étape  $n + 1$ .

On appelle  $t_n$  la demi-longueur du côté du  $n$ -ème polygone extérieur et  $s_n$  la demi-longueur d'un côté du polygone intérieur.

On appelle  $c_n$  la distance du centre à un côté du polygone intérieur.

 **En classe**

Vous pourrez vous reporter à l'activité C page 232 pour un exemple de travail sur cette méthode en classe de 2<sup>nde</sup>.

Calculons de deux façons l'aire du « cerf-volant » ACBD :

$$AB \times CD = AC \times AD$$

c'est-à-dire

$$2s_n = 2s_{n+1} \times 2c_{n+1}$$

On en déduit la relation

$$s_{n+1} = \frac{s_n}{2c_{n+1}}$$

Les triangles OAF et OCE sont isométriques donc  $t_n = AF = CE$ .

Or :

$$\frac{EC}{CO} = \frac{t_n}{1} = \frac{AG}{GO} = \frac{s_n}{c_n}$$

On en déduit la relation

$$t_n = \frac{s_n}{c_n}$$

Enfin, dans le triangle rectangle DAC,  $DA^2 = DG \times CD$ , c'est-à-dire  $4c_{n+1}^2 = (1 + c_n) \times 2$ . Finalement :

$$c_{n+1} = \sqrt{\frac{1 + c_n}{2}}$$

Le périmètre du polygone intérieur à  $2^{n+2}$  côtés vaut donc

$$S_{n+1} = 2^{n+2} s_{n+1} = 2^{n+1} \frac{s_n}{c_{n+1}} = \frac{S_n}{c_{n+1}}$$

Pour le périmètre extérieur :

$$T_{n+1} = 2^{n+2} t_{n+1} = 2^{n+2} \frac{s_{n+1}}{c_{n+1}} = \frac{S_{n+1}}{c_{n+1}}$$

### A 1 b Avec CAML en récursif

```
# let rec c(n)=if n=0 then 0.5*.sqrt(3.) else sqrt(0.5*.(1.+c(n-1)))
and s(n)=if n=0 then 3. else s(n-1)/.c(n)
and t(n)=s(n)/.c(n);;
```

Programme 292 – approximation de  $\pi$  par la méthode d'Archimède en récursif

Alors :

```
# s(20);;
- : float = 3.14159265358966211
# t(20)-.s(20);;
- : float = 3.91686683087755227e-13
```

**A 1 c Avec XCAS en impératif**

On rajoute un petit compteur d'itérations :

```

archi(d):={
local s,c,t,k,p;
p:=d+3;
epsilon:=1e-300;
s:=3;
c:=sqrt(3)/2;
t:=s/c;
k:=1;
while(evalf((t-s),p)>evalf(10^(-d),p)){
c:=evalf(sqrt((1+c)/2),p);
s:=evalf(s/c,p);
t:=evalf(s/c,p);
k:=k+1;
}
s,k
};

```

Programme 293 – approximation de  $\pi$  par la méthode d'Archimède en impératif

Par exemple :

```
archi(50)
```

renvoie :

3.14159265358979323846264338327950288419716939937510472,84

**A 2 Méthode de Nicolas de Cues****A 2 a La méthode**

On « cale » un polygone de  $2^n$  côtés et de périmètre 2 entre deux cercles : le cercle circonscrit de rayon  $r_n$  et le cercle inscrit tangent à chaque côté de rayon  $h_n$ . On a alors :

$$2\pi h_n \leq 2 \leq 2\pi r_n$$

c'est-à-dire :

$$\frac{1}{r_n} \leq \pi \leq \frac{1}{h_n}$$

Il reste à trouver une relation donnant  $r_n$  et  $h_n$ ...



```

cues(n):={
local h,r;
h:=0.25;
r:=1/sqrt(8);
for(k:=1;k<=n;k:=k+1){
h:=0.5*(r+h);
r:=sqrt(r*h);
}
return(1/h,evalf(1/h-1/r))
};

```

Programme 295 – approximation de  $\pi$  par la méthode de Cues en impératif

### A 2 d Avec Sage en impératif

```

def cues(n):
h=0.25
r=1./sqrt(8.)
for k in xrange(1,n):
h=0.5*(r+h)
r=sqrt(r*h)
return 1/h,(1/h-1/r)

```

Programme 296 – approximation de  $\pi$  par la méthode de Cues en impératif

Par exemple :

```

sage: cues(20)
(3.14159265359214, 3.52473605857995e-12)

```

## A 3 Méthode d'Al-Kashi

### A 3 a La méthode

Voir *Tangente* hors-série n°36 « le cercle » page 16.

### A 3 b Avec CAML en récursif

```

# let a(n)=
let rec b(n)=if n=0 then sqrt(3.) else sqrt(2.+b(n-1))
in 3.*2.**(float_of_int(n)).sqrt(4.-b(n)*b(n));;

```

Programme 297 – approximation de  $\pi$  par la méthode d'Al-Kashi en récursif

donne :

```

# a(20);;
- : float = 3.14167426502175751

```

### A 3 c Avec XCAS en impératif

```

alkashi(n) := {
  local k, b;
  b := sqrt(3.);
  for(k:=1; k<=n; k:=k+1) {
    b := sqrt(2+b)
  }
  return(3*2^n*sqrt(4-b^2))
};

```

Programme 298 – approximation de  $\pi$  par la méthode d'Al-Kashi en impératif

## B Approximation de $\pi$ par dénombrement des points de coordonnées entières d'un disque

## C Flocons fractales

On peut définir récursivement le flocon de VON KOCH.  
Pour simplifier, on se contentera de « floconiser » un segment.

### C1 Avec la tortue

```

koch(long, n) := {
  si (n==0) alors
  avance(long);
  sinon
  koch(long/3, n-1); tourne_gauche(60);
  koch(long/3, n-1); tourne_droite(120);
  koch(long/3, n-1); tourne_gauche(60);
  koch(long/3, n-1);
  fsi;
};

```

Programme 299 – flocon de Von Koch avec la tortue LOGO

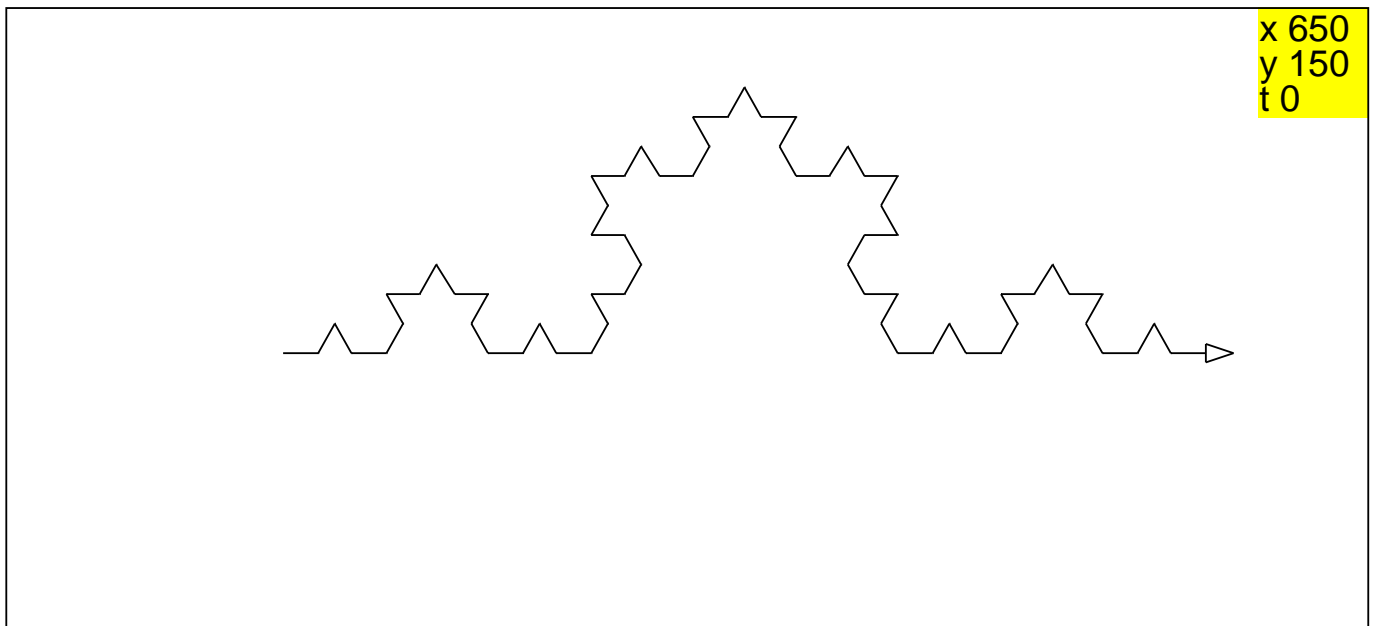
Alors

```

koch(500, 3)

```

renvoie

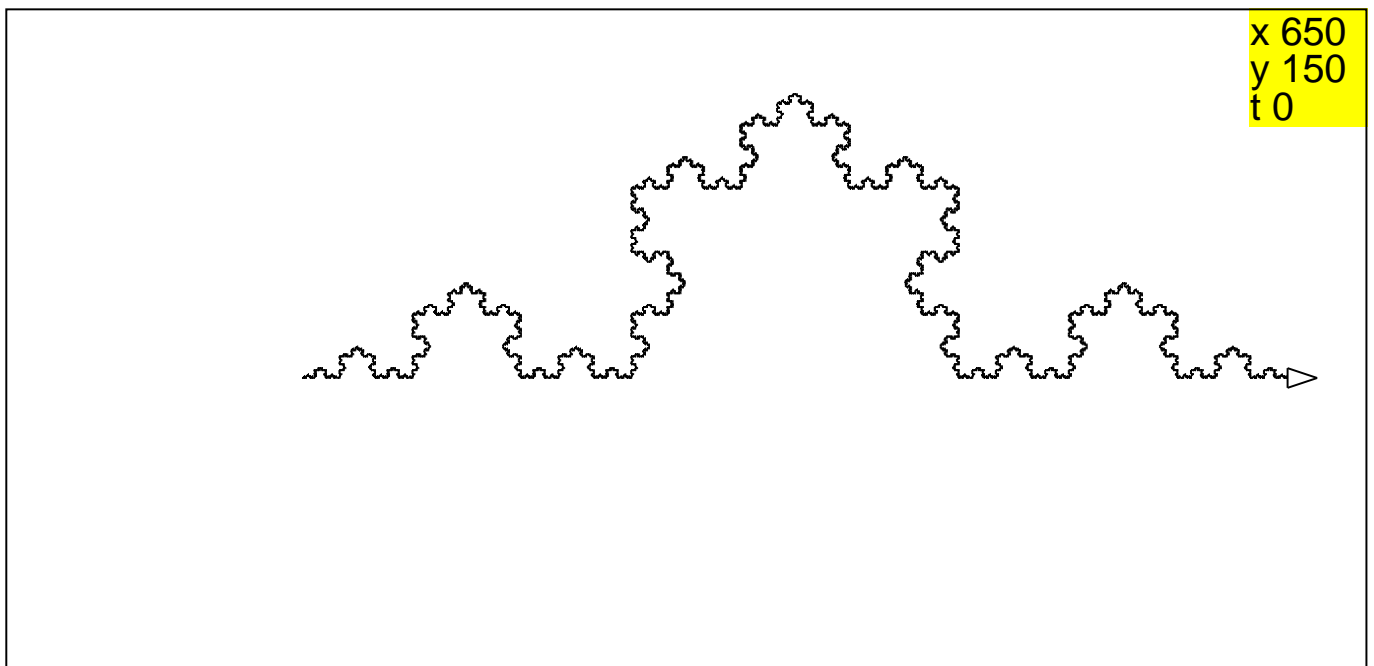


x 650  
y 150  
t 0

et

```
koch(500,6)
```

renvoie



x 650  
y 150  
t 0

## C2 Avec les outils géométriques de XCAS

```
KochBis(n,A,B):={
  if(n=0){segment(A,B)}
  else{
    KochBis(n-1,A,homothetie(A,1/3,B)),
    KochBis(n-1,homothetie(A,1/3,B),rotation(homothetie(A,1/3,B),pi/3,homothetie(A,2/3,B))),
    KochBis(n-1,rotation(homothetie(A,1/3,B),pi/3,homothetie(A,2/3,B)),homothetie(A,2/3,B)),

```

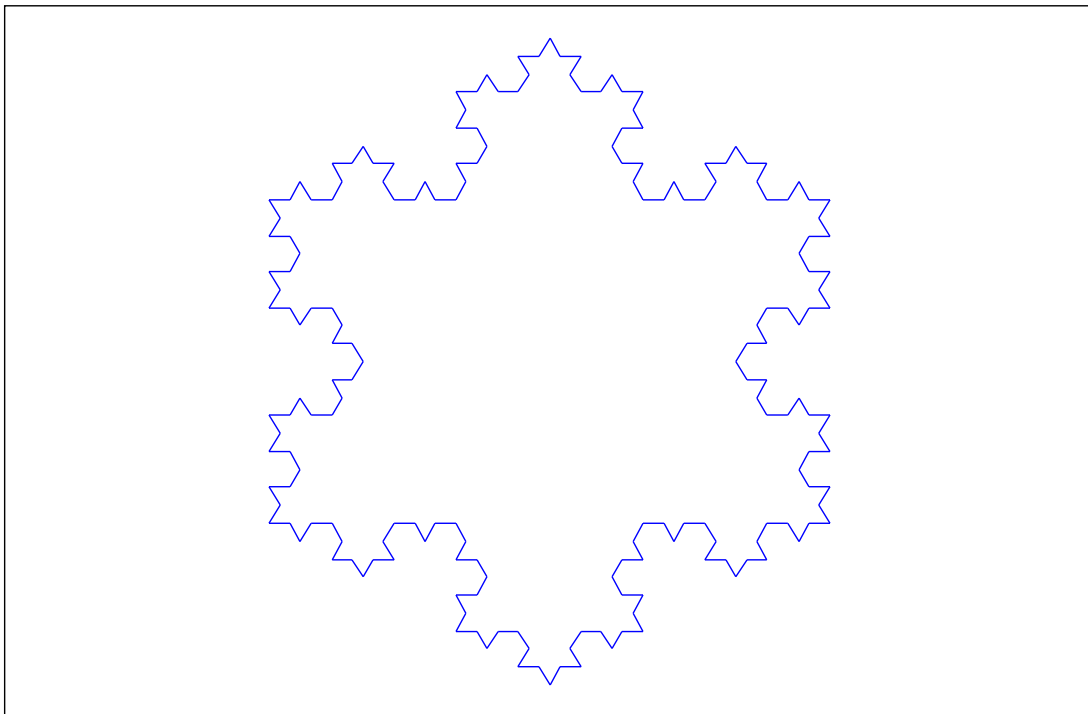


```
KochBis(n-1, homothetie(A, 2/3, B), B)
}
};;
```

Programme 300 – flocon de Von Koch avec XCAS

On peut alors s'amuser à dessiner un flocon à partir d'un triangle équilatéral :

```
KochBis(3, point(0,0), point(10,0)), KochBis(3, point(10,0), point(5, -10*sqrt(3.0)/2)), KochBis(3, point(5, -10*sqrt(3.0)/2), point(0,0))
```



## D Polygones de Sierpinski

Waclaw SIERPINSKI (1882 - 1969) fut un mathématicien polonais qui travailla sur des domaines mathématiques très ardues : fondements des mathématiques, construction axiomatique des ensembles, hypothèse du continu, topologie, théorie des nombres...

Il a également travaillé sur les premiers objets fractals qu'étudiera plus tard Benoît MANDELBROT, mathématicien français d'origine polonaise connu pour ses travaux dans ce domaine.

### D1 Jouons aux dés

Prenez un dé à 6 faces, un triangle ABC et un point G quelconque à l'intérieur du triangle.

Vous lancez le dé :

- si la face supérieure est 1 ou 2, vous faites une petite croix au niveau du milieu de G et de A ;
- si la face supérieure est 3 ou 4, vous faites une petite croix au niveau du milieu de G et de B ;
- si la face supérieure est 5 ou 6, vous faites une petite croix au niveau du milieu de G et de C ;

Ah, zut, on n'a pas de dé dans la salle d'info...mais on a XCAS ou Sage.

#### D1a Avec XCAS

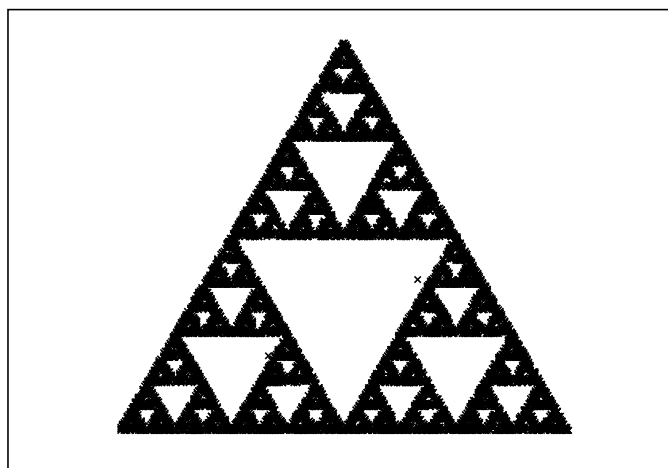
- hasard(0,1) renvoie un nombre aléatoirement choisi entre 0 et 1 ;
- hasard(n) renvoie un entier aléatoirement choisi entre 0 inclus et n exclus.

```

dede(n):={
local P,r,t,d,G,L,k;
P:=[[0,0],[1,0],[0.5,0.5*sqrt(3)]];
t:=hasard(0,1);
G:=t^2*P[0]+2*t*(1-t)*P[1]+(1-t)^2*P[2];
L:=G;
pour k de 1 jusque n faire
    G:=0.5*(G+P[hasard(3)]);
    L:=L,G;
fpour;
nuage_points([L]);
};

```

ce qui donne en lançant dede(10000) :



### D 1 b Avec Sage

- random() renvoie un nombre aléatoirement choisi entre 0 et 1;
- floor(n\*random()) renvoie un entier aléatoirement choisi entre 0 inclus et n exclus.

```

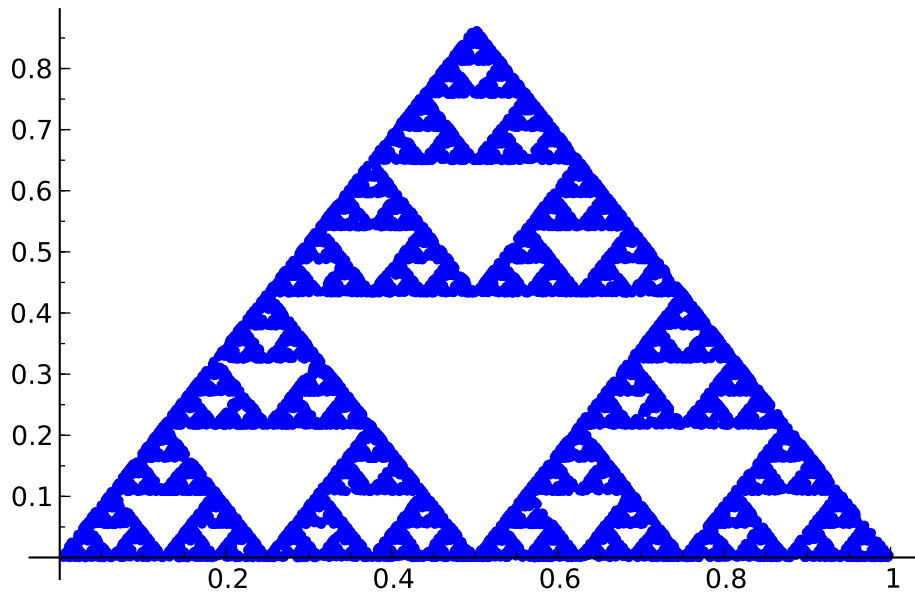
def dede(n):
    P=[[0,0],[1,0],[0.5,0.5*sqrt(3)]]
    t=random()
    G=map(lambda x,y,z:t^2*x+2*t*(1-t)*y+(1-t)^2*z,P[0],P[1],P[2])
    L=[G]
    for k in [1..n]:
        G=map(lambda x,y:0.5*(x+y),G,P[floor(3*random())])
        L+=[G]
    D=points(L)
    D.show()

```

Alors :

```
sage: dede(10000)
```

donne :



## D2 Étoiles et espaces

Que fait cette procédure :

```
pascal(n):={
local T, j, i;
T:=[[0n]n];
pour j de 0 jusque n-1 faire
pour i de 0 jusque n-1 faire
si j>i alors T[i,j]:="" ;
fsi;
fpour;
fpour;
retourne(T);
};;
```

Programme 301 – Tableau en triangle

Transformez-la un peu pour obtenir ceci avec  $n = 10$  par exemple :

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
```

Complétez cette dernière procédure afin de remplacer chaque nombre pair par une espace ' ' et chaque nombre impair par une étoile '\*'. Lancez-la pour  $n = 48$  : incroyable, non ?

Solution :

```
PascalT(n):={
local T, j, i;
T:=[[0n]n];
pour j de 0 jusque n-1 faire
```

```

pour i de 0 jusque n-1 faire
  si j>i alors T[i,j]:="" ;
  sinon si j==0 alors T[i,j]:=1 ;
    sinon si i==j alors T[i,j]:=1;
      sinon si (i>0) et (j>0) alors T[i,j]:=T[i-1,j]+T[i-1,j-1];
    fsi;fsi;fsi;fsi;
  fpour;
fpour;
pour i de 0 jusque n-1 faire
  pour j de 0 jusque i faire
    si irem(T[i,j],2)==0 alors T[i,j]:="";
    sinon T[i,j]:="*";
    fsi;
  fpour;
fpour;
retourne(T);
};

```

Programme 302 – Triangle de Pascal de Sierpinski

### D3 Tapis

Considérons les huit transformations de  $\mathbb{C}$  dans  $\mathbb{C}$  suivantes :

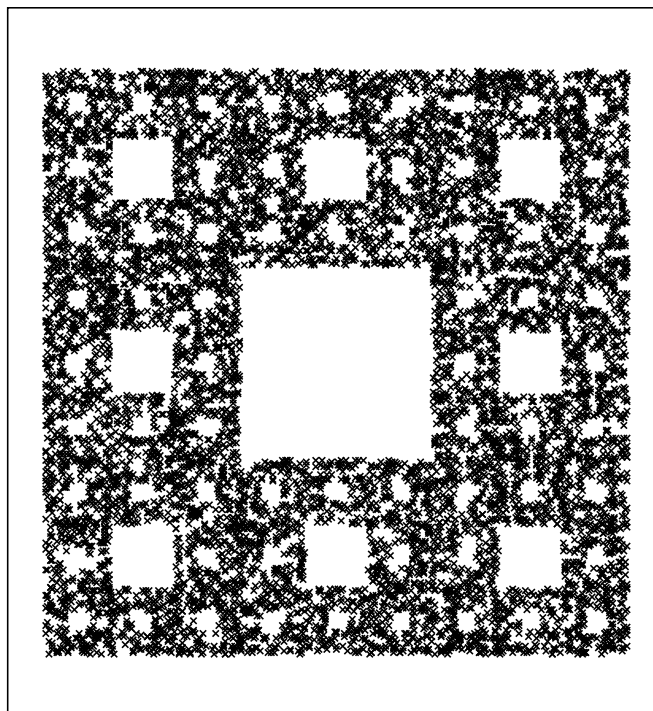
$$T_1(z) = \frac{1}{3}z \quad T_2(z) = \frac{1}{3}z + \frac{1}{3} \quad T_3(z) = \frac{1}{3}z + \frac{2}{3} \quad T_4(z) = \frac{1}{3}z + \frac{2}{3} + \frac{1}{3}i$$

$$T_5(z) = \frac{1}{3}z + \frac{2}{3} + \frac{2}{3}i \quad T_6(z) = \frac{1}{3}z + \frac{1}{3} + \frac{2}{3}i \quad T_7(z) = \frac{1}{3}z + \frac{2}{3}i \quad T_8(z) = \frac{1}{3}z + \frac{1}{3}i$$

Soit  $E_0$  le carré unité.

- Dessinez  $E_0$ ;
- Dessinez  $E_1 = T_1(E_0) \cup T_2(E_0) \cup \dots \cup T_8(E_0)$ ;
- Identifiez les  $T_k$ .

Comment obtenir ce joli dessin avec XCAS ?



Solution :

```

Tapis(n) := {
  local T, P, r, t, d, G, L, k, m, q, R, s, z;
  P := [0, 1, 1+i, i];
  t := hasard(0, 1);
  G := sum(binomial(3, q) * t^q * (1-t)^(3-q) * P[q], q=0..3); /* barycentre "intérieur" quelconque */
  L := [re(G), im(G)];
  T := [z -> z/3, z -> z/3+1/3, z -> z/3+2/3, z -> z/3+1/3+2*i/3, z -> z/3+2/3+2*i/3, z -> z/3+2/3+i/3, z -> z/3+i/3, z -> z/3+2*i/3];
  pour k de 1 jusque n faire
    G := T[hasard(8)](G);
    L := L, [re(G), im(G)];
  fpour;
  nuage_points([L]);
} :;

```

Programme 303 – Tapis de Sierpinski

## D4 Les mites de Sierpinski

Monsieur SIERPINSKI avait ramené d'un voyage en Orient un tapis carré de 1 mètre de côté dont il était très content. Jusqu'au jour où les mites s'introduisirent chez lui.

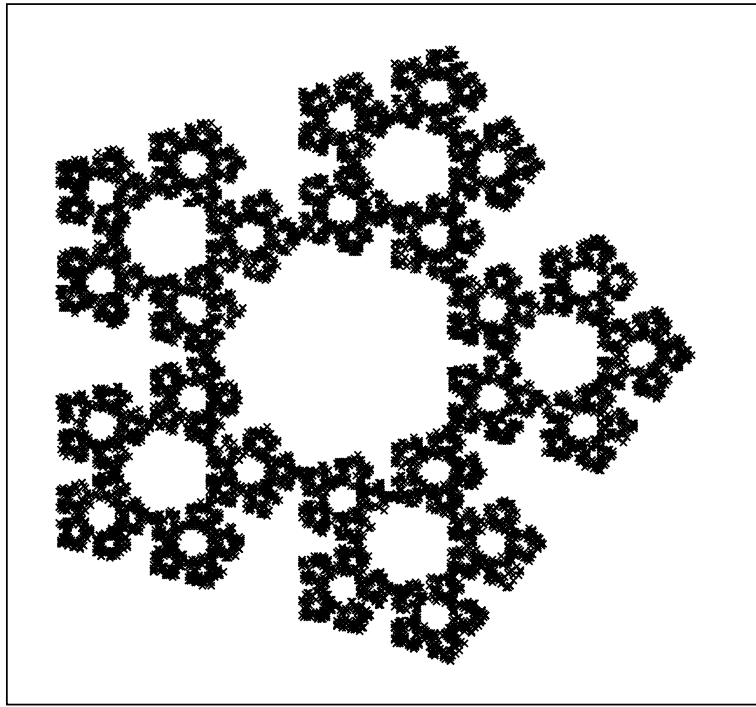
En 24 heures, elles dévorèrent dans le tapis un carré de côté trois fois plus petit, situé exactement au centre du tapis. En constatant les dégâts, Monsieur SIERPINSKI entra dans une colère noire ! Puis il se consola en se disant qu'il lui restait huit petits carrés de tapis, chacun de la taille du carré disparu. Malheureusement, dans les 12 heures qui suivirent, les mites avaient attaqué les huit petits carrés restants : dans chacun, elles avaient mangé un carré central encore trois fois plus petit. Et dans les 6 heures suivantes elles grignotèrent encore le carré central de chacun des tout petits carrés restants. Et l'histoire se répéta, encore et encore ; à chaque étape, qui se déroulait dans un intervalle de temps deux fois plus petit que l'étape précédente, les mites faisaient des trous de taille trois fois plus petite...

- Calculer le nombre total de trous dans le tapis de Monsieur SIERPINSKI après  $n$  étapes. Calculer la surface  $S_n$  de tapis qui n'a pas encore été mangée après  $n$  étapes. Trouver la limite de la suite  $(S_n)_{n \geq 0}$ . Que reste-t-il du tapis à la fin de l'histoire ?
- Calculer la durée totale du festin « mitique »...

Merci à *Frédéric Le Roux*

## D5 Dentelle de Varsovie

Nous voudrions obtenir le joli napperon en dentelle suivant :



Déterminez le rapport des similitudes qui transforment le grand pentagone en un des pentagones plus petit.  
En vous inspirant de ce qui a été fait pour le triangle de SIERPINSKI, faites tracer à XCAS ou Sage ce joli napperon.

### D5a Avec XCAS

En fait on peut montrer que pour un nombre  $c \geq 5$  de côtés, le rapport est de  $\frac{1}{2 \sum_{k=0}^{\lfloor \frac{c}{4} \rfloor} \cos\left(\frac{2k\pi}{c}\right)}$

Solution :

```
Serp(c,n):={
  local P,r,t,d,G,L,k,m,q,R,s;
  P:=[seq([re(exp(2*i*m*pi/c)),im(exp(2*i*m*pi/c))],m=1..c)]; /* sommets du polygone régulier */
  t:=hasard(0,1); /* paramètre pour le barycentre */
  R:=evalf(1/(2*sum(cos(2*s*pi/c),s,0,floor(c/4)))); /* rapport des similitudes */
  G:=sum(binomial(c-1,q)*t^q*(1-t)^(c-1-q)*P[q],q,0,c-1); /* barycentre "intérieur" quelconque */
  L:=G;
  pour k de 1 jusque n faire
    G:=R*(G+P[hasard(c)]);
    L:=L,G;
  fpour;
  nuage_points([L])
};;
```

Programme 304 – Dentelle de Sierpinski

### D5b Avec Sage

```
def serp(c,n):
  P=[[cos(2*m*pi/c),sin(2*m*pi/c)] for m in [1..c]]
  G=[0,0] # point de départ à l'intérieur du polygone
  s=var('s')
  R=1/(2.*sum(cos(2.*s*float(pi)/c),s,0,floor(c/4))) # rapport similitude
  L=[G]
```

```

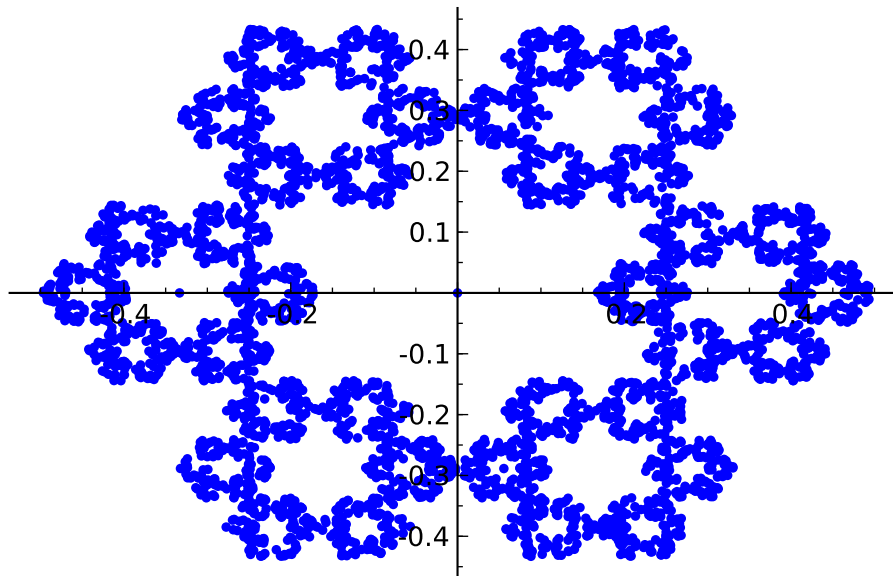
for k in [1..n]:
    G=map(lambda x,y:R*(x+y),G,P[floor(c*random())])
    L+=[G]
D=points(L)
D.show(aspect_ratio=1)

```

Programme 305 – Dentelle de Sierpinski (Sage)

Pour  $c = 6$  :

```
sage: serp(6,5000)
```



## E Végétation récursive

### E1 Avec XCAS

Analysez cet algorithme :

```

arbre(A,B,Rap,Ang,n):={
  si n>0 alors
    segment(point(A),point(B)),seq(arbre(B,B+Rap[q]*exp(Ang[q]*i)*(B-A),Rap,Ang,n-1),q=0..size(
      Rap)-1)
  sinon segment(point(A),point(B))
  fsi
};;

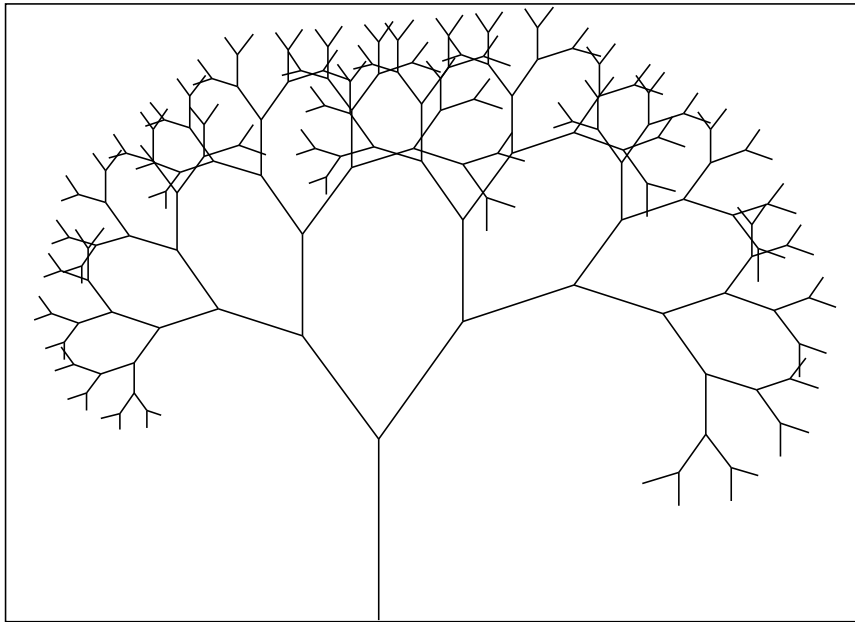
```

Programme 306 – L-System (XCAS)

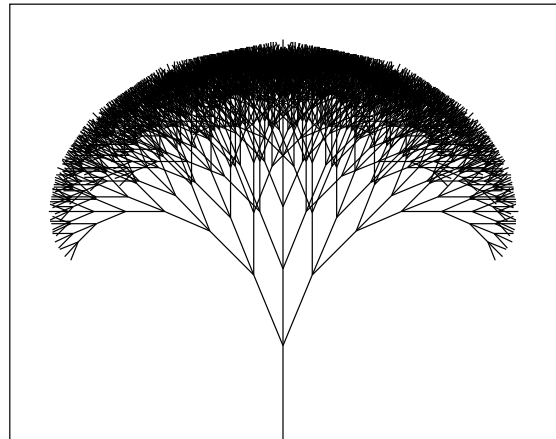
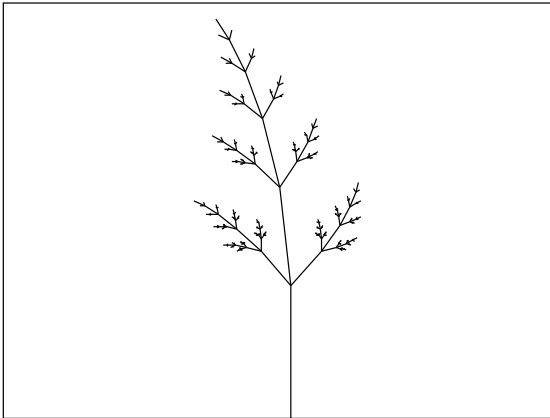
sachant que

```
arbre(0,i,[0.7,0.8],[pi/5.,-pi/5.],7)
```

donne



Comment obtenir cette fougère et ce brocolis ?



## E2 Avec Sage

```
def p_c(A):
    return [A.real(),A.imag()] # Donne les coordonnées de l'image d'un complexe

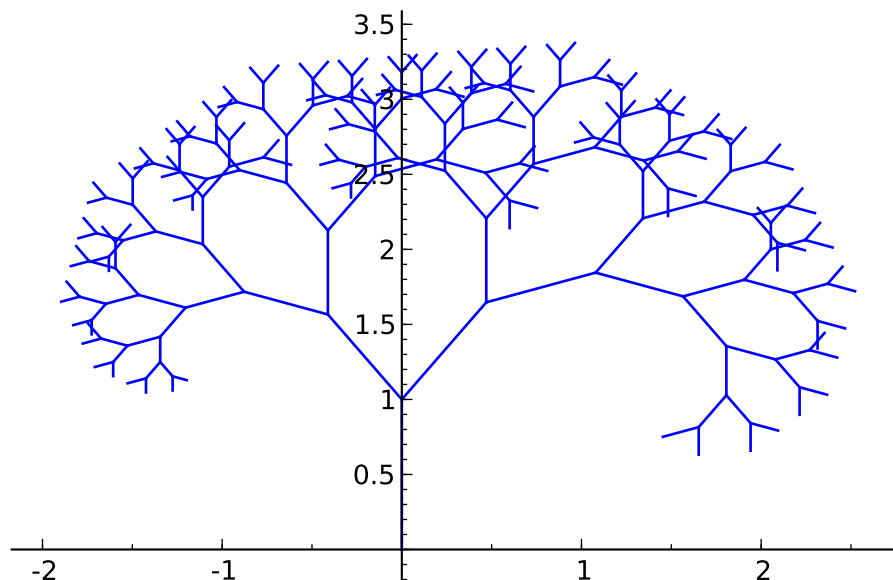
def arbre2(A,B,Rap,Ang,n):
    if n==0 : return line([p_c(A),p_c(B)])
    else :
        return line([p_c(A),p_c(B)])+arbre2(B,B+Rap[0]*exp(Ang[0]*I)*(B-A),Rap,Ang,n-1)+arbre2(B,B
        +Rap[1]*exp(Ang[1]*I)*(B-A),Rap,Ang,n-1)
```

Programme 307 – L-System (Sage)

Par exemple :

```
sage: P=arbre2(0.,1.*I,[0.6,0.8],[float(pi)/5.,-float(pi)/5.],7)
sage: P.show(axes=False)
```





## F Ensembles de Julia et de Mandelbrot

### F1 Ensemble de Julia

Gaston JULIA (1893 - 1978) est un mathématicien français qui a travaillé en particulier sur les fonctions à variable complexe. Ses travaux ont été largement utilisés par Benoît MANDELBROT (né en 1924) au début des années 1970. Nous allons survoler certains de leurs résultats les plus populaires.

Nous allons étudier les suites  $(z_n)$  définies par  $z_0$  et  $z_{n+1} = z_n^2 + c$  pour tout entier naturel  $n$  avec  $c$  un nombre complexe quelconque.

Nous avons déjà exploré une suite réelle semblable lors de notre incursion dans la dynamique des populations.

Le cas  $c = 0$  est assez simple à étudier : faites-le !

#### F1a Avec XCAS

Comment ce que vous avez obtenu est en lien avec cette procédure XCAS et son utilisation ?

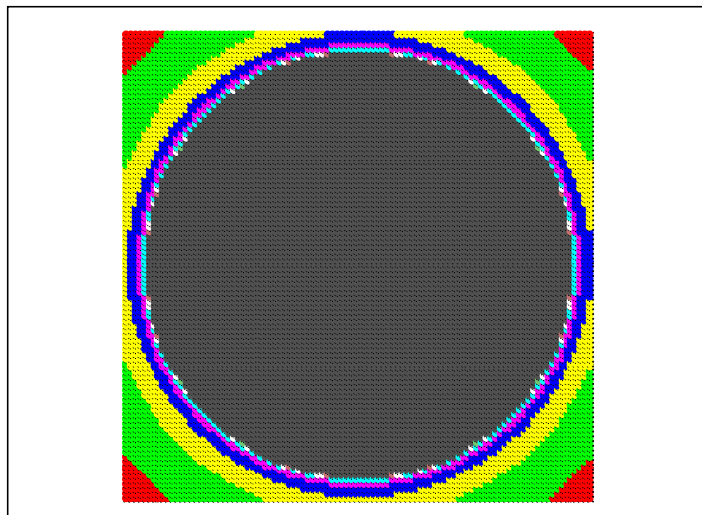
```

julia(n,c,X,Y):= {
local pasx,pasy,L,k,x,y,z,m;
pasx:=2.*X/n;
pasy:=2.*Y/n;
L:= [seq(0,k=1..n^2)]; /* une liste de n*n zéros */
k:=0;
pour x de -X jusque X pas pasx faire
  pour y de -Y jusque Y pas pasy faire
    z:=evalf(x+i*y)
    m:=0;
    tantque (abs(z)<2) et (m<30) faire
      z:=approx(z^2+c);
      m:=m+1;
    ftantque;
    L[k]:= couleur(point(x+i*y),m+point_point+epaisseur_point_2)
    k:=k+1;
  fpour;
fpour;
retourne(L)
};;

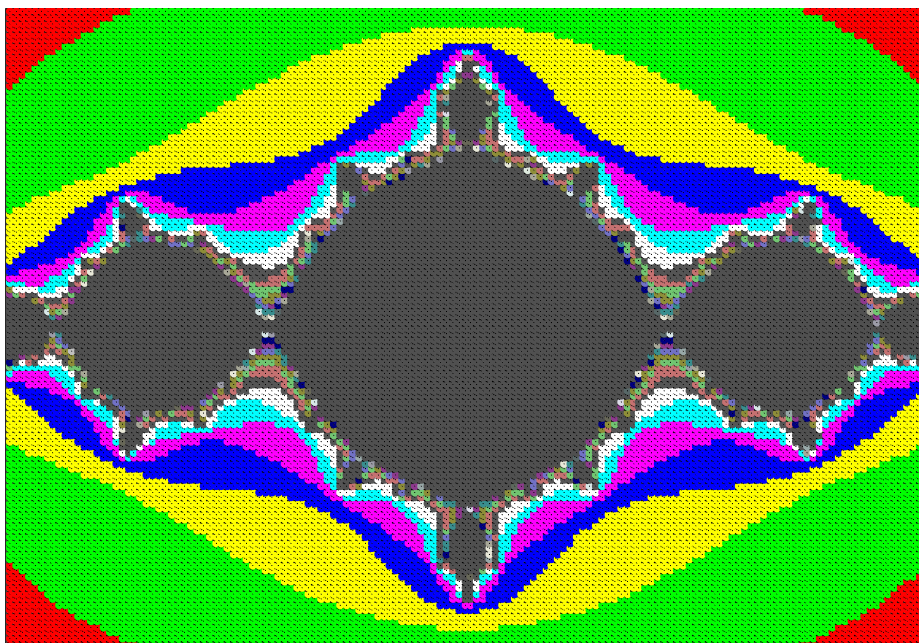
```

Programme 308 – ensembles de Julia

```
julia(100,0,1.1,1.1)
```



Testez différentes valeurs de  $c$  :  $-1, 0,32 + 0,043i$ ,  $-0,122561 + 0,744862i$ .  
Voici par exemple `julia(150,-1,1.7,0.9)` :



Quelles sont vos interprétations ?

### F1 b Avec Sage

```
def julia(c):
    c=complex(c)
    def juju(z):
        z=complex(z)
```

### 💡 for...break...else sur Python

Python contient une boucle for particulière :

```
for iteration
  if condition
    break # on sort de la boucle for
  action1 # si condition n'est pas remplie
else action2 # si on sort de for normalement sans break
```

### 🎵 Remarque 5 : complex

Il est plus rapide de calculer avec `complex(re,im)` qu'avec `re+I*im`

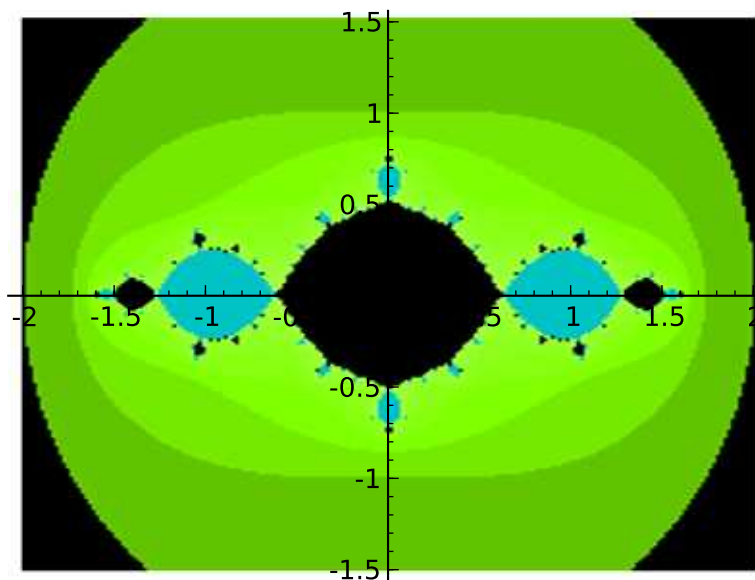
```
for k in xrange(100):
  if abs(z)>2:
    break
  z=z^2+c
else:
  return complex(z)
return complex(0,k)
return complex_plot(juju, (-2,2), (-1.5,1.5),plot_points=200)
```

Programme 309 – Ensemble de Julia (Sage)

Par exemple :

```
julia(-1)
```

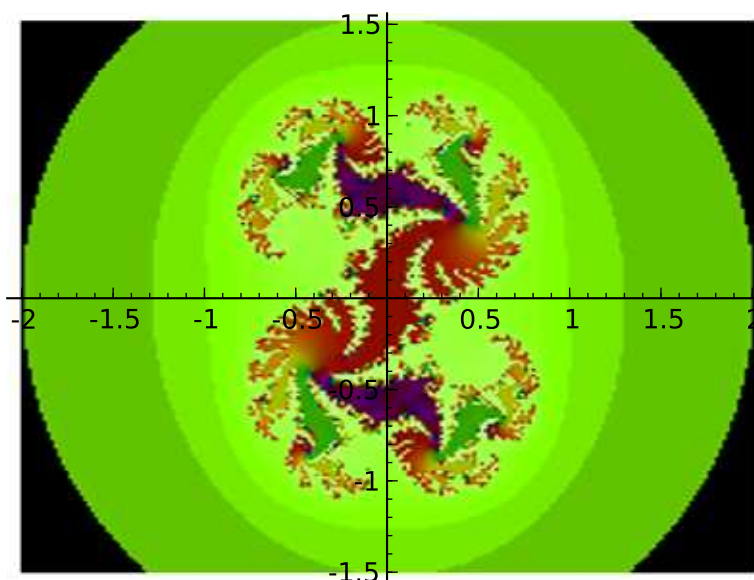
donne le joli dessin suivant :



et :

```
sage: julia(0.32+I*0.043)
```

donne :



## F2 Ensemble de Mandelbrot

Plutôt que de tester les valeurs de  $c$  une par une, Benoît MANDELBROT a cherché à représenter les valeurs de  $c$  qui faisaient converger la suite avec  $z_0 = 0$  : c'est l'ensemble de MANDELBROT.

Tracez-le!

*Solution :*

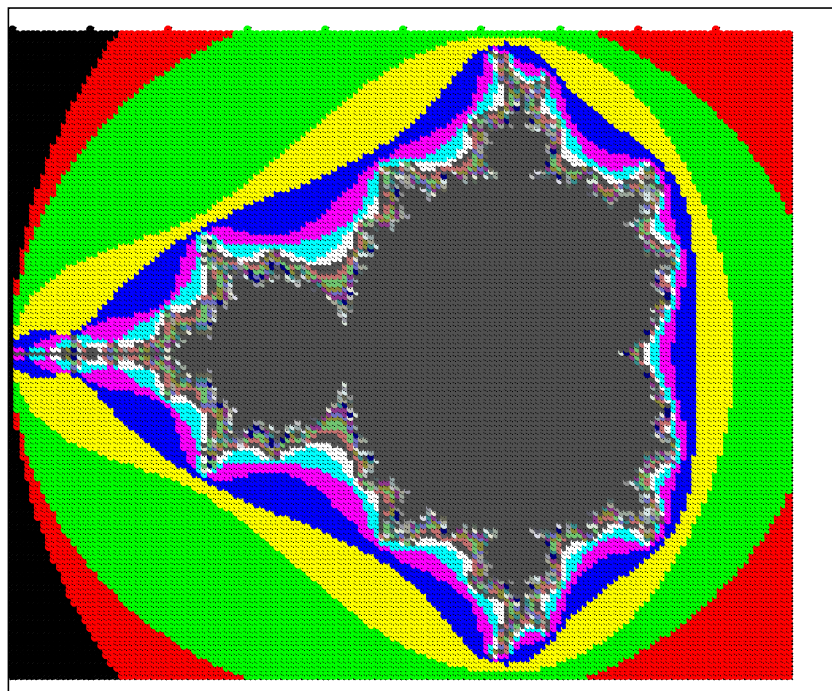
### F2 a Avec XCAS

```

MB(n):={
local pasx,pasy,L,k,x,y,c,z,m;
pasx:=2.9./n;
pasy:=2.4/n;
L:=seq(0,k=1..n^2)]; /* une liste de n*n zéros */
k:=0;
pour x de -2 jusque 0.9 pas pasx faire
  pour y de -1.2 jusque 1.2 pas pasy faire
    c:=evalf(x+i*y)
    z:=c;
    m:=0;
    tantque (abs(z)<2) et (m<30) faire
      z:=evalf(z^2+c);
      m:=m+1;
    ftantque;
    L[k]:= couleur(point(x+i*y),m+point_point+epaisseur_point_2)
    k:=k+1;
  fpour;
fpour;
retourne(L)
};;

```

Programme 310 – Ensemble de Mandelbrot



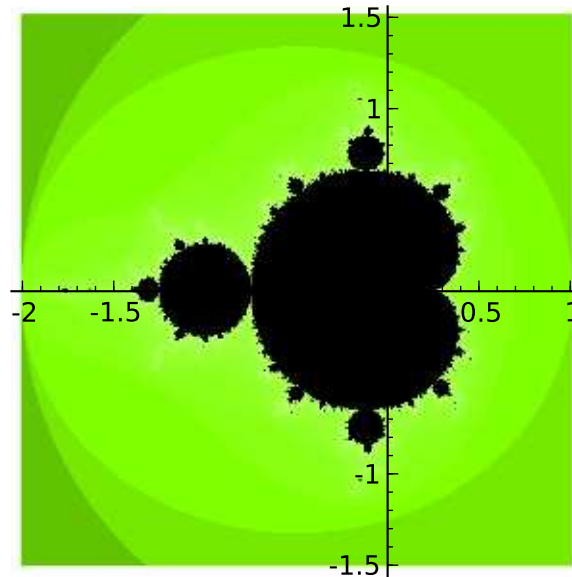
### F2 b Avec Sage

```
def mandelbrot(c):
    z=complex(0,0)
    c=complex(c)
    for k in xrange(100):
        if abs(z)>2:
            break
        z=z^2+c
    else:
        return complex(0,0)
    return complex(0,k)
```

Programme 311 – Ensemble de Mandelbrot (Sage)

On utilise ensuite `complex_plot(fonction, (xmin, xmax), (ymin, ymax), plot_points=nombre)` :

```
complex_plot(mandelbrot, (-2,1), (-1.5,1.5), plot_points=200)
```



## G Les vecteurs, le renard et le lapin

### G1 Première poursuite

Un lapin court vers le nord en ligne droite et un renard le poursuit en bondissant toujours dans sa direction.

On suppose que le lapin et le renard fond des bonds de longueurs respectives  $bl$  et  $br$ , que le lapin part du point de coordonnées  $(0;0)$  et le renard du point de coordonnées  $(x_0, y_0)$ .

On repèrera ensuite les animaux par le point  $R(x; y)$  et le point  $L(0; z)$ .

Calculons la distance Renard-Lapin : comme  $\overrightarrow{RL}(-x; z - y)$  alors la distance  $d$  qui les sépare vaut

$$d = \sqrt{x^2 + (y - z)^2}$$

Il est alors pratique de travailler dans un espace affine.

La position suivante du renard, après un bond est  $R + \frac{br}{d}\overrightarrow{RL}$

Celle du lapin est  $L + bl\vec{j}$ . Tout est en ordre pour pouvoir programmer...

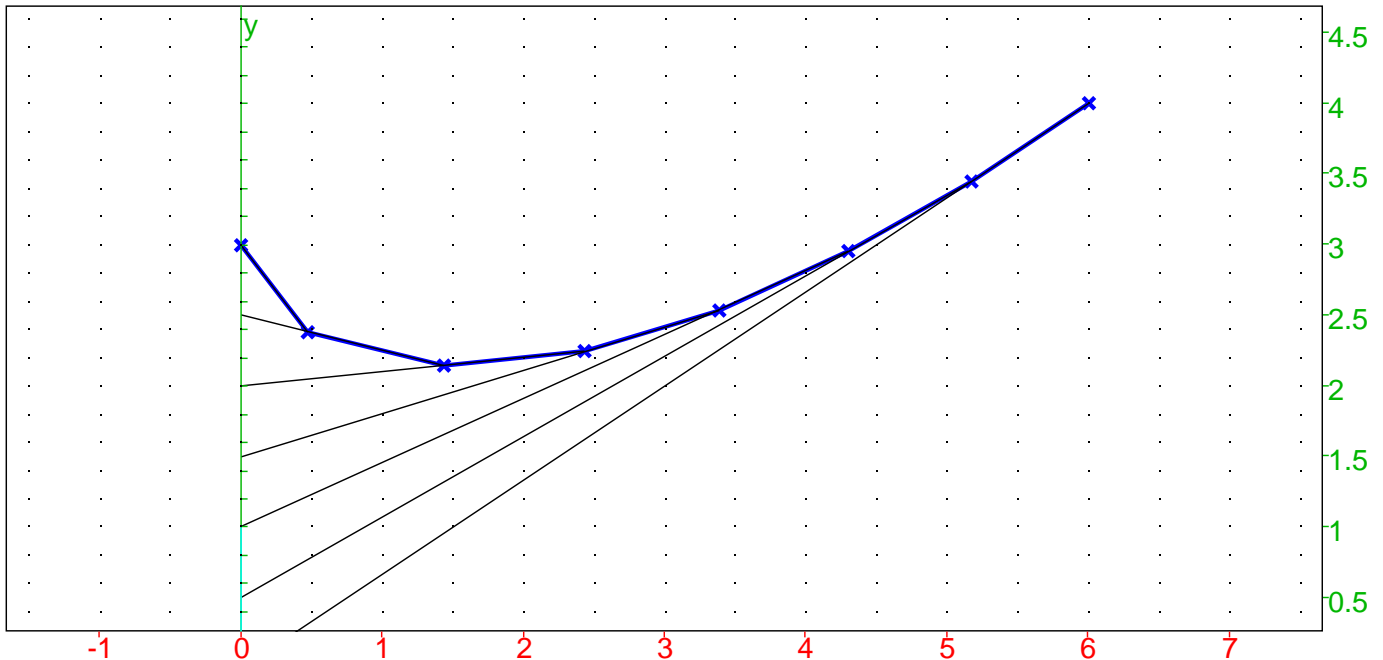
```
poursuite1(xo, yo, br, bl) := {
  local R, L, d, pR, Dir, nbsaut;
  R := [xo, yo];
  L := [0, 0];
  pR := NULL;
  Dir := NULL;
  nbsaut := 0;
  d := evalf(sqrt(xo^2 + yo^2));
  while((nbsaut < 1000) and (d > 0)) {
    pR := R, pR;
    Dir := Dir, segment(point(R), point(L));
    d := evalf(sqrt((L[0] - R[0])^2 + (L[1] - R[1])^2));
    if(d > br) then {
      R := R + (br/d) * (L - R);
      L := L + [0, bl];
    } else {R := L}
    nbsaut := nbsaut + 1;
  }
  print(if(nbsaut < 1000)
    then "Lapin attrapé en "+nbsaut+" bonds"
    else "Le lapin court toujours après 1000 bonds")
}
```

```
return(display(polygonscatterplot([pR]),blue+line_width_3+point_width_3),Dir)
};;
```

Programme 312 – poursuite renard/lapin (1)

Par exemple, avec des bonds du renard deux fois plus longs que ceux du lapin :

```
poursuite1(6,4,1,0.5,7)
```



## G2 Avec un lapin plus malin ?

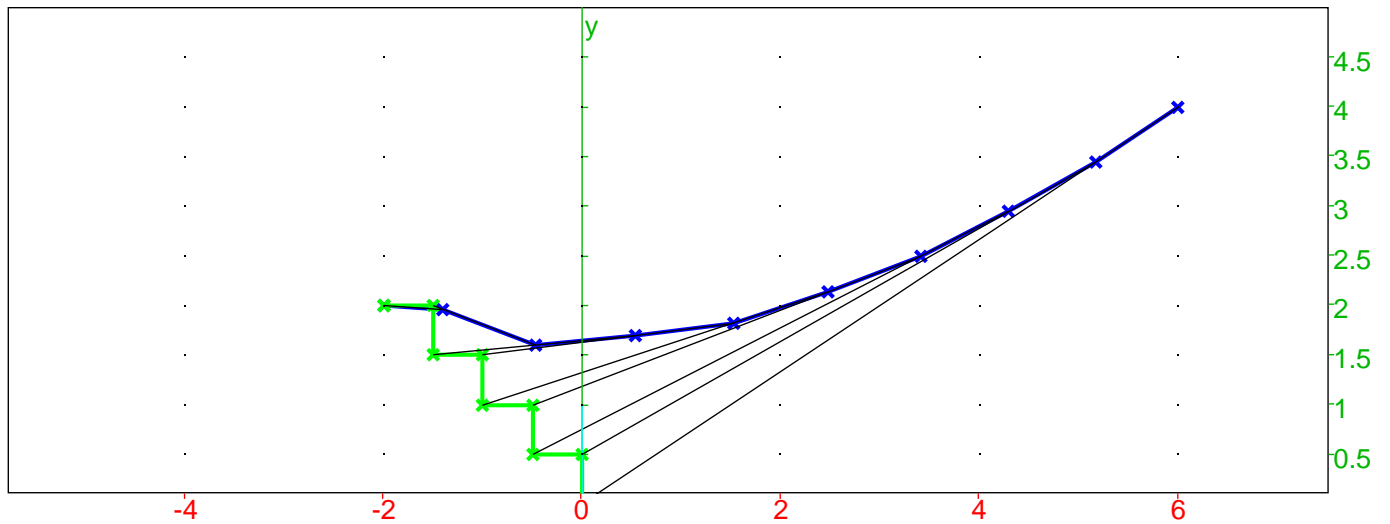
Le lapin fait maintenant des bonds en zigzag à angle droit.

```
poursuite2(xo,yo,br,bl):={
local R,L,d,pR,nbsaut,Dir;
R:=[xo,yo];
L:=[0,0];
pR:=NULL;
pL:=NULL;
Dir:=NULL;
nbsaut:=0;
d:=evalf(sqrt((L[0]-R[0])^2+(L[1]-R[1])^2));
while((nbsaut<1000) and (d>0)){
pR:=R,pR;
pL:=L,pL;
Dir:=Dir,segment(point(R),point(L));
d:=evalf(sqrt((L[0]-R[0])^2+(L[1]-R[1])^2));
if(d>br)then{
R:=R+(br/d)*(L-R);
L:=L+if(irem(nbsaut,2)==0)then{[0,bl]}else{[-bl,0]};
}else{R:=L}
nbsaut:=nbsaut+1;
}
print(if(nbsaut<1000)
then{"Lapin attrapé en "+nbsaut+" bonds"}
else{"Le lapin court toujours après 1000 bonds"})
```

```
return(display(polygonscatterplot([pR]),blue+line_width_3+point_width_3),  
display(polygonscatterplot([pL]),green+line_width_3+point_width_3),  
Dir)  
};;
```

Programme 313 – poursuite renard/lapin (2)

Le lapin gagne 2 bonds...





# 15 - Algorithmes en probabilité

## A Pile ou face

### A1 Avec XCAS

On lance trois fois de suite une pièce de monnaie. On compte combien de fois pile (ou face) tombe.  
Avec XCAS :

```
piece(n):={
T:=NULL;
pour k de 1 jusque n faire
  X:=0;
  si hasard(2)==1 alors X:=X+1; fsi;
  si hasard(2)==1 alors X:=X+1; fsi;
  si hasard(2)==1 alors X:=X+1; fsi;
  T:=T,X;
fpour;

retourne(evalf([count_eq(0,[T])/n,count_eq(1,[T])/n,count_eq(2,[T])/n,count_eq(3,[T])/n])
)};
```

Programme 314 – pile ou face

Sachant que :

- rand(k) renvoie un entier aléatoirement compris entre 0 et  $k - 1$  ;
- count\_eq(entier, liste) compte le nombre de fois ou entier apparaît dans liste

Par exemple :

```
piece(5000)
```

renvoie [0.1262,0.387,0.366,0.1208].

En effet, on peut vérifier avec binomial(n,k,p) qui renvoie  $\binom{n}{k} p^k (1-p)^{n-k}$  :

```
[seq(binomial(3,k,0.5),k=0..3)]
```

renvoie [0.125,0.375,0.375,0.125]

On peut faire la moyenne sur une dizaine de séries de 5000 séries de 3 lancers :

```
mean([seq(piece(5000),k=0..10)])
```

et on obtient [0.1272909091,0.3854,0.3637272727,0.1235818182]

### A2 Avec Sage

```
def count_eq(n,L): # pour compter les occurrences de n dans la liste L
  return len(filter(lambda x:x==n,L))
```

```
def piece(n):
    T=[]
    for k in xrange(1,n+1):
        X=0
        if floor(2*random())==1 : X+=1
        if floor(2*random())==1 : X+=1
        if floor(2*random())==1 : X+=1
        T+= [X]
    return([(count_eq(j,T)/n).n(digits=3) for j in xrange(0,4) ])
```

Par exemple, pour 100 000 lancers :

```
sage: piece(100000)
[0.125, 0.376, 0.374, 0.125]
```

## B Tirage de boules

### B1 Avec XCAS

On dispose de trois urnes, la première contenant 7 boules blanches et 4 noires, la deuxième 5 blanches et 2 noires, la troisième 6 blanches et 3 noires.

On tire une boule dans chaque urne et on note le nombre de boules blanches obtenues.

On adapte l'algorithme précédent :

```
boules(n):={
T:=NULL;
pour que de 1 jusque n faire
    X:=0;
    si hasard(11)<7 alors X:=X+1; fsi;
    si hasard(7)<5 alors X:=X+1; fsi;
    si hasard(9)<6 alors X:=X+1; fsi;
    T:=T,X;
fpour
retourne(evalf([count_eq(0,[T])/n,count_eq(1,[T])/n,count_eq(2,[T])/n,count_eq(3,[T])/n])
)};;
```

Programme 315 – exemple de tirage de boules

On obtient [0.0358, 0.2253818182, 0.4453272727, 0.2934909091]

Or, par exemple,  $\frac{7}{11} \times \frac{5}{7} \times \frac{6}{9} \approx 0,303$  et  $\frac{4}{11} \times \frac{2}{7} \times \frac{3}{9} \approx 0,034$ .

### B2 Avec Sage

```
def boules(n):
    T=[]
    for k in xrange(1,n+1):
        X=0
        if floor(11*random())<7 : X+=1
        if floor(7*random())<5 : X+=1
        if floor(9*random())<6 : X+=1
        T+= [X]
    return([(count_eq(j,T)/n).n(digits=3) for j in xrange(0,4) ])
```

Programme 316 – exemple de tirage de boules

Par exemple, pour 10 000 lancers :

```
sage: boules(100000)
[0.0345, 0.218, 0.445, 0.302]
```

## C Tirage de boules avec remise

Simulons le tirage successif de quatre boules avec remise dans une urne contenant 7 boules blanches et 3 boules noires. Comptons le nombre de tirages contenant

- exactement deux boules blanches;
- au moins une boule blanche.

### C1 Avec XCAS

```
boule(n,nb):={
local X,k,j,B;
X:=0;
pour k de 1 jusque n faire
  B:=0;
  pour j de 1 jusque 4 faire
    si hasard(10)<7 alors B:=B+1 fsi
  fpour
  si B==nb alors X:=X+1; fsi
fpour
retourne(evalf(X/n))
};;
```

Programme 317 – tirage de boules avec remise (XCAS)

Alors pour 1 000 000 tirages, on obtient pour deux boules blanches :

```
boule(1000000,2)
```

0.264818

En effet, le tirage se fait sans remise. La probabilité d'obtenir (N, N, B, B) est :

$$\frac{3}{10} \times \frac{3}{10} \times \frac{7}{10} \times \frac{7}{10} = \frac{441}{10000}$$

Les tirages de deux boules exactement sont les anagrammes de (N, N, B, B).

On multiplie donc le résultat précédent par  $\frac{4!}{2! \times 2!} = 6$ .

$$6 \times \frac{441}{10000} = 0,2646$$

Obtenir au moins une boule blanche est le contraire d'en obtenir aucune :

```
1-boule(1000000,0)
```

0.991846

En effet

$$1 - \left(\frac{3}{10}\right)^4 = 0,9919$$

## C2 Avec Sage

```
def boule(n,nb):
    X=0
    for k in xrange(1,n+1):
        B=0
        for j in xrange(1,5):
            if floor(10*random())<7 : B+=1
        if B==nb : X+=1
    return float(X/n)
```

Programme 318 – tirage de boules avec remise (XCAS)

Alors pour 100000 tirages, on obtient pour deux boules blanches :

```
sage: boule(100000,2)
0.26401999999999998
```

Obtenir au moins une boule blanche est le contraire d'en obtenir aucune :

```
sage: 1-boule(100000,0)
0.99189000000000005
```

## D Le Monty Hall

Le jeu oppose un présentateur à un candidat. Ce joueur est placé devant trois portes fermées. Derrière l'une d'elles se trouve une voiture et derrière chacune des deux autres se trouve une chèvre. Il doit tout d'abord désigner une porte. Puis le présentateur ouvre une porte qui n'est ni celle choisie par le candidat, ni celle cachant la voiture (le présentateur sait quelle est la bonne porte dès le début). Le candidat a alors le droit ou bien d'ouvrir la porte qu'il a choisie initialement, ou bien d'ouvrir la troisième porte.

Les questions qui se posent au candidat sont :

- Que doit-il faire ?
- Quelles sont ses chances de gagner la voiture en agissant au mieux ?

On peut avoir deux idées contradictoires :

- le présentateur ayant ouvert une porte, le candidat a une chance sur deux de tomber sur la voiture ;
- Au départ, le candidat a une probabilité de  $\frac{1}{3}$  de tomber sur la voiture. Ensuite, s'il garde son choix initial, il a toujours la même probabilité  $\frac{1}{3}$  de gagner donc s'il change d'avis, la probabilité qu'il gagne est donc  $1 - \frac{1}{3} = \frac{2}{3}$ .

Qui a raison ?

### D1 Avec XCAS

Simulons la situation à l'aide de XCAS.

```
monty(n):={
local gagne_sans_changer ,gagne_en_changeant , j ,voiture ,choix ,ouverte ,changement ;
gagne_sans_changer:=0;
gagne_en_changeant:=0;
pour j de 1 jusque n faire

    voiture:=hasard(3);
    choix:=hasard(3);

    si choix==voiture
    alors ouverte:=irem(voiture + (1+hasard(2)) , 3);
    sinon ouverte:=(0+1+2) - choix - voiture;
    fsi
```

```

changement:=(0+1+2) - choix - ouverte;

si choix==voiture
alors  gagne_sans_changer:=gagne_sans_changer+1
fsi

si changement==voiture
alors  gagne_en_changeant:=gagne_en_changeant+1
fsi
fpour
retourne("Gagne en changeant : "+(100.*gagne_en_changeant/n)+"%
Gagne sans changer : "+ (100.*gagne_sans_changer/n)+"%")
};
    
```

Programme 319 – jeu du Monty Hall (XCAS)

Alors pour 1000000 de parties :

```

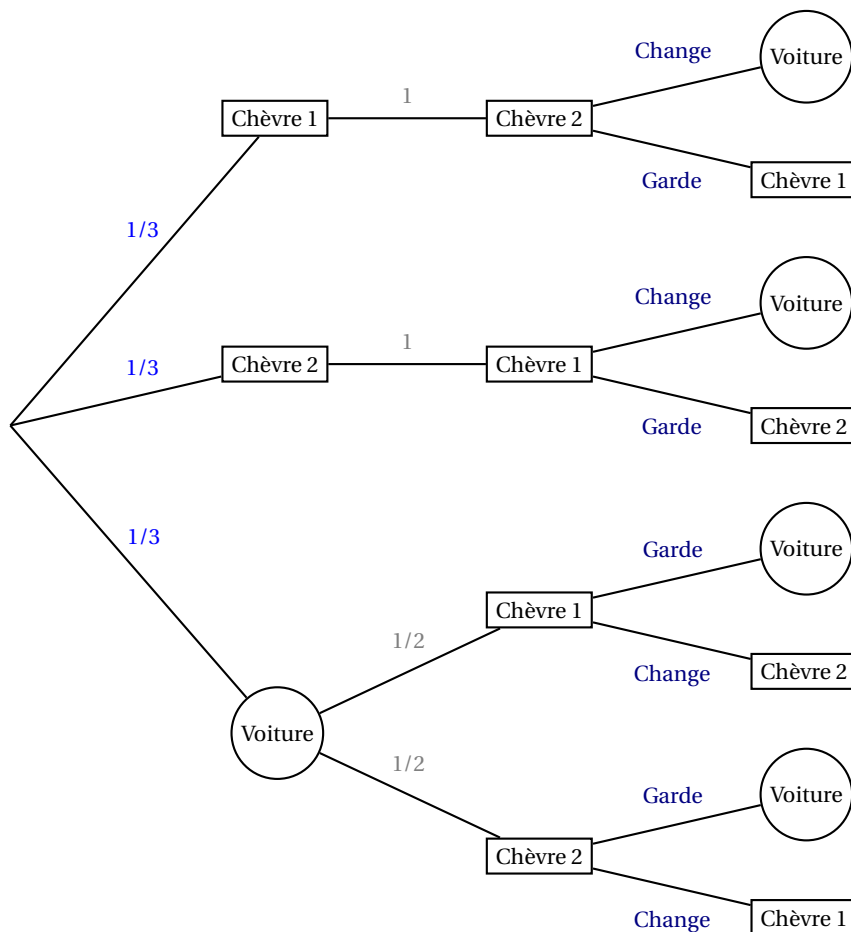
monty(1000000)
    
```

Gagne en changeant : 66.6898% Gagne sans changer : 33.3102%

La deuxième option semble la bonne...il reste à le prouver!

Dressons un arbre en trois étapes :

- Le candidat choisit d'abord une porte de manière équiprobable ;
- Le présentateur, sachant où se trouve la voiture, choisit une porte cachant une chèvre ;
- Le candidat prend ensuite la décision de garder ou de changer son choix initial.



Nous en déduisons que :

- s'il conserve son choix initial, la probabilité que le candidat gagne la voiture est :

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

- s'il change son choix initial, la probabilité que le candidat gagne la voiture est :

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

Il vaut donc mieux pour le candidat changer son choix initial.

## D2 Avec Sage

```
def monty(n):
    gagne_sans_changer=0
    gagne_en_changeant=0
    for j in xrange(1,n+1):
        voiture=floor(3*random())
        choix=floor(3*random())
        if choix==voiture:
            ouverte=(voiture+1+floor(2*random())) % 3
        else:
            ouverte=0+1+2-choix-voiture
            changement=0+1+2-choix-ouverte
        if choix==voiture:
            gagne_sans_changer+=1
        if changement==voiture:
            gagne_en_changeant+=1
    return 'Gagne en changeant : '+str(100.*gagne_en_changeant/n)+'%   Gagne sans changer : '+ str
        (100.*gagne_sans_changer/n)+'%
```

Programme 320 – Monty Hall (Sage)

Alors, pour 100000 expériences :

```
sage: monty(100000)
'Gagne en changeant : 66.39300000000000%   Gagne sans changer : 33.60700000000000%'
```

## E Problème du Duc de Toscane

Cosme II de Médicis (Florence 1590-1621), Duc de Toscane, fut le protecteur de l'illustre Galilée (né à Pise le 15 février 1564 et mort à Florence le 8 janvier 1642) son ancien précepteur. Profitant d'un moment de répit du savant entre l'écriture d'un théorème sur la chute des corps et la création de la lunette astronomique, le Grand Duc lui soumet le problème suivant : il a observé qu'en lançant trois dés cubiques et en faisant la somme des numéros des faces, on obtient plus souvent 10 que 9, alors qu'il y a autant de façons d'obtenir 9 que 10, à savoir six.

Après quelques réflexions, Galilée rédigea un petit mémoire sur les jeux de hasard en 1620 expliquant le phénomène.

### E1 Avec XCAS

#### E1a Méthode « naturelle »

On utilise une boucle *pour* :

```
toscane(n):={
local T,k;
T:=NULL;
pour k de 1 jusque n faire
```

```
T:=T,(((3+hasard(6))+hasard(6))+hasard(6))
fpour
retourne("Obtenir 9 : "+count_eq(9,[T])*100.0/n+"%", " Obtenir 10 : "+count_eq(10,[T])*100.0/n+"%")
};;
```

Programme 321 – problème du Duc de Toscane (XCAS 1)

Pour 100000 simulations sur mon ordinateur, il faut... énormément de temps!

### E 1 b Utilisation de l'opérateur =<

L'opérateur permet de stocker une valeur dans une liste déjà existante : on gagne donc du temps car on ne s'occupe que d'un terme à chaque fois et non pas de toute la liste.

```
toscanebis(n):={
local T,k;
T:=seq(0,k=1..n); /* on crée un tableau contenant n zéros */
pour k de 0 jusque n-1 faire
T[k]=<(((3+hasard(6))+hasard(6))+hasard(6)) /* on remplace petit à petit chaque zéro */
fpour
retourne("Obtenir 9 : "+count_eq(9,[T])*100.0/n+"%", " Obtenir 10 : "+count_eq(10,[T])*100.0/n+"%")
};;
```

Programme 322 – problème du Duc de Toscane (XCAS 2)

Cette fois il faut 1 seconde pour 100000 simulations sur mon ordinateur.

Evaluation time: 1.38

"Obtenir 9 :11.674%", " Obtenir 10 :12.56%"

### E 1 c Utilisation de ranm

```
toscaneater(n):={
T:=ranm(1,n,'(((3+hasard(6))+hasard(6))+hasard(6))');
retourne("Obtenir 9 : "+count_eq(9,T)*100.0/n+"%", " Obtenir 10 : "+count_eq(10,T)*100.0/n+"%")
};;
```

Programme 323 – problème du Duc de Toscane (XCAS 3)

Alors, pour 100000 tirages, on obtient en moins d'une seconde :

Evaluation time: 0.52

"Obtenir 9 :11.47%", " Obtenir 10 :12.477%"

On a utilisé `ranm(lignes,colonnes,'expérience')` qui renvoie une matrice lignes×colonnes où chaque terme est un résultat de expérience.

## E 2 Avec Sage

```
def toscane(n):
def count_eq(n,L):
return len(filter(lambda x:x==n,L))
T=n*[0]
for k in xrange(0,n):
T[k]=3+floor(6*random())+floor(6*random())+floor(6*random())
return 'Obtenir 9 :'+str(count_eq(9,T)*100.0/n)+'% , Obtenir 10 :'+str(count_eq(10,T)*100.0/n)+'%'
```

Programme 324 – problème du Duc de Toscane (Sage)

Pour 100000 lancers :

```
sage: toscane(100000)
'Obtenir 9 :11.744000000000000% , Obtenir 10 :12.460000000000000%'
```

## F Lancers d'une pièce et résultats égaux

On lance 10 fois de suite une pièce de monnaie et on s'intéresse au nombre maximal de résultats consécutifs égaux. On crée un programme qui simule autant de séries de lancers que l'on désire.

### F1 Avec XCAS

```
piece_max(essais):={
local S,k,P,j,H,m,M,s,p,h;
S:=[seq(0,k=1..essais)];
pour k de 1 jusque essais faire
  s:=NULL;
  P:=[seq(hazard(2),m=0..9)];
  p:=0;
  tantque p<9 faire
    j:=p+1;
    tantque (P[j]==P[p] and j<9) faire
      j+=1
    ftantque;
    s:=s,j-p;
    p:=j;
  ftantque;
  s:=SortD([s]);
  h:=head(s);
  S[k]=<h;
fpour;
m:=evalf(moyenne(S),2);
retourne("
Sur "+essais+" séries de 10 lancers, la moyenne du nombre maximal de résultats consécutifs égaux
est "+m)
};;
```

Programme 325 – pile ou face (2)

- On crée une liste S remplie de essais zéros pour y mettre les max;
- on boucle les essais;
- on crée une liste s enregistrant le nombre de résultats consécutifs égaux;
- on crée une liste P de 10 lancers de 0 ou 1;
- tant qu'on n'est pas au bout de la liste (p<9), on regarde le suivant (j=p+1)
- tant que le suivant est égal, on continue (j:=j+1);
- on rajoute le nombre de lancers égaux et on enlève p car on a commencé à p+1 (s:=s, j-p);
- on va ensuite voir le prochain résultat différent (p:=j);
- on classe dans l'ordre décroissant les résultats (SortD([s]));
- on prend le premier de la liste : c'est le plus grand (head(s));
- on le stocke dans une liste avant de refaire une série de 10 (S[k]=<h).

Par exemple :

```
piece_max(10000)
```

Sur 10000 séries de 10 lancers, la moyenne du nombre maximal de résultats consécutifs égaux est 3.51



## F2 Avec Sage

```
def piece_max(n):
    S=n*[1]
    for k in xrange(0,n):
        s=[]
        P=[floor(2*random()) for z in xrange(0,10)]
        p=0
        while p<9:
            j=p+1
            while (P[j]==P[p] and j<9):
                j+=1
            s+=[j-p]
            p=j
        s.sort(reverse=True)
        S[k]=s[0]
    m=(sum(S)/n).n(digits=3)
    return 'Sur '+str(n)+' series de 10 lancers, la moyenne du nombre maximal de resultats
    consecutifs egaux est '+str(m)
```

Programme 326 – lancers successifs de pièces (Sage)

Avec 10000 lancers :

```
sage: piece_max(10000)
'Sur 10000 series de 10 lancers, la moyenne du nombre maximal de resultats consecutifs egaux est
3.50'
```

## G Le lièvre et la tortue

On rappelle le jeu bien connu : on dispose d'un dé cubique. On le lance. Si un six sort, le lièvre gagne sinon la tortue avance d'une case. La tortue a gagné lorsqu'elle a avancé six fois de suite.

Il est aisé de montrer que la probabilité que la tortue gagne  $\mathbb{P}(T)$  est égale à  $(\frac{5}{6})^6 \approx 0,335$ .

Voici un exemple de simulation de  $n$  parties avec XCAS :

```
jeutortue(n):={
local simul,L,T,face,lancer;
L:=0;T:=0;
pour simul de 1 jusque n faire
face:=hasard(6)+1;
lancer:=1;
tantque(face<6) et (lancer<6) faire
face:=hasard(6)+1;
lancer+=1;
ftantque
si face==6
alors L+=1
sinon T+=1
fsi
fpour
retourne("Tortue : "+T*100.0/n+"% Lièvre : "+L*100.0/n+"%")
};;
```

Programme 327 – le lièvre et la tortue (1)

Alors on peut simuler 100000 parties en entrant :

```
jeutortue(100000)
```

et on obtient par exemple :

Tortue :33.639% Lievre : 66.361%

On peut modifier légèrement le programme précédent pour déterminer quel nombre maximum de lancers on doit fixer pour rendre le jeu favorable à la tortue :

```
jeutortue(n,lancer_max):={
local simul,L,T,face,lancer;
L:=0;T:=0;
pour simul de 1 jusque n faire
face:=hasard(6)+1; lancer:=1;
tantque(face<6) et (lancer<lancer_max)faire
face:=hasard(6)+1;
lancer:=lancer+1;
ftantque
si face==6
alors L:=L+1
sinon T:=T+1
fsi
fpour
retourne(T*100.0/n)
};;
```

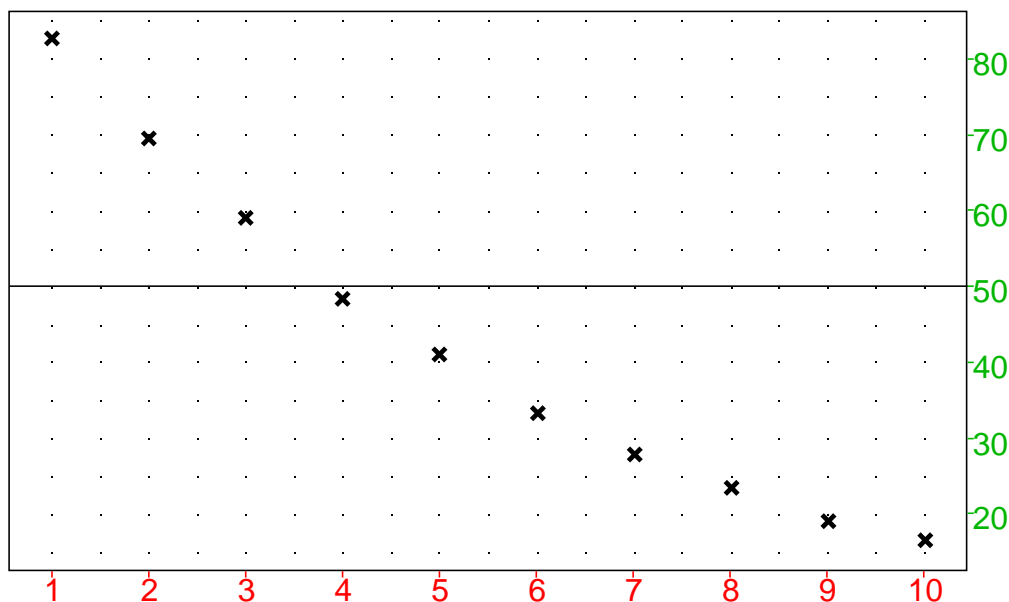
Programme 328 – le lièvre et la tortue (2) (XCAS)

On n'a cette fois en sortie que le pourcentage de parties gagnées par la tortue.

On trace ensuite le nuage de points de coordonnées (lancer\_max, jeutortue(10000,k)) avec k variant de 1 à 10 par exemple.

```
nuage_points([seq([k, jeutortue(10000,k)],k=1..10)],droite(y=50))
```

et on obtient :



## G1 Avec Sage

```
def jeutortue(n):
[L,T]=[0,0]
```

```

for simul in xrange(1,n+1):
    face=1+floor(6*random())
    lancer=1
    while face<6 and lancer<6:
        face=1+floor(6*random())
        lancer+=1
    if face==6 : L+=1
    else : T+=1
return 'Tortue :'+str(T*100.0/n)+'% Lièvre : '+str(L*100.0/n)+'%'

```

Programme 329 – lièvre et tortue 1 (Sage)

Sur 10000 parties :

```

sage: jeutortue(100000)
'Tortue :33.31000000000000% Li\xc3\xa8vre : 66.69000000000000%'

```

Quel nombre maximum de lancers doit-on fixer pour rendre le jeu favorable à la tortue ?

```

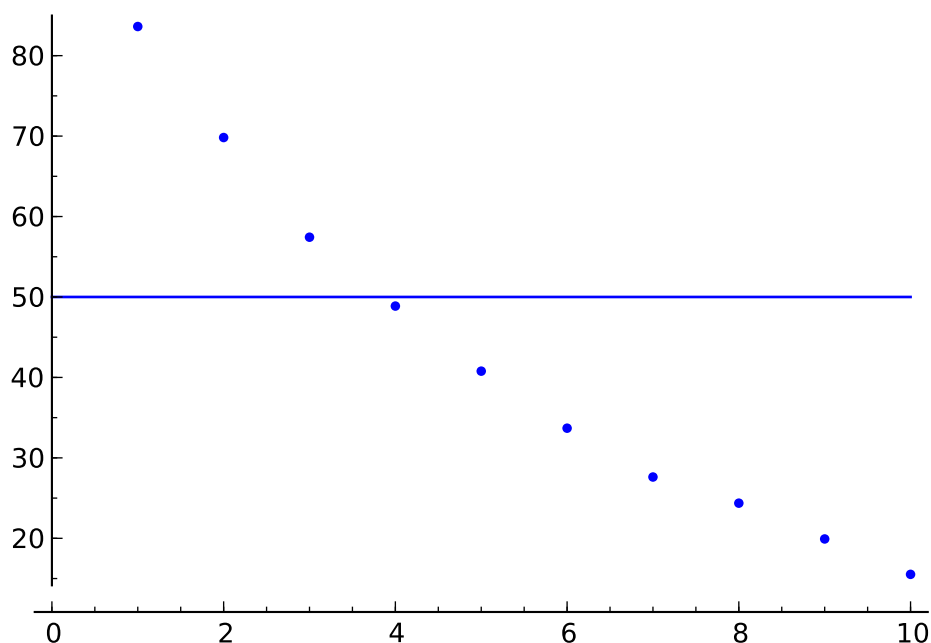
def jeutortuemax(n,lancer_max):
    [L,T]=[0,0]
    for simul in xrange(1,n+1):
        face=1+floor(6*random())
        lancer=1
        while face<6 and lancer<lancer_max:
            face=1+floor(6*random())
            lancer+=1
        if face==6 : L+=1
        else : T+=1
    return T*100.0/n

def trace_tortue():
    L=points([[k,jeutortuemax(10000,k)] for k in xrange(1,11)])
    D=line([[0,50],[10,50]])
    (L+D).show()

```

Programme 330 – lièvre et tortue 2 (Sage)

On lance trace\_tortue() et on obtient :



## H Générateurs de nombres aléatoires

Les machines ont leur propre générateur de nombre aléatoire. Pouvons-nous en fabriquer un nous-même ?

### H1 Méthode « middle square » de John Von Neumann

L'un des pères fondateurs de l'informatique avec Alan TURING et Alonzo CHURCH... des années avant l'invention du premier ordinateur.

On prend un nombre de  $n$  chiffres, on l'élève au carré et on forme un nouveau nombre avec les  $n$  chiffres situés « au milieu » du carré et on recommence.

Les nombres ainsi formés appartiennent à l'intervalle  $[[0; 10^n - 1]]$ . Il suffit alors de les diviser par  $10^n$  pour avoir des nombres entre 0 et 1.

Premier problème : comment extraire  $n$  chiffres « du milieu » ?

D'ailleurs, on ne sait même pas combien il y aura de chiffres :  $2n - 1$  ou  $2n$ .

Prenons par exemple  $n = 4$ .

En XCAS, on va utiliser exceptionnellement les commandes `input`, `goto` et `label` :

```
ms(N):={
S:=N;
k:=1;
print(N);
label truc;
N:=N^2;
  if(N>=10^7)
  then{N:=floor(N/10^2);
      N:=N-10^4*floor(N/10^4);
      S:=S+N;k:=k+1;
      afficher(N)
  }
  else{N:=floor(N/10);
      N:=N-10^(4)*floor(N/10^(4));
      S:=S+N;k:=k+1;
      afficher(N)
  }
input("Continuer : o sinon : n",rep);
if(rep==o)goto truc;
if(rep==n)return("moyenne : "+evalf(S/k));
};;
```

Programme 331 – générateur de nombres aléatoires

Une petite fenêtre apparaît à chaque génération pour demander si on doit continuer. Si on arrête en répondant n, on obtient la moyenne des résultats.

## I Calcul de $\pi$ avec une pluie aléatoire

### I1 La méthode

### I2 Avec XCAS en impératif

```
pluie(n):={
local r,k;
for(k:=1;k<=n;k:=k+1){
  r:=r+1-floor(rand(0,1)^2+rand(0,1)^2)
}
return(4.0*r/n)
```

};;

Programme 332 – calcul de  $\pi$  avec une pluie aléatoire en impératif (XCAS)

### I3 Avec XCAS en impératif

```
def pluie(n):
    r=0
    for k in xrange(1,n+1):
        r+=1-floor(random()*2+random()*2)
    return 4.*r/n
```

Programme 333 – calcul de  $\pi$  avec une pluie aléatoire en impératif (Sage)

```
sage: pluie(1000000)
3.143380000000000
```

### I4 Avec CAML en récursif

```
# let pluie(n)=
let rec pr(n)=
  if n=0
  then 0
  else 1+pr(n-1)-int_of_float(Random.float(1.))*2.+Random.float(1.))*2.)
in 4.*.float_of_int(pr(n)).float_of_int(n);;
```

Programme 334 – calcul de  $\pi$  avec une pluie aléatoire en récursif

## J Loi normale

### J1 Avec XCAS

Ne connaissant pas de primitive de  $t \mapsto \frac{1}{\sqrt{2\pi}}e^{-\frac{t^2}{2}}$ , on utilise la méthode des rectangles pour obtenir une approximation de

$$\int_0^X \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt$$

```
normale(T):={
DIGITS:=4;
dt:=0.001;
f:=t->(exp(-0.5*t^2))/sqrt(2*Pi);
t:=0;
S:=0;
while(t<=T){
S:=S+f(t)*dt;
t:=t+dt;
}
return(S)
};;
```

Programme 335 – approximation d'une probabilité élémentaire (loi normale)

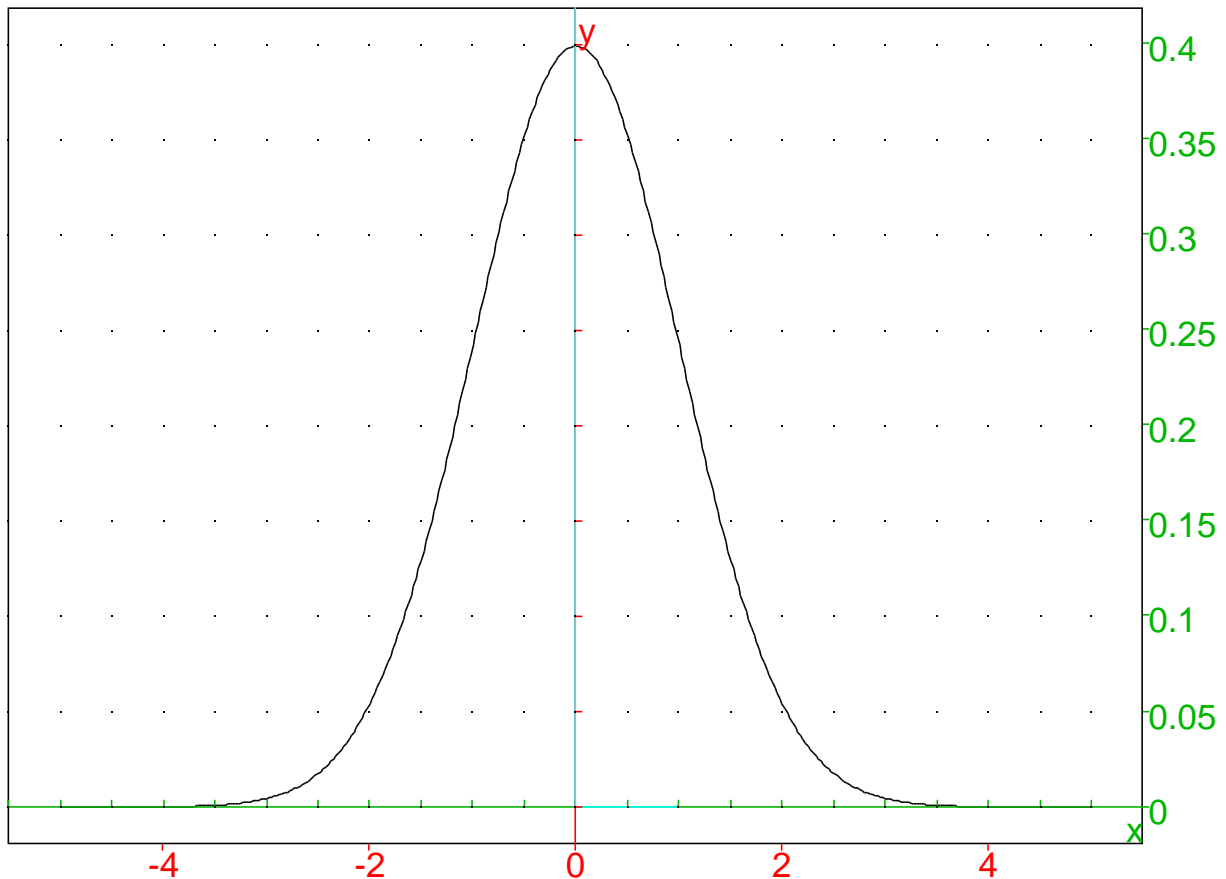
On s'aperçoit que cette intégrale *semble* converger vers  $\frac{1}{2}$  :

```
normale(10); normale(20);
```

0.5002,0.5002

Ça peut se comprendre (mais pas se démontrer en BTS...) en observant le graphe correspondant :

```
graphe((exp(-0.5*t^2))/sqrt(2*Pi),t=-5..5)
```



On peut ensuite utiliser notre procédure normale pour construire la table qu'on retrouve sur le formulaire des BTS :

```
tablenormale():={
local T,l,L,c;
T:=[ "", seq(k*0.01,k=0..9) ];
for(l:=0;l<3;l:=l+0.1){
L:=l;
for(c:=0;c<=0.09;c:=c+0.01){L:=L,0.5+normale(l+c)}
T:=T,[L]
}
return([T])
}::;
```

Programme 336 – table de la loi normale

et on obtient en 10 secondes :

	0.0	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
0	0.500399	0.503989	0.507978	0.511967	0.515954	0.519939	0.523923	0.527904	0.531882	0.535857
0.1	0.539829	0.543797	0.547760	0.551718	0.555672	0.559620	0.563562	0.567498	0.571427	0.575349
0.2	0.579264	0.583171	0.587069	0.590959	0.594841	0.598712	0.602575	0.606427	0.610269	0.614100
0.3	0.617920	0.621729	0.625526	0.629311	0.633083	0.636842	0.640589	0.644322	0.648041	0.651746
0.4	0.655437	0.659113	0.662774	0.666420	0.670050	0.673664	0.677262	0.680843	0.684408	0.687956
0.5	0.691486	0.694999	0.698493	0.701970	0.705429	0.708868	0.712289	0.715691	0.719074	0.722437
0.6	0.725780	0.729103	0.732406	0.735689	0.738951	0.742192	0.745412	0.748611	0.751789	0.754945
0.7	0.758080	0.761192	0.764283	0.767352	0.770398	0.773422	0.776423	0.779401	0.782357	0.785290
0.8	0.788199	0.791086	0.793949	0.796789	0.799605	0.802398	0.805167	0.807913	0.810634	0.813332
0.9	0.816006	0.818656	0.821282	0.823884	0.826462	0.829016	0.831546	0.834052	0.836533	0.838990
1.0	0.841423	0.844072	0.846454	0.848812	0.851146	0.853455	0.855741	0.858002	0.860240	0.862453
1.1	0.864642	0.866808	0.868949	0.871067	0.873160	0.875230	0.877277	0.879300	0.881299	0.883275
1.2	0.885227	0.887156	0.889062	0.890945	0.892804	0.894641	0.896455	0.898246	0.900015	0.901761
1.3	0.903485	0.905186	0.906865	0.908523	0.910158	0.911772	0.913364	0.914934	0.916483	0.918011
1.4	0.919518	0.921003	0.922468	0.923913	0.925336	0.926740	0.928123	0.929486	0.930830	0.932153
1.5	0.933457	0.934742	0.936007	0.937253	0.938480	0.939689	0.940879	0.942050	0.943203	0.944338
1.6	0.945456	0.946555	0.947637	0.948702	0.949749	0.950779	0.951793	0.952789	0.953769	0.954733
1.7	0.955681	0.956613	0.957529	0.958429	0.959314	0.960183	0.961038	0.961878	0.962702	0.963513
1.8	0.964309	0.965090	0.965858	0.966612	0.967352	0.968079	0.968792	0.969492	0.970179	0.970854
1.9	0.971516	0.972165	0.972802	0.973427	0.974040	0.974641	0.975231	0.975809	0.976376	0.976932
2.0	0.977476	0.978010	0.978534	0.979047	0.979549	0.980042	0.980524	0.980997	0.981460	0.981913
2.1	0.982357	0.982792	0.983218	0.983634	0.984042	0.984442	0.984832	0.985215	0.985589	0.985955
2.2	0.986314	0.986664	0.987007	0.987342	0.987670	0.987991	0.988304	0.988611	0.988910	0.989203
2.3	0.989490	0.989769	0.990043	0.990310	0.990571	0.990825	0.991074	0.991317	0.991555	0.991787
2.4	0.992013	0.992234	0.992450	0.992660	0.992866	0.993067	0.993262	0.993453	0.993640	0.993821
2.5	0.993999	0.994171	0.994340	0.994504	0.994665	0.994821	0.994973	0.995122	0.995267	0.995408
2.6	0.995545	0.995679	0.995809	0.995937	0.996060	0.996181	0.996298	0.996413	0.996524	0.996632
2.7	0.996738	0.996840	0.996940	0.997038	0.997132	0.997224	0.997314	0.997401	0.997486	0.997568
2.8	0.997648	0.997726	0.997802	0.997876	0.997947	0.998017	0.998085	0.998150	0.998214	0.998276
2.9	0.998337	0.998395	0.998452	0.998507	0.998561	0.998613	0.998664	0.998713	0.998761	0.998807

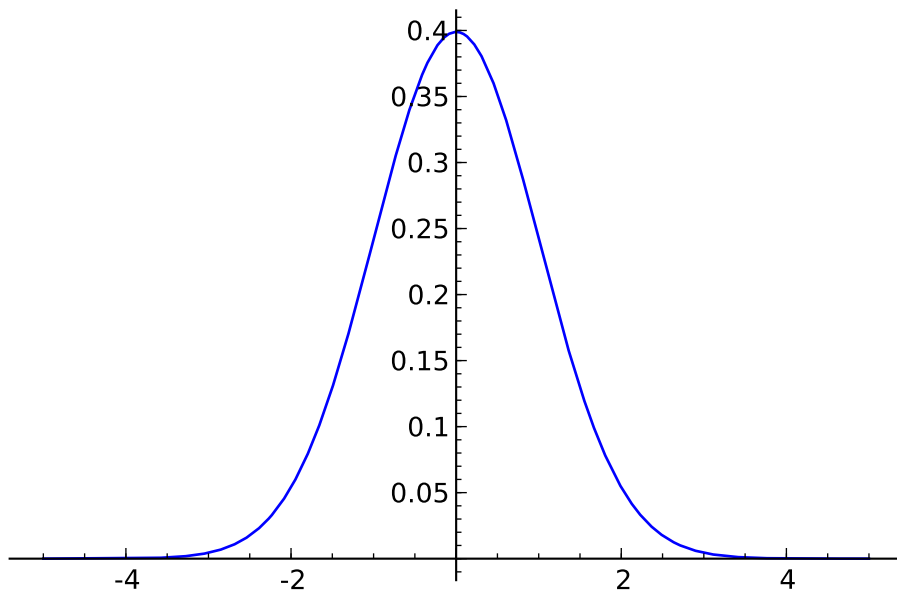
## J2 Avec Sage

```
def normale(T):
    dt=0.0001
    def f(t):
        return (exp(-0.5*t^2))/sqrt(2*float(pi))
    t=0
    S=0
    while t<=T:
        S+=f(t)*dt
        t+=dt
    return S
```

Programme 337 – loi normale (Sage)

Pour le graphique :

```
sage: t=var('t')
sage: g=plot((exp(-0.5*t^2))/sqrt(2*pi),t,-5,5)
sage: g.show()
```





## A Lissage par moyennes mobiles

On donne une liste d'abscisses, une liste d'ordonnées, un ordre et on demande le tracé de la courbe initiale et le tracé de la courbe lissée.

### A1 Avec XCAS en impératif

On crée<sup>a</sup> une fonction `som_list(liste, debut, fin)` qui calcule la somme des éléments d'une liste du rang `debut` jusqu'au rang `fin`.

```
som_list(liste,debut,fin):={
  local S,k;
  S:=0;
  for(k:=debut;k<=fin;k:=k+1){
    S+=liste[k]
  }
  return(S)
}::;
```

Programme 338 – somme des éléments d'une liste en impératif

Ensuite, il nous faut une commande `ligne(liste abs, liste ord)` qui trace la ligne brisée passant par les points dont on donne la liste des coordonnées.

```
trace_ligne(X,Y,coul):={
  local P,k;
  if(size(X)!=size(Y))then{return("Probleme de taille...")}
  P:=[seq(0,n=1..size(X))];
  for(k:=0;k<size(X);k:=k+1){
    P[k]=<[X[k],Y[k]];
  }
  couleur(polygone_ouvert(P),coul);
}::;
```

Programme 339 – tracé d'une ligne brisée

Sur XCAS, c'est `polygone_ouvert` qui relie des points par une ligne brisée; on adapte la commande au langage utilisé. Voici le programme proprement dit :

```
lmm(X,Y,k):={
  Xm:=NULL;
  Ym:=NULL;
  n:=size(X);
  if(k mod 2==0){
    p:=k/2;
```

a. Cette commande existe déjà sur XCAS mais on apprend ici à programmer quelque soit le langage...

```

for(j:=p;j<n-p;j++){
  Ym:=Ym,(0.5*Y[j-p]+som_list(Y,j-p+1,j+p-1)+0.5*Y[j+p])/k;
  Xm:=Xm,X[j];
}
}else{
p:=(k-1)/2;
for(j:=p;j<n-p;j++){
  Ym:=Ym,(som_list(evalf(Y),j-p,j+p))/k;
  Xm:=Xm,X[j];
}
}
Xm:=[Xm];
Ym:=[Ym];
C:=trace_ligne(X,Y,red);
Cm:=trace_ligne(Xm,Ym,blue);
return(C,Cm)
};

```

Programme 340 – lissage par moyenne mobile en impératif (XCAS)

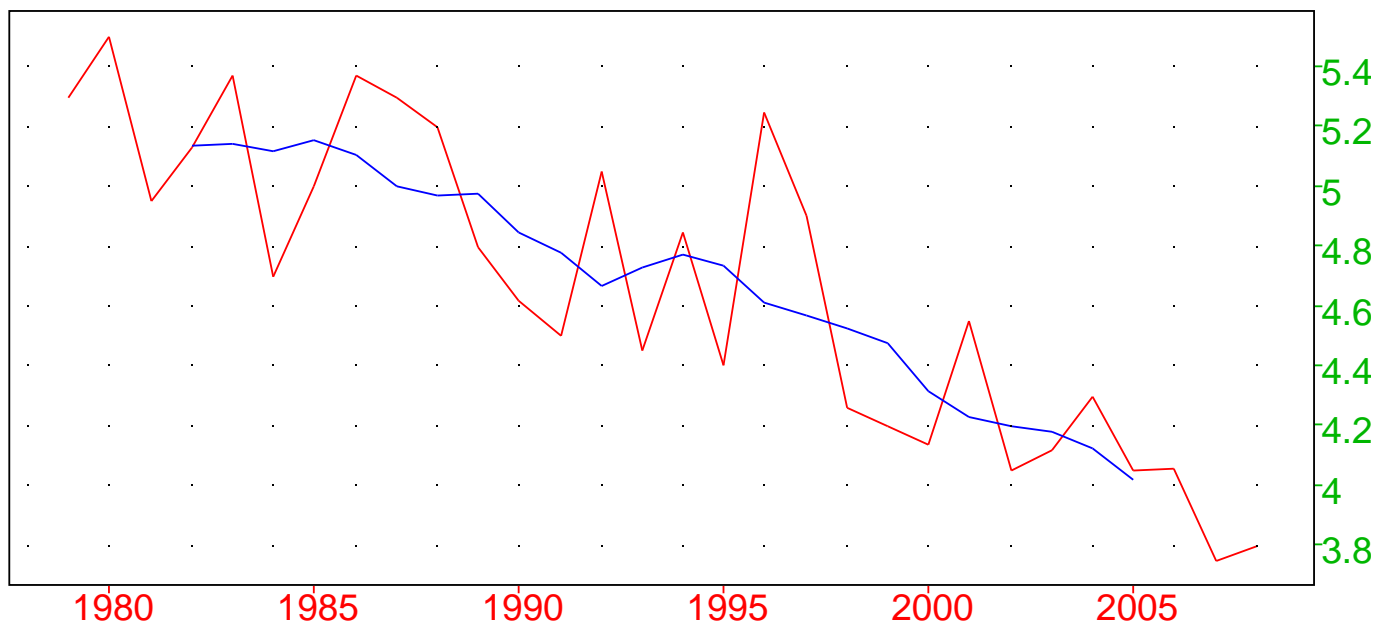
Par exemple, voici ce que cela donne pour la moyenne lissée d'ordre 7 de la série de mesure de l'étendue de la banquise au mois de septembre :

```

lmm([1979,1980,1981,1982,1983,1984,1985,1986,1987,1988,1989,1990,1991,
1992,1993,1994,1995,1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,
2006,2007,2008],[5.3,5.5,4.95,5.13,5.37,4.7,5.5,5.37,5.3,5.2,4.8,4.62,4.5,
5.05,4.45,4.85,4.4,5.25,4.9,4.26,4.2,4.14,4.55,4.05,4.12,4.3,4.05,4.06,
3.75,3.8],7)

```

On obtient l'alarmant graphique :



## A2 Avec Sage en impératif

On suit la même tactique :

```

def som_list(liste,debut,fin):
  S=0
  for k in xrange(debut,fin+1):
    S+=liste[k]

```

```

return S

def trace_ligne(X,Y,coul):
    P=len(X)*[0]
    for k in xrange(0,len(X)):
        P[k]=[X[k],Y[k]]
    return line(P,color=coul)

def lmm(X,Y,k):
    Xm=[]
    Ym=[]
    n=len(X)
    if k%2==0:
        p=k//2
        for j in xrange(p,n-p):
            Ym+=[(0.5*Y[j-p]+som_list(Y,j-p+1,j+p-1)+0.5*Y[j+p])/k]
            Xm+=[X[j]]
    else:
        p=(k-1)//2
        for j in xrange(p,n-p):
            Ym+=[(som_list(Y,j-p,j+p))/k]
            Xm+=[X[j]]
    C=trace_ligne(X,Y,'red')
    Cm=trace_ligne(Xm,Ym,'blue')
    LMM=C+Cm
    LMM.show()

```

Programme 341 – lissage par moyenne mobile en impératif (Sage)

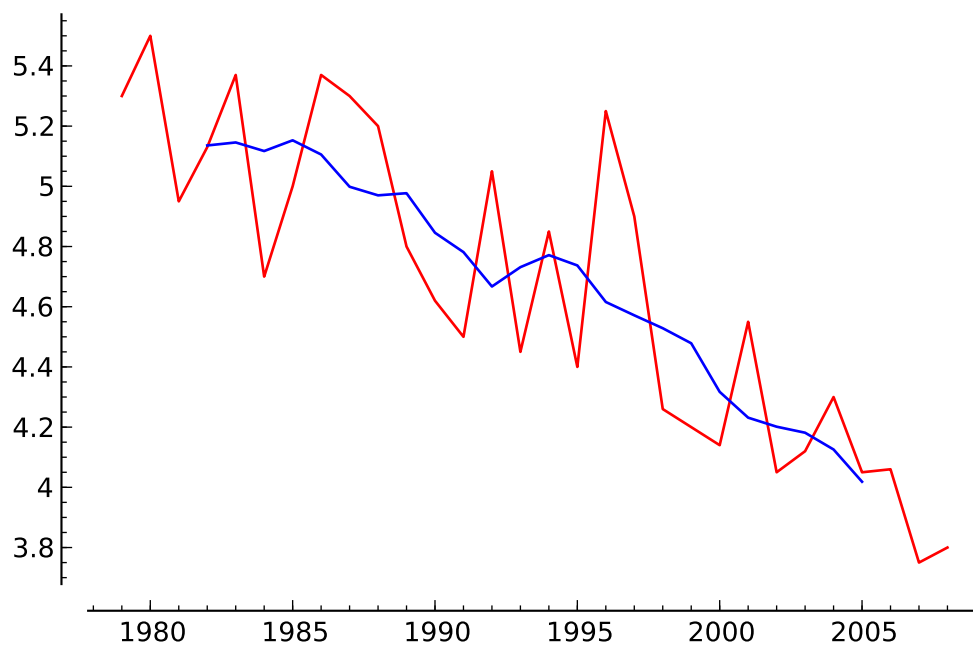
Alors avec la même série :

```

sage: lmm
([1979,1980,1981,1982,1983,1984,1985,1986,1987,1988,1989,1990,1991,1992,1993,1994,1995,1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011],
[5.3,5.5,4.95,5.13,5.37,4.7,5,5.37,5.3,5.2,4.8,4.62,4.5,5.05,4.45,4.85,4.4,5.25,4.9,4.26,4.2,4.14,4.55,4.05,4.12,4.3,4.05,4.06,3.75,3.8],7)

```

On obtient :



### A3 Avec CAML en récursif

Amusons-nous à fabriquer tous nos outils...

D'abord une fonction qui extrait une liste d'une liste existante entre des rangs donnés :

```
# let extrait(liste,debut,fin)=
  let rec extract=function
    |([],debut,fin,n)->[]
    |tete::queue,debut,fin,n->
      if (n>=debut) & (n<=fin)
      then tete::extract(queue,debut,fin,n+1)
      else if n<=fin
          then extract(queue,debut,fin,n+1)
          else []
  in extract(liste,debut,fin,0);;
```

Programme 342 – extraction d'une sous-liste en récursif

Par exemple :

```
# extrait([0;1;2;3;4;5;6;7;8;9],3,7);;
- : int list = [3; 4; 5; 6; 7]
```

Ensuite calculons la longueur d'une liste :

```
# let rec longueur=function
  | [] -> 0
  | tete::queue -> 1+longueur(queue);;
```

Programme 343 – longueur d'une liste en récursif

Puis la somme des éléments d'une liste :

```
# let rec somme=function
  | [] -> 0.
  | tete::queue -> tete +. somme(queue);;
```

Programme 344 – somme des éléments d'une liste en récursif

Calculons maintenant la moyenne mobile d'ordre  $k$  d'une liste donnée avec  $k$  un entier *impair* :

```
# let moyenne_mobile_impair(y,k)=
  let rec mobile(y,k,n)=
    if n+k/2>=longueur(y) then []
    else somme(extrait(y,n-k/2,n+k/2)).float_of_int(k)::mobile(y,k,n+1)
  in mobile(y,k,k/2);;
```

Programme 345 – moyenne mobile en récursif

Par exemple :

```
# moyenne_lisse_mobile([1.;2.;3.;4.;5.;6.;7.;8.;9.;10.;11.],5);;
- : float list = [3.; 4.; 5.; 6.; 7.; 8.; 9.]
```

## B Intervalle de fluctuation en 2<sup>nde</sup>

### B1 Le fameux théorème vu par XCAS

Le document d'accompagnement affirme que pour des échantillons de taille  $n$  obtenus à partir d'un modèle de Bernoulli, 95% des mesures des fréquences mesurées sont comprises dans l'intervalle  $\left[p - \frac{1}{\sqrt{n}}, p + \frac{1}{\sqrt{n}}\right]$  avec  $p$  la proportion à mesurer.

On simule ici  $N$  échantillons de taille  $n$  d'un modèle de Bernoulli ayant une probabilité  $p$ .  
 On simule une expérience de probabilité  $p$  : `evalf(hasard(0,1))<p`. En effet, on tire au hasard un décimal entre 0 et 1 et on compte un point si ce nombre est inférieur à  $p$ .  
 On trace le nuage des points d'ordonnées les fréquences d'obtention d'un résultat favorable.  
 On trace également les droites d'équations :

$$y = p \pm \frac{1}{\sqrt{n}}$$

```

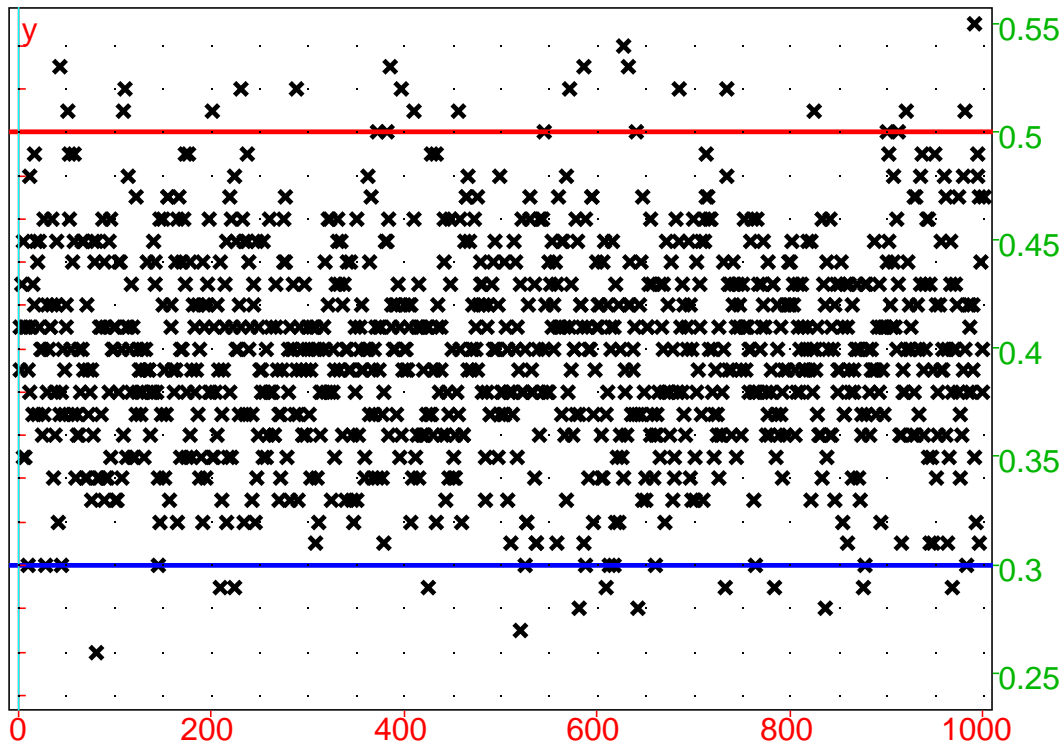
echantillon(n,p,N):={
Freq:=NULL;
pour k de 1 jusque N faire
  S:=0;
  pour j de 1 jusque n faire
    si evalf(hasard(0,1))<p alors S+=1; fsi
  fpour
  Freq:=Freq,[k,evalf(S/n)];
fpour
P:=affichage(nuage_points([Freq]),epaisseur_point_3);
D1:=affichage(droite(y=p-1./sqrt(n)),bleu+epaisseur_ligne_3);
D2:=affichage(droite(y=p+1./sqrt(n)),rouge+epaisseur_ligne_3);
retourne(P,D1,D2)
};
    
```

Programme 346 – intervalle de fluctuation en 2<sup>nde</sup> (XCAS)

Par exemple, pour 1000 tirages d'échantillons de taille  $n = 100$  dans une urne contenant une proportion  $p = 0,4$  de boules rouges :

```

echantillon(100,0.4,1000)
    
```



On observe qu'une immense majorité des fréquences sont entre  $p - \frac{1}{\sqrt{n}}$  et  $p + \frac{1}{\sqrt{n}}$ .  
 Le programme dit qu'au moins 95% des fréquences se trouvent dans cet intervalle. Vérifions-le :

```

echantillon2(n,p,N):={
ok:=0;
    
```

```

pour k de 1 jusque N faire
  S:=0;
  pour j de 1 jusque n faire
    si evalf(hazard(0,1))<p alors S+=1; fsi
  fpour
  si evalf(S/n)>=(p-1./sqrt(n)) et evalf(S/n)<=(p+1./sqrt(n))
    alors ok+=1
  fsi
fpour
retourne ok*100./N
};

```

Programme 347 – intervalle de fluctuation en 2<sup>nde</sup> bis (XCAS)

```
echantillon(100,0.4,10000)
```

95.57

On relance autant de fois que l'on veut le calcul pour observer que le nombre obtenu est supérieur à 95. Nous ne discuterons pas ici de la pertinence d'une telle expérience devant des élèves de 2<sup>nde</sup>...

## B2 Le fameux théorème vu par SAGE

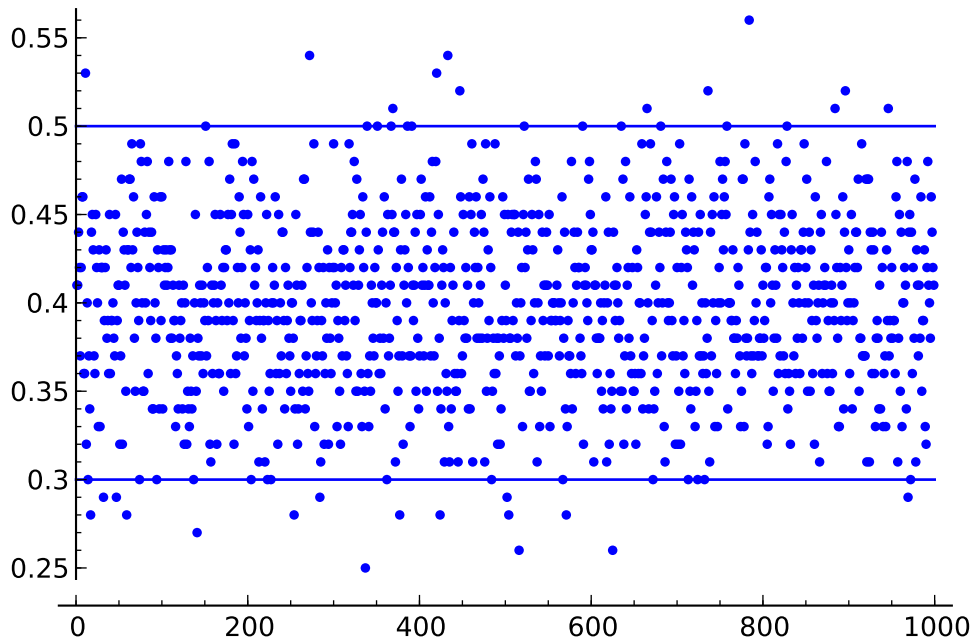
```

def echantillon(n,p,N):
  Freq=[]
  for k in xrange(1,N):
    S=0
    for j in xrange(1,n):
      if random()<p : S+=1
    Freq+=[[k,float(S/n)]]
  P=points(Freq)
  D1=line([[0,p-1./sqrt(n)],[N,p-1./sqrt(n)]])
  D2=line([[0,p+1./sqrt(n)],[N,p+1./sqrt(n)]])
  (P+D1+D2).show()

```

Programme 348 – intervalle de fluctuation en 2<sup>nde</sup> (Sage)

```
sage: echantillon(100,0.4,1000)
```



```
def echantillon2(n,p,N):
    ok=0
    for k in xrange(1,N):
        S=0
        for j in xrange(1,n):
            if random()<p : S+=1
        if (S/n)>=(p-1/sqrt(n)) and (S/n)<=(p+1/sqrt(n)):
            ok+=1
    return ok*100./N
```

Programme 349 – intervalle de fluctuation en 2<sup>de</sup> bis (Sage)

```
sage: echantillon2(100,0.4,10000)
95.77000000000000
sage: echantillon2(100,0.4,10000)
95.79000000000000
```

## B3 Un exemple de TD en 2<sup>de</sup>

### B3a Sondages et urnes

#### 1. Expérience

Une urne contient un très grand nombre de boules dont 40% sont rouges. On en tire un échantillon de taille 100 au hasard. On mesure la fréquence de boules rouges obtenues.

Un résultat classique en statistique affirme que la probabilité que la fréquence observée soit dans l'intervalle  $\left[0,40 - \frac{1}{\sqrt{100}}, 0,40 + \frac{1}{\sqrt{100}}\right]$  est au moins supérieure à une valeur que nous allons essayer de déterminer expérimentalement.

#### 2. Que voulons-nous que l'ordinateur fasse ?

Nous allons généraliser le problème.

Nous allons demander à l'ordinateur d'effectuer un grand nombre  $N$  de tirages d'échantillons de taille  $n$ , sachant que la proportion de boules rouges est  $p$ .

Nous allons représenter le nuage des points d'ordonnées les fréquences de boules rouges obtenues par échantillon.

On trace également les droites d'équations :

$$y = p \pm \frac{1}{\sqrt{n}}$$

### 3. Procédure

Essayez d'analyser la procédure suivante :

```

echantillon(n,p,N) := {
  Freq:=NULL;
  pour k de 1 jusque N faire
    S:=0;
    pour j de 1 jusque n faire
      si evalf(hasard(0,1))<p alors S+=1; fsi
    fpour
    Freq:=Freq, [k, evalf(S/n)];
  fpour
  P:=affichage(nuage_points([Freq]), epaisseur_point_3);
  D1:=affichage(droite(y=p-1./sqrt(n)), bleu+epaisseur_ligne_3);
  D2:=affichage(droite(y=p+1./sqrt(n)), rouge+epaisseur_ligne_3);
  retourne(P, D1, D2)
}::;

```

puis analyser le résultat.

### 4. À vous de jouer

Il faudrait légèrement modifier la procédure précédente pour obtenir le pourcentage de fréquences observées de boules rouges.

### 5. Le théorème

Le résultat suivant sera admis :

Lorsqu'on prélève un échantillon de taille  $n$  dans une population où la fréquence d'un caractère est  $p$ , alors, sous certaines conditions, la probabilité que cet échantillon fournisse une fréquence appartenant à

$$I = \left[ p - \frac{1}{\sqrt{n}}, p + \frac{1}{\sqrt{n}} \right]$$

est **au moins égale** à ..... %

## B 3 b Sondages et urnes : les élections

Vu à la radio : « En perte de vitesse ces derniers mois, la cote du président syldave est remontée de 49% à 52% après son mariage avec la princesse Carlotta de Bordurie qui a su séduire par ses sourires les électeurs de Syldavie. »

1. La véritable cote de popularité du président auprès des 60 millions de Syldave est inconnue. Désignons-la par  $p$ .

On connaît deux fréquences observées sur deux échantillons de 1 000 personnes :  $f_1 = 489\%$  et  $f_2 = 52\%$ .

Montrez que  $f_i \in I_i = \left[ p - \frac{1}{\sqrt{n}}, p + \frac{1}{\sqrt{n}} \right]$  équivaut à  $p \in J_i = \left[ f_i - \frac{1}{\sqrt{n}}, f_i + \frac{1}{\sqrt{n}} \right]$ .

2. Écrivez les intervalles  $J_1$  et  $J_2$  correspondant à  $f_1$  et  $f_2$ . Représentez-les sur une droite graduée. Commentez alors l'annonce faite à la radio.

## B 3 c Sondage personnel

Lu à la syldavision :

Le président syldave a affirmé hier lors de sa conférence de presse que sa cote de popularité est de 52%.

1. Un groupe dissident soutenu par la Bordurie effectue un sondage auprès de 625 personnes et obtient une cote de 47% d'opinions favorables : le président syldave aurait-il menti ?

2. Un autre sondage effectué auprès de 625 autres personnes donne une cote de 49%. Cela permet-il de confirmer la cote de 52% annoncée par le président syldave ?



**B 3 d** Arnaque ?

La princesse Carlotta anime un jeu à la télévision. Elle annonce que dans une urne se trouvent 50 boules noires et 50 boules blanches. Elle offrira une photo dédiée de son mari à toute personne obtenant au moins 11 boules blanches en tirant 30 boules au hasard de l'urne.

L'ambassadeur bordure participe au jeu. Il tire 30 boules et obtient 10 blanches seulement. Furieux, il décide de déclarer la guerre à la Syldavie affirmant que le jeu est truqué. Qu'en pensez-vous ? Parviendrez-vous à éviter la guerre ?

# 17 - Un peu de logique

## A Fonction mystérieuse

Observez ces fonctions et décrivez leur action sachant que `&&` signifie « et » et que `||` signifie « ou » :

```
# let rec truc(n) = n<>1 && ( n=0 || machin(n-1) )
and machin(n) = n<>0 && ( n=1 || truc(n-1) ) ;;
```

Programme 350 – fonction mystérieuse (1)

Pour vous mettre sur la voie, voici trois fonctions plus compliquées mais aux noms plus explicites...

```
# let rec multiple_3(n) = n<>1 && n<>2 && ( n=0 || reste_3_2(n-1) )
and reste_3_2(n) = n<>0 && n<>1 && ( n=2 || reste_3_1(n-1) )
and reste_3_1(n) = n<>0 && n<>2 && ( n=1 || multiple_3(n-1) ) ;;
```

Programme 351 – fonction mystérieuse (2)

# 18 - Problèmes d'optimisation

## A Algorithme glouton et rendu de monnaie

Une première idée pour optimiser une certaine situation est d'effectuer, étape par étape un choix optimum local dans l'espoir que le résultat global obtenu soit optimum : l'algorithme obtenu est appelé *algorithme glouton*.

Une illustration classique de ce problème est le rendu de monnaie qui est lui-même un cas particulier du problème du sac à dos. On dispose d'au moins deux types de pièces de valeurs premières entre elles (pourquoi?... ) et on veut rendre la monnaie en utilisant le moins de pièces possibles.

Une première idée est de répéter le choix de la pièce de plus grande valeur qui ne dépasse pas la somme restante : c'est un algorithme glouton.

```
Entrées : somme n à rendre et liste L des pièces disponibles
Initialisation :
somme partielle ← 0
liste des pièces ← vide
début
  tant que somme partielle < n faire
    k ← taille de la liste
    tant que k > 0 et pièce n° k est supérieure à la somme restante faire
      k ← k - 1 : on regarde la pièce de valeur inférieure
    retourner liste des pièces
fin
```

Algorithme 47 : algorithme glouton de rendu de monnaie

En XCAS, cela donne :

```
piece_glouton(n,L):={
local somme_partielle,liste_pieces,k;
somme_partielle:=0;
liste_pieces:=NULL;
while(somme_partielle<n){
k:=size(L)-1;
while((k>0) and (L[k]>(n-somme_partielle))){
k:=k-1
}
somme_partielle:=somme_partielle+L[k];
liste_pieces:=liste_pieces,L[k];
}
}::;
```

Programme 352 – rendu de monnaie par algorithme glouton

Ainsi, pour savoir comment former 39 centimes avec nos pièces européennes, on entre :

```
piece_glouton(39,[1,2,5,10,20,50,100,200])
```

et on obtient 20,10,5,2,2.

Imaginons maintenant un pays dans lequel on dispose uniquement de pièces de valeur 1, 4 et 5 et que nous voulions rendre 8 unités monétaires. On entre donc :

```
piece_glouton(8, [1, 4, 5])
```

qui répond 5, 1, 1, 1 alors qu'il aurait été plus simple de rendre 2 pièces de 4 unités : **un algorithme glouton n'est pas forcément optimum.**

En fait, l'algorithme glouton est optimum si la suite des valeurs des pièces est *super-croissante* (on peut s'amuser à le démontrer...), c'est-à-dire si chaque terme est supérieur à la somme des précédents.

Par exemple,  $20 > 10 + 5 + 2 + 1$  mais  $5 \leq 4 + 1$  et la deuxième liste n'est pas super-croissante.

Pour une application du problème du sac à dos à la cryptographie, se reporter au paragraphe [C page 207](#)

# 19 - Manipulation de chaînes de caractères

## A Qu'est-ce qu'une chaîne de caractères ?

Une chaîne de caractère est une suite de caractères dont l'ordre est fixé. Par exemple, la chaîne « bonjour » est différente de la chaîne « jourbon ».



### chaîne vide

Une espace n'est pas le vide !

Chaque caractère est en effet associé à un codage binaire, en général sur 8 bits (un octet) ce qui donne  $2^8 = 256$  caractères possibles.

En fait, un américain a mis au point en 1961 l'ASCII (on prononce « aski ») pour « American Standard Code for Information Interchange » qui codait 128 caractères sur 7 bits ce qui est suffisant pour les américains qui n'ont pas de lettres accentuées. Il a ensuite fallu étendre ce code pour les langues comportant des caractères spéciaux et là, les normes diffèrent, ce qui crée des problèmes de compatibilité.

Depuis quelques années, le codage *Unicode* tente de remédier à ce problème en codant tous les caractères existant. C'est le standard utilisé sur les systèmes Linux par exemple.

La plupart des langages de programmation ont une commande qui associe le code ASCII (éventuellement étendu) à un caractère. On pourra donc l'utiliser dans les algorithmes généraux.

Caractère		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	0	1	2	
Code		32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
Caractère	3	4	5	6	7	8	9	:	;	<	=	>	?	@	A	B	C	D	E	
Code	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	
Caractère	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	
Code	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	
Caractère	Y	Z	[	\	]	^	_	'	a	b	c	d	e	f	g	h	i	j	k	
Code	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	
Caractère	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
Code	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	

TABLE 19.1 – Table des caractères ASCII affichables

### Au lycée

L'un des intérêts majeurs de la manipulation de chaînes de caractères au lycée est son application en cryptographie mais un autre domaine très friand de ces chaînes est la génétique : chaîne d'ADN, protéines...

## B Quelques exercices basiques sur les chaînes

- Écrire un algorithme qui transforme un entier entre 20 et 59 en sa transcription littérale.

### B1 Algorithme

```

Entrées : Saisir n : entier entre 20 et 59
début
  si n < 20 ou n > 59 alors
    retourner Il faut entrer un entier entre 20 et 59
  // petite précaution en option...
  unite ← [" ", "-et-un", "-deux", "-trois", "-quatre", "-cinq", "-six", "-sept", "-huit", "-neuf"]
  dizaine ← ["vingt", "trente", "quarante", "cinquante"]
  N ← chaîne(n) // On transforme le nombre n en une chaîne formée de deux caractères
  retourner concaténation(dizaine[N[0]-2], unite[N[1]])
fin

```

Algorithme 48 : écriture littérale d'un entier

Concaténer deux chaînes c'est les mettre bout à bout pour n'en former qu'une seule.

### B2 Traduction XCAS

```

litt(n):={
local unite,dizaine,N;

si (n<20)or(n>59) alors retourne("Il faut entrer un entier entre 20 et 59");fsi;

unite:=[" ", "-et-un", "-deux", "-trois", "-quatre", "-cinq", "-six", "-sept", "-huit", "-neuf"];
dizaine:["vingt", "trente", "quarante", "cinquante"];
N:=string(n);

retourne(dizaine[expr(N[0])-2]+unite[expr(N[1])])
};;

```

Programme 353 – écriture littérale d'un entier

Sur XCAS, la commande `string()` retourne l'expression évaluée sous forme de chaîne de caractères et la commande `expr()` transforme une chaîne en une commande ou un nombre.

```
litt(59)
```

retourne cinquante-neuf.

### B3 Variante

Donner un algorithme qui conjugue à toutes les personnes de l'imparfait du subjonctif un verbe régulier du premier groupe...

- Donner un algorithme qui reçoit un mot non accentué en majuscules et le renvoie en minuscules sauf la première lettre.

**B4** Algorithme

**Entrées** : C : chaîne de caractères

**Initialisation** : c ← chaîne vide

**début**

**pour** j de 1 jusqu'à taille(C)-1 **faire**

    concatène(c, chaîne(code(C[j])+32)

**retourner** (concatène(C[0],c)) // On garde la première lettre en majuscule

**fin**

**Algorithme 49** : mise en minuscules

Une minuscule a pour code ASCII le code de la majuscule correspondant plus 32.

**B5** Traduction XCAS

```
capitale(C):={  
c:="";  
pour j de 1 jusque size(C)-1 faire  
  c:=c+char(asc(C[j])+32);  
fpour;  
retourne(C[0]+c);  
};
```

Programme 354 – mise en minuscules

et capitale(MAMAN) donne bien Maman.

## A Test de croissance

### A1 Niveau

À partir de la seconde.

### A2 Prérequis

Test-Boucles-Procédures-Listes.

### A3 Principe

On entre une liste et on voudrait créer un algorithme qui permette de tester si cette liste est croissante.

### A4 Intérêt

Outre l'aspect technique informatique, on travaille sur la notion de *négation*. On doit comprendre que la négation de l'assertion « la suite est croissante » est « il existe au moins deux termes consécutifs classés dans l'ordre décroissant ».

### A5 Algorithme

Voici un exemple en utilisant une boucle pour et on sort de la boucle dès qu'un contre-exemple apparaît.

```
Entrées : Saisir L : liste
Initialisation : T ← "croissante" // La liste est croissante tant qu'on n'a pas prouvé le contraire
début
  pour k de 0 jusque taille(L)-2 faire
    si L[k]>L[k+1] alors
      T ← "pas croissante" // un contre-exemple suffit
      retourner T // On sort tout de suite de la boucle
  retourner T
fin
```

Algorithme 50 : test de croissance

### A6 Traduction XCAS

On utilise d'abord le code en français.

```
test(L):={
local k,T;
T:="croissante";
pour k de 0 jusque size(L)-2 do
```



```

    si L[k]>L[k+1] alors T:= "pas croissante"; retourne(T); fsi;
fpour;
retourne(T);
};;

```

Programme 355 – croissance d'une liste en impératif

La même chose avec un langage plus standard :

```

test(L):={
local k,T;
T:="croissante";
for(k:=0;k<size(L)-1;k:=k+1){
    if(L[k]>L[k+1]){T:= "pas croissante"; return(T)};
};
return(T);
};;

```

Programme 356 – croissance d'une liste en impératif (VO)

## A7 Version récursive avec CAML

```

# let rec contient_terme_avec_successeur_inferieur (lx)
=
    if lx = []
    then false
    else if List.tl (lx) = []
    then false
    else if List.hd (lx) > List.hd (List.tl (lx))
    then true
    else contient_terme_avec_successeur_inferieur (List.tl (lx))

# let est_sans_decroissance (lx)
=
    not (contient_terme_avec_successeur_inferieur (lx))

```

Programme 357 – croissance d'une liste en récursif (1)

(Merci à René THORAVAL)

ou bien sans recours au module « List »

```

let rec croissante = function
| [] -> true
| x::[] -> true
| x::y::queue -> if x > y
    then false
    else croissante (y::queue);;

```

Programme 358 – croissance d'une liste en récursif (2)

alors on obtient :

```

# croissante([1;2;3;4;5;6;4;7]);;
- : bool = false
# croissante([1;2;3;4;5;6;7]);;
- : bool = true

```

## B Tri à bulles

### B1 Niveau

À partir de la seconde.

### B2 Prérequis

Test-Boucles-Procédures-Listes.

### B3 Principe

On veut trier une liste de nombre par ordre croissant.

### B4 Intérêt

Comprendre le principe d'incrémement. Comprendre le facteur chronologique d'un algorithme : les affectations ne doivent pas être faites dans n'importe quel ordre. Découvrir l'« astuce » informatique consistant à introduire une variable temporaire. Étoffer sa culture scientifique en découvrant les algorithmes de tri.

### B5 Algorithme impératif

```

Entrées : Saisir L : liste
Initialisation : LT ← L
compteur ← 0
début
  pour k de 0 à taille(L) avec un pas de 1 faire
    si LT[k]>LT[k+1] alors
      temp ← LT[k]
      LT[k] ← LT[k+1]
      LT[k+1] ← temp
      k ← -1 // Pour revenir à zéro après incrémement
      compteur ← compteur+1
  retourner LT,compteur
fin

```

Algorithme 51 : tri à bulles

### B6 Version impérative avec XCAS

En français :

```

bulle(L):={
LT:=L;
compteur:=0; /* pour compter les permutations */
pour k de 0 jusque size(LT)-1 faire
  si LT[k]>LT[k+1] alors
    temp:=LT[k]; /* on stocke L[k] */
    LT[k]:=LT[k+1]; /* le deuxieme devient premier */
    LT[k+1]:=temp; /* et vice versa */
    k:=-1; /* on recommence au debut */
    compteur:=compteur+1; /* une permutation de plus */
  fsi;
fpour;

```

```
retourne(LT,compteur);
};;
```

Programme 359 – tri à bulle

Voici le même algorithme en anglais et avec la liste des permutations effectuées :

```
bulle(L):={
LT:=L;
compteur:=0;
T:=LT; /* la liste sans permutation */
for(k:=0;k<size(LT)-1;k:=k+1){
  if(LT[k]>LT[k+1]){
    temp:=LT[k];
    LT[k]:=LT[k+1];
    LT[k+1]:=temp;
    k:=-1;
    compteur:=compteur+1;
    T:=T,LT; /* on rajoute la permutation effectuee dans la liste */
  }
}
return([T]); /* une liste de listes est un tableau (une matrice) */
};;
```

Programme 360 – tri à bulle (VO)

Ce qui permet de visualiser les permutations permettant de passer de [4,3,2,1] à [1,2,3,4].

```
bulle([4,3,2,1])
```

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


### Commandes XCAS

Des algorithmes beaucoup plus efficaces sont déjà implémentés dans XCAS : il s'agit des commandes `sort(liste)` et `SortA(liste)` qui trient dans l'ordre croissant et `SortD(liste)` qui trie dans l'ordre décroissant.

En fait, `sort()` est beaucoup plus puissant car on peut mettre en deuxième argument une fonction de tri (qui doit remplir certaines propriétés).

## B7 Version récursive avec CAML

Nous allons d'abord créer une fonction qui extrait le k-eme opérande d'une liste :

```
# exception Liste_Vide;;

# let rec op = function
| (0,x::liste) -> x
| (k,x::liste) -> op(k-1,liste)
| (_,[]) -> raise Liste_Vide;;
```

Programme 361 – k-eme opérande d'une liste avec CAML

Par exemple :

```
# op(2,[0;10;20;30;40;50]);;
- : int = 20
```

On crée ensuite une fonction qui calcule la longueur d’une liste :

```
# let rec longueur = function
| [] -> 0
| _::queue -> 1+longueur(queue);;
```

Programme 362 – longueur d’une liste avec CAML

Puis une fonction qui concatène deux listes :

```
# let rec concat = function
| [],liste2 -> liste2
| (tete1::queue1),liste2 -> tete1::(concat(queue1,liste2));;
```

Programme 363 – concaténation de deux listes

On crée une fonction qui extrait une sous-liste en indiquant le premier et le dernier opérande :

```
# let rec extrait(liste,debut,fin) =
  if debut=fin then [op(fin,liste)]
  else concat([op(debut,liste)],extrait(liste,debut+1,fin));;
```

Programme 364 – extraction d’une sous-liste avec CAML

Par exemple :

```
# extrait([0;1;2;3;4;5;6;7],2,6);;
- : int list = [2; 3; 4; 5; 6]
```

Ensuite, on crée une fonction qui échange deux opérandes successifs :

```
# let échange(liste,k)=
  if k=0
  then concat([op(1,liste)],concat([op(0,liste)],extrait(liste,2,longueur(liste)-1)))
  else if k=longueur(liste)-2
  then concat(concat(extrait(liste,0,k-1),[op(k+1,liste)]),[op(k,liste)])
  else concat(concat(extrait(liste,0,k-1),[op(k+1,liste)]),concat([op(k,liste)],extrait(
    liste,k+2,longueur(liste)-1)));;
```

Programme 365 – échange de 2 opérandes successifs avec CAML

Il ne reste plus qu’à buller :

```
# let rec bulle(liste,k)=
  if k >= longueur(liste)-1 then liste
  else if op(k,liste) > op(k+1,liste) then bulle(échange(liste,k),0)
  else bulle(liste,k+1);;
```

Programme 366 – tri à bulle avec CAML

Et alors :

```
# bulle([4;3;2;1],0);;
- : int list = [1; 2; 3; 4]
```

## C Tri par insertion

C’est le plus naturel : dans une collection déjà classée, on insère un nouvel élément à sa place. C’est un algorithme naturellement récursif..

## C1 Avec CAML

D'abord on apprend comment insérer un nouvel élément dans une liste :

```
# let rec insere element = function
  | [] -> [element]
  | tete::queue -> if element <= tete
                    then element::tete::queue
                    else tete::(insere element queue);;
```

Programme 367 – tri par insertion

Par exemple, pour insérer 7 dans [1;2;4;5;7;9;13] (qui est déjà classée...) :

```
# insere 7 [1;2;4;5;9;13];;
- : int list = [1; 2; 4; 5; 7; 9; 13]
```

Quand on sait insérer, on prend les éléments 1 par 1 dans la liste et on les insère au reste :

```
# let rec tri_insertion = function
  | [] -> []
  | tete::queue -> insere tete (tri_insertion queue);;
```

## C2 Avec XCAS

```
insere(element,liste):={
if(liste==[])
then{[element]}
else{if(element<=head(liste))
      then{prepend(liste,element)}
      else{prepend(insere(element,tail(liste)),head(liste))}
}
};;
```

Programme 368 – tri par insertion (XCAS)

Puis :

```
tri_insertion(liste):={
if(liste==[])
then{[]}
else{insere(head(liste),tri_insertion(tail(liste)))}
};;
```

## D Tri rapide

### D1 Principe

On choisit un nombre de la liste (le pivot) et on permute les éléments pour avoir les plus petits que le pivot à gauche et les plus grands à droite et on recommence sur chacune des sous-listes : c'est très...récuratif!

### D2 Algorithme récursif avec CAML

Une première fonction qui met les plus petits que le pivot à gauche et les autres à droite :

```
# let rec partition n = function
| [] -> []
| tete::queue -> if tete <= n
                  then tete::(partition n queue)
                  else (partition n queue)@[tete];;
```

Programme 369 – partition d'une liste en récursif

On sépare les petits et les grands dans deux listes à partir d'une liste « partitionnée » :

```
# let rec grand n = function
| [] -> []
| tete::queue -> if tete > n then tete::queue
                  else grand n queue;;

# let rec petit n = function
| [] -> []
| tete::queue -> if tete > n then []
                  else tete::(petit n queue);;
```

Programme 370 – partition d'une liste en récursif (suite)

On résume :

```
# let grande_partition n liste = grand n (partition n liste);;
# let petite_partition n liste = petit n (partition n liste);;
```

Par exemple :

```
# grand_partition 5 [5;2;45;7;4;5;8;2;1;5;3;4;5;7;9;9;6;5;3;8;2;4;5;2];;
- : int list = [8; 6; 9; 9; 7; 8; 7; 45]
# petite_partition 5 [5;2;45;7;4;5;8;2;1;5;3;4;5;7;9;9;6;5;3;8;2;4;5;2];;
- : int list = [5; 2; 4; 5; 2; 1; 5; 3; 4; 5; 5; 3; 2; 4; 5; 2]
```

et pour finir, le tri rapide lui-même :

```
# let rec rapide = function
| [] -> []
| pivot::reste -> rapide (petite_partition pivot reste) @ [pivot] @ rapide (grande_partition
pivot reste);;
```

Programme 371 – tri rapide en récursif

alors :

```
# rapide [5;2;45;7;4;5;8;2;1;5;3;4;5;7;9;9;6;5;3;8;2;4;5;2];;
- : int list =
[1; 2; 2; 2; 2; 3; 3; 4; 4; 4; 5; 5; 5; 5; 5; 5; 6; 7; 7; 8; 8; 9; 9; 45]
```

Trop fortement mathématique!!...

## A Codage de César

Le codage de César, on connaît. On va créer un algorithme qui prend une chaîne de caractère et une clé de « décalage » en argument.

### A1 En récursif avec CAML

La commande `int_of_char` renvoie le code ASCII d'un caractère (non accentué : A comme *american...*).

Il y a 95 caractères qui commencent à 32 et finissent à 126 (les autres codes correspondent à des touches d'action comme le « retour chariot »).

Nous allons donc ajouter la clé au code des caractères modulo 95 mais il faudra avant enlever 32 puis le rajouter à la fin pour travailler avec des caractères et non pas des actions.

Travailler avec des chaînes de caractères est moins pratique que la manipulation des listes. Nous pouvons créer une fonction « queue » qui donne la chaîne de caractère privée de sa « tête » et la donner toute faite aux élèves car elle n'est pas intéressante à construire en elle-même :

```
# let queue(chaine) = String.sub chaine 1 (String.length(chaine)-1);;
```

Programme 372 – queue d'un chaîne

On utilise `String.sub chaine debut longueur` qui remplace la chaîne `chaine` par la sous-chaîne extraite à partir du rang `debut` et de longueur `longueur`.

Ensuite, on a besoin de transformer le caractère (type `char`) renvoyé par `char_of_int` pour en faire une chaîne de caractère (type `string`). On utilise `String.make longueur caractere` qui fabrique une chaîne de longueur `longueur` remplie du caractère `caractere`.

Pour notre situation, cela donne :

```
# let code(car,cle) =  
  String.make 1 (char_of_int(((int_of_char(car)-32+cle) mod 95)+32));;
```

Programme 373 – codage numérique d'une chaîne

Enfin, voici le programme principal. On utilise `^` qui concatène des chaînes et `mot.[0]` qui renvoie le caractère numéro 0 de la chaîne `mot` :

```
# let rec cesar(mot,cle) =  
  if mot="" then ""  
  else (code(mot.[0],cle))^cesar(queue(mot),cle);;
```

Programme 374 – codage de César en récursif

On code avec la clé 3 :

```
# cesar("Essayons de coder cette phrase de plus 8 mots !",3);;  
- : string = "Hvvd|rqv#gh#frghu#fhwwh#skudvh#gh#soxv#;#prwv#"
```

On décode avec la clé -3 :

```
# cesar("Hvvd|rqv#gh#frghu#fhwwh#skudvh#gh#soxv#;#prvv#
",-3);;-: string = "Essayonsdecodercettephrasedeplus8mots!"
```

## A2 En impératif avec XCAS

Voici le code proposé par Bernard Parisse dans l'aide de XCAS :

```
code:= (c)->{ return(asc(c)-32) };;
decode:= (k)->{return(char(k+32)) };;
jules_cesar:= (message,cle)->
{ local s,j,messcode;
  s:=size(message);
  messcode:="";
  for (j:=0;j<s;j:=j+1) {
    messcode:=append(messcode,decode(irem(cle+code(message[j]),95)));
  };
  return(messcode);
};;
```

Programme 375 – codage de César en impératif

## B RSA

Nos amis Ronald Rivest, Adi Shamir et Leonard Adleman mirent au point en 1977 un système de codage alors qu'ils essayaient au départ de montrer que ce qu'ils allaient développer étaient une impossibilité logique : aah, la beauté de la recherche... Et savez-vous quel est la base du système?... le petit théorème de Fermat, si cher au programme de spé maths en TS! Mais replaçons dans son contexte les résultats de R, S et A. Jusqu'il y a une trentaine d'années, la cryptographie était l'apanage des militaires et des diplomates. Depuis, les banquiers, les mega business men et women, les consommateurs internautes, les barons de la drogue et j'en passe ont de plus en plus recours aux messages codés. Oui, et alors? Le problème, c'est que jusqu'ici, l'envoyeur et le destinataire partageaient une même clé secrète qu'il fallait faire voyager à l'insu de tous : les états et l'armée ont la valise diplomatique, mais les autres?

C'est ici que Whitfield Diffie et Martin Hellman apparaissent avec leur idée de principe qu'on peut résumer ainsi : votre amie Josette veut recevoir de vous une lettre d'amour mais elle a peur que le facteur l'intercepte et la lise. Elle fabrique donc dans son petit atelier une clé et un cadenas. Elle vous envoie le cadenas, ouvert mais sans la clé, par la poste, donc à la merci du facteur : le cadenas est appelé *clé publique*. Vous recevez le cadenas et l'utilisez pour fermer une boîte contenant votre lettre : le facteur ne peut pas l'ouvrir car il n'a pas la clé. Josette reçoit donc la boîte fermée : elle utilise sa clé, qu'elle seule possède, pour ouvrir la boîte et se pâmer devant vos élans épistolaires.

En termes plus mathématiques, cela donne : je fabrique une fonction  $\pi$  définie sur  $\mathbb{N}$  qui possède une réciproque  $\sigma$ . On suppose qu'on peut fabriquer de telles fonctions mais que si l'on ne connaît que  $\pi$ , il est (quasiment) impossible de retrouver  $\sigma$ . La fonction  $\pi$  est donc la clé publique : vous envoyez  $\pi(\text{message})$  à Josette. Celle-ci calcule  $\sigma(\pi(\text{message})) = \text{message}$ . Josette est la seule à pouvoir décoder car elle seule connaît  $\sigma$ .

Tout ceci était très beau en théorie, mais Diffie et Hellman n'arrivèrent pas à proposer de telles fonctions. Mais voici qu'arrivent Ronald, Adi et Leonard...

## B1 En impératif avec XCAS

Nous aurons tout d'abord besoin d'un programme code qui transforme un message écrit en majuscule en un nombre entier et un programme decode qui effectuera la démarche inverse.

On choisit les majuscules pour être sûrs d'avoir des codes ASCII de 2 chiffres pour pouvoir décoder...

```
code(c):={
  local L,C,k;
  L:=asc(c);
```



```
C:="";
for(k:=0;k<size(L);k:=k+1){
  C:=cat(C,string(L[k]));
}
return(expr(C))
};;
```

Programme 376 – codage numérique d'une chaîne en impératif

On a adapté un peu ce qui a été fait pour César mais cette fois on veut transformer la liste des codes en un nombre entier en concaténant les chiffres qui la composent.

La commande `string(expression)` transforme n'importe quelle expression en une chaîne de caractère.

La commande `cat(chaîne, caractère)` concatène caractère à droite de la chaîne.

La commande `expr(chaîne)` est la réciproque de `string`.

Passons à `decode` qui transforme un entier en une chaîne de caractère :

```
decode(n):={
local L,C,k;
L:=string(n);
C:="";
for(k:=0;k<size(L)-1;k:=k+2){
  C:=cat(C,char(expr(cat(L[k],L[k+1]))))
}
return(C)
};;
```

Programme 377 – décodage d'un entier en chaîne de caractères

Par exemple :

```
code("I LOVE YOU")
```

73327679866932897985

Vérifiez que `decode` décode bien :

```
decode(code("I LOVE YOU"))
```

Ceci étant dit, je vais vous montrer comment quelqu'un peut m'envoyer un message crypté en utilisant le protocole RSA, que personne ne peut comprendre sauf moi, puis comment je vais décrypter ce message.

Avant de m'envoyer un message, on doit connaître ma *clé publique*, connue de tous et constituée d'un couple de nombres  $(n, e)$ .

Et avant de connaître cette clé, il faut que je la fabrique - secrètement - de la manière suivante :

- je commence par fabriquer un nombre premier quelconque mais suffisamment grand. J'utilise `nextprime(n)` qui renvoie le plus petit entier premier (ou supposé tel...) supérieur à  $n$ . Par exemple :

```
p:=nextprime(857452321345678945615348675153785313513545313135435153) ;;
```

- puis un autre

```
q:=nextprime(9574138786465743137483136984548364156838461638468343648634) ;;
```

Ces deux nombres ne sont connus que de moi !

- je forme le produit de ces deux nombres

```
n:=p*q ;;
```

Ce nombre  $n$  est mon module public. Il reste à former le nombre  $\phi$  qui est ma puissance de cryptage publique. Là encore, cela se fait par étapes.

- je commence par former le nombre  $\phi_n$  (tiens, tiens...) de la manière suivante :

```
phi_n:=(p-1)*(q-1);;
```

- c'est là qu'il faut jouer avec la chance : je dois choisir un nombre `pui` premier avec `phi_n`. En pratique, je choisis un grand nombre premier au hasard :

```
pui:=nextprime(4568531538443156358743168741353);;
```

Je vérifions quand même que `pui` et `phi_n` sont premiers entre eux :

```
gcd(pui, phi_n);
```

1

Ouf!...Au fait, pouvais-je en être sûr?...

- il me reste à calculer ma puissance de décryptage secrète : je l'obtiens en cherchant l'inverse de `pui` modulo `phi_n`, c'est-à-dire en déterminant un coefficient de Bézout grâce à l'algorithme d'Euclide étendu.

Le problème, c'est que nos nombres sont un peu grands pour faire l'algorithme à la main, mais...heureusement, il y a XCAS. Il existe en effet une fonction `igcdex(x, y)` qui renvoie le PGCD de  $x$  et  $y$ , ainsi que des coefficients de Bézout  $u$  et  $v$  qui vérifient donc  $ux + vy = x \wedge y$  :

```
igcdex(pui, phi_n)
```

Ceci veut dire que  $upui + vphi_n = 1$ , avec  $u$  le premier nombre renvoyé par XCAS et  $v$  le deuxième. Ainsi  $upui \equiv 1 [phi_n]$  mais  $u$  n'est pas forcément l'inverse de `pui` modulo `phi_n` car il n'appartient pas forcément à l'intervalle entier  $[[0, phi_n - 1]]$ . Il suffit de calculer sa classe modulo `phi_n` c'est-à-dire le reste de sa division par `phi_n` :

```
d:=irem(ans()[0], phi_n);;
```

Je suis maintenant prêt à recevoir des messages cryptés selon le protocole RSA et vous êtes prêt(e) à m'en envoyer car j'ai rendu publique ma clé (`pui, n`).

- d'abord vous « numérisez » votre message.

```
text_num:=code("ESSAYE DONC DE DECRYPTER CE MESSAGE");;
```

- comme nous allons calculer modulo `n`, il faut vérifier que `text_num` est plus petit que `n`, en utilisant par exemple `lengtha`.

```
text_num < n
```

- ensuite, vous calculez ce nombre puissance `pui` et vous regardez quelle est sa classe modulo `n`. On utilise `powermod(a, m, p)` qui calcule la classe de  $a^m$  modulo  $p$  :

```
crypte:=powermod(text_num, pui, n);;
```

- je reçois ce message et je le décrypte grâce à ma clé secrète  $d$  que je suis le seul à connaître dans tout l'univers. Je calcule `crypted[n]` :

```
decrypte:=powermod(crypte, d, n);;
```

- il ne me reste plus qu'à retrouver mon texte original grâce à la procédure `decode` :

```
decode(decrypte)
```

ESSAYE DONC DE DECRYPTER CE MESSAGE

Ça marche!!

Il reste à prouver notre méthode (indicateur d'Euler, théorème d'Euler, etc.), à voir si on peut casser le code. Au fait, on a failli parler du RSA en classe de Seconde...

## C Cryptosystème du sac à dos

<sup>a</sup>. Et qu'est-ce qu'on peut faire si `text_num` est trop grand ?

# 22 - Graphes

# 23 - Suites définies par une relation de récurrence

## A Suite de Fibonacci

Léonard de Pise, fils de Bonacci, s'est intéressé en 1202 au fameux problème des lapins : un couple donne naissance, à partir du deuxième mois à un couple chaque mois, les nouveaux couples suivant la même loi de reproduction ; combien y aura-t-il de lapins (supposés immortels...) au bout de  $n$  mois ?

Il s'agit donc d'étudier la suite  $(F_n)$  définie par :

$$F_0 = F_1 = 1 \quad F_{n+2} = F_n + F_{n+1}$$

### A1 VersionS récursives

On peut écrire la (double!) récursion directement :

```
# let rec fibo(n)=  
  if n<2  
    then 1  
    else fibo(n-1)+fibo(n-2);;
```

Programme 378 – suite de Fibonacci en récursif (1)

```
fibo(n):={  
  if(n<2)  
    then{1}  
    else{fibo(n-1)+fibo(n-2)}  
};;
```

Programme 379 – suite de Fibonacci en récursif (1) avec XCAS

Très facile....mais peu efficace car il faut une vingtaine de secondes à XCAS pour calculer  $\text{fibo}(30)$  (une seconde avec CAML). En effet, la récursion est double et nécessite une pile bien plus importante.

Mieux vaut recourir à une fonction auxiliaire pour avoir une récursion terminale :

```
# let fibr(n)=  
  let rec fkplus2(n, fk, fkplus1)=  
    if n<2 then fkplus1  
    else fkplus2(n-1, fkplus1, fk+.fkplus1)  
  in fkplus2(n, 1., 1.);;
```

Programme 380 – suite de Fibonacci avec une récursion terminale

Nous avons de plus choisi de travailler avec des flottants pour pouvoir parler ensuite du Nombre d'Or.

Voici un autre moyen utilisant l'itération d'une fonction de deux variables.

```
# let rec iter n f x =  
  if n=0 then x  
  else f(iter (n-1) f x);;
```

## Programme 381 – itération polymorphe d’une fonction

On crée une fonction qui renvoie la première composante d’un couple :

```
# let projx (x,y)=x;;
```

Enfin, sachant que  $(u_{n+2}, u_{n+1}) = f(u_{n+1}, f(u_n))$  avec  $f(x, y) = (x + y, x)$  et  $(u_1, u_0) = (1, 0)$  définit la suite de Fibonacci, on obtient :

```
# let fib3(n)=
  projx( iter n (fun(x,y)->(x+.y,x)) (1.,0.) );;
```

## Programme 382 – suite de Fibonacci : troisième version

Alors :

```
# fib3(1000);;
- : float = 7.03303677114227653e+208
```

## A2 Version impérative

```
fibi(n):={
local fk, fkplus1, k;
if(n<2)
then{return(1)}
else{
fk:=1;
fkplus1:=1;
for(k:=1;k<n;k:=k+1){
fkplus1:=fk+fkplus1;
fk:=fkplus1-fk;
}
return(fkplus1)
}
};;
```

## Programme 383 – suite de Fibonacci en impératif

Alors par exemple :

```
seq(fibi(j), j=0..10)
```

donne les onze premiers termes de la suite :

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

## A3 Suite des quotients

L’étude de la suite  $q_n = \frac{F_{n+1}}{F_n}$  permet d’obtenir une approximation du nombre d’or  $\varphi$ . En effet :

$$q_n = \frac{F_{n+1}}{F_n} = \frac{F_n + F_{n-1}}{F_n} = 1 + \frac{1}{q_{n-1}}$$

Après avoir montré que la suite  $(q_n)$  converge, sa limite vérifie  $\ell = 1 + \frac{1}{\ell}$  et donc  $\ell = \varphi$ .

Version récursive avec CAML :

```
# let rec nbdor(n)=
  if n=0
    then 1.
    else 1.+1./nbdor(n-1);;
```

Programme 384 – quotients des termes d'une suite de Fibonacci en récursif

Notez que la récursion est terminale en utilisant la nouvelle relation de récurrence définissant  $(q_n)$ . Pour avoir des approximations, il ne faut pas oublier de travailler avec des flottants.

Version impérative avec XCAS :

```
nbdor(n):={
  local fk,fkplus1,k,qk;
  if(n==0)then{return(1.0)}
  else{
    fk:=1.0;
    fkplus1:=1.0;
    for(k:=0;k<=n;k:=k+1){
      qk:=fkplus1/fk;
      fkplus1:=fk+fkplus1;
      fk:=fkplus1-fk;
    }
    return(qk)
  }
};;
```

Programme 385 – quotients des termes d'une suite de Fibonacci en impératif (1)

ou plus simplement :

```
nbdorbis(n):={
  local k,qk;
  qk:=1.0;
  if(n==0)then{return(qk)}
  else{
    for(k:=1;k<=n;k:=k+1){
      qk:=1+1/qk;
    }
    return(qk)
  }
};;
```

Programme 386 – quotients des termes d'une suite de Fibonacci en impératif (2)

## B Suite de Syracuse

On ne la présente plus....

Avec CAML, on demande d'afficher tous les termes avec une espace entre chaque jusqu'à obtenir 1 :

```
# let rec syracuse(n)=
  print_int(n);
  print_string(" ");
  if n>1 then syracuse(if n mod 2=0 then n/2 else 3*n+1);;
```

Programme 387 – suite de Syracuse en récursif

Pour 187514 :

```
# syracuse(187514);;
```

on obtient :

```
187514 93757 281272 140636 70318 35159 105478 52739 158218 79109 237328 118664 59332 29666 14833
44500 22250 11125 33376 16688 8344 4172 2086 1043 3130 1565 4696 2348 1174 587 1762 881 2644
1322 661 1984 992 496 248 124 62 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274
137 412 206 103 310 155 466 233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336 668 334
167 502 251 754 377 1132 566 283 850 425 1276 638 319 958 479 1438 719 2158 1079 3238 1619
4858 2429 7288 3644 1822 911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866
433 1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1 - :
unit = ()
```

Avec XCAS en impératif :

```
syracuse(n):={
S:=n;
N:=n;
while(N>1){
  N:=if(irem(N,2)==0)then{N/2}else{3*N+1}
  S:=S,N;
}
return(S)
};;
```

Programme 388 – suite de Syracuse en impératif

## C Encore des approximations de $\pi$

### C1 Suite de Viète

François VIÈTE proposa au XVI<sup>e</sup> siècle une approximation de  $\pi$  qui revient en langage moderne à étudier la suite :

$$u_0 = \frac{1}{\sqrt{2}} \quad u_n = \sqrt{\frac{1}{2} + \frac{1}{2}u_{n-1}}$$

Alors

$$\pi = \frac{2}{\prod_{n=0}^{+\infty} u_n}$$

```
# let rec viete(n)=
  if n=0 then 1./sqrt(2.)
  else sqrt(0.5*(1.+viete(n-1)));;

# let rec produit_viete(n)=
  if n=0 then viete(0)
  else viete(n)*.produit_viete(n-1);;

# let pi_viete(n)=2./produit_viete(n);;
```

Programme 389 – suite de Viète en récursif

Alors :

```
# pi_viete(20);;
- : float = 3.1415926535895
```

**C1 a** En impératif avec XCAS

```

viete(n):={
local u,P,k;
u:=1/sqrt(2.);
P:=u;
for(k:=1;k<=n;k:=k+1){
u:=sqrt(0.5+0.5*u)
P:=P*u
}
return(2/P)
};

```

Programme 390 – suite de Viète en impératif

**C2** Euler et la suite des inverses des carrés

Chacun connaît la formule :

$$\sum_{n=1}^{+\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

**C2 a** En récursif avec CAML

```

# let rec somme_euler(n)=
  if n=1 then 1.
  else 1./.(float_of_int(n)**2. +. somme_euler(n-1));;

# let pi_euler(n)=sqrt(6.*.somme_euler(n));;

```

Programme 391 – suite des inverses des carrés en récursif

La convergence n'est pas très rapide...

```

# pi_euler(20000);;
- : float = 3.14154490793769403

```

**C2 b** En impératif avec XCAS

```

pi_euler(n):={
local S,k;
S:=0.;
for(k:=1;k<=n;k:=k+1){
S:=S+1/k^2;
}
return(sqrt(6*S))
};

```

Programme 392 – suite des inverses des carrés en impératif



## D Suites logistiques

### D1 L'effet papillon

Vers 1950, le météorologue américain Edward LORENZ étudie la circulation atmosphérique. Reprenant les travaux des mathématiciens français HADAMARD et POINCARÉ sur l'imprédictibilité des systèmes simples. Même en modélisant l'atmosphère en un système ayant une dizaine de degrés de liberté, il montre la sensibilité exponentielle par rapport aux conditions initiales. Il illustre son propos en montrant que le battement d'aile d'un papillon en un point de la Terre est suffisant pour déclencher deux semaines plus tard une tornade à l'autre bout de la planète. Au même moment, le russe Vladimir ARNOLD, disciple de KOLMOGOROV, modélise le mouvement d'un fluide parfait comme celui d'un point sur une « hypersurface » de dimension infinie. Il part lui aussi de travaux d'HADAMARD sur les trajectoires d'une bille sur certaines surface : par exemple, si on lâche une bille sur un paraboloïde hyperbolique (une selle de cheval), sa trajectoire va dépendre énormément de sa position de départ, à tel point qu'il s'avère impossible de prévoir sa trajectoire au-delà d'un temps très court.

Bref, l'effet papillon envahit les esprits et nombres de disciplines pour imposer une vision chaotique d'un monde complexe et imprédictible. C'est ce phénomène que nous allons observer et commenter aujourd'hui, non pas en étudiant le domaine des turbulences atmosphériques « physiquement » hors de notre portée, mais en explorant, de manière très simplifiée, la *dynamique des populations*.

Nous évoquerons en conclusion les apports récents de la *mécanique statistique* qui replace dans leur contexte les travaux de LORENZ et ARNOLD.

### D2 Le Zrājdz

Comme vous le savez tous, le Zrājdz commun à ailette mouchetée est l'animal emblématique de la Syldavie. Aussi paisible que les habitants de ce bucolique pays, le Zrājdz se nourrit exclusivement des baies du bleurtschzrn, arbre qui pousse en abondance dans les verts sous-bois syldaves. Si l'on ne considérait que cette idéale situation, la population  $u$  de Zrājdzs suivrait la loi suivante

$$u_{n+1} = Ru_n$$

Cette relation traduit le fait que la population de l'année  $n + 1$  est proportionnelle à l'année  $n$  : on applique à  $u_n$  le taux de natalité et le taux de mortalité. Le coefficient  $R$  résume ces proportions.

On prend  $R = 1,5$  et on suppose qu'il y avait 10 000 couples de Zrājdzs au départ de notre étude. On exprimera la population en centaine de milliers d'individus.

La suite étant définie par récurrence, la première idée est de la définir de la même manière sur XCAS :

```
u(n):={
  si n=0 alors 0.2;
  sinon R*u(n-1);
  fsi;
};
```

On fixe  $R$  :

```
R:=1.5;
```

On demande par exemple les 15 premiers termes de la suite :

```
seq(u(k),k=0..15);
```

et on trace la représentation graphique associée :

```
nuage_points([seq([n,u(n)],n=0..15)]);
```

## E Trop de Zrājdzs nuit aux Zrājdzs

Il est assez aisé d'objecter au modèle précédent que l'évolution ne peut pas rester proportionnelle à la population de l'année précédente : au bout d'un moment la nourriture et l'espace vital, par exemple, viennent à manquer. On peut alors modéliser l'évolution de la population selon la loi

$$u_{n+1} = R u_n (1 - u_n)$$

1. Définissez  $u(n)$  comme cela a été fait au paragraphe précédent et calculez les 25 premiers termes de la suite. On prendra encore  $R = 1,5$  et  $u_0 = 0,2$ .

```
u(n):={
  si n=0 alors 0.2;
  sinon R*u(n-1)*(1-u(n-1));
  fsi;
}::;
```

On préférera sur XCAS une procédure impérative :

```
u(n):={
  local u;
  u:=0.2;
  pour k de 1 jusque n faire
    u:=R*u*(1-u);
  fpour;
  retourne(u)
}::;
```

2. Tracez également la graphique donnant la population en fonction de l'année. On observera le phénomène sur une cinquantaine d'années.

```
nuage_points([seq([n,u(n)],n=0..50)]);
```

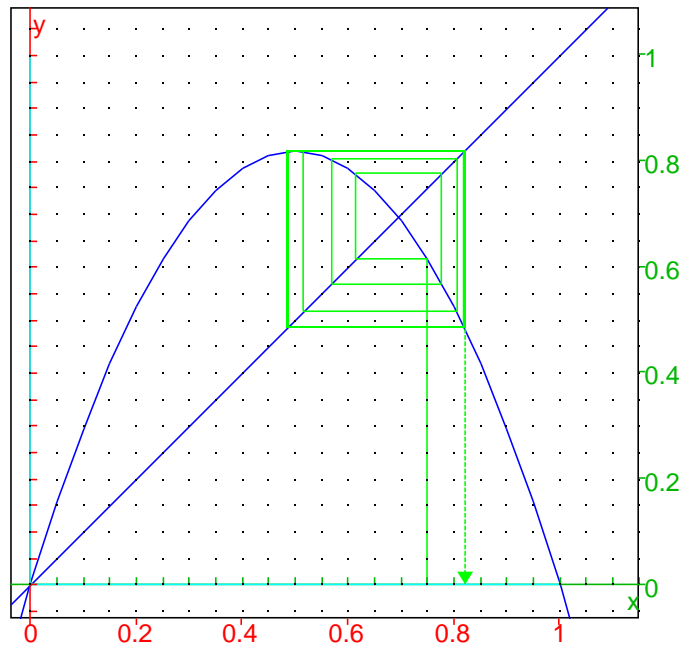
3. Observez maintenant ce qui se passe pour ces graphiques quand on fait varier le coefficient  $R$ . On regardera en particulier les graphiques correspondant à  $R=2,0$ ;  $R=0,9$ ;  $R=3,44$ ;  $R=3,5$ ;  $R=3,544$ ;  $R=3,831874048$ .
4. Ce n'est pas tout! Regardons maintenant ce qui se passe pour une même valeur de  $R$  lorsqu'on fait varier de manière infime  $u(0)$ . On prendra par exemple  $R = 3,831874048$  et  $u(0)$  successivement égal à  $0,2$  puis à  $0,20001$ . C'est l'illustration du fameux « effet papillon » : un battement d'aile de papillon à Rezé est susceptible de déclencher beaucoup plus tard une tornade au Texas. La très faible perturbation créée par le vol d'un papillon pourrait en principe faire varier les conditions initiales du système atmosphérique et engendrer plus tard un changement climatique en un endroit quelconque de notre planète.
5. On peut avoir une vision plus spectaculaire avec l'habituelle représentation des suites définies par une relation de récurrence.

On rajoute un curseur pour faire varier  $R$ .

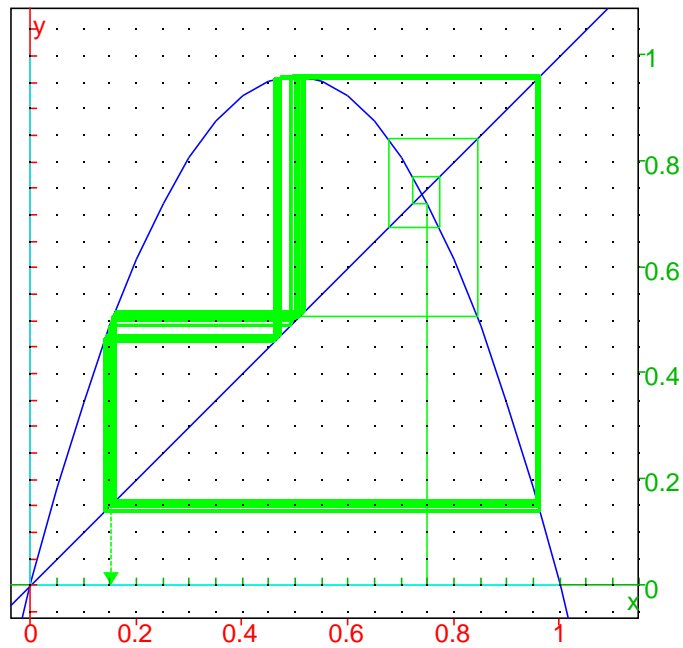
On ouvre une fenêtre de géométrie en tapant  + .

```
R:=element(0 .. 4,0.01) /* R varie de 0 à 4 avec un pas de 0,01 */
f:=x->R*x*(1-x);
graphe_suite(f(x),0.75,2500) /* 2500 termes représentés avec uo=0,75 */
```

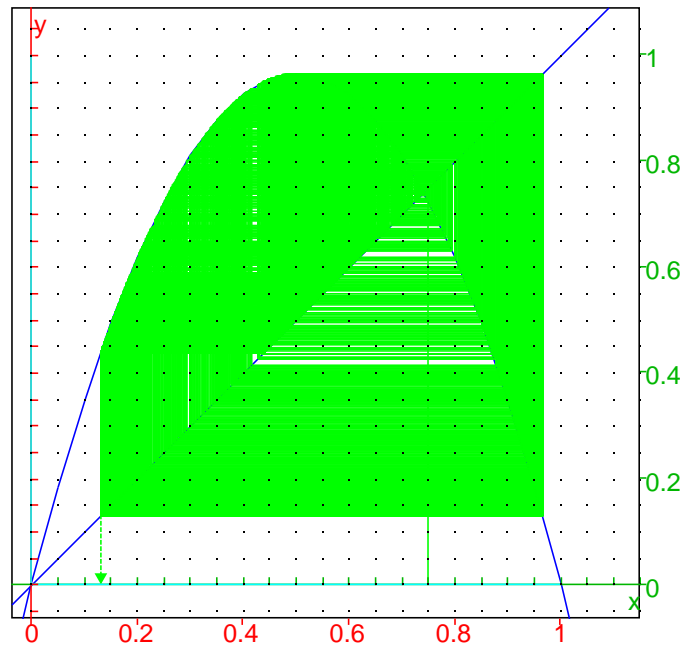
Pour  $R = 3,28$  :



Pour  $R = 3,85$  :



Pour  $R = 3,86$  :



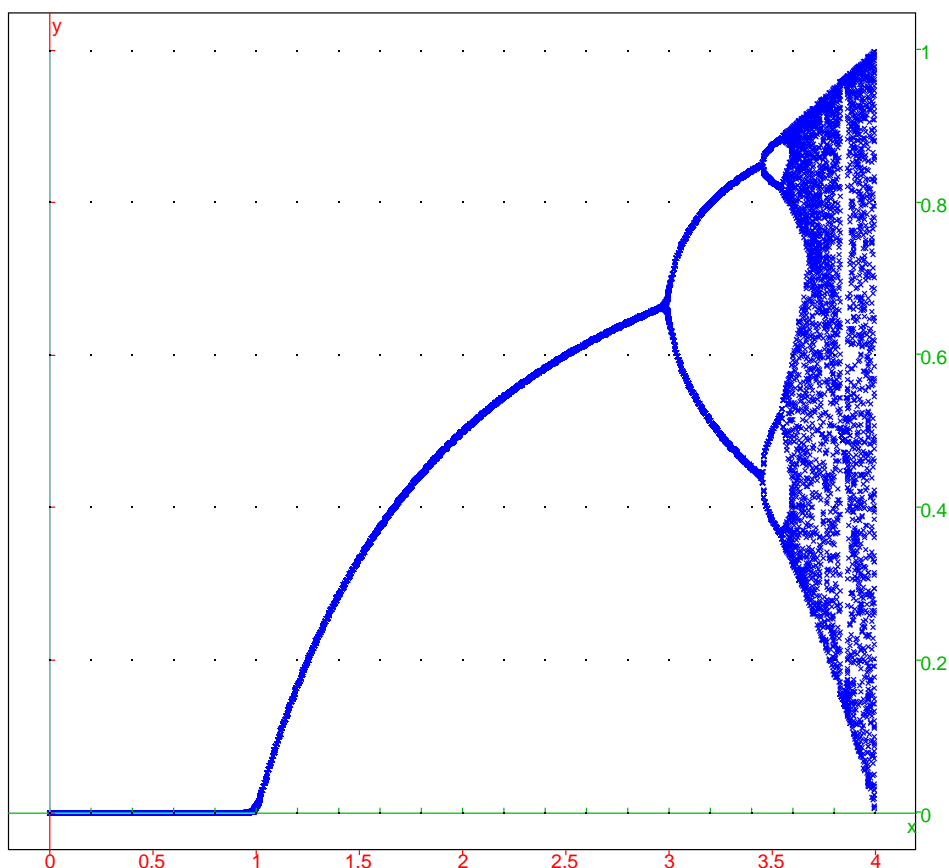
6. Ce phénomène est mis plus clairement en évidence par un **diagramme de bifurcation**. Cette fois, le travail est inversé : je vous livre le programme et vous essayez de le commenter et de découvrir à quoi correspond le graphique obtenu. Qu'est-ce que représente  $r$  ?  $res$  ?  $ligne$  ?  $tmp$  ?

```

Feig(R,N):={
  local k,r,tmp,ligne;
  tmp:=floor(R*N);
  /*comme on avance de pas=1/N, il y aura R*N points tracés */
  res:=[0tmp];/* il y aura 20 points par ligne car on fait varier k de 100 à 120 */ ligne := [020];
  for (r:=0;r<R;r+=1/N){
    tmp:=0.2;
    /* calcul de u100 */
    for (k:=0;k<100;k++){
      tmp:=r*tmp*(1-tmp);
    }
    /* calcul des suivants et stockage */
    for (k:=0;k<20;k++){
      ligne[k] := point(r,tmp);
      tmp:=r*tmp*(1-tmp);
    }
    res[r*N] =< ligne;
  }
  return res;
};

```

Ensuite, observez ce qui se passe pour  $0 \leq R < 3$ ,  $3 \leq R < 3,45$ ,  $3,45 \leq R < 3,57$ ,  $3,57 \leq R \leq 4$ .



## F Exposant de Lyapounov

Pour mesurer la sensibilité d'un système dynamique aux conditions initiales, on mesure l'exposant de LYAPOUNOV introduit par le mathématicien russe Alexandre LYAPOUNOV à la fin du XIX<sup>e</sup> siècle.

On considère une suite définie par la relation  $u_{n+1} = f(u_n)$ . Quelle est l'influence d'un écart  $e_0$  sur  $u_0$  pour la suite des itérés ? Après une itération, l'écart absolu vérifie  $|e_1| = |f(u_0 + e_0) - f(u_0)|$  et l'écart relatif vaut  $\frac{|e_1|}{|e_0|} = \frac{|f(u_0 + e_0) - f(u_0)|}{|e_0|} \approx |f'(u_0)|$  pour  $|e_0|$  suffisamment petit.

Après  $n$  itérations, l'écart relatif vaut

$$\frac{|e_n|}{|e_0|} = \frac{|e_1|}{|e_0|} \times \frac{|e_2|}{|e_1|} \times \dots \times \frac{|e_n|}{|e_{n-1}|} = \prod_{k=1}^n |f'(u_{k-1})|$$

Si un des écarts devient nul, notre étude est sans intérêt. Comme effectuer un produit est une opération coûteuse, informatiquement parlant, nous allons pouvoir considérer le logarithme de ce produit.

Notre problème est de savoir si les écarts s'amplifient et donc le produit est supérieur à 1, ou bien si le système est stable et donc le produit est inférieur à 1.

$$\prod_{k=1}^n |f'(u_{k-1})| < 1 \iff \sum_{k=1}^n \ln |f'(u_{k-1})| < 0$$

Pour relativiser le rôle du choix de  $n$ , nous allons normer cette somme en la divisant par  $n$ .

On définit alors l'exposant de LYAPOUNOV :

$$\lambda = \lim_{n \rightarrow +\infty} \frac{1}{n} \sum_{k=1}^n \ln |f'(u_{k-1})|$$

Écrivez une procédure `lyapounov(u0, R, n)` qui calcule une approximation numérique de l'exposant de Lyapounov qui dépend de la donnée de  $u_0$ ,  $R$  et le nombre  $n$  d'itérations.

*Solution :*

```

lyapounov(u0,R,n)={
local u,lyap,derivee,tmp,k;
u:=u0;
lyap:=0;
pour k de 1 jusque n faire
  derivee:=evalf(R-2*R*u);
  si derivee==0
    alors retourne(-infinity);
    sinon lyap:=lyap+ln(abs(derivee));
  fsi;
u:=R*u*(1-u);
fpour;
retourne(lyap/n)
};

```

Programme 393 – Exposant de Lyapounov

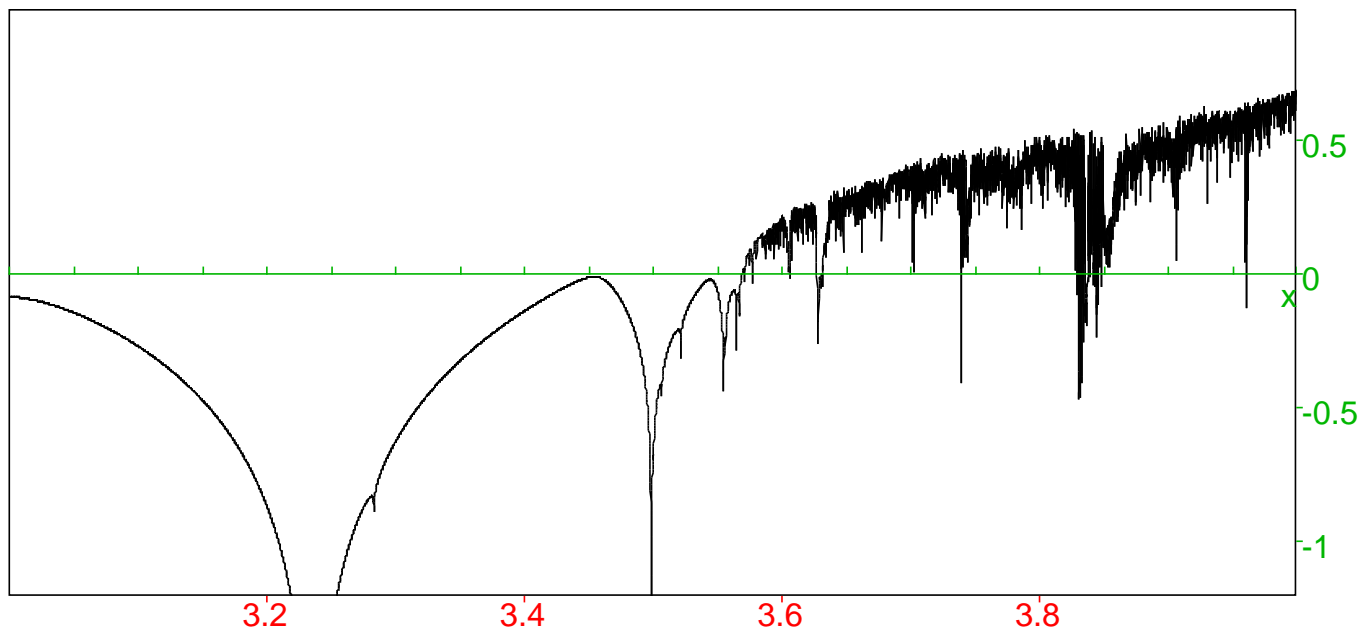
Ensuite, vous l'utiliserez pour représenter l'exposant en fonction de  $R$ .

Par exemple, avec  $u_0 = 0,45$  et 50 itérations :

```

ligne_polygone([seq([r*0.001,lyapounov(0.45,r*0.001,50)],r=3000..4000)]);

```



Comment interpréter ce graphique ?

# 24 - Quelques particularités de CAML...

## A Toutes sortes de fonctions

Par exemple, la longueur d'une liste :

```
let rec longueur = function
  | [] -> 0
  | _::queue -> 1+longueur(queue);;
```

Programme 394 – longueur d'une liste

Par exemple :

```
# longueur(["machin";"truc"]);;
- : int = 2
```

Concaténation de 2 listes :

```
let rec concat = function
  | [],liste2->liste2
  | (tete1::queue1),liste2->tete1::(concat(queue1,liste2));;
```

Programme 395 – concaténation de deux listes

Alors :

```
# concat(["machin";"truc"],["bidule";"chose"]);;
- : string list = ["machin"; "truc"; "bidule"; "chose"]
```

Je ne m'en lasse pas....

Retournons une liste :

```
let rec retourne = function
  | [] -> []
  | tete::queue -> concat(retourne(queue),[tete]);;
```

Programme 396 – retournement d'une liste

Trop fun :

```
# retourne([1;2;3;4;5]);;
- : int list = [5; 4; 3; 2; 1]
```

Appliquons une fonction aux éléments d'une liste :

```
let rec applique f = function
  | [] -> []
  | tete::queue -> (f(tete))::(applique f queue);;
```

Programme 397 – application d'une fonction aux éléments d'une liste

Et hop :

```
# applique (function x->x*x) [1;2;3;4;5];;
- : int list = [1; 4; 9; 16; 25]
```

## B Fonctions d'ordre supérieur

### B1 Somme

On peut créer une fonction somme qui calcule la somme des  $f(x)$  avec  $x$  un élément d'une certaine liste.

```
let rec somme(f) = function
| [] -> 0
| x :: l -> f(x) + somme(f)(l);;
```

Programme 398 – somme des images par une fonction des éléments d'une liste

Par exemple, on peut l'utiliser pour avoir la somme des carrés des éléments de la liste [1;3;5;7] :

```
# somme(function x->x*x) ([1;3;5;7]);;
- : int = 84
```

Si on veut travailler avec des flottants, il faut arranger un peu :

```
# let rec somme_r(f) = function
| [] -> 0.
| x :: l -> f(x) +. somme_r(f)(l);;
```

Pour avoir la somme des inverses des carrés des éléments de la liste [1;3;5;7] :

```
# somme_r(function x->1./.(x*.x)) ([1.;3.;5.;7.]);;
- : float = 1.17151927437641712
```

Pour avoir la somme de leurs sinus :

```
# somme_r(function x->sin(x)) ([1.;3.;5.;7.]);;
- : float = 0.680653316923414353
```

### B2 Composée

On définit une fonction composant deux fonctions :

```
# let rond(f,g)=function x->f(g(x));;
```

Programme 399 – composée de deux fonctions

Définissons la fonction « carré » :

```
# let carre=function x->x*x;;
```

Programme 400 – fonction carré

Petit jeu : comment interprétez-vous le résultat suivant ?

```
# rond(carre, somme(carre))([1;2;3;4;5]);;
- : int = 3025
```

Rigolo...

Continuons à nous amuser. Définissons l'itérée de la composée d'une fonction :



```
# let rec iteree f n =
  if n = 0 then function x -> x
  else rond(f, (iteeree f (n - 1)));;
```

Programme 401 – itérée de la composée d’une fonction

On a omis les parenthèses pour les variables (ce qui est courant sous CAML) pour ne pas se perdre...  
Par exemple, on calcule  $\sin \circ \sin \circ \dots \circ \sin(1)$  :

```
# iteree sin 15 1.;;
- : float = 0.395376546988834565
```

## C Dérivée formelle

On va fabriquer assez simplement une machine à dériver formellement.

On crée d’abord un type de variable (en plus des types existant comme int, float, list, etc.) qu’on appelle formel par exemple et qui va faire la liste et définir les expressions formelles que l’on veut dériver :

```
# type formel =
  | Int of int
  | Var of string
  | Add of formel * formel
  | Sous of formel*formel
  | Mul of formel * formel
  | Div of formel*formel
  | Ln of formel
  | Cos of formel
  | Sin of formel
  | Puis of int*formel
  | Rac of formel
  | Exp of formel;;
```

Programme 402 – dérivée formelle - les types de dérivations

Par exemple, Var (comme variable...) sera de type formel mais prendra comme argument une chaîne de caractère (string). On entrera donc les variables à l’aide de guillemets.

De même, Add est de type formel\*formel, c’est-à-dire que c’est un objet formel qui prend comme argument un couple d’objets formels.

On définit ensuite récursivement une fonction deriv qui va dériver nos expressions selon des règles que nous allons établir :

```
# let rec deriv(f, x) =
  match f with
  | Var y when x=y -> Int 1
  | Int _ | Var _ -> Int 0
  | Add(f, g) -> Add(deriv(f, x), deriv(g, x))
  | Sous(f, g) -> Sous(deriv(f, x), deriv(g, x))
  | Mul(f, g) -> Add(Mul(f, deriv(g, x)), Mul(g, deriv(f, x)))
  | Div(f,g) -> Div(Sous(Mul(deriv(f,x),g),Mul(f,deriv(g,x))),Mul(g,g))
  | Ln(f) -> Div(deriv(f,x),f)
  | Cos(f) -> Mul(deriv(f,x),Mul(Sin(f), Int(-1)))
  | Sin(f) -> Mul(deriv(f,x),Cos(f))
  | Puis(n,f) -> Mul(deriv(f,x),Mul(Int(n),Puis(n-1,f)))
  | Rac(f) -> Mul(deriv(f,x),Div(Int(1),Mul(Int(2),Rac(f))))
  | Exp(f) -> Mul(deriv(f,x),Exp(f));;
```

Programme 403 – méthodes de dérivation formelle

Par exemple, quand la variable `Var` est en fait la *variable* au sens mathématique, sa dérivée est 1. En effet, `Var` regroupe à la fois la variable de la fonction mathématique mais aussi les paramètres formels éventuels.

La ligne suivante dit que toute autre variable (donc paramètre) et toute constante numérique entière (`Int`) a pour dérivée 0. Ensuite on explique comment dériver une somme, une différence, un cosinus, etc.

Par exemple, dérivons  $x \mapsto kx^2$ .

On définit deux variables `k` et `x` :

```
# let x , k = Var "x", Var "k";;
```

Puis on demande la dérivée :

```
# deriv(Mul(k,Puis(2,x)), "x");;
```

On obtient :

```
- : formel =
Add (Mul (Var "k", Mul (Int 1, Mul (Int 2, Puis (1, Var "x")))),
    Mul (Puis (2, Var "x"), Int 0))
```

qu'il reste à « déchiffrer »...

$$k \times 1 \times 2 \times x^1 + x^2 \times 0 = 2kx$$

Pour  $\ln\left(\frac{1}{x}\right)$  :

```
# deriv(Ln(Div(Int(1),x)), "x");;
```

On obtient :

```
- : formel =
Div
(Div (Sous (Mul (Int 0, Var "x"), Mul (Int 1, Int 1)),
    Mul (Var "x", Var "x")),
    Div (Int 1, Var "x"))
```

C'est-à-dire :

$$\frac{\frac{0 \times x - 1 \times 1}{x \times x}}{\frac{1}{x}}$$

Just for fun : dérivons  $\ln(\ln(x))$

```
# deriv(Ln(Ln(Ln(x))), "x");;
```

On obtient :

```
- : formel = Div (Div (Div (Int 1, Var "x"), Ln (Var "x")), Ln (Ln (Var "x")))
```

C'est-à-dire :

$$\frac{\frac{\frac{1}{x}}{\ln(x)}}{\ln(\ln(x))}$$

On devrait être extrêmement impressionné par ce qui vient d'être fait ici ! À partir de la définition d'un type de variable formel et d'une fonction `deriv`, on a pu calculer de manière symbolique des dérivées de toutes les fonctions étudiées au lycée ! Et on est parti de pratiquement rien sur les fonctions numériques ! On a juste précisé quelle était la règle de dérivation basique pour chacune d'elles et leur somme/produit/rapport.

La seule petite entorse à notre règle ascétique est la fonction `Puis` qui utilise la somme des entiers. On aurait pu l'écrire :

```
| Puis(n,f) -> Mul(d(f,x),Mul(Int(n),Puis(Sous(Int(n),Int(1)),f)))
```

ou ne pas la définir du tout et se contenter de `Mul(x,x)` mais bon, on a un peu gagné en lisibilité...

Il reste d'ailleurs à effectuer un traitement informatique de ces deux fonctions pour qu'elles affichent leurs résultats de manière plus lisible. Cependant, cela prendrait du temps et l'exercice consistant à décrypter les réponses peut être formateur mathématiquement.

# 25 - Exemples de TP d'algorithmique

## A Fractions et PGCD en 2<sup>nde</sup>

### A1 Division

#### A1a Comme en primaire

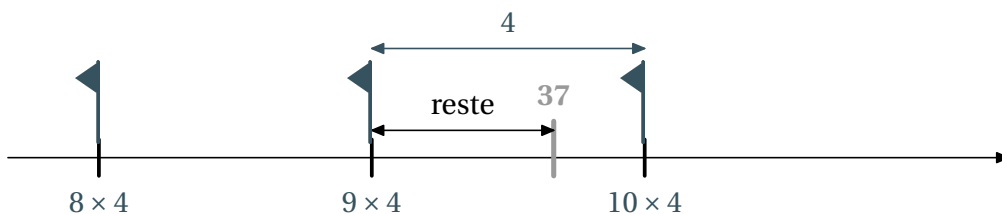
Vous voulez diviser 37 par 4 : à l'école primaire, vous avez appris à vous poser la question :

*En 37 combien de fois 4 ?*

Puis vient le temps du calcul :

- une fois 4, 4 : trop petit...
- 2 fois 4, 8 : trop petit...
- 3 fois 4, 12 : trop petit...
- ...
- 9 fois 4, 36 : trop petit...
- 10 fois 4, 40 : Ah! Trop grand!

« Il y va » donc 9 fois mais on n'y est pas encore : il me reste un petit pas à faire de 36 jusqu'à 37.



Comment traduire cette division à l'aide d'une somme et d'un produit ?

Comment s'appelle chaque membre de cette division ?

#### A1b Parlons à l'ordinateur

Imaginez maintenant que vous devez expliquer ce calcul à un ordinateur en l'imaginant comme un petit lutin se promenant sur la droite des nombres.

Vous pourriez commencer par lui dire :

« tu pars de 0 et tu vas compter tes enjambées »,

puis

« tu avances de 4 unités tant que tu n'auras pas dépassé 37 ».

Enfin,

« dès que tu as dépassé 37, tu t'arrêtes : le quotient est le nombre d'enjambées moins une ».

Il ne reste plus qu'à lui dire dans sa langue... Nous étudierons principalement deux langues informatiques cette année : le XCAS et le CAML.

Par exemple, en XCAS, cela donne :

```

quotient(dividende,diviseur):={
  enjambees:=0;
  while(enjambees*diviseur<=dividende){
    enjambees:=enjambees+1
  }
  return(enjambees-1)
}

```

Programme 404 – quotient euclidien naïf en impératif

Écrivez un script qui renvoie cette fois le reste de la division euclidienne de deux nombres.

### A 1 c Parlons à un ordinateur plus malin

Les mathématiques, c'est pour les fainéants... Le problème, c'est que pour pouvoir se payer le luxe d'être fainéant, il faut pas mal réfléchir!

Par exemple, je suis malin et passablement fainéant. Je dois diviser  $a$  par  $b$  donc savoir « en  $a$  combien de fois  $b$  ». Je me dis :

*Ben déjà, si  $a$  est plus petit que  $b$ , en  $a$  il y va zéro fois : le quotient vaut donc zéro*

Puis je me dis :

*Si je recule de  $b$ , il ira une fois de moins  $b$  dans  $a$ . Cela signifie que  $\text{quotient}(a,b)=1+\text{quotient}(a-b,b)$*

Eh ben, avec ça, un ordinateur intelligent va se débrouiller...

On lui demande le quotient de 37 par 4 :

- est-ce que 37 est plus petit que 4? Non, donc je garde en mémoire que  $\text{quotient}(37,4)=1+\text{quotient}(37-4,4)$  et je m'occupe de  $\text{quotient}(33,4)$  ;
- est-ce que 33 est plus petit que 4? Non, donc je garde en mémoire que  $\text{quotient}(33,4)=1+\text{quotient}(33-4,4)$  et je m'occupe de  $\text{quotient}(29,4)$  ;
- etc.
- est-ce que 5 est plus petit que 4? Non, donc je garde en mémoire que  $\text{quotient}(5,4)=1+\text{quotient}(5-4,4)$  et je m'occupe de  $\text{quotient}(1,4)$  ;
- est-ce que 1 est plus petit que 4? Oui, donc je sais que  $\text{quotient}(1,4)=0$  ;
- donc je vais pouvoir calculer  $\text{quotient}(5,4)$  puis  $\text{quotient}(9,4)$  puis etc.

Ça paraît plus long mais ce sale boulot de gestion de la mémoire est fait par des logiciels intelligents. Nous pouvons nous contenter de donner juste à l'ordinateur un cas simple et un moyen pour aller d'un cas compliqué vers le cas simple.

Regardez plutôt!

```

quotient_malin(dividende,diviseur):={
  if(dividende<diviseur)
  then{0}
  else{1+quotient_malin(dividende-diviseur,diviseur)}
}

```

Programme 405 – quotient euclidien en récursif

CAML parle à peu près le même langage :

```

let rec quotient_malin(dividende,diviseur)=
  if dividende<diviseur
  then 0
  else 1+quotient_malin(dividende-diviseur,diviseur)

```

Programme 406 – quotient euclidien en récursif (CAML)

Écrivez un script qui renvoie cette fois le reste malin de la division euclidienne de deux nombres.

**A 2 PGCD****A 2 a Diviseur**

Le PéGéCéDé de deux nombres, c'est leur plus grand diviseur commun.

D'abord, qu'est-ce qu'on appelle un diviseur dans ce contexte ?

Par exemple, 4 divise 80 car le reste de la division de 80 par 4 vaut 0 : « la division tombe juste » comme vous disiez en CM1.

On peut aussi dire que 4 divise 32 : 4 est donc un *diviseur commun* de 32 et 80. Mais est-ce le plus grand ?

**A 2 b Euclide**

Euclide, c'est un savant grec qui a eu beaucoup de bonnes idées il y a 2300 ans. On utilise encore largement ses résultats au lycée.

Bon, vous voulez connaître le plus grand diviseur commun de deux nombres  $a$  et  $b$ .

Est-ce qu'il en existe au moins un ?

L'idée d'Euclide a été d'effectuer la division...euclidienne de  $a$  par  $b$ . Soit  $a$  le plus grand des deux.

$$a = b \times q + r \quad \text{avec} \quad 0 \leq r < b$$

Appelons  $d$  un diviseur commun de  $a$  et  $b$ .

Pourquoi  $d$  est aussi un diviseur commun de  $b$  et  $r$  ?

Et vice-versa et dans l'autre sens ?

Donc chercher le PGCD de  $a$  et  $b$  revient à chercher celui de  $b$  et  $r$ .

Quel est l'avantage ? Qu'est-ce qui se passe au bout d'un moment ? Y a-t-il un cas simple ? Peut-on utiliser notre méthode de fainéant ?

Testez vos conjectures sur la machine.

**A 3 Opérations sur les nombres rationnels****A 3 a Fraction simplifiée**

Quelle différence existe-t-il entre  $\frac{7}{3}$  et  $\frac{21}{9}$  ?

Quel rôle peut jouer le PGCD dans cette histoire ?

On va créer un programme simplifiant les fractions.

Au lieu d'écrire un nombre rationnel sous la forme  $\frac{\text{numérateur}}{\text{dénominateur}}$ , on va l'entrer dans la machine sous la forme [numérateur, dénominateur].

Ainsi  $\frac{21}{9}$  s'écrira [21, 9].

On peut alors créer un programme qui simplifiera une fraction donnée.

Par exemple sur XCAS :

```
simp(Fraction):={
a:=Fraction[0]; /* le premier terme de la fraction */
b:=Fraction[1]; /* le second */
if(b==0)
then{"On peut pôt diviser par 0"}
else{[a/pgcd(a,b),b/pgcd(a,b)]}
};;
```

Programme 407 – simplification de fractions

Des commentaires ?

Sur CAML, on procède à quelques aménagements.

Pour prévenir toute division par zéro on va créer un opérateur traitant les exceptions qu'on nommera `Division_par_zero` et qui nous servira de garde-fou :

```
# exception Division_par_zero;;
```

On invoquera cette exception avec la commande `raise` : étant donné votre niveau d'anglais, cette commande est naturelle !...

On peut créer un opérateur qui simplifie les fractions :

```
# let simp([a;b]) =
  if b=0 then raise Division_par_zero
  else [a/pgcd(a,b);b/pgcd(a,b)];;
```

### A 3 b Opérations sur les fractions

On crée ensuite une somme simplifiée de fractions :

```
# let som([a;b],[c;d]) =
  if b=0 or d=0 then raise Division_par_zero else
  simp([a*d+b*c;b*d]);;
```

Par exemple :

```
# som([2;3],[1;6]);;
- : int list = [5; 6]
```

Bon : occupez-vous de la multiplication, du calcul de l'inverse et de la division.

Quand tout sera prêt, comment entrer

$$1 + \frac{2 + \frac{3}{4}}{1 - \frac{5}{6}}$$

et

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

## B Suite de Fibonacci en 2<sup>nde</sup>

### B 1 Des animaux, des plantes et des nombres

#### B 1 a Des lapins

Voici l'énigme des lapins qui fut proposée en 1202 par Léonard de Pise, fils de Bonacci :



#### Exploration 1 : Lapins de Fibonacci

Partant d'un couple, combien de couples de lapins obtiendrons-nous après un nombre donné de mois sachant que chaque couple produit chaque mois un nouveau couple, lequel ne devient productif qu'après deux mois.

Alors...le mois n°0, il y a 1 couple de bébés lapin...le mois n°1 encore le même couple...le mois n°2, hop, un couple de bébés naît, ça fait deux couples en tout...le mois n°3, un couple de plus, ça fait 3 couples...

#### B 1 b Des fourmis et des abeilles



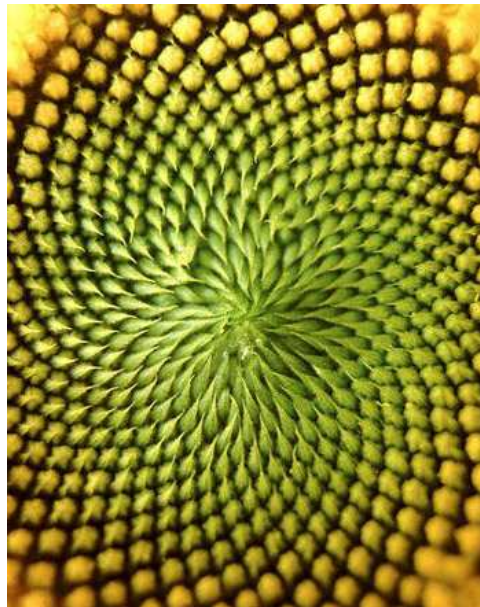
#### Exploration 2 : arbre généalogique

Les abeilles et les fourmis ne naissent pas comme nous dans les roses ou les choux mais sortent du ventre de leur reine qui passe sa vie à pondre.  
Lorsque l'œuf est fécondé, le bébé est femelle, sinon, ce sera un mâle.  
Ainsi, une femelle a un papa et une maman alors qu'un mâle n'a qu'une maman.  
On voudrait compter les ancêtres d'un mâle de la septième génération.

Bon, alors, un mâle a une seule maman, cette maman a un papa et une maman et... vaudrait mieux dresser un arbre généalogique, ce sera plus clair.

Appelons  $A(n)$  le nombre d'ancêtres de la génération  $n$ ,  $F(n)$  le nombre de femelles et  $M(n)$  le nombre de mâles.

### B 1 c Des tournesols



Comptez les spirales...

### B 1 d Des pommes de pin



Comptez les spirales...

**B 1 e** Des ordinateurs**Exploration 3 : Fibonacci : le retour**

On appelle suite de Fibonacci la « séquence » de nombres :

$$1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - 34 - 55 - 89 - \dots$$

Quel est le suivant ?

Essayez de trouver un algorithme malin afin que CAML puisse calculer le  $n$ -eme nombre de cette séquence. Utilisez cet algorithme pour étudier le quotient d'un terme de cette séquence par le précédent.

**B 2** Des hommes et un nombre**B 2 a** Un rectangle**Exploration 4 : un rectangle en or**

- Tracez un segment ;
- tracez un autre segment, perpendiculaire au premier et de longueur double. On obtient ainsi un triangle rectangle ;
- prolongez le premier segment à partir de son extrémité libre d'une longueur égale à l'hypoténuse du triangle précédemment tracé ;
- on obtient à présent un rectangle. Calculez de manière exacte le quotient entre la largeur et la longueur de ce rectangle puis donnez-en une valeur approchée.

Cela vous rappelle-t-il quelque chose ?

**B 2 b** Une spirale**Exploration 5 : une spirale en or**

On reprend le rectangle précédent. On trace à l'intérieur le carré de côté la largeur du rectangle. Que pensez-vous de petit rectangle restant ? Et si on continuait ?

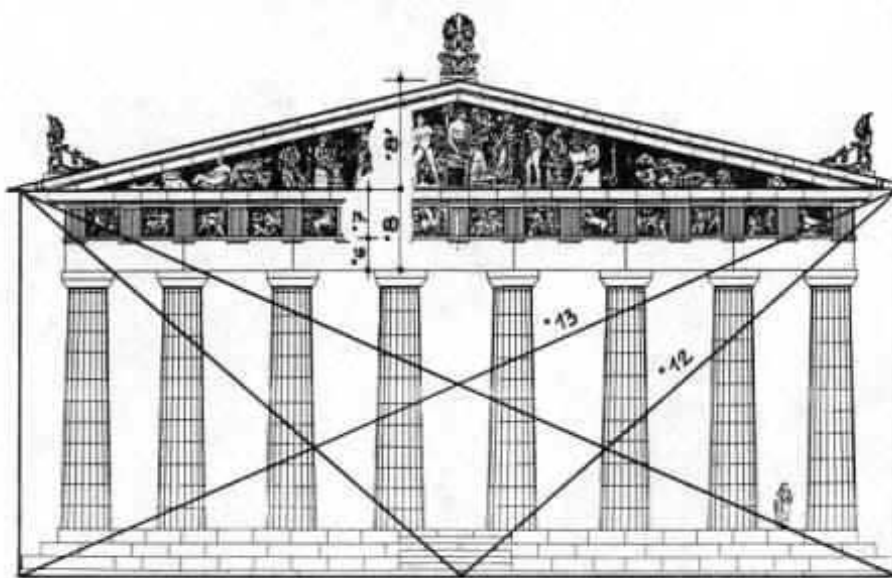
À partir du coin supérieur droit du carré, tracez un arc de cercle de rayon le côté du carré et d'extrémités les coins supérieur gauche et inférieur droit puis réitérez le processus pour obtenir une spirale.





**B 2 c Le Parthénon****😊 Exploration 6 : Parthénon en or**

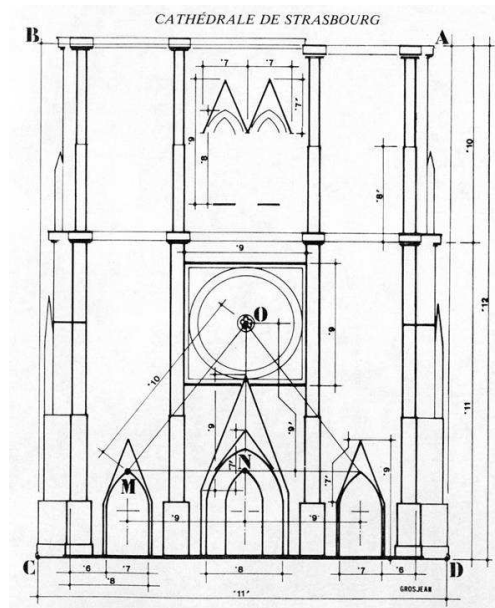
Retrouvez des proportions d'Or dans le Parthénon.

**B 2 d Khéops****😊 Exploration 7 : Pyramide en or**

La hauteur de la pyramide de Khéops est de 148,2m. Le côté de sa base carré mesure 232,8m. Calculez la hauteur d'une de ses faces triangulaires. Retrouvez alors une proportion en or...

**B 2 e Strasbourg****😊 Exploration 8 : Parthénon en or**

Retrouvez des proportions d'Or dans la cathédrale de Strasbourg.

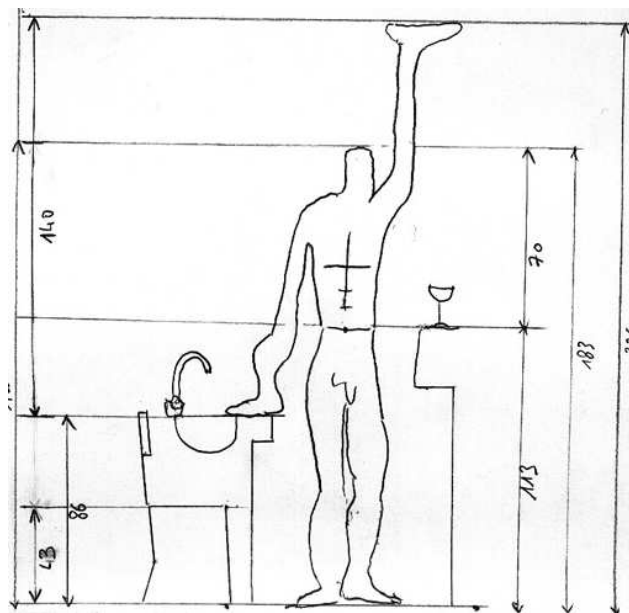


**B 2 f** Rézé



**Exploration 9 : Le Corbusier**

L'architecte Le Corbusier qui a conçu notre Maison Radieuse municipale a conçu les dimensions de ses appartements selon le principe du *Modulor* représenté dans la figure suivante. Où se cache l'or?...



**B 3** Des calculs et un nombre

**B 3 a** Sans calculatrice



**Exploration 10 :  $\Phi$**

On note  $\Phi$  le nombre d'or. Simplifiez l'écriture de  $1 + \frac{1}{\Phi}$ . Que remarquez-vous? Simplifiez l'écriture de  $\Phi^2$  puis simplifiez l'écriture de  $1 + \Phi$ . Que remarquez-vous?

**B 3 b Avec ordinateur****Exploration 11 : racines**

Donnez une approximation de

1.  $a_0 = 1$

2.  $a_1 = \sqrt{1 + \sqrt{1}}$

3.  $a_2 = \sqrt{1 + \sqrt{1 + \sqrt{1}}}$

4.  $a_3 = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1}}}}$

5.  $a_4 = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1}}}}}$

Que pensez-vous des algorithmes suivant :

```
# let rec racor(n)=
  if n=0 then 1.
  else sqrt(1. +. racor(n-1));;
```

Programme 408 – suite  $u_n n + 1 = \sqrt{u_n + 1}$

```
racor(n) := {
  r := 1.;
  for(k:=1; k<=n; k:=k+1) {
    r := sqrt(1+r)
  }
  return(r)
};;
```

Programme 409 – suite  $u_n n + 1 = \sqrt{u_n + 1}$  avec XCAS

**Exploration 12 : inverses**

Donnez une approximation de

1.  $b_0 = 1$

2.  $b_1 = 1 + \frac{1}{1}$

3.  $b_2 = 1 + \frac{1}{1 + \frac{1}{1}}$

4.  $b_3 = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}$

5.  $b_4 = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}}$

6.  $b_5 = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}}}$

Trouvez un algorithme malin pour calculer  $b_n$  avec CAML et un algorithme impératif avec XCAS.

**C****L'approximation de  $\pi$  d'Archimède en 2<sup>nde</sup>****C1****Dodécagone**

À partir de l'hexagone régulier ABCDEF inscrit dans le cercle de rayon 1 et de centre O que nous avons étudié précédemment, nous voudrions tracer un polygone inscrit à 12 côtés (dodécagone) pour avoir un peu plus de précision.

Commencez d'abord par zoomer sur le triangle AOB.

Le nouveau sommet  $A'$  du dodécagone est bien sûr sur le cercle et nous le choisissons tel que  $A'A = A'B$  : pourquoi ?

Que représente la droite  $(OA')$  pour le segment  $[AB]$  ? Pourquoi ?

La droite  $(OA')$  coupe la droite  $[AB]$  en  $I$  : que pouvez-vous dire sur  $I$  ?

Calculez les distances  $OI$  puis  $IA'$  puis  $AA'$  puis le périmètre du dodécagone.

Quelle nouvelle valeur approchée de  $\pi$  obtient-on ?

## C2 Tétraicosagone

On double encore le nombre de côtés. Zoomez comme tout à l'heure sur le triangle  $OAA'$  et essayez de vous inspirer de la méthode que nous venons d'employer pour calculer le périmètre du tétraicosagone. On appellera  $J$  le milieu de  $[AA']$ .

## C3 96 côtés... et plus

### C3 a Un vieux théorème

Reprenez la figure du dodécagone en traçant le cercle en entier et en rajoutant le point  $K$ , milieu de  $[BA']$ .

Notons  $A_d$  le point diamétralement opposé à  $A'$ .

Que pensez-vous des segments  $[OK]$  et  $[A_dB]$  ?

### C3 b Un nouveau théorème

On considère un triangle  $TRI$  rectangle en  $R$  et  $H$  le pied de la hauteur issue de  $R$ .

Combien de triangles rectangles sont maintenant dessinés sur la figure ?

Appliquez le théorème de Pythagore à chacun de ces triangles.

On voudrait montrer que  $TR^2 = TH \times TI$ .

Dans une des trois équations écrites grâce au théorème de Pythagore, isolez  $TR^2$  à gauche puis débrouillez-vous avec les deux autres équations pour n'avoir que des  $TH$  et des  $TI$  à droite.

Retournez vers notre cercle et essayez de montrer que  $OJ = \sqrt{\frac{1+OI}{2}}$ .

### C3 c Un cerf-volant

Un cerf-volant, c'est un quadrilatère non croisé dont les diagonales sont perpendiculaires.

Soit  $ABCD$  un cerf-volant. Calculez l'aire de  $ABCD$  de deux manières différentes.

Revenez au cercle. Trouvez un cerf-volant caché dans la figure. Calculez son aire de deux manières différentes pour montrer que  $AA' = \frac{AB}{2OJ}$ .

### C3 d Héritage

On suppose connus  $OI$  et  $AB$  : comment calculer  $OJ$  et  $AA'$  ?

En quoi cela peut nous aider à continuer à nous approcher du cercle ?

### C3 e Un ordinateur

Tout ça nous demande beaucoup de calculs or un ordinateur sait calculer donc nous allons lui laisser faire la sale besogne. Le problème, c'est qu'un ordinateur est stupide : il faut lui expliquer précisément quels calculs il doit effectuer.

Appelons l'étape 0 le cas étudié avec l'hexagone, l'étape 1 le cas du dodécagone, l'étape 2 le cas du tétraicosagone, etc.

Nous venons de voir comment, connaissant les dimensions des côtés et la distance du centre au côté à une certaine étape, nous pouvons calculer les dimensions des côtés et la distance du centre au côté à l'étape suivante.

Appelons  $c(n)$  la longueur du côté du polygone de l'étape  $n$  et  $d(n)$  la distance de  $O$  à un côté du polygone de l'étape  $n$ .

Que valent  $c(0)$  et  $d(0)$  ?

Combien le polygone de l'étape  $n$  a-t-il de côtés ?

Calculez  $d(n+1)$  en fonction de  $d(n)$  puis  $c(n+1)$  en fonction de  $c(n)$  et de  $d(n+1)$ .

Voici à l'aide de CAML un algorithme malin qui permet de calculer  $d(n)$  à n'importe quelle étape  $n$  :

```
# let rec d(n)=  
  if n=0 then 0.5*.sqrt(3.)  
  else sqrt(0.5*(1.+d(n-1)));;
```

Comment dit-on carré en anglais ? Et racine ? À votre avis, à quoi correspond la commande `sqrt`.

À vous de trouver un programme pour calculer  $c(n)$  puis pour donner une valeur approchée par défaut de  $\pi$ ...

# Liste des programmes

1	Totue LOGO	14
2	attention aux affectations	15
3	affectation et image par une fonction	16
4	affectation et composition de fonctions	16
5	exemple de procédure	16
6	exemple de programme récursif	18
7	exemple de programme récursif en anglais	19
8	successeur d'un entier	19
9	successeur version terminale (étape 1)	19
10	successeur version terminale (étape 2)	20
11	composition de fonction	20
12	création de fonctions simples	20
13	composée de fonctions avec XCAS	20
14	mouvement élémentaire	21
15	résolution récursive du problème des tours de Hanoi	21
16	suite récurrente	22
17	suite récurrente (VO)	22
18	suite récurrente avec CAML	23
19	suite récurrente avec Sage	23
20	suite récurrente version impérative	23
21	suite récurrente version impérative (VO)	23
22	suite récurrente version impérative en Sage	23
23	somme des entiers avec XCAS	24
24	somme des entiers avec XCAS (VO)	24
25	somme des entiers avec CAML	24
26	somme des images des entiers par une fonction avec CAML	24
27	somme des entiers avec Sage	25
28	somme des images des entiers par une fonction avec Sage	25
29	somme des entiers version impérative avec XCAS	25
30	somme des entiers version impérative (VO)	25
31	somme des entiers version impérative avec Sage	25
32	partie entière récursive	26
33	partie entière récursive (VO)	26
34	partie entière récursive avec CAML	26
35	partie entière récursive avec Sage	27
36	partie entière impérative (XCAS)	27
37	partie entière impérative (VO)	28
38	partie entière impérative (Sage/Python)	28
39	fractions continues en récursif	29
40	partie entière en récursif (VO)	29
41	partie entière en récursif avec CAML	29
42	partie entière en récursif (Sage)	29
43	partie entière en impératif	30
44	partie entière en impératif (VO)	30
45	partie entière en impératif (Sage)	30
46	dichotomie en récursif	31

47	dichotomie en récursif (VO)	31
48	dichotomie en récursif avec CAML	31
49	dichotomie en récursif (Sage/Python)	32
50	dichotomie en impératif (XCAS)	32
51	dichotomie en impératif (Sage)	33
52	longueur d'une liste en récursif (CAML)	33
53	longueur d'une liste en récursif (Sage)	34
54	somme d'entiers en récursif (CAML)	34
55	somme d'entiers en récursif (Sage)	34
56	valeur absolue	36
57	valeur absolue (VO)	36
58	valeur absolue (CAML)	37
59	valeur absolue (Sage)	37
60	partie entier en récursif	37
61	partie entière en récursif (VO)	37
62	partie entière en récursif (CAML)	38
63	partie entière récursive avec Sage	38
64	partie entière en impératif	38
65	partie entière en impératif (VO)	39
66	partie entière impérative (Sage/Python)	39
67	arrondi au centième	40
68	arrondi au centième avec CAML	40
69	somme des entiers en récursif	40
70	somme des entiers en récursif avec CAML	41
71	somme des images des entiers par une fonction (CAML)	41
72	somme des entiers avec Sage	41
73	somme des images des entiers par une fonction avec Sage	41
74	somme des entiers en impératif	41
75	somme des entiers en impératif (VO)	42
76	somme des entiers version impérative avec Sage	42
77	maximum d'une liste en récursif (1)	42
78	maximum d'une liste en récursif (2)	43
79	maximum d'une liste en récursif (3)	43
80	maximum d'une liste en récursif (XCAS)	43
81	maximum d'une liste en récursif (Sage)	43
82	maximum d'une liste en impératif (XCAS)	44
83	maximum d'une liste en impératif (Sage)	44
84	taux de remise variable en impératif	45
85	taux de remise variable en impératif (VO)	45
86	taux de remise variable en récursif (CAML)	46
87	taux de remise variable (Sage)	46
88	quotient euclidien en récursif (XCAS)	47
89	quotient euclidien en récursif (Sage)	47
90	quotient euclidien en récursif (CAML)	47
91	quotient euclidien en impératif (XCAS)	48
92	quotient euclidien en impératif (Sage)	48
93	reste euclidien en récursif	49
94	reste euclidien en récursif (CAML)	49
95	reste euclidien en récursif (Sage)	49
96	reste euclidien en impératif (XCAS)	50
97	reste euclidien en impératif (Sage)	50
98	divisibilité (XCAS)	50
99	divisibilité (CAML)	50
100	divisibilité (Sage)	51
101	nombre de diviseurs en récursif (1)	51
102	nombre de diviseurs en récursif (2)	51
103	nombre de diviseurs en impératif (1) XCAS	52
104	nombre de diviseurs en impératif (2) XCAS	52
105	représentation graphique du nombre de diviseurs (XCAS)	52

106	nombre de diviseurs en impératif (1) Sage	53
107	nombre de diviseurs en impératif (2) Sage	53
108	représentation graphique du nombre de diviseurs(Sage)	53
109	liste des diviseurs en récursif (1)	54
110	somme des termes d'une liste en récursif (1)	55
111	nombres parfaits	55
112	liste des diviseurs d'un entier (2)	55
113	somme des termes d'une liste en récursif (2)	55
114	liste des nombre parfait en récursif	55
115	liste des nombres parfaits en récursif (version terminale)	56
116	Liste des diviseurs en impératif	56
117	nombres parfaits en impératif	56
118	liste des nombres parfaits en impératif	57
119	liste des diviseurs d'un entier (Sage)	57
120	test entier parfait (Sage)	57
121	Liste des entiers parfaits (Sage)	57
122	PGCD en récursif (1)	58
123	PGCD en récursif (1) avec CAML	58
124	PGCD en récursif (1) avec Sage	58
125	PGCD en impératif (1) (XCAS)	59
126	PGCD en impératif (1) (Sage)	59
127	PGCD en récursif (2)	60
128	PGCD en récursif (2) avec CAML	60
129	PGCD en récursif (2) avec Sage	60
130	algorithme d'Euclide étendu en récursif	61
131	algorithme d'Euclide étendu en récursif (CAML)	61
132	algorithme d'Euclide étendu en récursif (Sage)	62
133	algorithme d'Euclide étendu en impératif (XCAS)	62
134	somme de listes terme à terme (Sage)	62
135	multiplication d'une liste par un scalaire (Sage)	63
136	algorithme d'Euclide étendu en impératif (Sage)	63
137	fractions continues en récursif (1)	64
138	fractions continues en récursif (1) avec CAML	64
139	fractions continues en récursif (1) avec Sage	64
140	fractions continues en impératif (1) avec XCAS	64
141	fractions continues en impératif (1) avec Sage	65
142	fractions continues en impératif (2) avec XCAS	65
143	fractions continues en impératif (2) avec Sage	66
144	fractions continues en récursif (2)	66
145	fractions continues en récursif (2) avec CAML	66
146	fractions continues en récursif (2) avec Sage	67
147	test de primalité en impératif (1)	67
148	test de primalité en impératif (1) en VO	67
149	test de primalité en récursif (1) avec CAML	68
150	test de primalité en récursif (1bis) avec CAML	68
151	test de primalité en impératif (1) avec Sage	68
152	test de primalité en récursif (1) avec Sage	68
153	test de primalité en impératif (2) avec XCAS	69
154	test de primalité en impératif (2) en VO	69
155	test de primalité en récursif (2) avec CAML	69
156	test de primalité en impératif (2) avec Sage	70
157	test de primalité en récursif (2) avec Sage	70
158	reste symétrique	70
159	test de primalité en impératif (3)	70
160	test de primalité en récursif (3)	71
161	crible d'Ératosthène en impératif	71
162	crible d'Ératosthène en impératif (VO)	71
163	crible d'Ératosthène en impératif (Sage)	72
164	entiers compris entre deux valeurs	72



165	pour retirer les multiples d'un entier dans une liste donnée (1)	72
166	pour retirer les multiples d'un entier dans une liste donnée (2)	72
167	crible d'Ératosthène en récursif	73
168	décomposition en facteurs premiers en récursif (XCAS)	73
169	décomposition en facteurs premiers en récursif avec CAML (1)	74
170	décomposition en facteurs premiers en récursif avec CAML (2)	74
171	décomposition en facteurs premiers en récursif (Sage)	74
172	nombre d'occurrences d'un nombre dans une liste en récursif	75
173	liste des éléments d'une liste avec leur valuation	75
174	décomposition en facteurs premiers en récursif avec CAML (2)	75
175	décomposition en facteurs premiers en impératif (1)	76
176	décomposition en facteurs premiers en impératif (2)	76
177	nombres jumeaux en impératif	77
178	nombre de chiffres d'un entier (XCAS)	77
179	nombre de chiffres d'un entier (CAML)	77
180	nombre de chiffres d'un entier (Sage)	77
181	décomposition en base 2 en impératif (XCAS)	78
182	décomposition en base 2 en impératif (Sage)	79
183	décomposition en base 2 en récursif (XCAS)	79
184	décomposition en base 2 en récursif (CAML)	80
185	décomposition en base 2 en récursif (Sage)	80
186	calcul du jour de la semaine d'une date donnée (XCAS)	80
187	calcul du jour de la semaine d'une date donnée (Sage)	81
188	addition de deux entiers en récursif	82
189	multiplication de deux entiers en récursif	82
190	calcul du PGCD en récursif	82
191	simplification de fraction	83
192	somme de fractions	83
193	produit de fractions	83
194	inverse d'une fraction	83
195	division de fractions	83
196	développement de fractions continues (1)	84
197	développement de fractions continues (2)	84
198	algorithme de Hörner en récursif (XCAS)	85
199	algorithme de Hörner en récursif (CAML)	86
200	algorithme de Hörner en récursif (Sage)	86
201	algorithme de Hörner version impérative	87
202	exponentiation rapide version impérative (XCAS)	87
203	exponentiation rapide version impérative (Sage)	87
204	exponentiation rapide version récursive (XCAS)	88
205	exponentiation rapide version récursive (CAML)	88
206	exponentiation rapide version récursive (Sage)	88
207	exponentiation rapide en récursif « à la Hörner »	89
208	exponentiation rapide en récursif « à la Hörner »	89
209	exponentiation rapide en récursif « à la Hörner »	89
210	exponentiation rapide en impératif (3) (XCAS)	90
211	exponentiation rapide en impératif (3) (Sage)	90
212	exponentiation rapide en récursif (3)	91
213	exponentiation rapide en récursif (3) (Sage)	91
214	la machine légo en récursif (CAML)	92
215	la machine légo en récursif 1ère étape (XCAS)	92
216	la machine légo en récursif 2ème étape (XCAS)	93
217	la machine légo en impératif	93
218	signe d'un produit de facteurs affines	95
219	dérivation numérique	98
220	liste de coordonnées de points d'un graphe de fonction en récursif (XCAS)	98
221	liste de coordonnées de points d'un graphe de fonction en récursif (Sage)	99
222	liste aléatoire de point du graphe d'une fonction (XCAS)	100
223	liste aléatoire de point du graphe d'une fonction (Sage)	100

224	fonction « dent de scie »	101
225	fonction dent de scie (2)	102
226	fonction « dent de scie »	102
227	minimum d'une fonction en récursif 1 (XCAS)	103
228	minimum d'une fonction en récursif 1 (CAML)	103
229	minimum d'une fonction en récursif 1 (Sage)	103
230	minimum d'une fonction en impératif (XCAS)	104
231	minimum d'une fonction en impératif (Sage)	104
232	minimum d'une fonction en récursif 2 (XCAS)	105
233	minimum d'une fonction en récursif 2 (CAML)	105
234	minimum d'une fonction en récursif 2 (Sage)	105
235	distance d'un réel à l'entier le plus proche	106
236	fonction « blanc-manger »	106
237	courbe du blanc-manger avec Sage	107
238	escalier de Cantor en récursif (XAS)	108
239	escalier de Cantor en récursif( CAML)	109
240	escalier de Cantor en récursif( Sage)	109
241	zig-zag de Cantor en récursif)	110
242	zigzag de Cantor en récursif( Sage)	111
243	dichotomie récursive (XCAS)	112
244	dichotomie récursive (CAML)	112
245	dichotomie en récursif (Sage/Python)	113
246	dichotomie impérative	113
247	dichotomie en impératif (Sage)	114
248	partie proportionnelles en récursif	114
249	parties proportionnelles en impératif (XCAS)	114
250	méthode de Newton-Raphson en récursif (XCAS)	117
251	méthode de Newton-Raphson en récursif (CAML)	117
252	méthode de Newton-Raphson en récursif (Sage)	118
253	NEWTON-RAPHSON : version impérative (Sage)	118
254	méthode d'Euler générale en récursif (XCAS)	121
255	méthode d'Euler générale en récursif (Sage)	122
256	méthode d'Euler générale en impératif	123
257	méthode d'Euler générale en impératif (Sage)	124
258	méthode d'Euler pour l'exponentielle en impératif (XCAS)	125
259	méthode d'Euler pour l'exponentielle en récursif (XCAS)	125
260	méthode d'Euler pour l'exponentielle en impératif (Sage)	126
261	méthode d'Euler pour l'exponentielle en récursif (Sage)	126
262	construction ln (XCAS)	127
263	construction ln (Sage)	127
264	méthode des rectangles en récursif (CAML)	128
265	méthode des rectangle en récursif terminal étape 1	129
266	méthode des rectangles en récursif terminal étape 2	129
267	méthode des rectangles en récursif (Sage)	129
268	méthode des rectangles en impératif (XCAS)	129
269	méthode des rectangles en impératif (Sage)	130
270	méthode des trapèzes en récursif (CAML)	130
271	méthode des trapèzes en récursif (Sage)	130
272	méthode des trapèzes en impératif (XCAS)	131
273	méthode des trapèzes en impératif (Sage)	131
274	méthode de Simpson en récursif (CAML)	132
275	méthode de Simpson en récursif (Sage)	132
276	méthode de Simson en impératif (XCAS)	132
277	méthode de Simpson en impératif (Sage)	132
278	système de Cramer en impératif (XCAS)	134
279	système de Cramer en impératif (Sage)	134
280	pivot de Gauss (XCAS)	135
281	pivot de Gauss (Sage)	136
282	algorithme de Héron en impératif (XCAS)	138

283	algorithme de Héron en récursif	138
284	décimales de e en récursif	139
285	décimale de e en impératif (XCAS)	139
286	300 décimales de e en impératif (Sage)	140
287	seuil pour obtenir n décimales de e	140
288	n décimales de e en impératif (Sage)	140
289	approximation de $\sqrt{a}$ par la méthode d'Euler en récursif	141
290	approximation de $\sqrt{a}$ par la méthode d'Euler en impératif	142
291	approximation de $\sqrt{a}$ par la méthode d'Euler en impératif (Sage)	142
292	approximation de $\pi$ par la méthode d'Archimède en récursif	144
293	approximation de $\pi$ par la méthode d'Archimède en impératif	145
294	approximation de $\pi$ par la méthode de Cues en récursif	146
295	approximation de $\pi$ par la méthode de Cues en impératif	147
296	approximation de $\pi$ par la méthode de Cues en impératif	147
297	approximation de $\pi$ par la méthode d'Al-Kashi en récursif	147
298	approximation de $\pi$ par la méthode d'Al-Kashi en impératif	148
299	flocon de Von Koch avec la tortue LOGO	148
300	flocon de Von Koch avec XCAS	149
301	Tableau en triangle	152
302	Triangle de Pascal de Sierpinski	152
303	Tapis de Sierpinski	154
304	Dentelle de Sierpinski	155
305	Dentelle de Sierpinski (Sage)	155
306	L-System (XCAS)	156
307	L-System (Sage)	157
308	ensembles de Julia	158
309	Ensemble de Julia (Sage)	159
310	Ensemble de Mandelbrot	161
311	Ensemble de Mandelbrot (Sage)	162
312	poursuite renard/lapin (1)	163
313	poursuite renard/lapin (2)	164
314	pile ou face	166
315	exemple de tirage de boules	167
316	exemple de tirage de boules	167
317	tirage de boules avec remise (XCAS)	168
318	tirage de boules avec remise (XCAS)	169
319	jeu du Monty Hall (XCAS)	169
320	Monty Hall (Sage)	171
321	problème du Duc de Toscane (XCAS 1)	171
322	problème du Duc de Toscane (XCAS 2)	172
323	problème du Duc de Toscane (XCAS 3)	172
324	problème du Duc de Toscane (Sage)	172
325	pile ou face (2)	173
326	lancers successifs de pièces (Sage)	174
327	le lièvre et la tortue (1)	174
328	le lièvre et la tortue (2) (XCAS)	175
329	lièvre et tortue 1 (Sage)	175
330	lièvre et tortue 2 (Sage)	176
331	générateur de nombres aléatoires	177
332	calcul de $\pi$ avec une pluie aléatoire en impératif (XCAS)	177
333	calcul de $\pi$ avec une pluie aléatoire en impératif (Sage)	178
334	calcul de $\pi$ avec une pluie aléatoire en récursif	178
335	approximation d'une probabilité élémentaire (loi normale)	178
336	table de la loi normale	179
337	loi normale (Sage)	180
338	somme des éléments d'une liste en impératif	182
339	tracé d'une ligne brisée	182
340	lissage par moyenne mobile en impératif (XCAS)	182
341	lissage par moyenne mobile en impératif (Sage)	183

342	extraction d'une sous-liste en récursif	185
343	longueur d'une liste en récursif	185
344	somme des éléments d'une liste en récursif	185
345	moyenne mobile en récursif	185
346	intervalle de fluctuation en 2 <sup>nde</sup> (XCAS)	186
347	intervalle de fluctuation en 2 <sup>nde</sup> bis (XCAS)	186
348	intervalle de fluctuation en 2 <sup>nde</sup> (Sage)	187
349	intervalle de fluctuation en 2 <sup>nde</sup> bis (Sage)	188
350	fonction mystérieuse (1)	191
351	fonction mystérieuse (2)	191
352	rendu de monnaie par algorithme glouton	192
353	écriture littérale d'un entier	195
354	mise en minuscules	196
355	croissance d'une liste en impératif	197
356	croissance d'une liste en impératif (VO)	198
357	croissance d'une liste en récursif (1)	198
358	croissance d'une liste en récursif (2)	198
359	tri à bulle	199
360	tri à bulle (VO)	200
361	k-eme opérande d'une liste avec CAML	200
362	longueur d'une liste avec CAML	201
363	concaténation de deux listes	201
364	extraction d'une sous-liste avec CAML	201
365	échange de 2 opérandes successifs avec CAML	201
366	tri à bulle avec CAML	201
367	tri par insertion	202
368	tri par insertion (XCAS)	202
369	partition d'une liste en récursif	203
370	partition d'une liste en récursif (suite)	203
371	tri rapide en récursif	203
372	queue d'un chaîne	204
373	codage numérique d'une chaîne	204
374	codage de César en récursif	204
375	codage de César en impératif	205
376	codage numérique d'une chaîne en impératif	205
377	décodage d'un entier en chaîne de caractères	206
378	suite de Fibonacci en récursif (1)	209
379	suite de Fibonacci en récursif (1) avec XCAS	209
380	suite de Fibonacci avec une récursion terminale	209
381	itération polymorphe d'une fonction	209
382	suite de Fibonacci : troisième version	210
383	suite de Fibonacci en impératif	210
384	quotients des termes d'une suite de Fibonacci en récursif	211
385	quotients des termes d'une suite de Fibonacci en impératif (1)	211
386	quotients des termes d'une suite de Fibonacci en impératif (2)	211
387	suite de Syracuse en récursif	211
388	suite de Syracuse en impératif	212
389	suite de Viète en récursif	212
390	suite de Viète en impératif	213
391	suite des inverses des carrés en récursif	213
392	suite des inverses des carrés en impératif	213
393	Exposant de Lyapounov	219
394	longueur d'une liste	220
395	concaténation de deux listes	220
396	retournement d'une liste	220
397	application d'une fonction aux éléments d'une liste	220
398	somme des images par une fonction des éléments d'une liste	221
399	composée de deux fonctions	221
400	fonction carré	221

401 itérée de la composée d'une fonction . . . . .	222
402 dérivée formelle - les types de dérivations . . . . .	222
403 méthodes de dérivation formelle . . . . .	222
404 quotient euclidien naïf en impératif . . . . .	225
405 quotient euclidien en récursif . . . . .	225
406 quotient euclidien en récursif (CAML) . . . . .	225
407 simplification de fractions . . . . .	226
408 suite $u_n n + 1 = \sqrt{u_n + 1}$ . . . . .	232
409 suite $u_n n + 1 = \sqrt{u_n + 1}$ avec XCAS . . . . .	232

- BARTHE, DANIEL: La mesure du cercle. *Tangente*, avril 2009, Nr. 36 HS, pp 15–16
- CONNAN, GUILLAUME ET GROGNET, STÉPHANE: Guide du calcul avec des logiciels libres. Dunod, 2008
- COUSINEAU, GUY ET MAUNY, MICHEL: Approche fonctionnelle de la programmation. Ediscience, 1995
- COUTURIER, ALAIN ET JEAN-BAPTISTE, GÉRALD: Programmation fonctionnelle - Spécifications & applications. Cépaduès-Éditions, 2003
- DARTE, ALAIN ET VAUDENAY, SERGE: Algorithmique et optimisation. Dunod, 2001
- DONZEAU-GOUGE, VÉRONIQUE *et al.*: Informatique - Programmation - Tomes 1, 2 et 3. Masson, 1986
- ENGEL, ARTHUR ET REISZ, DANIEL: Mathématique élémentaire d'un point de vue algorithmique. CEDIC, 1979
- GACÔGNE, LOUIS: Programmation par l'exemple en Caml. Ellipses, 2004
- HABRIAS, HENRI: Spécification avec B. Cours IUT de Nantes, département informatique, octobre 2006
- LEROY, XAVIER ET WEIS, PIERRE: Manuel de référence du langage Caml. InterEditions, 1993
- LEROY, XAVIER ET WEIS, PIERRE: Le langage Caml. Dunod, 1999
- MONASSE, DENIS: Option informatique : cours complet pour la sup MPSI. Vuibert, 1996
- MONASSE, DENIS: Option informatique : cours complet pour la spé MP et MP\*. Vuibert, 1997
- QUERCIA, MICHEL: Algorithmique. Vuibert, 2002
- SAUX PICART, PHILIPPE: Cours de calcul formel - Algorithmes fondamentaux. Ellipses, 1999
- VEIGNEAU, SÉBASTIEN: Approches impérative et fonctionnelle de l'algorithmique. Springer, 1999