Chapitre 1

1.1 Memento Programmation Xcas et Python

	affectation	puissance	quotient (ds $\mathbb{Z}$)	reste (ds $\mathbb{Z}$)	$\mathbb{Z}/n\mathbb{Z}$	commentaires
giac/Xcas, (mode xcas)	:=	^ ou **	`iquo(4,3)`	`irem(2,3)`	`2 % 3` ou `2 mod 3`	`//` ou `/* */`
Python	=	**	4 // 3	2 % 3		#
Xcas mode Python	:=	^ ou **	`iquo(4,3)`	`irem(4,3)` ou 2 % 3	`2 mod 3`	`//` ou #

Modes Xcas et Python

— ⚠ En 2019 de nombreuses versions d'Xcas ont des problèmes lorsque la configuration par défaut est en mode Python. Il faut donc **choisir le mode Xcas lors du premier lancement**, même si l'on souhaite utiliser une syntaxe comme en python. Si au lancement vous êtes en mode python, allez dans le menu `Cfg>config du Cas` changez le mode vers xcas, **sauvez** la config et **quittez** Xcas puis relancez le.

— Pour utiliser un symbole dans une fonction (par exemple `X`) il vaut mieux [1] faire

(giac/xcas)

```
local X;
purge(X)
```

— Dans Xcas, commencer une cellule par le symbole `#` bascule son contenu en mode python quelle que soit la configuration globale (et commencer par `//` bascule en mode Xcas).

(giac/xcas)

```
# La cellule sera en mode python, ceci donnera donc 0 (le quotient de 3 par 4)
3//4
A=5   # on peut alors utiliser l'affectation comme en python. NB := marche aussi.
```

(giac/xcas)

```
// La cellule sera en mode Xcas, ceci donnera donc 3 suivi du commentaire 4
3//4
A:=5; // ici l'affectation doit se faire avec :=
```

— ⚠ **Mais attention** à ne pas valider une cellule qui ne contient que `#` car cela peut planter [2] Xcas immédiatement.

— NB : Dans une fonction definie par def on est automatiquement en mode python. L'exemple ci dessous donnera toujours 0.

(giac/xcas)

```
def test():
    return 3//4
```

Quelques instructions souvent utiles dans Xcas :

— `purge` : Libère une variable.

— `unapply` : Crée une fonction à partir d'un symbole.

— `op` : Enlève un niveau de crochets.

1. certaines "anciennes" versions déclarerait automatiquement toutes les variables comme locales dans une fonction python, et leur affecterait 0, d'où le besoin de faire purge/del pour avoir un symbole.

2. semble OK avec la vieille version 1.4.9 sur clefagreg 2019

1.1.1 Séquences : T-uples, listes, ensembles, dictionnaires

On utilisera essentiellement les listes. On retiendra juste l'existence des ensembles et dictionnaires et comment s'informer rapidement sur ces notions dans la documentation en cas de besoin.

Dans xcas il n'y a que des listes (modifiables). On peut utiliser () ou [], c'est sur la façon de traiter les emboitements que les choses diffèrent.

(giac/xcas)

```
9>> a:=(11,22,"texte",(1,2),11) // Ici les niveaux de () ne comptent pas.
11,22,"texte",1,2,11
10>> b:=[11,22,"texte",[1,2],11] // Les niveaux de []  comptent
[11,22,"texte",[1,2],11]
11>> a[0]:=111  // Dans xcas les objets sont modifiables
111,22,"texte",1,2,11
12>> b[0]:=111
[111,22,"texte",[1,2],11]
13>> b[3][0]; // ou bien b[3,0].
1
14>> set[1,3,2,2] // Dans xcas, les ensembles sont aussi notes: %{1,3,2,2%}
set[1,2,3]
15>> bureau:=table("han"="9D11","hermann"="9D23","danila"="9D3",112="pas de bureau");
//NB: Dans une table tous les types sont acceptes. Cf dictionnaire en python
table(
112 = "pas de bureau",
"danila" = "9D3",
"han" = "9D11",
"hermann" = "9D23"
)
16>> bureau["han"];bureau[112] //On accede aux elements par des objets
"9D11","pas de bureau"
```

En Python on a :

(python)

```
>>> a=(11,22,"peu importe",44,11)  # un t-uple. Ses entrees ne sont pas modifiables.
>>> a[0]
11
>>> b=[11,22,"peu importe le type",44,11,range(3,8)]  # une liste.
>>> b[1]=222;b   #Les entrees d'une liste sont modifiables.
[11, 222, 'peu importe', 44, 11,[3,4,5,6,7]]
>>> b[0],b[3],b[5]
(11, 44, [3,4,5,6,7])
>>> b[5][0] #L'element d'indice 0 de la liste b[5]. En python b[5,0] est incorrect.
3
>>> c=set(a);c # on peut creer un ensemble a partir d'une liste ou d'un t-uple
set(['peu importe', 11, 44, 22])
>>> c[0] # Il n'y a pas d'indices pour les ensembles en python.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>> bureau={"han":"9D11","hermann":"9D23","danila":"9D3",112:"pas de bureau"}
>>> bureau["hermann"] #on a cree un dictionnaire. On accede aux elements
'9D23'
>>> bureau[112] #par des objets de type quelconque et non par des indices.
'pas de bureau'
```

Création automatique d'une liste avec xcas utiliser `seq`.

⚠ Bien selon les syntaxes fournies `seq` peut rendre un résultat entre crochets ou non. (probablement pour des raisons de compatibilité maple, TI)

(giac/xcas)

```
1>> L2:=[ seq( j,  j = 1..9 ) ]; // avec des .. pas de crochets donc on en ajoute
[1,2,3,4,5,6,7,8,9]
2>> L4:=[ seq( j*j, j = L2)] ; // la liste des carres de la liste L2
[1,4,9,16,25,36,49,64,81]
3>> L3:=seq( j,  j , 0, 9, 2 ) ; // avec des , il y a deja des crochets
[0,2,4,6,8]
```

En python on utilise `range` qui peut avoir de 1 jusque 3 arguments selon que l'on précise le début (il est inclus), la fin (exclue), le pas. En Python3 `range` est optimisé, et n'affichera donc pas tous les éléments (ils seront juste crées lorsque l'on en aura besoin)

(python)

```
>>> range(10)  #on peut l'utiliser comme une liste de 0 jusque 9
range(0, 10)
>>> list(range(10)) # en python3 on peut afficher un range en composant avec list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L2=range(1,10)
>>> L4=[ j*j for j in L2] # la liste des carres de L2
[1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> L4=[ j*j for j in range(1,10)] # on pouvait utiliser range directement
>>> L4
[1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
```

1.1.2 Affectations : valeur ou adresse ?

Une première difficulté ; des questions à se poser avec chaque logiciel ou language :
Que se passe t'il lorsque l'on fait ce genre de transformations ?

(giac/xcas)

```
a:=1:;b:=a:;b:=2:; //Dans xcas pour ne pas afficher la reponse on utilise:;
print(a,b); // le ; sert a separer les instructions dans un programme
la:=[11,22,33]:;lb:=la:;lb[0]:=-7:;
afficher(la,lb)  // le ; est facultatif en mode interactif.
```

Remarque 1.1.1 *xcas possède une autre affectation que* `:=` *appelée affectation sur place dans la documentation. Elle se fait par le symbole* `=<`

(giac/xcas)

```
la=<[11,22,33]:;lb=<la:;lb[0]=<-7:;
afficher(la,lb)  // le ; est facultatif en mode interactif.
```

Exercice 1.1.2 *Essayez* `lb[0]:=7` *au lieu du* `=<`

et en Python ?

(python)

```
a=1;b=a;b=2;print(a,b)
la=[11,22,33];lb=la;lb[0]=-7;print(la,lb)
```

1.2 Instructions de contrôle

⚠ En python il n'y a pas de délimiteurs de blocs, c'est l'indentation qui compte. NB : Depuis 2017-2018 Xcas supporte les syntaxes python usuelles pour la programmation de base. Les opérations sont alors faites avec les types d'xcas ce qui permet de faire du calcul formel.

1.2.1 Tests

Xcas possède une syntaxe proche du C (Les boucles et tests coïncident à part l'affectation qui est `:=` sous xcas et `=` en C) :

(giac/xcas)

```
if(a<7){
  print("a<7")
}
else {
  print("a>=7")
}
```

On peut aussi utiliser un syntaxe de type algorithmique.

(giac/xcas)

```
si (a<7) alors
  afficher("a<7")
sinon
  print("a>=7")
fsi
```

Les instructions de comparaison en python et xcas sont les mêmes :

```
x == y       # x est \'egal \`a y
x != y       # x est diff\'erent de y. NB <> est obsolete en python.
x > y        # x est plus grand que y
x < y        # x est plus petit que y
x >= y       # x est plus grand que, ou \'egal \`a y
x <= y       # x est plus petit que, ou \'egal \`a y
```

(python)

```
if (a<7):
   print("a inferieur a 7")
   print("a inferieur a 7")
else:
   print("a vaut au moins 7")
   print("a vaut au moins 7")
```

1.2.2 Boucles

(giac/xcas)

```
a:=0;
while(a<7) // faire shift entree pour aller a la ligne sans evaluer
  {
    a:=a+1; // ou bien a++ ou bien a+=1
    print(a);
  }
tantque a<17 faire
   a:=a+2;
   print(a);
ftantque;  //en fait xcas tolere l'absence de ; mais pas giac
```

(python)

```
>>> a = 0
>>> while (a < 7):        # (n'oubliez pas le double point !)
...     a = a + 1         # (n'oubliez pas l'indentation !)
...     print(a)
```

En python il faut créer une liste pour pouvoir faire une boucle `for`.

(python)

```
>>> range(10) # En python3 range n'affiche rien. Faire list(range(10)) pour afficher
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(3,10,2) # help(range) donne les options de range
[3, 5, 7, 9]
```

On calcule donc $\sum_{i=0}^{9} 3^i$ ainsi :

(python)

```
s=0
for i in range(10):
    s=s+3**i //le corps de la boucle est donne par l'indentation.
print(s)
```

NB : Sous Xcas [3] la lettre i est réservée au nombre complexe.

3. sauf en mode maple

(giac/xcas)

```
s:=0;
for(j:=0;j<10;j++){  //pour aller de 2 en 2 on remplace j++ par j:=j+2 (ou j+=2)
  s:=s+3^j;
}
```

(giac/xcas)

```
s:=0;
pour j de 0 jusque 9 faire
 s:=s+3^j;
fpour
```

(giac/xcas)

```
s:=0;
pour j de 0 jusque 9 pas 2 faire // pour aller de 2 en 2
 s:=s+3^j;
fpour
```

Exercice 1.2.1 *En utilisant dans xcas la syntaxe* `for( ; ; )` *créez une bouble de type* `while`*.*

On peut aussi utiliser des listes avec xcas. L'analogue python de `range` est `seq`

(giac/xcas)

```
2>> s:=seq(j^2,j,1,10)  // seq donne une suite entre ( )
(1,4,9,16,25,36,49,64,81,100)
3>> seq(j^2,j=1..10)
1,4,9,16,25,36,49,64,81,100
4>> seq(j^2,j=1..10,2) // pour aller de 2 en 2 on ajoute un argument
1,9,25,49,81
5>>[s] // pour passer d'une suite entre ( ) a une liste entre [ ]
```

(giac/xcas)

```
s:=0;
L:=seq(t,t,0,9);
for j in L do //ne marche pas en francais
  s:=s+3^j;
end;
```

1.3 Fonctions

1.3.1 Définition et type de variables

En python et xcas il n'y a pas de déclaration de type pour les arguments d'une fonction, ni pour les valeurs de retour. On peut retourner n'importe quel objet (même une liste)

(python)

```
def discri(a,b,c):
   """Cette fonction calcule le discriminant de ax^2+bx+c""" # aide facultative
   d=b**2-4*a*c #En python, variables locales par defaut. (Sinon utiser global)
   return d
```

(python)

```
>>> discri(1,2,1)
0
>>> help(discri)
Help on function discri in module __main__:

discri(a, b, c)
    Cette fonction calcule le discriminant de ax^2+bx+c
```

(giac/xcas)

```
discri(a,b,c):={
  local d; //par defaut les variables sont globales mais un warning s'affiche
  d:=b^2-4*a*c;
  return(d);// ou bien: d;  le retour est la derniere evaluation
  }
```

(giac/xcas)

```
fonction discri(a,b,c)
  local d; //par defaut les variables sont globales mais un warning s'affiche
  d:=b^2-4*a*c;//ou bien b**2-4*a*c
  retourne(d);//ou bien: d; ou bien: retourne d;
ffonction
```

1.3.2 Arguments des fonctions et affectations en Python

(python)

```
def test1(b):
   b=b+1
   print("b dans la fonction",b)
   return b
```

(python)

```
>>> a=1
>>> test1(a)
('b dans la fonction', 2)
2
>>> print(a)
1
```

(python)

```
def test2(b):
   b=[0]
   print("b dans la fonction",b)
   return b
```

(python)

```
>>> a=[1]
>>> test2(a)
('b dans la fonction', [0])
[0]
>>> print(a)
[1]
>>> a=1
```

(python)

```
def test3(b):
   b[0]=b[0]+1
   print("b dans la fonction",b)
   return b
```

(python)

```
>>> a=[1]
>>> test3(a)
('b dans la fonction', [2])
[2]
>>> print(a)
[2]
```

1.3.3 Arguments des fonctions et affectations sous Xcas

(giac/xcas)

```
test1(b):={
  b:=b+1;
  print("b dans la fonction",b);
  return b;
};
```

(giac/xcas)

```
test2(b):={
  b:=[2];//il y a recopie locale de b
  print("b dans la fonction",b);
  return b;
}
```

(giac/xcas)

```
test3(b):={
  b[0]:=b[0]+1;//il y a tout de meme recopie locale de b
  print("b dans la fonction",b);
  return b;
}
```

En utilisant l'affectation sur place =< on obtient le même comportement qu'en Python. (ie que les listes sont des pointeurs, mais que l'argument d'une fonction est recopié localement et donc non modifié.)

(giac/xcas)

```
test4(b):={
  b=<[2];//affectation sur place
  print("b dans la fonction",b);
  return b;
}
```

(giac/xcas)

```
test5(b):={
  b[0]=<b[0]+1;//affectation sur place
  print("b dans la fonction",b);
  return b;
}
```

(giac/xcas)

```
4>> (a:=1),test1(a),a ;
"b dans la fonction",2
1,2,1
5>> (a:=[1]),test2(a),a ;
"b dans la fonction",[1]
[1],[2],[1]
6>> (a:=[1]),test3(a),a ;
"b dans la fonction",[2]
[1],[2],[1]
7>> (a:=[1]),test4(a),a ;
"b dans la fonction",[2]
[1],[2],[1] //a n'est pas modifie car c'etait l'argument.
8>> (a:=[1]);test5(a);a ;
"b dans la fonction",[2]
[2],[2],[2]  //l'affichage  a lieu apres evaluation de toute la ligne ce qui
explique le premier 2. Le contenu de la liste a  est modifie car on a modifie
une entree de la liste alors que l'argument de la fonction etait la liste.
```

Chapitre 2

Exemples d'algorithmes itératifs et récursifs

Dans ce chapitre on va mettre l'accent sur l'écriture des algorithmes et leur justification (l'algorithme se termine et produit le bon résultat).

2.1 Deux versions de l'algorithme d'Euclide

Proposition 2.1.1 *Soient : $a \in \mathbb{Z}$, $b \in \mathbb{Z}$. On a pour tout $m \in \mathbb{Z}$:*

$$PGCD(a, b) = PGCD(b, a - mb) \ .$$

On peut en particulier appliquer la proposition précédente au cas où $m = q$ est le quotient dans la division euclidienne de a par b.

Entrée: Deux entiers relatifs : a, b;
Sortie: Un entier pgcd de a et b;
Fonction PGCD(a, b) ;
si *b est nul* **alors**
  retourner a;
sinon
  $r \leftarrow a$ mod b (reste de la division euclidienne) ;
  retourner PGCD(b, r);
fsi

Algorithme 1: Euclide, forme récursive

Entrée: Deux entiers relatifs : a, b;
Sortie: Un entier pgcd de a et b;
Fonction PGCD(a, b) ;
tantque *b est non nul* **faire**
  $r \leftarrow a$ mod b (reste de la division euclidienne) ;
  $a \leftarrow b$;$b \leftarrow r$;
ftantque
retourner a;

Algorithme 2: Euclide, forme itérative

2.2 Algorithme d'Euclide étendu aux coefficients de Bezout

Proposition 2.2.1 *Soient : $a \in \mathbb{Z}$, $b \in \mathbb{Z}$, $d = PGCD(a, b)$. Il existe deux entiers u et v tels que $d = ua + vb$.*

Entrée: 2 entiers a et b
Sortie: Une liste de 3 entiers : (u, v, d) tels que $a.u + v.b = d$
Fonction Bezout(a, b);
$A \leftarrow (1, 0, a)$; //On notera A_i la i^{ieme} coordonnée de A ;
$B \leftarrow (0, 1, b)$; //On notera B_i la i^{ieme} coordonnée de B ;
$reste \leftarrow b$;
tantque *reste*$\neq 0$ **faire**
  $q \leftarrow$ le quotient de A_3 par $reste$;
  $temp \leftarrow A - q \times B$;
  $A \leftarrow B$;
  $B \leftarrow temp$;
  $reste \leftarrow B_3$;
ftantque
retourner A

Algorithme 3: Euclide étendu : version itérative

Proposition 2.2.2 *On considère deux entiers a, b et l'on pose : $r_0 = a$ et $r_1 = b$. On définit par récurrence tant que r_i est non nul : $r_{i+1} = r_{i-1} - q_i.r_i$ où q_i est le quotient de la division Euclidienne de r_{i-1} par r_i. 0n pose $u_0 = 1, u_1 = 0$ et $v_0 = 0, v_1 = 1$. Alors les suites (u_i, v_i) définies par*

$$u_{i+1} = u_{i-1} - q_i.u_i \text{ et } v_{i+1} = v_{i-1} - q_i.v_i$$

tant que r_i est non nul, vérifient la relation suivante :

$$u_i.a + v_i.b = r_i$$

pour toute les valeurs de i où elles sont définies.

Exercice 2.2.3 *Notons N le nombre d'itérations de l'algorithme d'Euclide du calcul de $d = pgcd(a, b)$ avec $0 < b < a$. Alors les suites $(u_i)_{N \geq i \geq 1}$ et $(v_i)_{N \geq i \geq 1}$ sont de signe alterné et $(|u_i|)_{N \geq i \geq 1}$ et $(|v_i|)_{N \geq i \geq 1}$ sont croissantes. On a de plus $u_{N+1} = (-1)^{N+1} \dfrac{b}{d}$.*

2.3 Exemples de coûts

2.3.1 Résumé des couts classiques

Modèle à coût fixe : Ex les flottants, les opérations dans $\mathbb{Z}/N\mathbb{Z}$. Modèle à coût bilinéaire :

Opérations dans $\mathbb{Z}$:	
$M \pm N$	$O(\sup(\log N, \log M))$
$M.N, M \leq N$	Méthode classique : $O(\log N. \log(M))$; T.F discrète : $O(\log N. \log^3(\log N))$
$M/N, M \leq N$	$O(\log M. \log(N/M))$
$N = a^n$	$O(\log n)$ multiplications, donc le coût est en : $O(n^2) = O((\log N)^2)$
$u \wedge v$, Bezout, $u, v \leq N$	$O((\log N)^2)$. Cas le pire et suite de Fibonacci

(NB : pour TF discrete, on peut ameliorer le $O(\log N. \log^3(\log N))$. Cf Knuth vol2 page 311)

Opérations dans $\mathbb{Z}/N\mathbb{Z}$	
$\pm$	$O(\log N)$
. ou /	$O((\log N)^2)$
a^n	$O((\log n.(\log N)^2)$

Pivot de Gauss sur une matrice de taille n : $O(n^3)$ multiplications et additions de coefficients. Symbole de Jacobi : même ordre de grandeur que le pgcd.

2.3.2 Euclide

Lemme 2.3.1 *Pour des entiers a, b tels que $0 < b < a$. Notons $d = pgcd(a, b)$, et (F_n) la suite de Fibonacci définie par $F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n$. Alors si l'algorithme d'Euclide pour caluler le pgcd de a et b se termine en N tours on a :*

$$a > d \cdot F_{N+2},\ b > d \cdot F_{N+1}$$

NB : Comme pour tout entier n on a $F_n = \dfrac{\phi^n - \phi^{-n}}{\sqrt{5}}$ où $\phi = \dfrac{1+\sqrt{5}}{2}$, c'est l'entier le plus proche de $\dfrac{\phi^n}{\sqrt{5}}$, et donc $F_{N+1} > \dfrac{\phi^N}{\sqrt{5}} > F_N$. Comme $\dfrac{1}{\log_2(\phi)} < 1.5$, on obtient alors

$$N \leq \frac{3}{2}\log_2(b) + 1$$

Proposition 2.3.2 *Soient a, b deux entiers naturels inférieurs à un entier M. Alors le nombre de tours effectué par l'algorithme d'Euclide itératif pour calculer le pgcd de a et b ou un couple de bezout est un $O(\log(M))$.*

On peut aussi montrer avec l'Exercice2.2.3 que si l'on prend en compte le cout des opérations dans $\mathbb{Z}$, le cout total est un $O((\log(M))^2)$.

2.3.3 Puissance rapide

Proposition 2.3.3 *Soit $(G, *)$ un groupe commutatif dont on note la loi multiplicativement. On peut calculer a^n en $O(\log(n))$ opérations $*$.*

Remarque 2.3.4 *On distinguera deux situations assez différentes selon que le coût de $x * y$ est indépendant des éléments x, y de G ou pas. Donnez 2 exemples de G dans chaque situation.*

Entrée: Un élément g de G et un naturel n.
Sortie: Un élément de G : g^n
Fonction puiss(g, n) ;
$u \leftarrow 1; v \leftarrow g$; //u,v : 2 variables locales ;
tantque $n > 0$ **faire**
  si n *pair* **alors**
    $v \leftarrow v * v$; $n \leftarrow n/2$;
  sinon
    $u \leftarrow u * v; v \leftarrow v * v; n \leftarrow (n-1)/2$;
  fsi
ftantque
retourner u;

Algorithme 4: Puissance rapide.