

Algorithmique et traduction pour xcas

Renée De Graeve

10 juillet 2008

Chapitre 1

Vue d'ensemble de xcas pour le programmeur

1.1 Installation de xcas

Le programme `xcas` est un logiciel libre écrit en C++, (disponible sous licence GPL). La version à jour se récupère sur :

`http://www-fourier.ujf-grenoble.fr/~parisse/giac.html` ou

`ftp://fourier.ujf-grenoble.fr/pub/hp48`

où l'on trouve le code source (`giac.tgz`) ou des versions précompilées pour Linux (PC ou ARM) ou Windows (`xcas.zip`, `xcas_user.tgz`, `xcas_root.tgz`, `xcas_ipaq.tgz`).

1.2 Éditer, sauver, exécuter un programme avec xcas

On édite un programme ou un script (i.e. une suite de commandes séparées par des `;`) avec son éditeur préféré : on peut écrire, dans un même fichier, la définition de plusieurs fonctions séparées par des points virgules (`;`) (que l'on sauve par exemple sous le nom de `bidon`), puis dans `xcas` on tape `:read("bidon")` ; et cela a pour effet, de compiler les différentes fonctions de `bidon`, de les mettre comme réponse (avec `Success..` dans la zone des résultats intermédiaires pour indiquer les fonctions valides).

En rééditant le programme, ou le script, avec son éditeur préféré, on peut le corriger, le sauver sous un autre nom etc..., mais il est préférable de le recopier dans un éditeur de programmes (que l'on ouvre avec `Alt+p`) pour cela :

- on écrit directement le programme (ou le script), dans l'éditeur de programmes,
- on utilise le menu `Fich` sous-menu `Charger` de l'éditeur de programmes, si le programme est dans un fichier,
- on le recopie avec la souris, si le programme est dans la ligne de commande (par exemple après avoir fait `Charger` du menu `Fich` de la session) ou si le programme est dans son éditeur préféré,

En effet, depuis l'éditeur de programmes, on peut tester facilement si le programme est syntaxiquement correct grâce au bouton `OK` : la ligne où se trouve la faute de syntaxe est indiquée dans la fenêtre des messages du bandeau général. On cor-

4 CHAPITRE 1. VUE D'ENSEMBLE DE XCAS POUR LE PROGRAMMEUR

rige les fautes. Quand le programme est syntaxiquement correct, il y a `Success compiling ...` dans la fenêtre des messages du bandeau général. On peut alors exécuter le programme dans une ligne de commande.

Vous sauvez le programme avec le bouton `save` de l'éditeur de programmes sous le nom que vous voulez lui donner (ce nom s'inscrit alors à côté du bouton `save` de l'éditeur de programmes. Si vous voulez lui donner un autre nom il faut le faire avec le menu `Fich` sous-menu `Sauver` comme de l'éditeur de programmes.

1.3 Débugger un programme avec xcas

Vous avez par exemple un programme syntaxiquement correct, mais qui ne fait pas ce qu'il devrait faire, il faut donc le corriger. Pour utiliser le débogueur, il faut que ce programme soit syntaxiquement correct.

Avec le débogueur, on a la possibilité d'exécuter le programme au pas à pas (`sst`), ou d'aller directement (`cont`) à une ligne précise marquée par un `breakpoint` (`break`), de voir (`voir`) les variables que l'on désire surveiller, d'exécuter au pas à pas les instructions d'une fonction utilisateur (`dans`), ou de sortir brutalement du débogueur (`tuer`).

On tape : `debug(nom _du_programme(valeur_des_ arguments))`.

Il faut bien sûr que le programme soit validé : si ce n'est pas le cas, on tape, `read("toto")` si `toto` est le nom du fichier où se trouve ce programme.

Par exemple, si `pgcd` a été validé, on tape :

```
debug(pgcd(15, 25))
```

L'écran du débogueur s'ouvre : il est formé par trois écrans :

- dans l'écran du haut, le programme est écrit et la ligne en surbrillance sera exécutée grâce au bouton `sst`.

- dans l'écran du bas, coté gauche :

on trouve les points d'arrêts, le numéro de la ligne du curseur et la dernière valeur renvoyée. Les points d'arrêts permettent d'aller directement à un point précis. On marque les points d'arrêts grâce au bouton `break` ou à la commande `breakpoint` d'arguments le nom du programme et le numéro de la ligne ou l'on veut un point d'arrêt : par exemple `breakpoint(pgcd, 3)`. Pour faciliter son utilisation, il suffit de cliquer sur la ligne où l'on veut le point d'arrêt pour avoir : `breakpoint` dans la ligne de commande, avec le nom du programme et le bon numéro de ligne, puis de valider la commande. Il suffit donc de cliquer et de valider !

- dans l'écran du bas, coté droit, on voit l'évolution des valeurs que l'on a choisit de voir grâce au bouton `voir` qui écrit la commande `watch`. On tape alors, les arguments de `watch` qui sont les noms des variables que l'on veut surveiller, par exemple : `watch(a, b, r)` et on valide la commande.

Remarques :

On sera obligé, pour réutiliser d'autres points d'arrêts, d'effacer les points d'arrêts utilisés précédemment avec la commande `rmbreakpoint` qui a les mêmes arguments que `breakpoint`. Là encore, pour faciliter son utilisation, il suffit de cliquer sur la ligne où l'on veut enlever le point d'arrêt pour avoir : `rmbreakpoint`

dans la ligne de commande, avec le nom du programme et le bon numéro de ligne (Attention si il n'y a pas de point d'arrêt à cet endroit il en mettra un !).

Pour voir d'autres variables il faut aussi effacer les variables désignées précédemment avec la commande `rmwatch` qui a les mêmes arguments que `watch`.

Lorsqu'une instruction du programme utilise une fonction définie précédemment, le bouton `dans` permet d'exécuter aussi pas à pas les instructions de cette fonction (`sst_in`).

Le bouton `tuer` permet d'interrompre le debugage (on sort de l'écran du debugger), et, le bouton `cont` permet d'aller directement jusqu'au prochain point d'arrêt ou à la fin du programme et de sortir du debugger.

1.4 Présentation générale des instructions avec xcas

Les commentaires sont des chaînes de caractères, ils sont précédés de `//` ou sont parenthésés par `/* */`

Les variables sont les endroits où l'on peut stocker des valeurs, des nombres, des expressions, des objets.

Le nom des variables est formé par une suite de caractères et commence par une lettre : attention on n'a pas droit aux mots réservés ...ne pas utiliser par exemple la variable `i` dans un `for` car `i` représente le nombre complexe de module 1 et d'argument $\frac{\pi}{2}$.

Une action ou `bloc` est une séquence d'une ou plusieurs instructions.

Quand il y a plusieurs instructions il faut les parenthéser avec `{ }` et séparer les instructions par un point virgule `;`

Un `bloc` est donc parenthésé par `{ }` et commence éventuellement par la déclaration des variables locales (`local . . .`).

En effet les variables locales doivent être déclarées au début d'un bloc par le mot réservé `local` puis on met les noms des variables séparés par des virgules `,`. Ces variables locales peuvent être initialisées lors de leur déclaration : l'initialisation des variables locales faites dans la ligne `local` se fait en utilisant le contexte d'évaluation global, par exemple :

```
f() := {
  local (n:=0), (d:=n+1);
  return d;
}
```

`f()` renvoie la valeur de `n+1` ou `n` est global et non 1. Il faut initialiser `d` après la déclaration locale pour utiliser le contexte local en tapant :

```
f() := {
  local (n:=0), d;
  d:=n+1;
  return d;
}
```

et alors `f()` renvoie 1.

6 CHAPITRE 1. VUE D'ENSEMBLE DE XCAS POUR LE PROGRAMMEUR

Attention Les variables locales sont toujours affectées : on ne définira donc pas, les variables formelles, avec `local`.

Voici des exemples :

- variables locales et variables formelles,

Voici comme exemple le programme qui donne la valeur de la suite de Fibonacci définie par $u_0 = u0, u_1 = u1, u_{n+2} = u_{n+1} + u_n$. Dans ce programme on utilise les variables formelles x, A, B qui doivent être purgées.

On tape :

```
u(n,u0,u1):={
local L,a,b;
//verifier que A,B,x ne sont pas affectées
[a,b]:=solve(x^2-x-1,x);
L:=linsolve([A+B=u0,A*a+B*b=u1],[A,B]);
return normal(L[0]*a^n+L[1]*b^n);
};
```

On tape :

```
u(3,0,1)
```

On obtient :

```
2
```

Dans ce programme, les variables x, A, B ne doivent pas être déclarées comme variables locales car ce sont des variables formelles : il ne faut donc pas tenir compte lors de la compilation du warning : `// Warning : x A B declared as global variable(s) compiling u`

- variables locales internes à un bloc,

Voici comme exemple le programme de la fonction qui donne le quotient et le reste de la division euclidienne de 2 entiers (c'est la fonction `iquorem` de `xcas`):

```
idiv2(a,b):={
local (q:=0),(r:=a);
if (b!=0) {
q:=iquo(a,b);
r:=irem(a,b);
}
return [q,r];
};
```

Voici le programme de la même fonction mais avec les variables locales internes au bloc du `if` :

```
idiv2(a,b):={
if (b==0) {return [b,a];}
if (b!=0) {
local q,r;
q:=iquo(a,b);
r:=irem(a,b);
return [q,r];
}
};
```

ou encore avec les variables locales internes au bloc du `else` :

```
idiv2(a,b):={
```

```

    if (b==0) {return [b,a];}
    else {
        local q,r;
        q:=iquo(a,b);
        r:=irem(a,b);
        return [q,r];
    }
};

```

Les paramètres sont mis après le nom de la fonction entre parenthèses :

par exemple `f(a,b) :=...`

Ces paramètres sont initialisés lors de l'appel de la fonction et se comportent comme des variables locales.

L'affectation se fait avec `:=` (par exemple `a :=2 ; b :=a ;`).

Les entrées se font par passage de paramètres ou avec `input`.

Il n'y a pas de distinction entre programme et fonction : les sorties se font en mettant le nom de la variable à afficher (ou la séquence des variables à afficher ou entre crochets les variables à afficher séparées par une virgule) précédé du mot réservé `return`.

Les tests et boucles ont une syntaxe similaire au langage C++.

```

testif(a,b):={
if ((a==10) or (a<b))
    b:=b-a;
else
    a:=a-b;
return [a,b];
};

```

```

testfor1(a,b):={
local j,s:=0;
for (j:=a;j<=b;j++)
    s:=s+1/j^2;
return s;
};

```

```

testfor2(a,b):={
local j,s:=0;
for (j:=b;j>=a;j--)
    s:=s+1/j^2;
return s;
};

```

```

testwhile(a,b):={
while ((a==10) or (a<b))

```

8 CHAPITRE 1. VUE D'ENSEMBLE DE XCAS POUR LE PROGRAMMEUR

```
    b:=b-a;  
return [a,b];  
};
```

Un exemple : le PGCD d'entiers

- Version itérative

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

-Version récursive

```
pgcdr(a,b) := {  
    if (b==0) return a;  
    return pgcdr(b,irem(a,b));  
};
```

ou

```
pgcdr(a,b) := if (b==0) return a;  
               else return pgcdr(b,irem(a,b));
```


Chapitre 2

Les différentes instructions

`xcas` propose un mode de compatibilité avec `Maple`, `MuPAD` et la `TI89/92` : pour cela, il suffit de le spécifier dans `Prog style` du menu de configuration du `cas` (bouton rouge `cas`). On peut choisir, en cliquant sur la flèche située à côté de `Prog style` : `xcas` ou `Maple` ou `MuPAD` ou `TI89/92`.

On a aussi la possibilité de traduire un programme écrit en `Maple` ou en `MuPAD` ou en `TI89/92` en choisissant `Traduire` du menu `Fich` d'un éditeur de programmes. On présente ici le mode natif le plus proche de la syntaxe `C`

2.1 Les commentaires

2.1.1 Traduction Algorithmique

Il faut prendre l'habitude de commenter les programmes. En algorithmique un commentaire commence par `//` et se termine par un passage à la ligne.

Exemple :

```
//ceci est un commentaire
```

2.1.2 Traduction xcas

Avec `xcas` un commentaire commence par `//` et se termine par un passage à la ligne.

Exemple :

```
//ceci est un commentaire
```

2.1.3 Traduction MapleV

Un commentaire commence par `#` et se termine par un passage à la ligne.

Exemple :

```
# ceci est un commentaire
```

2.1.4 Traduction MuPAD

Un commentaire est entouré de deux `#` ou commence par `/*` et se termine par `*/`.

Exemple :

```
# ceci est un commentaire #
/* ceci est un commentaire */
```

2.1.5 Traduction TI89 92

Le commentaire commence par © (F2 9) et se termine par un passage à la ligne.

Exemple :

```
© ceci est un commentaire
```

2.2 Les variables

2.2.1 Leurs noms

Ce sont les endroits où l'on peut stocker des valeurs, des nombres, des expressions.

Avec `xcas` les noms des variables ou des fonctions commencent par une lettre et sont formés par des lettres ou des chiffres.

Par exemple :

`azertyuiop := 2` met la valeur 2 dans la variable `azertyuiop`

`azertyuio? := 1` renvoie un message d'erreur car `azertyuio?` contient ? qui n'est pas une lettre.

Étant donné que `xcas` fait du calcul formel il faut quelquefois purger une variable pour qu'elle redevienne formelle.

On tape :

```
x := 2; x contient 2 et n'est pas formelle.
```

On tape :

`purge(x)` : cela renvoie la valeur `x` si `x` est affecté et `x` redevient une variable formelle, et, si `x` n'est pas affecté `purge(x)` renvoie "`x not assigned`".

Si Avec `MapleV` et `MuPAD`, un nom doit commencer par une lettre, ne pas contenir d'espace, de symbole opératoire (+, -, ...) et ne pas être un mot réservé comme `D`, `I`, ... pour `MapleV`.

On peut utiliser des noms ayant plus de 8 caractères.

Pour `TI89/92` un nom doit commencer par une lettre, ne pas contenir d'espace, de symbole opératoire (+, -, ...), ne pas être un mot réservé et ne doit pas avoir plus de 8 caractères.

2.2.2 Notion de variables locales

Pour `xcas` il faut définir les variables locales en début de programme en écrivant :

```
local a, b, c ;
```

Ces variables peuvent être initialisées : `local a, (b := 1), c ;`

Pour `MapleV`, il faut définir les variables locales en début de programme en écrivant :

```
local a, b
```

Pour MuPAD, il faut définir les variables locales en début de programme en écrivant :

```
local a,b
```

Pour la TI89/92 il faut définir les variables locales en début de programme en écrivant :

```
:local a,b
```

2.3 Les paramètres

Quand on écrit une fonction il est possible d'utiliser des paramètres.

Par exemple si A et B sont les paramètres de la fonction PGCD on écrit :

```
PGCD(A,B)
```

Ces paramètres se comportent comme des variables locales, la seule différence est qu'ils sont initialisés lors de l'appel de la fonction. L'exécution se fait en demandant par exemple : PGCD(15, 75)

2.3.1 Traduction xcas

Avec xcas on peut définir une fonction :

- en lui donnant un nom :

on met alors le nom de la fonction puis, entre parenthèses et séparés par une virgule, le nom des paramètres, par exemple :

```
addition(a,b) :=a+b;
```

L'exécution se fera alors en tapant : addition(2, 5).

- en mettant le nom des paramètres entre parenthèses puis -> puis entre des accolades le corps de la fonction, par exemple :

```
(a,b)->a+b ;.
```

cette façon d'écrire une fonction peut être utile quand on doit mettre une fonction comme argument d'une commande par exemple :

```
makelist((j)->j^2+1,1..3) ou makelist((j)->j^2+1,1..9,2).
```

Remarque Le langage de xcas est fonctionnel puisque on peut passer des programmes ou des fonctions en paramètre.

Autre exemple :

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

ou encore :

```
pgcd:=(a,b)-> {
  local r;
  while (b!=0) {
    r:=irem(a,b);
    a:=b;
    b:=r;
  }
  return(a);
};
```

L'exécution se fera alors en tapant : `pgcd(15, 75)`.

2.3.2 Traduction MapleV

```
PGCD :=proc(A,B)
....
....
end :
```

Attention

Ces paramètres NE se comportent PAS comme des variables locales : ils sont initialisés lors de l'appel de la fonction mais ne peuvent pas être changés au cours de la procédure. L'exécution se fait en demandant par exemple : `PGCD(15, 75)`

2.3.3 Traduction MuPAD

```
PGCD :=proc(A,B)
begin
....
end_proc :
```

Ces paramètres se comportent comme des variables locales, la seule différence est qu'ils sont initialisés lors de l'appel de la fonction. L'exécution se fait en demandant par exemple : `PGCD(15, 75)`

2.3.4 Traduction TI89/92

Pour les TI 89/92 on met le nom des paramètres dans le nom de la fonction par exemple :

```
:addition(a,b)
:pgcd(a,b)
```

2.4 Les Entrées

2.4.1 Traduction Algorithmique

Pour que l'utilisateur puisse entrer une valeur dans la variable A au cours de l'exécution d'un programme, on écrira, en algorithmique :

```
saisir A
```

Et pour entrer des valeurs dans A et B on écrira :

```
saisir A,B
```

2.4.2 Traduction xcas

On écrit :

```
input(A) ;
```

2.4.3 Traduction MapleV

On peut utiliser :

```
A := readline() ; ou
```

```
A :=readstat('A=?') ;
```

2.4.4 Traduction MuPAD

```
input("A=", A)
```

```
input("A=", A, "B=", B )
```

2.4.5 Traduction TI89/92

```
:Prompt A
```

```
:Prompt A,B
```

ou encore :

```
:Input "A=", A
```

2.5 Les Sorties

2.5.1 Traduction Algorithmique

En algorithmique on écrit :

```
Afficher "A=", A
```

2.5.2 Traduction xcas

On écrit :

```
print(A) ;
```

2.5.3 Traduction MapleV

```
print('A=' . A) :
```

On préférera ' (accent grave) à " (guillemet), comme délimiteur de chaîne, car le premier n'est pas affiché, alors que le deuxième l'est.

2.5.4 Traduction MuPAD

```
print("A=", A)
```

Il fait savoir que lorsqu'une procédure est appelée, la suite d'instructions entre `begin` et `end_proc` est exécutée, et le résultat est égal au résultat de la dernière évaluation.

2.5.5 Traduction TI89/92

```
:Disp "A=", A
:ClrIO efface l'écran.
:ClrHome efface l'écran pour la TI 83+.
:Pause arrête le programme (on appuie sur ENTER pour reprendre l'exécution).
```

2.6 La séquence d'instructions ou action

Une action est une séquence d'une ou plusieurs instructions.

En langage algorithmique, on utilisera l'espace ou le passage à la ligne pour terminer une instruction.

2.6.1 Traduction xcas

Avec `xcas` la séquence d'instructions est parenthésées par `{ }` et est appelée un bloc et `;` termine chaque instruction. Seule la dernière instruction génère la réponse. Si on veut des sorties intermédiaires il faudra le faire à l'aide de la commande `print` et ces sorties se feront alors avant la réponse en écriture bleue.

Par exemple si on écrit :

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

Alors `pgcd(15, 25)` renvoie 5 comme réponse et :

```
r :15
r :10
r :5
r :0
```

s'écrivent en bleu dans un écran qui se met avant la réponse.

En géométrie aussi, si la dernière instruction est géométrique, elle génère une sortie dans un écran géométrique. Les instructions géométriques intermédiaires se feront dans l'écran géométrique `DispG`.

Par exemple si on écrit :

```
pgcdg(a,b):={
  local r;
  while (b!=0){
    r:=irem(a,b);
    point(a+i*b);
    segment(a+i*b,b+i*r);
    a:=b;
    b:=r;
  }
  return(A:=point(a+i*b));
};
```

Un écran de graphique s'ouvre avec le point A de coordonnées (0;5) et les différents segments et les points (sans la lettre A) se trouvent dans l'écran géométrique `DispG` (que l'on ouvre avec la commande `DispG()` pour tout voir, vous devez changer la configuration avec le bouton `cfg` de `DispG`).

2.6.2 Traduction MapleV

: ou ; indique la fin d'une instruction.

Une instruction terminée par point-virgule (;) génère une sortie et celle terminée par deux points (:) n'en génère pas.

2.6.3 Traduction MuPAD

Le : ou ; est un séparateur d'instructions.

Le ; génère une sortie alors que le : n'en génère pas.

2.6.4 Traduction TI89/92

: indique la fin d'une instruction. Il faut noter qu'à chaque passage à la ligne le : est mis automatiquement.

2.7 L'instruction d'affectation

L'affectation est utilisée pour stocker une valeur ou une expression dans une variable.

2.7.1 Traduction Algorithmique

En algorithmique on écrira par exemple :

$3 \rightarrow A$

$2 * A \rightarrow B$

pour stocker 3 dans A, $2 * A$ (c'est à dire 6) dans B

2.7.2 Traduction xcas

Avec `xcas` on écrira :

`a :=3 ;`

$b := 2 * a$; En tapant le nom d'une variable dans `xcas` on peut voir le contenu de cette variable. Si on tape le nom d'une fonction on verra la définition de la fonction.

Attention

On peut écrire :

$(a, b) := (1, 2)$ ou $(a, b) := [1, 2]$ qui est équivalent à $a := 1 ; b := 2$ mais

$(a, b) := (a+b, a-b)$ ou $(a, b) := [a+b, a-b]$ est équivalent à $c := a ; a := a+b ; b := c-b$

Donc si on tape :

$(a, b) := (1, 2) ; (a, b) := (a+b, a-b)$

On obtient :

3 dans a et -1 dans b

Donc si on tape :

$a := 1 ; (a, b) := (2, a)$ On obtient :

2 dans a et 1 dans b

mais `purge(a)` ; $(a, b) := (2, a)$

On obtient :

2 dans a et 2 dans b

2.7.3 Traduction Maple

Avec `Maple` on écrit :

$b := 2 * a$ pour stocker $2 * a$ dans b .

2.7.4 Traduction MuPAD

Avec `MuPAD` on écrit :

$b := 2 * a$ pour stocker $2 * a$ dans b .

2.7.5 Traduction TI89/92

On écrit $2 * A \rightarrow B$ pour stocker $2 * A$ dans B .

2.8 L'instruction pour faire des hypothèses sur une variable formelle

L'affectation est utilisée pour stocker une valeur ou une expression dans une variable.

2.8.1 Traduction Algorithmique

En algorithmique on écrira par exemple :

`supposons(n, entier)`

`supposons(a > 2)`

2.8.2 Traduction xcas

Avec `xcas` on écrira :

`assume(n, integer) ;`

2.9. L'INSTRUCTION POUR CONNAITRE LES CONTRAINTES D'UNE VARIABLE 17

```
assume(a>2) ;
```

Il faut noter que si on écrit :

```
assume(a=2) ;
```

ou encore

```
assume(a :=2) ;
```

cela a pour effet d'ouvrir l'écran géométrique en mettant en haut et à droite un curseur noté a . En effet, cela veut dire que l'on va faire une figure de géométrie en donnant à a la valeur 2, mais que les différents calculs se feront avec a formelle. Bien sur la valeur donnée à a pourra être modifiée à l'aide du curseur et ainsi modifier la figure selon les valeurs de a .

2.8.3 Traduction Maple

Avec Maple on écrit :

```
assume(n, integer) ;
```

```
assume(a>2) ;
```

2.8.4 Traduction MuPAD

Avec MuPAD on écrit :

```
assume(n, integer) ;
```

```
assume(a>2) ;
```

2.8.5 Traduction TI89/92

Il n'y a pas d'hypothèse en mode TI.

2.9 L'instruction pour connaître les contraintes d'une variable

2.9.1 Traduction Algorithmique

En algorithmique on écrira par exemple :

```
domaine(A)
```

2.9.2 Traduction xcas

Avec xcas on écrira :

```
about(a) ou
```

```
assume(a)
```

2.9.3 Traduction Maple

Avec Maple on écrit :

```
about(a) ou
```

```
assume(a)
```

2.9.4 Traduction MuPAD

Avec MuPAD on écrit :

```
getprop(a)
```

2.9.5 Traduction TI89/92

Il n'y a pas d'hypothèse en mode TI.

2.10 Les instructions conditionnelles

2.10.1 Traduction Algorithmique

Si *condition* alors *action* fsi

Si *condition* alors *action1* sinon *action2* fsi

Exemple :

Si $A = 10$ ou $A < B$ alors $B-A \rightarrow B$ sinon $A-B \rightarrow A$ fsi

2.10.2 Traduction xcas

```
if (condition) {action;}
if (condition) {
action1;
} else {
action2;
}
```

Exemples

```
if ((a==10) or (a<b)) {b :=b-a;} else {a :=a-b;}
```

```
essaiif(a,b):={
  if ((a==10) or (a<b)) {
    b:=b-a;
  } else {
    a:=a-b;
  }
  return([a,b]);
};
```

```
idiv2(a,b):={
  local (q:=0),(r:=a);
  if (b!=0){
    q:=iquo(a,b);
    r:=irem(a,b);
  }
  return([q,r]);
};
```

Avec xcas, on peut aussi utiliser un "case" qui se traduit par :

```

switch (<nom_de_variable>){
case val_de_la_variable : {
.....
break;
}
case val_de_la_variable : {
.....
break;
} default : {
...
}
}

```

Exemple :

```

s(a):={
local r;
switch(a) {
case 1 :{
r:=1;
break;
}
case 2 :{
r:=-1;
break;
}
default :{
r:=0;
}
}
return r;
}

```

2.10.3 Traduction MapleV

```

si ... alors
if <condition> then <action> fi;
si ... alors ... sinon ...
if <condition> then <action1> else <action2> fi;
Le schéma général à n cas :
if <condition_1> then <action_1>
elif <condition_2> then <action_2>
... elif <condition_n-1> then <action_n-1> else <action_n>
fi;

```

2.10.4 Traduction MuPAD

```

if <condition> then <action> end_if
if <condition> then <action1> else <action2> end_if

```

Exemple :

```
if a = 10 or A < B then b :=b-a else a :=a-b end_if
```

Lorsque il y a plusieurs if else à la suite on écrit elif au lieu de else if.

2.10.5 Traduction TI89/92

Si ..alors

```
:If condition Then : action : EndIf
```

Si..alors sinon

```
:If condition Then : action1 : Else : action2 : EndIf
```

Exemple :

```
:If A = 10 or A < B Then : B-A->B : Else : A-B->A : EndIf
```

2.11 Les instructions "Pour"

2.11.1 Traduction Algorithmique

```
pour I de A a B faire action fpour
```

```
pour I de A a B (pas P) faire action fpour
```

2.11.2 Traduction xcas

Attention on n'a pas le droit d'employer la variable *i* puisque *i* représente le nombre complexe de module 1 et d'argument $\pi/2$.

```
for (j :=1 ; j<=b ; j :=j+1) {action;} ou encore
```

```
for (j :=1 ; j<=b ; j :=j++) {action;} 
```

```
for (j :=1 ; j<=b ; j :=j+p) {action;} 
```

Exemples

```
essaifor(a,b):={
  local s:=0;
  for (j:=a; j<b+1; j++){
    s:=s+1/j^2;
  }
  return(s);
};
```

```
essaiford(a,b):={
  local s:=0;
  for (j:=b; j>a-1; j--){
    s:=s+1/j^2;
  }
  return(s);
};
```

2.11.3 Traduction MapleV

Voici la syntaxe exacte :

```
for <nom> from <expr> by <expr> to <expr>
```

```
do <action> od;
```

Par exemple

```
for i from 1 to n do <action :> od
for i from 1 by p to n do <action :> od
```

2.11.4 Traduction MuPAD

```
for i from a to b do <action> end_for
for i from b downto a do <action> end_for
for i from a to b step p do <action> end_for
```

Vous pouvez aussi ouvrir le menu MuPAD sous menu Shapes et sélectionner for.

2.11.5 Traduction TI89 92

```
:For I,A,B : action : EndFor
:For I,A,B,P : action : EndFor
```

2.12 L'instruction "Tant que"

2.12.1 Traduction Algorithmique

```
tantque condition faire action ftantque
```

2.12.2 Traduction xcas

```
while (condition) {
action;
}
```

Exemple

```
essaiwhile(a,b):={
  while ((a==10) or (a<b)) {
    b:=b-a;
  }
  return([a,b]);
};
```

2.12.3 Traduction MapleV

```
while <condition> do <action> od:
```

2.12.4 Traduction MuPAD

```
while <condition> do <action> end_while
```

Vous pouvez aussi ouvrir le menu MuPAD sous menu Shapes et sélectionner while.

2.12.5 Traduction TI89/92

```
:While condition : action : EndWhile
```

2.13 Les conditions ou expressions booléennes

Une condition est une fonction qui a comme valeur un booléen, à savoir elle est soit `vraie` soit `fausse`.

2.13.1 Les opérateurs relationnels

Traduction Algorithmique

Pour exprimer une condition simple on utilise en algorithmique les opérateurs :
`= > < ≤ ≥ ≠`

Traduction xcas

Ces opérateurs se traduisent pour `xcas` par :

`== > < <= >= !=`

Attention pour `xcas` l'égalité se traduit comme en langage C par : `==`

Traduction MapleV, MuPAD, TI89/92

Ces opérateurs se traduisent pour `MapleV`, `MuPAD` et `TI89/92` par :

`= > < <= >= <>`

2.13.2 Les opérateurs logiques

Pour traduire des conditions complexes, on utilise en algorithmique, les opérateurs logiques :

`ou et non`

Pour `xcas`, ces opérateurs se traduisent par :

`or and not` ou encore par `|| && !`

Pour `MapleV`, `MuPAD` et `TI89/92`, ces opérateurs se traduisent par :

`or and not`

2.14 Les fonctions

Dans une fonction on ne fait pas de saisie de données : on utilise des paramètres qui seront initialisés lors de l'appel.

Les entrées se font donc par passage de paramètres.

Dans une fonction on veut pouvoir réutiliser le résultat :

en algorithmique, on n'utilise pas la commande `affichage` mais la commande `retourne`.

Si la fonction est utilisée seule, sa valeur sera affichée.

2.14.1 Traduction Algorithmique

On écrit par exemple en algorithmique :

```
fonction addition(A,B)
retourne A+B
ffonction
```

Cela signifie que :

- Si on fait exécuter la fonction, ce qui se trouve juste après `retourne` sera la valeur de la fonction, mais les instructions qui suivent `retourne` seront ignorées (`retourne` fait sortir immédiatement de la fonction).
- On peut utiliser la fonction dans une expression ou directement dans la ligne de commande et, dans ce cas, sa valeur sera affichée.

2.14.2 Traduction xcas

```
retourne se traduit par return
addition(a,b) := {
  return(a+b) ;
}
```

Remarque : `return` fait sortir immédiatement de la fonction.

2.14.3 Traduction MapleV

```
retourne se traduit par RETURN
addition:= proc(a,b)
RETURN(a+b) ;
end:
```

Remarque : `RETURN` fait sortir immédiatement de la fonction.

2.14.4 Traduction MuPAD

```
retourne se traduit MuPAD par return :
addition:=proc(a,b)
begin
return(a+b)
end_proc;
```

Remarque : `return` fait sortir immédiatement de la fonction.

2.14.5 Traduction TI89 92

```
:addition(a,b)
:Func
:Return a+b
:EndFunc
```

Remarque : `Return` fait sortir immédiatement de la fonction.

2.15 Les listes

2.15.1 Traduction Algorithmique

On utilise les `{ }` pour délimiter une liste.

Attention !!!

En algorithmique, on a choisi cette notation car c'est celle qui est employée par les calculatrices...à ne pas confondre avec la notion d'ensemble en mathématiques : dans un ensemble l'ordre des éléments n'a pas d'importance mais dans une liste l'ordre est important...

Par exemple `{ }` désigne la liste vide et `{ 1, 2, 3 }` est une liste de 3 éléments.

`concat` sera utilisé pour concaténer 2 listes ou une liste et un élément ou un élément et une liste :

`{ 1, 2, 3 } -> TAB`

`concat (TAB, 4) -> TAB` (maintenant `TAB` désigne `{ 1, 2, 3, 4 }`)

`TAB [2]` désigne le deuxième élément de `TAB` ici 2.

2.15.2 Traduction xcas

Avec `xcas` il existe différentes notions : la liste ou le vecteur, la séquence et l'ensemble.

- Les listes et les vecteurs

Une liste ou un vecteur est délimité par `[]` et les éléments situés à l'intérieur des crochets sont séparés par des virgules.

- Les séquences

Une séquence n'a pas de délimiteurs, les éléments sont séparés par des virgules, mais on doit parfois parenthéser par `()` (on écrit `1, 2, 3, 4` ou `(1, 2, 3, 4)` ou encore `seq[1, 2, 3]`).

- Les ensembles

Un ensemble est délimité par `%{ % }` et les éléments situés à l'intérieur des délimiteurs sont séparés par des virgules (on écrit `%{ 1, 2, 3, 4% }` ou encore `set[1, 2, 3, 4]`).

Les fonctions pour les listes

La liste vide est désignée par `[]` et la séquence vide par `NULL`.

La commande `makelist` permet de fabriquer une liste à partir d'une fonction f . Les paramètres sont : la fonction, l'intervalle de variation de l'argument de f et le pas de son incrémentation.

On tape :

`f(x) := x2`

`l := makelist(f, 2, 10, 3)` ou encore

`l := makelist(x -> x2, 2, 10, 3)` ou encore

`l := makelist(x -> x2, 2..10, 3)` ou encore

`l := makelist(sq, 2, 10, 3)`

On obtient `l = [4, 25, 64]`

Pour avoir une liste constante on peut taper par exemple :

`l := makelist(3, 1..5)` on obtient `l = [3, 3, 3, 3, 3]`

On peut écrire `l := [1, 2, 3]`.

Attention Les éléments sont indicés à partir de zéro (contrairement à Maple ou MuPAD qui commencent les indices à 1) :

dans l'exemple `l[0]` vaut 1.

Si on tape ensuite :

`l[0] := 4` : après cette instruction `l` sera la liste `[4, 2, 3]`.

La commande `append(l, elem)` permet de mettre à la fin d'une liste `l`, un élément (ou une liste) `elem`.

La commande `prepend(l, elem)` permet de mettre au début d'une liste `l`, un élément (ou une liste) `elem`.

La commande `tail(l)` renvoie la liste `l` privée de son premier élément et, la commande `head(l)` renvoie le premier élément de la liste.

La commande `concat` permet de concaténer deux listes ou une liste et un élément.

La commande `augment` permet de concaténer deux listes.

La commande `size` ou `nops` renvoie la longueur d'une liste ou d'une séquence.

Les fonctions pour les séquences

La commande `op` transforme une liste en une séquence.

On a la relation :

si `l` est une liste `op(l)` est une séquence.

Exemple :

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

`s := op(l)` (`s` est la séquence `1, 2, 3`),

`a := [s]` (`a` est la liste `l` égale à `[1, 2, 3]`),

Pour concaténer deux séquences il suffit d'écrire :

`s1 := (1, 2, 3)`

`s := (s1, 4, 5)`

ou encore

`s := s1, 4, 5` car la virgule (,) est prioritaire par rapport à l'affectation (`:=`).

La commande `seq` permet de fabriquer une séquence à partir d'une expression. Les paramètres sont : l'expression, la variable=l'intervalle de variation (le pas d'incrément de la variable est toujours 1).

On tape :

`seq(j^2, j=1..4)`

On obtient ;

`(1, 4, 9, 16)`

On peut aussi utiliser `$` qui est une fonction infixée.

On tape :

`(j^2) $ (j=1..4)`

On obtient ;

`(1, 4, 9, 16)`

Les fonctions pour les ensembles

Soit `A := set[1, 2, 3, 4]` ; `B := set[3, 4, 4, 6]` ; `union(A, B)` désigne l'union de `A` et `B`,

`intersect(A, B)` désigne l'intersection de A et B,

`minus(A, B)` désigne la différence de A et B.

On a :

`union(A, B) = set[1, 2, 3, 4, 5, 6]`

`intersect(A, B) = set[3, 4]`

`minus(A, B) = set[1, 2]`

2.15.3 Traduction MapleV

En Maple on utilise `{ }`, comme en mathématiques, pour représenter un **ensemble**.

Exemple : `De := {1, 2, 3, 4, 5, 6}`

Dans un ensemble, l'ordre n'a pas d'importance. La répétition est interdite.

Pour délimiter une **liste**, on utilise `[]`.

L'ordre est pris en compte, la répétition est possible.

Exemple : `[Pile, Face, Pile]`

Une **séquence** est une suite d'objets, séparés par une virgule.

Si C est un ensemble ou une liste, `op(C)` est la séquence des objets de C.

`nops(C)` est le nombre d'éléments de C.

Ainsi, si L est une liste, `{op(L)}` est l'ensemble des objets (non répétés) de L.

Exemples

On écrit :

`S := NULL` : (pour la séquence vide)

`S := S, A` : (pour ajouter un élément à S)

Une liste est une séquence entourée de crochets :

`L := [S]` :

`L[i]` est le ième élément de la liste L.

On peut aussi revenir à la séquence : `S := op(L)` :

2.15.4 Traduction MuPAD

Une liste est une suite d'expressions entre un crochet ouvrant `[` et un crochet fermant `]`.

`[]` désigne la liste vide.

Exemple :

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

`nops(l)` renvoie le nombre d'éléments de la liste l.

`l[1]` ou `op(l, 1)` renvoie le premier élément de la liste l : les éléments sont numérotés de 1 à `nops(l)`.

`op(l)` renvoie `1, 2, 2, 3`

`append(l, 4)` ajoute l'élément 4 à la fin de la liste l.

De plus, les listes peuvent être concaténées avec le signe `.` (un point), par exemple

`[1, 2] . [2, 3] = [1, 2, 2, 3]`.

Attention une suite d'expressions entre une accolade ouvrante `{` et une accolade fermante `}` désigne un ensemble.

Exemple d'ensembles et de fonctions agissant sur les ensembles :

$A := \{a, b, c\}$; $B := \{a, d\}$

$A \text{ union } B$ désigne $\{a, b, c, d\}$

$A \text{ intersect } B$ désigne $\{a\}$

$A \text{ minus } B$ désigne $\{b, c\}$

2.15.5 Traduction TI89/92

`augment` permet de concaténer deux listes.

`{ }` désigne la liste vide. Pour travailler avec des listes, on peut initialiser une liste de n éléments avec la commande `newlist`, par exemple :

`newlist(10) → L` (L est alors une liste de 10 éléments nuls).

On peut utiliser les commandes suivantes :

`seq(i*i, i, 1, 10)` qui désigne la liste des carrés des 10 premiers entiers, ou `seq(i*i, i, 0, 10, 2)` qui désigne la liste des carrés des 5 premiers entiers pairs (le pas est ici égal à 2).

Exemple :

`seq(i*i, i, 0, 10, 2) → L` va par exemple créer la liste :

$\{0, 4, 16, 36, 64, 100\}$ c'est à dire la liste des carrés de 0 à 10 avec un pas de 2 que l'on stocke dans L .

$L[i]$ qui désigne le i ème élément de la liste L .

On peut aussi écrire :

`2 → L[2]`

La liste L est alors $\{0, 2, 16, 36, 64, 100\}$

ou si L est de longueur n on peut rajouter un élément (par exemple 121) à L en écrivant :

`121 → L[n+1]`

Dans l'exemple précédent $n=6$ on peut donc écrire :

`121 → L[7]` (L est alors égale à $\{0, 2, 16, 36, 64, 100, 121\}$).

`left(L, 5)` désigne les 5 premiers éléments de la liste L .

2.16 Un exemple : le crible d'Eratosthène

2.16.1 Description

Pour trouver les nombres premiers inférieurs ou égaux à N :

1. On écrit les nombres de 1 à N dans une liste.
2. On barre 1 et on met 2 dans la case P .
Si $P \times P \leq N$ il faut traiter les éléments de P à N .
3. On barre tous les multiples de P à partir de $P \times P$.
4. On augmente P de 1.
Si $P \times P$ est inférieur ou égal à N , il reste à traiter les éléments non barrés de P à N .
5. On met le plus petit élément non barré de la liste dans la case P .
6. On refait les points 3 4 5 tant que $P \times P$ reste inférieur ou égal à N .

2.16.2 Écriture de l'algorithme

```

Fonction crible(N)
local TAB PREM I P
// TAB et PREM sont des listes
{} ->TAB
{} ->PREM
pour I de 2 a N faire
    concat(TAB, I) -> TAB
fpour
concat(0, TAB) -> TAB
2 -> P
// On a fait les points 1 et 2
//barrer 1 a ete realise en le remplaçant par 0
//TAB est la liste 0 2 3 4 ...N

tantque  $P \times P \leq N$  faire

    pour I de P a E(N/P) faire
//E(N/P) designe la partie entiere de N/P
        0 -> TAB[I*P]
    fpour
// On a barre tous les multiples de P a partir de P*P
    P+1 -> P
//On cherche le plus petit nombre <= N non barre (non nul) entre P et

    tantque ( $P \times P \leq N$ ) et (TAB[P]=0) faire

        P+1 -> P
    ftantque
ftantque
//on ecrit le resultat dans une liste PREM
pour I de 2 a N faire

    si TAB[I]  $\neq$  0 alors

        concat(PREM, I) -> PREM
    fsi
fpour
retourne PREM

```

2.16.3 Traduction xcas

```

//renvoie la liste des nombres premiers<=n selon erathostene
crible(n):={
    local tab,prem,p;
    tab:=[0,0];
    prem:=[];
    for (j:=2;j<=n;j++){
        tab:=append(tab,j);
    }
}

```

```

p:=2;
while (p*p<=n) {
  for (j:=p;j*p<=n;j++){
    tab[eval(j*p)]:=0;
  }
  p:=p+1;
  while ((p*p<=n) and (tab[p]==0)) {
    p:=p+1;
  }
}
for (j:=2;j<=n;j++) {
  if (tab[j]!=0) {
    prem:=append(prem,j);
  }
}
return(prem);
};

```

2.16.4 Traduction TI89/92

Voici la fonction crible :

- n est le paramètre de cette fonction.
- crible(n) est égal à la liste des nombres premiers inférieurs ou égaux à n.

```

:crible(n)
:Func
:local tab,prem,i,p
:newList(n)->tab
:newList(n)->prem
:seq(i,i,1,n) ->tab
:0 -> tab[1]
:2 -> p

:While p*p ≤ n

:For i,p,floor(n/p)
:0 -> tab[i*p]
:EndFor
:p+1 -> p

:While p*p ≤ n and tab[p]=0

:p+1 -> p
:EndWhile
:EndWhile
:0 -> p
:For i,2,n

:If tab[i] ≠ 0 Then

```

```

:p+1 ->p
:i ->prem[p]
:EndIf
:EndFor
:Return left(prem,p)
:EndFunc

```

2.17 Un exemple de fonction vraiment récursive

2.17.1 La définition

Voici la définition de la fonction a dite fonction de ackermann qui est une fonction de $\mathbb{N} \times \mathbb{N}$ dans \mathbb{N} :

$$\begin{aligned}
 a(0, y) &= y+1, \\
 a(x, 0) &= a(x-1, 1) \text{ si } x > 0, \\
 a(x, y) &= a(x-1, a(x, y-1)) \text{ si } x > 0 \text{ et si } y > 0.
 \end{aligned}$$

Ainsi on a :

$$\begin{aligned}
 a(0, 0) &= 1 \\
 a(1, 0) &= a(0, 1) = 2 \\
 a(1, 1) &= a(0, a(1, 0)) = a(0, 2) = 3 \\
 a(1, 2) &= a(0, a(1, 1)) = 4 \\
 a(1, n) &= a(0, a(1, n-1)) = 1 + a(1, n-1) = \dots = n+2 \\
 a(2, 0) &= a(1, 1) = 3 \\
 a(2, 1) &= a(1, a(2, 0)) = a(1, 3) = 5 \\
 a(2, 2) &= a(1, a(2, 1)) = 2 + a(2, 1) = 7 \\
 a(2, n) &= a(1, a(2, n-1)) = 2 + a(2, n-1) = 2n+3 \\
 a(3, 0) &= a(2, 1) = 5 \\
 a(3, 1) &= a(2, a(3, 0)) = 2 * a(3, 0) + 3 = 13 \\
 a(3, 2) &= a(2, a(3, 1)) = 2 * a(3, 1) + 3 = 29 \\
 a(3, n) &= a(2, a(3, n-1)) = 2 * a(3, n-1) + 3 = 2^{(n+1)} + 3 * (2^n + 2^{(n-1)} + \dots + 1) = 2^{(n+1)} + 3 * 2^n - 1
 \end{aligned}$$

On a donc par exemple : $a(3, 5) = 2^8 - 3 = 253$

Les calculs sont vite gigantesques on a par exemple :

$$\begin{aligned}
 a(4, 1) &= a(3, a(4, 0)) = a(3, a(3, 1)) = a(3, 13) = 65533 \\
 a(4, 2) &= a(3, a(4, 1)) = a(3, 65533) = 2^{65536} - 3 \\
 a(4, 3) &= a(3, a(4, 2)) = a(3, 2^{65536} - 3) = 2^{(2^{65536} - 3)} - 3
 \end{aligned}$$

On a donc :

$$a(4, y) = a(3, a(4, y-1)) = 2^{(a(4, y-1) + 3)} - 3 = 2^{(2^{(a(4, y-2) + 3)} - 3 + 3)} - 3 = 2^{(2^{(a(4, y-2) + 3)} - 3)} - 3$$

et donc

$$a(4, y) = 2^{(2^{(2^{(2^{(a(4, 0) + 3)} \dots)} - 3)} - 3)} - 3 = 2^{(2^{(2^{(2^{16} \dots)} - 3)} - 3)} - 3,$$

avec 2^y qui se répète y fois, et comme $16 = 2^4$ on a,

$$a(4, y) = 2^{(2^{(2^{(2^2 \dots)} - 3)} - 3)} - 3,$$

avec 2 qui se répète $y + 3$ fois.

2.17.2 Le programme

Voici un premier programme :

```

akc(x,y):={
  if (x==0) return y+1;
  if (y==0) return akc(x-1,1);
  return ack(x-1,ack(x,y-1));
}

```

ou bien en utilisant ifte :

```

ack(x,y):=ifte(x==0,y+1,ifte(y==0,ack(x-1,1),ack(x-1,ack(x,y-1))))

```

On remarque que le temps pour calculer la valeur pour $a(3, 5)$ est de 5.06s ce qui est très long, mais on peut donc améliorer le programme en arrêtant la récursivité lorsque $x==3$.

On écrit :

```

a(x,y):={
  if (x==0) return y+1;
  if (x==1) return y+2;
  if (x==2) return 2*y+1;
  if (x==3) return 2^(y+3)-3;
  if (y==0) return a(x-1,1);
  return a(x-1,a(x,y-1));
}

```

On peut aussi améliorer le programme en arrêtant la récursivité lorsque $x==4$.

On écrit :

```

a(x,y):={
  if (x==0) return y+1;
  if (x==1) return y+2;
  if (x==2) return 2*y+1;
  if (x==3) return 2^(y+3)-3;
  if (x==4) {
    local p:=1;
    for (j:=1;j<=y+3;j++) p:=2^p;
    return p-3;
  }
  if (y==0) return a(x-1,1);
  return a(x-1,a(x,y-1));
}

```

Essayez $a(4, 1)=65533$, $a(4, 2)$

Chapitre 3

Les programmes d'arithmétique

3.1 Quotient et reste de la division euclidienne

3.1.1 Les fonctions iquo, irem et smod de xcas

Si a et b sont des entiers ou des entiers de Gauss :

`iquo(a, b)` renvoie le quotient q de la division euclidienne de a par b et

`irem(a, b)` renvoie le reste r de la division euclidienne de a par b .

q et r vérifient :

si a et b sont entiers $a = b * q + r$ avec $0 \leq r < b$

si a et b sont des entiers de Gauss $a = b * q + r$ avec $|r|^2 \leq \frac{|b|^2}{2}$.

Par exemple si $a = 3 + 6 * i$ et si $b = 1 + 3 * i$ on a :

$q = 1 + i$ et $r = 1 + 2 * i$

Si a et b sont des entiers relatifs `smod(a, b)` renvoie le reste symétrique rs de la division euclidienne de a par b .

q et rs vérifient :

$a = b * q + rs$ avec $-\frac{b}{2} < rs \leq \frac{b}{2}$

Exemples :

`smod(7, 4) = -1`

`smod(-10, 4) = -2`

`smod(10, 4) = 2`

Remarque `mod` (ou `%`) est une fonction infixée et désigne un élément de $\mathbb{Z}/n\mathbb{Z}$.

On a : $7 \bmod 4 = -1 \% 4$ désigne un élément de $\mathbb{Z}/4\mathbb{Z}$

3.1.2 Activité

le texte de l'exercice

1. Vérifier que :

$$\frac{13}{18} = \frac{1}{2} + \frac{1}{5} + \frac{1}{45}$$

2. On donne deux entiers a et b vérifiant : $0 < b < a$. On note q et r le quotient et le reste de la division euclidienne de a par b ($a = bq + r$ avec $0 \leq r < b$).

Démontrer que :

$$q > 0$$

$$\frac{1}{q+1} < \frac{b}{a} \leq \frac{1}{q}$$

3. On définit u et v par :

$$\frac{b}{a} - \frac{1}{q+1} = \frac{v}{u}$$

et

$$u = a(q+1)$$

Exprimer v en fonction de b et r .

Démontrer que :

$$v \leq b < a < u$$

Si $r = 0$, vérifier que :

$$\frac{b}{a} = \frac{1}{q}$$

4. Si r est différent de zéro, on pose : $a_1 = u$ et $b_1 = v$.

Puis, on recommence : on divise a_1 par b_1 .

On trouve un quotient q_1 et un reste r_1 . Si r_1 est nul, vérifier que :

$$\frac{b}{a} = \frac{1}{q+1} + \frac{1}{q_1}$$

Si r_1 n'est pas nul, on recommence.

Montrer qu'il existe une suite finie d'entiers Q_0, Q_1, \dots, Q_n strictement croissante telle que :

$$\frac{b}{a} = \frac{1}{Q_0} + \frac{1}{Q_1} + \dots + \frac{1}{Q_n}$$

5. Rédiger l'algorithme décrit ici et l'appliquer à la fraction :

$$\frac{151}{221}$$

L'algorithme

On suppose que la fraction $\frac{\text{NUM}}{\text{DENOM}} \in]0; 1[$.

L'algorithme s'écrit en langage non fonctionnel :

```

DENOM → A
NUM → B
Quotient(A, B) → Q
Reste(A, B) → R
tantque R ≠ 0 faire
    Q+1 → D
    Afficher D
    B-R → B
    A*D → A
    Quotient(A, B) → Q
    Reste(A, B) → R
ftantque
Afficher Q

```

Traduction xcas

On écrit la fonction `decomp` qui va décomposer selon l'algorithme la fraction `frac`. Cette fonction va renvoyer la liste `lres` égale à $[Q_0, \dots, Q_n]$ avec $0 < Q_0 < \dots < Q_n$ et $\text{frac} = 1/Q_0 + \dots + 1/Q_n$.

Attention $\text{frac} = b/a$ et donc $\text{fxnd}(\text{frac}) = \text{fxnd}(b/a) = [b, a]$.

```
decomp(frac) := {
  local a,b,l,q,r,lres;
  l:=fxnd(frac);
  b:=l[0];
  a:=l[1];
  q:=iquo(a,b);
  r:=irem(a,b);
  lres:=[];
  while (r!=0) {
    lres:= concat(lres, q+1);
    b:=b-r;
    a:=a*(q+1);
    q:=iquo(a,b);
    r:=irem(a,b);
  }
  lres:=concat(lres,q);
  return lres;
}
```

Application à $\frac{151}{221}$

On tape :

```
decomp(151/221)
```

On obtient :

```
[2,6,61,5056,40895962,4181199228867648]
```

On vérifie :

```
1/2+1/6+1/61+1/5056+1/40895962+1/4181199228867648
```

On obtient :

```
151/221
```

On peut écrire un programme pour faire la vérification :

`size(l)` est égal à la longueur de la liste `l`

```
verifie(l) := {
  local s,k,res;
  s:=size(l);
  res:=0;
  for (k:=0;k<s;k++) {
    res:=res+1/l[k];
  }
  return res;
}
```

On tape :

```
verifie([2,6,61,5056,40895962,4181199228867648])
```

On obtient :

$$\frac{151}{221}$$

3.2 Calcul du PGCD par l'algorithme d'Euclide

Soient A et B deux entiers positifs dont on cherche le $PGCD$.
L'algorithme d'Euclide est basé sur la définition récursive du $PGCD$:

$$\begin{aligned} PGCD(A, 0) &= A \\ PGCD(A, B) &= PGCD(B, A \bmod B) \text{ si } B \neq 0 \end{aligned}$$

où $A \bmod B$ désigne le reste de la division euclidienne de A par B .

Voici la description de cet algorithme :

on effectue des divisions euclidiennes successives :

$$\begin{aligned} A &= B \times Q_1 + R_1 & 0 \leq R_1 < B \\ B &= R_1 \times Q_2 + R_2 & 0 \leq R_2 < R_1 \\ R_1 &= R_2 \times Q_3 + R_3 & 0 \leq R_3 < R_2 \\ &\dots\dots \end{aligned}$$

Après un nombre fini d'étapes, il existe un entier n tel que : $R_n = 0$.

on a alors :

$$PGCD(A, B) = PGCD(B, R_1) = \dots$$

$$PGCD(R_{n-1}, R_n) = PGCD(R_{n-1}, 0) = R_{n-1}$$

3.2.1 Traduction algorithmique

-Version itérative

Si $B \neq 0$ on calcule $R = A \bmod B$, puis avec B dans le rôle de A (en mettant B dans A) et R dans le rôle de B (en mettant R dans B) on recommence jusqu'à ce que $B = 0$, le $PGCD$ est alors égal à A .

```
fonction PGCD(A,B)
local R

tantque B ≠ 0 faire
    A mod B->R
    B->A
    R->B
ftantque
retourne A
ffonction
```

-Version récursive

On écrit simplement la définition récursive vue plus haut.

```
fonction PGCD(A,B)

Si B ≠ 0 alors
```

```

    retourne PGCD(B,A mod B)
  sinon
    retourne A
  fsi
ffonction

```

3.2.2 Traduction xcas

- Version itérative :

```

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

```

- Version récursive.

```

pgcdr(a,b) := {
  if (b==0)
    return(a);
  else
    return(pgcdr(b,irem(a,b)));
};

```

3.2.3 Traduction MapleV

-Version itérative :

```

pgcd:=proc(x,y)
local a,b,r:
a:=x:
b:=y:
while (b>0) do
r:=irem(a,b):
a:=b:
b:=r:
od:
RETURN(a):
end:

```

-Version récursive :

```

pgcd:=proc(a,b)
if (b=0) then
RETURN(a)
else

```

```

RETURN (pgcd(b, irem(a, b))) :
fi:
end:

```

3.2.4 Traduction MuPAD

-Version itérative :

```

pgcd:=proc(a,b)
local r:
begin
while (b>0) do
r:=a mod b:
a:=b:
b:=r:
end_while:
return(a):
end_proc;

```

-Version récursive :

```

pgcd:=proc(a,b)
begin
if (b=0) then
return(a)
else
return(pgcd(b, a mod b)):
end_if:
end_proc;

```

3.2.5 Traduction TI89 92

-Version itérative

```

:pgcd(a,b)
:Func
:Local r

:While b ≠ 0

:mod(a,b)->r
:b->a
:r->b
:EndWhile
:Return a
:EndFunc

```

-Version récursive

```

:pgcd(a,b)
:Func

```

```

:If b  $\neq$  0 Then
:Return pgcd(b, mod(a,b))
:Else
:Return a
:EndIf
:EndFunc

```

3.3 Identité de Bézout par l'algorithme d'Euclide

Dans ce paragraphe la fonction `Bezout(A,B)` renvoie la liste $\{U, V, PGCD(A, B)\}$ où U et V vérifient : $A \times U + B \times V = PGCD(A, B)$.

3.3.1 Version itérative sans les listes

L'algorithme d'Euclide permet aussi de trouver un couple U et V vérifiant :

$$A \times U + B \times V = PGCD(A, B)$$

En effet, si on note A_0 et B_0 les valeurs de A et de B du début on a :

$$A = A_0 \times U + B_0 \times V \text{ avec } U = 1 \text{ et } V = 0$$

$$B = A_0 \times W + B_0 \times X \text{ avec } W = 0 \text{ et } X = 1$$

Puis on fait évoluer A, B, U, V, W, X de façon à ce que ces deux relations soient toujours vérifiées. Voici comment A, B, U, V, W, X évoluent :

- on pose : $A = B \times Q + R$ $0 \leq R < B$ ($R = A \bmod B$ et $Q = E(A/B)$)

- on écrit alors :

$$R = A - B \times Q = A_0 \times (U - W \times Q) + B_0 \times (V - X \times Q) = A_0 \times S + B_0 \times T$$

avec $S = U - W \times Q$ et $T = V - X \times Q$

Il reste alors à recommencer avec B dans le rôle de A ($B \rightarrow A$ $W \rightarrow U$ $X \rightarrow V$) et R dans le rôle de B ($R \rightarrow B$ $S \rightarrow W$ $T \rightarrow X$) d'où l'algorithme :

```

fonction Bezout(A,B)
local U,V,W,X,S,T,Q,R
1->U 0->V 0->W 1->X

tantque B  $\neq$  0 faire

A mod B->R
E(A/B)->Q
//R=A-B*Q
U-W*Q->S
V-X*Q->T
B->A W->U X->V
R->B S->W T->X
ftantque
retourne {U, V, A}
ffonction

```

3.3.2 Version itérative avec les listes

On peut simplifier l'écriture de l'algorithme ci-dessus en utilisant moins de variables : pour cela on utilise les listes LA , LB , LR pour mémoriser les triplets $\{U, V, A\}$, $\{W, X, B\}$ et $\{S, T, R\}$. Ceci est très commode car les logiciels de calcul savent ajouter des listes de même longueur (en ajoutant les éléments de même indice) et savent aussi multiplier une liste par un nombre (en multipliant chacun des éléments de la liste par ce nombre).

```

fonction Bezout(A,B)
local LA LB LR
{1, 0, A}->LA
{0, 1, B}->LB

tantque LB[3] ≠ 0 faire

LA-LB*E(LA[3]/LB[3])->LR
LB->LA
LR->LB
ftantque
retourne LA
ffonction

```

3.3.3 Version récursive sans les listes

Si on utilise des variables globales pour A , B , D , U , V , T , on peut voir la fonction `Bezout` comme calculant à partir de A , B , des valeurs qu'elle met dans U , V , D ($AU+BV=D$), grâce à une variable locale Q .

On écrit donc une fonction sans paramètre : seule la variable Q doit être locale à la fonction alors que les autres variables A , B . . . sont globales.

`Bezout` fabrique U , V , D vérifiant $A*U+B*V=D$ à partir de A et B . Avant l'appel récursif (on préserve $E(A/B)=Q$ et on met A et B à jour (nouvelles valeurs), après l'appel les variables U , V , D vérifient $A*U+B*V=D$ (avec A et B les nouvelles valeurs), il suffit alors de revenir aux premières valeurs de A et B en écrivant : $B*U+(A-B*Q)*V=A*V+B*(U-V*Q)$

On écrit alors :

```

fonction Bezout
local Q

Si B ≠ 0 faire

E(A/B)->Q
A-B*Q->R
B->A
R->B
Bezout
U-V*Q->W
V->U
W->V

```



```

sinon
1->U
0->V
A->D
fsi
ffonction

```

3.3.4 Version récursive avec les listes

On peut définir récursivement la fonction Bezout par :

$$\text{Bezout}(A, 0) = \{1, 0, A\}$$

Si $B \neq 0$ il faut définir $\text{Bezout}(A, B)$ en fonction de $\text{Bezout}(B, R)$ lorsque $R = A - B \times Q$ et $Q = E(A/B)$.

On a :

$$\begin{aligned} \text{Bezout}(B, R) = LT &= \{W, X, \text{pgcd}(B, R)\} \\ \text{avec } W \times B + X \times R &= \text{pgcd}(B, R) \end{aligned}$$

Donc :

$$\begin{aligned} W \times B + X \times (A - B \times Q) &= \text{pgcd}(B, R) \text{ ou encore} \\ X \times A + (W - X \times Q) \times B &= \text{pgcd}(A, B). \end{aligned}$$

D'où si $B \neq 0$ et si $\text{Bezout}(B, R) = LT$ on a :

$$\text{Bezout}(A, B) = \{LT[2], LT[1] - LT[2] \times Q, LT[3]\}.$$

```

fonction Bezout(A,B)
local LT Q R

Si B ≠ 0 faire

E(A/B)->Q
A-B*Q->R
Bezout(B,R)->LT
retourne {LT[2], LT[1]-LT[2]*Q, LT[3]}
sinon retourne {1, 0, A}
fsi
ffonction

```

3.3.5 Traduction xcas

- Version itérative avec les listes

```

bezout(a,b):={
//renvoie [u,v,d] tels que a*u+b*v=pgcd(a,b) (fct iterative)
local la,lb,lr,q,lb2;
la:=[1,0,eval(a)];
lb:=[0,1,eval(b)];
lb2:=eval(b);
while (lb2 !=0){

```

```

q:=iquo(la[2],lb2);
lr:=la+(-q)*lb;
la:=lb;
lb:=lr;
lb2:=lb[2];
}
return(la);
};

```

- Version récursive avec les listes

```

bezoutr(a,b):={
//renvoie [u,v,d] tels que a*u+b*v=pgcd(a,b) (fct recursive)
local lb,q,r;
if (b!=0) {
q:=iquo(a,b);
r:=irem(a,b);
lb:=bezoutr(b,r);
return([lb[1],lb[0]+(-q)*lb[1],lb[2]]);
} else
return([1,0,a]);
};

```

3.4 Décomposition en facteurs premiers d'un entier

Dans cette section, on ne suppose pas connue une table de nombres premiers : on ne se sert donc pas du programme crible.

3.4.1 Les algorithmes et leurs traductions algorithmiques

- Premier algorithme

Soit N un entier.

On teste, pour tous les nombres D de 2 à N , la divisibilité de N par D .

Si D divise N , on cherche alors les diviseurs de N/D etc... N/D joue le rôle de N et on s'arrête quand $N = 1$

On met les diviseurs trouvés dans la liste FACT.

```

fonction factprem(N)
local D FACT
2 -> D
{} -> FACT

tantque N ≠ 1 faire
    si N mod D = 0 alors
        concat(FACT,D) -> FACT
        N/D -> N
    sinon
        D+1 -> D

```

```

    fsi
  ftantque
  retourne FACT
ffonction

```

- Première amélioration

On ne teste que les diviseurs D entre 2 et $E(\sqrt{N})$.

En effet si $N = D1 * D2$ alors on a :

soit $D1 \leq E(\sqrt{N})$, soit $D2 \leq E(\sqrt{N})$ car sinon on aurait :
 $D1 * D2 \geq (E(\sqrt{N}) + 1)^2 > N$.

```

fonction factprem1(N)
local D FACT
2 -> D
{} -> FACT

tantque D*D ≤ N faire
  si N mod D = 0 alors
    concat(FACT,D) -> FACT
    N/D-> N
  sinon
    D+1 -> D
  fsi
ftantque
concat(FACT,N) -> FACT
retourne FACT
ffonction

```

Dans la liste FACT, on a les diviseurs premiers éventuellement plusieurs fois, par exemple :

factprem1(12)={2, 2, 3}. - Deuxième amélioration

On cherche si 2 divise N , puis on teste les diviseurs impairs D entre 3 et $E(\sqrt{N})$.

Dans la liste FACT, on fait suivre chaque diviseur premier par son exposant, par exemple :

factprem2(12)={2, 2, 3, 1}.

```

fonction facprem2(N)
local K D FACT
{}->FACT
0 -> K
tantque N mod 2 = 0 faire
  K+1 -> K
  N/2 -> N
ftantque

si K ≠ 0 alors
  concat(FACT,{2 K}) -> FACT
fsi
3 ->D

```

```

tantque D*D ≤ N faire
    0 -> K
    tantque N mod D = 0 faire
        K+1 -> K
        N/D -> N
    ftantque
    si K ≠ 0 alors
        concat(FACT, {D K}) -> FACT
    fsi
    D+2 -> D
ftantque
si N ≠ 1 alors
    concat(FACT, {N 1}) -> FACT
fsi
retourne FACT
ffonction

```

- Troisième amélioration

On cherche si 2 et 3 divisent N , puis on teste les diviseurs D entre 5 et $E(\sqrt{N})$ de la forme $6 * k - 1$ ou $6 * k + 1$.

On remarque que si :

$D = 6 * k - 1$ on a $D + (4 * D \bmod 6) = 6 * k + 1$

et que si :

$D = 6 * k + 1$ on a $D + (4 * D \bmod 6) = 6 * (k + 1) - 1$

Dans la liste FACT, on fait suivre chaque diviseur par son exposant, par exemple :

$\text{factprem3}(12) = \{2, 2, 3, 1\}$.

```

fonction factprem3(N)
local J, D, FACT
2->D
{}->FACT
tantque (D*D≤N) faire
    0->J
    tantque (N mod D=0) faire
        N/D->N
        J+1->J
    ftantque
    si (J ≠ 0) alors concat(FACT, {D, J})->FACT fsi
    si (D<4) alors
        2*D-1->D
    sinon
        D+(4*D mod 6)->D
    fsi
ftantque
si (N ≠ 1) alors concat(FACT, {N, 1})->FACT fsi
retourne(FACT)
ffonction

```

3.5. DÉCOMPOSITION EN FACTEURS PREMIERS EN UTILISANT LE CRIBLE 45

3.4.2 Traduction xcas

On traduit la troisième amélioration.

```
factprem(n) := {
//decompose n en facteur premier dans la liste l de dimension s
local j,d,s,l;
d:=2;
s:=0;
l:=[];
while (d*d<=n) {
j:=0;
while (irem(n,d)==0) {
n:=iquo(n,d);
j:=j+1;
}
if (j!=0) {
l:=concat(l,[d,j]);
s:=s+2;
}
if (d<4) {
d:=2*d-1;
}
else {
d:=d+irem(4*d,6);
}
}
if (n!=1) {
l:=concat(l,[n,1]);
s:=s+2;
}
return([l,s]);
};
```

3.5 Décomposition en facteurs premiers en utilisant le crible

Pour effectuer la décomposition en facteurs premiers de n , on utilise la table des nombres premiers fabriquée par le crible : on ne teste ainsi que des nombres premiers.

Si on peut écrire $N = A * D^J$ avec $PGCD(A, D) = 1$ et $J > 0$ alors D^J est un facteur de la décomposition de N .

On écrit tout d'abord la fonction $ddiv(N, D)$ qui renvoie :

- soit la liste :

[N , []] si D n'est pas un diviseur de N ,

- soit la liste :

[A , [D , J]] si $N = A * D^J$ avec $PGCD(A, D) = 1$ et $J > 0$.
 D^J est alors un diviseur de N et $A = N/D^J$.

3.5.1 Traduction Algorithmique

```

fonction ddiv(N,D)
//ddiv renvoie [a,[d,j]] (n=a*d^j, pgcd(a,d)=1) si j!=0 sinon [n,[]]
local L,J
0->J
tantque (N mod D)=0 faire
N/D->N
J+1->J
ftantque
si (J=0) alors
{N,{}}->L
sinon
{N,{D,J}}->L
fsi
retourne(L)
ffonction

```

On cherche la liste des nombres premiers plus petit que \sqrt{N} et on met cette liste dans la variable *PREM*. Lorsque $N > 1$, on teste si ces nombres premiers sont des diviseurs de N en utilisant *ddiv*.

```

fonction criblefact(N)
//decomposition en facteurs premiers de n
//en utilisant ddiv et crible
local D,PREM,S,LD,LDIV;
PREM:=crible(floor(sqrt(N)));
S:=dim(PREM);
LDIV:={};
1->K
tantque (K<=S et N>1) faire
ddiv(N,PREM[K])->LD
concat(LDIV,ld[2])->LDIV;
LD[1]->N
K+1->K
}

si (N ≠ 1){
concat(LDIV,[N,1])->LDIV;
}
retourne(LDIV);
}

```

3.5.2 Traduction xcas

```

ddiv(n,d):={
//ddiv renvoie [a,[d,j]] (n=a*d^j, pgcd(a,d)=1) si j!=0
//sinon [n,[]]
local l,j;

```

```

j:=0;
while (irem(n,d)==0) {
n:=iquo(n,d);
j:=j+1;
}
if (j==0) {
l:=[n, []];
} else {
l:=[n, [d, j]];
}
return(l);
}

criblefact(n):={
//decomposition en facteurs premiers de n
//en utilisant ddiv et crible
local d,prem,s,ld,ldiv;
prem:=crible(floor(sqrt(n)));
s:=size(prem);
ldiv=[];
for (k:=0;k<s;k++){
ld:=ddiv(n,prem[k]);
ldiv:=concat(ldiv,ld[1]);
n:=ld[0];
k:=k+1;
}
if (n!=1) {
ldiv:=concat(ldiv,[n,1]);
}
return(ldiv);
}

```

3.6 La liste des diviseurs

3.6.1 Les programmes avec les élèves

L'algorithme naïf :

```

pour i de 1 a n faire
  si (i divise n) alors
    afficher i
  fsi
fpour

```

Les élèves remarquent que l'on peut avoir les diviseurs deux par deux.

```

pour i de 1 a E( $\sqrt{n}$ ) faire
  si (i divise n) alors
    afficher i, n/i
  fsi

```

fpour

Malheureusement, lorsque l'entier n est le carré de p , p figure deux fois dans l'affichage des diviseurs.

On améliore donc l'algorithme :

```

1 → i
tantque i < √n faire
    si (i divise n) alors
        afficher i, n/i
    fsi
    i+1 → i
ftantque
si i·i=n alors
    afficher i
fsi

```

3.6.2 Le nombre de diviseurs d'un entier n

On décompose n en facteurs premiers, puis on donne aux exposants de ces facteurs premiers toutes les valeurs possibles. Si $n = a^\alpha * b^\beta * c^\gamma$ l'exposant de a peut prendre $\alpha + 1$ valeurs ($0.. \alpha$), celui de b peut prendre $\beta + 1$ valeurs et celui de c peut prendre $\gamma + 1$ valeurs donc le nombre de diviseurs de n est $(\alpha + 1) * (\beta + 1) * (\gamma + 1)$.

3.6.3 L'algorithme sur un exemple

Déscription de l'algorithme sur un exemple :

$$n = 360 = 2^3 * 3^2 * 5$$

n a donc $(3+1)*(2+1)*(1+1)=24$ diviseurs.

On les écrit en faisant varier le triplet représentant les exposants avec l'ordre :

$(0,0,0), (1,0,0), (2,0,0), (3,0,0),$

$(0,1,0), (1,1,0), (2,1,0), (3,1,0),$

$(0,2,0), (1,2,0), \dots,$

$(0,2,1), (1,2,1), (2,2,1), (3,2,1)$

On a $(a_1, \beta, \gamma) < (b_1, b_2, b_3)$ si :

$\gamma < b_3$ ou

$\gamma = b_3$ et $\beta < b_2$ ou

$\gamma = b_3$ et $\beta = b_2$ et $a_1 < b_1$.

On obtient les $4*3*2=24$ diviseurs de 360 :

1,2,4,8,3,6,12,24,9,18,36,72,5,10,20,40,15,30,60,120,45,90,180,360.

que l'on peut écrire en le tableau suivant :

1,2,4,8 (les puissances de 2)

3,6,12,24 (3*les puissances de 2)

9,18,36,72 (3*3*les puissances de 2)

5,10,20,40 (5*les puissances de 2)

15,30,60,120 (5*3*les puissances de 2)

45,90,180,360 (5*3*3*les puissances de 2).

Comment obtient-on la liste des diviseurs de $a^\alpha * b^\beta * c^\gamma$ à partir de la liste L1 des

diviseurs de $a^\alpha * b^\beta$?

Il suffit de rajouter à L1 la liste L2 constituée par :

$c * L1, \dots, c^\gamma * L1$

Dans le programme cette liste de diviseurs (L1) sera donc constituée au fur et à mesure au moyen d'une liste (L2) qui correspond au parcours de l'arbre.

On initialise L1 avec $\{1\}$, puis on rajoute à L1 la liste L2 formée par :

$a * L1, \dots, a^\alpha * L1$.

Puis on recommence avec le diviseur suivant :

on rajoute à L1 la liste L2 formée par $b * L1, \dots, b^\beta * L1$ etc...

3.6.4 Les algorithmes donnant la liste des diviseurs de n

La liste L1 est la liste destinée à contenir les diviseurs de N.

Au début $L1 = \{1\}$ et $L2 = \{\}$.

Pour avoir la liste des diviseurs de N, on cherche A le premier diviseur de N et on cherche a la puissance avec laquelle A divise N.

On définit la liste L2 :

L2 est obtenue en concaténant, les listes $L1 * A, L1 * A^2, \dots, L1 * A^a$: au début $L1 = \{1\}$ donc $L2 = \{A, A^2, \dots, A^a\}$.

On modifie la liste L1 en lui concaténant la liste L2, ainsi $L1 = \{1, A, A^2, \dots, A^a\}$.

Puis, on vide la liste L2. On cherche B le deuxième diviseur éventuel de N et on cherche b la puissance avec laquelle B divise N.

On définit la nouvelle liste L2 :

L2 est obtenue en concaténant, les listes $L1 * B, L1 * B^2, \dots, L1 * B^b$ (c'est à dire $L2 = \{B, B * A, B * A^2, \dots, B * A^a B^2, B^2 * A, B^2 * A^2, \dots, B^2 * A^a, \dots, B^b * A, B^b * A^2, \dots, B^b * A^a\}$)

On modifie la liste L1 en lui concaténant la liste L2, ainsi :

$L1 = \{1, A, A^2, \dots, A^a, B, B * A, B * A^2, \dots, B * A^a, \dots, B^b, B^b * A, B^b * A^2, \dots, B^b * A^a\}$.

Et ainsi de suite, jusqu'à avoir épuisé tous les diviseurs de N.

Traduction Algorithmique

```

fonction NDIV1(N)
  local D, L1, L2, K
  2 ← D
  {1} ← L1
  tantque (N ≠ 1) faire
    {} → L2
    0 ← K :
    tantque ((N MOD D) = 0) faire
      N/D ← N
      K + 1 ← K
    concat(L2, L1 * D^K) → L2
  ftantque
  concat(L1, L2) → L1
  D + 1 ← D
  ftantque
  retourne(L1)

```

Traduction xcas

```

ndiv1(n) := {
  local d, l1, l2, k;
  d:=2;
  l1:=[1];
  while (n!=1) {
    l2:=[];
    k:=0;
    while (irem(n,d)==0) {
      n:=iquo(n,d);
      k:=k+1;
      l2:=concat(l2, l1*d^k);
    }
    l1:=concat(l1, l2);
    d:=d+1;
  }
  return(l1);
}

```

On peut améliorer ce programme si on tient compte du fait qu'après avoir éventuellement divisé N par 2 autant de fois qu'on le pouvait, les diviseurs potentiels de N sont impairs.

On remplace alors :

$D + 1 - > D$

par :

si $D=2$ alors $D + 1 - > D$ sinon

$D + 2 - > D$ fsi

On améliore encore le programme précédent en remarquant que, si le diviseur potentiel D est tel que $D > \sqrt{N}$, c'est que N est premier ou vaut 1. On ne continue donc pas la recherche des diviseurs de N et quand N est différent de 1 on complète $L1$ par $L1 * N$.

Et aussi, on ne teste comme diviseur potentiel de N , que les nombres 2, 3, puis les nombres de la forme $6 * k - 1$ ou de la forme $6 * k + 1$ (pour $k \in \mathbb{N}$).

On remplace donc :

si $D=2$ alors $D + 1 - > D$ sinon

$D + 2 - > D$ fsi

par :

si $D < 4$ alors $2 * D - 1 - > D$ sinon

$D + (4 * D \bmod 6) - > D$ fsi

Traduction Algorithmique

```

fonction ndiv2(N)
local D, L1, L2, K
2- > D
{1}- > L1
tantque ( $D \leq \sqrt{N}$ ) faire

```

3.7. LA LISTE DES DIVISEURS AVEC LA DÉCOMPOSITION EN FACTEURS PREMIERS 51

```

{}->L2
0->K:
tantque ((N MOD D) =0) faire
N/D->N
K+1->K
concat(L2,L1*D^K)->L2
ftantque
concat(L1,L2)->L1
si D<4 alors 2*D-1->D sinon
D+(4*D mod 6)->D fsi
ftantque
si N ≠ 1 alors
concat(L1,L1*N)->L1
fsi
retourne(L1)

```

3.7 La liste des diviseurs avec la décomposition en facteurs premiers

3.7.1 FPDIV

On utilise le programme `factprem` (qui donne la liste des facteurs premiers de N (cf 3.4.2) pour obtenir la liste des diviseurs de N selon l'algorithme utilisé dans `NDIV1`.

Traduction Algorithmique

```

fonction fpdiv(N)
//renvoie la liste des diviseurs de n en utilisant factprem
local L1,L2,L3,D,ex,S
factprem(N)->L3
dim(L3)->S
{1}->L1
pour K de 1 a S-1 pas 2 faire
{}->L2
L3[K]->D
L3[K+1]->ex
pour J de 1 a ex faire
concat(L2,L1*(D^J))->L2
}
concat(L1,L2)->L1
}
retourne(L1)
}

```

Traduction xcas

```

fpdiv(n):={

```

```
//renvoie la liste des diviseurs de n en utilisant factprem
local l1,l2,l3,d,ex,s;
l3:=factprem(n);
s:=size(l3);
l1:=[1];
for (k:=0;k<s-1;k:=k+2) {
l2:=[];
d:=l3[k];
ex:=l3[k+1];
for (j:=1;j<=ex;j++) {
l2:=concat(l2,l1*(d^j));
}
l1:=concat(l1,l2);
}
return(l1);
}
```

3.7.2 CRIBLEDIV

Pour obtenir la liste des diviseurs de N selon l'algorithme utilisé dans `NDIV1`, on utilise le programme `criblefact` (cf 3.5.2) qui donne la liste des facteurs premiers de N .

3.7.3 Traduction Algorithmique

```
fonction criblediv(N)
//renvoie la liste des diviseurs de n en utilisant factprem
local L1,L2,L3,D,ex,S
criblefact(N)->L3
dim(L3)->S
{1}->L1
pour K de 1 a S-1 pas 2 faire
{}->L2
L3[K]->D
L3[K+1]->ex
pour J de 1 a ex faire
concat(L2,L1*(D^J))->L2
}
concat(L1,L2)->L1
}
retourne(L1)
}
```

Traduction xcas

```
criblediv(n):={
//renvoie la liste des diviseurs de n en utilisant criblefact
local l1,l2,l3,d,ex;
```

```

l3:=criblefact(n);
s:=size(l3);
l1:=[1];
for (k:=0;k<s-1;k:=k+2) {
  l2:=[];
  d:=l3[k];
  ex:=l3[k+1];
  for (j:=1;j<=ex;j++) {
    l2:=concat(l2,l1*(d^j));
  }
  l1:=concat(l1,l2);
}
return(l1);
}

```

3.8 Calcul de $A^P \text{ mod } N$

3.8.1 Traduction Algorithmique

-Premier algorithme

On utilise deux variables locales PUIS et I.

On fait un programme itératif de façon qu'à chaque étape, PUIS représente $A^I \text{ mod } N$

```

fonction puimod1 (A, P, N)
local PUIS, I
1->PUIS
pour I de 1 a P faire
  A*PUIS mod N ->PUIS
fpour
retourne PUIS
ffonction

```

-Deuxième algorithme

On n'utilise ici qu'une seule variable locale PUI, mais on fait varier P de façon qu'à chaque étape de l'itération on ait :

$PUI * A^P \text{ mod } N = \text{constante}$. Au début $PUI = 1$ donc $\text{constante} = A^P \text{ mod } N$ (pour la valeur initiale du paramètre P , c'est à dire que cette *constante* est égale à ce que doit retourner la fonction), et, à chaque étape, on utilise l'égalité $PUI * A^P \text{ mod } N = (PUI * A \text{ mod } N) * A^{P-1} \text{ mod } N$, pour diminuer la valeur de P , et pour arriver à la fin à $P = 0$, et alors on a la *constante* = PUI .

```

fonction puimod2 (A, P, N)
local PUI
1->PUI
tantque P>0 faire
  A*PUI mod N ->PUI
  P-1->P
ftantque

```

```
retourne PUI
ffonction
```

-Troisième algorithme

On peut aisément modifier ce programme en remarquant que :

$$A^{2^*P} = (A * A)^P.$$

Donc quand P est pair, on a la relation :

$$PUI * A^P = PUI * (A * A \bmod N)^{P/2} \bmod N$$

et quand P est impair, on a la relation :

$$PUI * A^P = (PUI * A \bmod N) * A^{P-1} \bmod N.$$

On obtient alors, un algorithme rapide du calcul de $A^P \bmod N$.

```
fonction puimod3 (A, P, N)
local PUI
1->PUI
tantque P>0 faire
    si P mod 2 =0 alors
        P/2->P
        A*A mod N->A
    sinon
        A*PUI mod N ->PUI
        P-1->P
    fsi
ftantque
retourne PUI
ffonction
```

On peut remarquer que si P est impair, $P - 1$ est pair.

On peut donc écrire :

```
fonction puimod4 (A, P, N)
local PUI
1->PUI
tantque P>0 faire
    si P mod 2 =1 alors
        A*PUI mod N ->PUI
        P-1->P
    fsi
P/2->P
A*A mod N->A
ftantque
retourne PUI
ffonction
```

-Programme récursif

On peut définir la puissance par les relations de récurrence : $A^0 = 1$
 $A^{P+1} \bmod N = (A^P \bmod N) * A \bmod N$

```
fonction puimod5(A, P, N)
si P>0 alors
```

```

retourne puimod5(A, P-1, N)*A mod N
sinon
retourne 1
fsi
ffonction

```

-Programme récursif rapide

```

fonction puimod6(A, P, N)
si P>0 alors
  si P mod 2 =0 alors
    retourne puimod6((A*A mod N), P/2, N)
  sinon
    retourne puimod6(A, P-1, N)*A mod N
  fsi
sinon
retourne 1
fsi
ffonction

```

3.8.2 Traduction xcas

```

puimod(a,p,n):={
//calcule recursivement la puissance rapide a^p modulo n
  if (p==0){
    return(1);
  }
  if (irem(p,2)==0){
    return(puimod(irem(a*a,n),iquo(p,2),n));
  }
  return(irem(a*puimod(a,p-1,n),n));
}

```

3.9 La fonction "estpremier"

3.9.1 Traduction Algorithmique

- Premier algorithme

On va écrire une fonction booléenne de paramètre N , qui sera égale à *VRAI* quand N est premier, et, à *FAUX* sinon.

Pour cela, on cherche si N possède un diviseur différent de 1 et inférieur ou égal à $E(\sqrt{N})$ (partie entière de racine de N).

On traite le cas $N=1$ à part !

On utilise une variable booléenne *PREM* qui est au départ à *VRAI*, et qui passe à *FAUX* dès que l'on rencontre un diviseur de N .

```

Fonction estpremier(N)
local PREM, I, J

```

```

E( $\sqrt{N}$ ) -> J
Si N = 1 alors
    FAUX->PREM
    sinon
        VRAI->PREM
fsi
2->I

tantque PREM et  $I \leq J$  faire
    si N mod I = 0 alors
        FAUX->PREM
        sinon
            I+1->I
    fsi
ftantque
retourne PREM
ffonction

```

-Première amélioration

On peut remarquer que l'on peut tester si N est pair, et ensuite, tester si N possède un diviseur impair.

```

Fonction estpremier(N)
local PREM, I, J
E( $\sqrt{N}$ ) -> J
Si (N = 1) ou (N mod 2 = 0) et  $N \neq 2$  alors
    FAUX->PREM
    sinon
        VRAI->PREM
fsi
3->I

tantque PREM et  $I \leq J$  faire
    si N mod I = 0 alors
        FAUX->PREM
        sinon
            I+2->I
    fsi
ftantque
retourne PREM
ffonction

```

- Deuxième amélioration

On regarde si N est divisible par 2 ou par 3, sinon on regarde si N possède un diviseur de la forme $6 \times k - 1$ ou $6 \times k + 1$ (pour $k \in \mathbb{N}$).

```

Fonction estpremier(N)
local PREM, I, J

```


$E(\sqrt{N}) \rightarrow J$

```

Si (N = 1) ou (N mod 2 = 0) ou (N mod 3 = 0) alors
  FAUX  $\rightarrow$  PREM
  sinon
    VRAI  $\rightarrow$  PREM
  fsi
si N=2 ou N=3 alors
  VRAI  $\rightarrow$  PREM
  fsi
5  $\rightarrow$  I

```

tantque PREM et $I \leq J$ faire

```

  si (N mod I = 0) ou (N mod I+2 = 0) alors
    FAUX  $\rightarrow$  PREM
    sinon
      I+6  $\rightarrow$  I
    fsi
  ftantque
  retourne PREM
ffonction

```

3.9.2 Traduction xcas

```

estprem(n) := {
//teste si n est premier
  local prem, j, k;
  if ((irem(n,2)==0) or (irem(n,3)==0) or (n==1)) {
    return(false);
  }
  if ((n==2) or (n==3)) {
    return(true);
  }
  prem:=true;
  k:=5;
  while ((k*k<=n) and prem) {
    if (irem(n,k)==0 or irem(n,k+2)==0) {
      prem:=false;
    }
    else {
      k:=k+6;
    }
  }
  return(prem);
}

```

3.10 La fonction estpremc en utilisant le crible

3.10.1 Traduction algorithmique

```

fonction estpremc(N)
//utilise la fonction crible pour tester si n est premier
local PREM,S;
crible(floor(sqrt(N)))->PREM
dim(PREM)->S
si (N=1) retourne(FAUX)
pour K de 1 a S faire
    si (N mod ,PREM[K])=0)
        retourne(FAUX);
    fsi
fpour
retourne(VRAI)
ffonction

```

3.10.2 Traduction xcas

```

estpremc(n):={
//utilise la fonction crible pour tester si n est premier
local prem,s;
prem:=crible(floor(sqrt(n)));
s:=size(prem);
if (n==1) return(false);
for (k:=0;k<s;k++){
    if (irem(n,prem[k])==0){
        return(false);
    }
}
return(true);
}

```

3.11 Méthode probabiliste de Mr Rabin

Si N est premier alors tous les nombres K strictement inférieurs à N sont premiers avec N , donc d'après le petit théorème de Fermat on a :

$$K^{N-1} = 1 \pmod{N}$$

Par contre, si N n'est pas premier, les entiers K ($1 < K < N$) vérifiant :

$$K^{N-1} = 1 \pmod{N}$$

sont peu nombreux.

La méthode probabiliste de Rabin consiste à prendre au hasard un nombre K dans l'intervalle $[2 ; N - 1]$ ($1 < K < N$) et à calculer :

$$K^{N-1} \pmod{N}$$

Si $K^{N-1} = 1 \pmod{N}$ on refait un autre tirage du nombre K , et, si $K^{N-1} \neq 1 \pmod{N}$ on est sûr que N n'est pas premier.

Si on obtient $K^{N-1} = 1 \pmod{N}$ pour 20 tirages successifs de K on peut conclure que N est premier avec une probabilité d'erreur faible :

on dit alors que N est pseudo-premier.

Bien sûr cette méthode est employée pour savoir si de grands nombres sont pseudo-premiers mais on préfère utiliser la méthode de Miller-Rabin (cf 3.12) qui est aussi une méthode probabiliste mais qui donne N premier avec une probabilité d'erreur plus faible (inférieure à $(0.25)^{20}$ si on a effectué 20 tirages, soit, une erreur de l'ordre de 10^{-12}).

3.11.1 Traduction Algorithmique

On suppose que :

`hasard(N)` donne un nombre entier au hasard entre 0 et $N - 1$.

Le calcul de $K^{N-1} \bmod N$ se fait grâce à l'algorithme de la puissance rapide (cf page 53).

On suppose que :

`powmod(K, P, N)` calcule $K^P \bmod N$

```
Fonction estprem(N)
local K, I, P
1->I
1->P
tantque P = 1 et I < 20 faire
  hasard(N-2)+2->K
  powmod(K, N-1, N)->P
  I+1->I
ftantque
Si P =1 alors
  retourne VRAI
sinon
  retourne FAUX
fsi
ffonction
```

3.11.2 Traduction xcas

La fonction `powmod` existe dans `xcas` : il est donc inutile de la programmer.

```
rabin(n):={
//teste par la methode de Rabin si n est pseudo-premier
local k,j,p;
j:=1;
p:=1;
while ((p==1) and (j<20)) {
k:=2+rand(n-2);
p:=powmod(k,n-1,n);
j:=j+1;
}
if (p==1) {
return(true);
}
```

```
return(false);
}
```

3.12 Méthode probabiliste de Mr Miller-Rabin

3.12.1 Un exemple

Rappel Le théorème de Fermat :

Si n est premier et si k est un entier quelconque alors $k^n = k \bmod n$.
et donc

Si n est premier et si k est premier avec n alors $k^{n-1} = 1 \bmod n$.

Soit $N = 561 = 3 * 11 * 17$. Il se trouve que l'on a : pour tout A ($A < N$), on a $A^N = A \bmod N$, donc si A est premier avec N on a $A^{N-1} = 1 \bmod N$, le test de Rabin est donc en défaut, seulement pour A non premier avec N . Par exemple on a :

$$3^{560} = 375 \bmod 561$$

$$11^{560} = 154 \bmod 561$$

$$17^{560} = 34 \bmod 561$$

$$471^{560} = 375 \bmod 561$$

mais pour tous les nombres A non multiples de 3, 11 ou 17 on a :

$$A^{N-1} = 1 \bmod 561.$$

Par exemple on a :

$$5^{560} = 1 \bmod 561.$$

$$52^{N-1} = 1 \bmod 561.$$

On risque donc de dire avec le test de Rabin que 561 est pseudo-premier.

Il faut donc affiner le test en remarquant que si N est premier l'équation : $X^2 = 1 \bmod N$ n'a pour solution que $X = 1 \bmod N$ ou $X = -1 \bmod N$.

Le test de Miller-Rabin est basé sur cette remarque.

Pour $N = 561$, $N - 1 = 560$, on a : $560 = 35 * 2^{16}$

$$13^{35} = 208 \bmod 561$$

$$13^{35*2} = 67 \bmod 561$$

$$13^{35*4} = 1 \bmod 561$$

$$13^{35*8} = 1 \bmod 561...$$

On vient de trouver que 67 est solution de $X^2 = 1 \bmod 561$ donc on peut affirmer que 561 n'est pas premier.

$A = 13$ vérifie le test de Rabin car $13^{560} = 1 \bmod 561$

mais ne vérifie pas le test de Miller-Rabin car

$$13^{35*2} \neq -1 \bmod 561 \text{ et } 13^{35*2} \neq 1 \bmod 561$$

$$\text{et pourtant } 13^{35*4} = 13^{35*4} = 1 \bmod 561$$

Par contre ce test ne suffit pas pour affirmer qu'un nombre est premier car :

$101^{35} = 560 = -1 \bmod 561$ et donc $101^{35*2} = 1 \bmod 561$ et cela ne fournit pas de solutions autre que 1 ou -1 à l'équation $X^2 = 1 \bmod 561$.

3.12.2 L'algorithme

L'algorithme est basé sur :

1/ Le petit théorème de Fermat :

$A^{N-1} = 1 \pmod N$ si N est premier et si $A < N$.

2/ Si N est premier, l'équation $X * X = 1 \pmod N$ n'a pas d'autres solutions que $X = 1 \pmod N$ ou $X = -1 \pmod N$.

En effet il existe un entier k vérifiant $X * X - 1 = (X + 1) * (X - 1) = k * N$ donc,

puisque N est premier, N divise $X + 1$ ou $X - 1$. On a donc soit $X = 1 \pmod N$ ou $X = -1 \pmod N$.

On élimine les nombres pairs que l'on sait ne pas être premiers.

On suppose donc que N est impair et donc que $N - 1$ est pair et s'écrit :

$N - 1 = 2^t * Q$ avec $t > 0$ et Q impair.

Si $A^{N-1} = 1 \pmod N$ c'est que $A^{N-1} \pmod N$ est le carré de $B = A^{\frac{N-1}{2}} = A^{2^{t-1}Q} \pmod N$.

Si on trouve $B \neq 1 \pmod N$ et $B \neq -1 \pmod N$ on est sûr que N n'est pas premier.

Si $B = -1 \pmod N$ on recommence avec une autre valeur de A .

Si $B = 1 \pmod N$ on peut recommencer le même raisonnement si $\frac{N-1}{2}$ est encore pair ($B = A^{\frac{N-1}{2}} = (A^{\frac{N-1}{4}})^2 \pmod N$) ou si $\frac{N-1}{2}$ est impair, on recommence avec une autre valeur de A .

On en déduit que :

si $N - 1 = 2^t * Q$ et

si $A^{N-1} = 1 \pmod N$ et

si $A^Q \neq 1 \pmod N$ et

si pour $0 \leq ex < t$ on a $A^{2^{ex} * Q} \neq -1 \pmod N$ c'est que N n'est pas premier.

D'où la définition :

Soit N un entier positif impair égal à $1 + 2^t * Q$ avec Q impair.

On dit que N est pseudo-premier fort de base A si :

soit $A^Q = 1 \pmod N$

soit si il existe e , $0 \leq e < t$ tel que $A^{Q * 2^e} = -1 \pmod N$.

On voit facilement qu'un nombre premier impair est pseudo-premier fort dans n'importe quelle base A non divisible par N .

Réciproquement on peut montrer que si $N > 4$ n'est pas premier, il existe au plus $N/4$ bases A ($1 < A < N$) pour lesquelles N est pseudo-premier fort de base A .

L'algorithme va choisir au hasard au plus 20 nombres A_k compris entre 2 et $N - 1$: si N est pseudo-premier fort de base A_k pour $k = 1..20$ alors N est premier avec une très forte probabilité égale à $(1/4)^{20} (< 10^{-12})$.

Bien sûr cette méthode est employée pour savoir si de grands nombres sont pseudo-premiers.

3.12.3 Traduction Algorithmique

On suppose que :

hasard(N) donne un nombre entier au hasard entre 0 et $N - 1$.

Le calcul de $K^{N-1} \pmod N$ se fait grâce à l'algorithme de la puissance rapide (cf page 53).

On notera :

powmod(K , P , N) la fonction qui calcule $K^P \pmod N$

Fonction Miller(N)

```

local Q,P,t,C,A,B,ex
si (N=2) alors retourne FAUX
si (N mod 2)==0) alors retourne FAUX
N-1->Q
0->t
tantque (Q mod 2 =0) faire
t+1->t
E(Q/2)->Q
ftantque
//N-1=2^t*Q
20->C
VRAI->P
tantque (C>0 et P) faire
hasard(N-2)+2->A
0->ex
powmod(A, Q, N)->B
si B<>1 alors
tant que (B<>1) et (B<>N-1) et (ex<t-1) faire
ex+1->ex
powmod(B,2,n)->B
ftantque
si (B<>N-1) alors
FAUX->P
fsi
C-1->C
ftantque
retourne P
ffonction

```

3.12.4 Traduction xcas

La fonction powmod existe dans xcas : il est donc inutile de la programmer.

```

miller(n):={
local p,q,t,c,a,b,ex;
if (n==2){return(true);}
if (irem(n,2)==0) {return(false);}
q:=n-1;
t:=0;
while (irem(q,2)==0) {
t:=t+1;
q:=iquo(q,2);
}
//ainsi n-1=q*2^t
c:=20;
p:=true;
while ((c>0) and p) {
//rand(k) renvoie un nombre entier de [0,k-1] si k<999999999
if (n<=10^9) {a:=2+rand(n-2);} else {a:=2+rand(999999999);}

```

```

ex:=0;
b:=powmod(a,q,n);
//si b!=1 on regarde si b^{2^ex}=-1 mod n (ex=0..t-1)
if (b!=1) {
while ((b!=1) and (b!=n-1) and (ex<=t-2)) {
b:=powmod(b,2,n);
ex:=ex+1;}
//si b!=n-1 c'est que n n'est pas premier
if (b!=n-1) {p:=false;}
}
c:=c-1;
}
return(p);
};

```

3.13 Numération avec xcas

On a besoin ici des fonctions de `xcas` :

- `asc` qui convertit un caractère ou une chaîne de caractères, en une liste de nombres et,
- `char` qui convertit un nombre ou une liste de nombres en un caractère ou une chaîne de caractères.

On a :

`char(n)` pour n entier, ($0 \leq n \leq 255$) donne le caractère ayant comme code ASCII l'entier n .

`char(l)` pour une liste d'entiers l ($0 \leq l[j] \leq 255$), donne la chaîne de caractères dont les caractères ont pour code ASCII les entiers $l[j]$ qui composent la liste l .

`asc(mot)` renvoie la liste des codes ASCII des lettres composant le mot.

Exemples

```
asc("A")=[65]
```

```
char(65)="A"
```

```
asc("Bonjour")=[66,111,110,106,111,117,114]
```

```
char([66,111,110,106,111,117,114])="Bonjour"
```

Remarque :

Il existe aussi la fonction `ord` qui a pour argument une chaîne de caractères mais qui renvoie le code ASCII de la première lettre de la chaîne de caractères :

```
ord("B")= 66 ord("Bonjour")= 66
```

3.13.1 Passage de l'écriture en base dix à une écriture en base b

La base b est inférieure ou égale à 10

- Version itérative

Si $n < b$, il n'y a rien à faire : l'écriture en base b est la même que l'écriture en base dix et est n . On divise n par b : $n = b * q + r$ avec $0 \leq r < b$.

Le reste r de la division euclidienne de n par b ($r := \text{irem}(n, b)$) donne le dernier chiffre de l'écriture en base b de n . L'avant dernier chiffre de l'écriture en base

b de n sera donné par le reste de la division euclidienne de q ($q := \text{iquo}(n, b)$) par b . On fait donc une boucle en remplaçant n par q ($n := \text{iquo}(n, b)$) tant que $n \geq b$ en mettant à chaque étape $r := \text{irem}(n, b)$ au début de la liste qui doit renvoyer le résultat.

On écrit la fonction itérative `ecritu` qui renvoie la liste des chiffres de n en base b :

```
ecris(n,b):={
//n est en base 10 et b<=10, ecrit est une fonction iterative
//renvoie la liste des caracteres de l'écriture de n en base b
local L;
L:=[];
while (n>=b){
L:=concat([irem(n,b)],L);
n:=iquo(n,b);
}
L:=concat([n],L);
return(L);
}
```

- Version récursive

Si $n < b$, l'écriture en base b est la même que l'écriture en base dix et est n .

Si $n \geq b$, l'écriture en base b de n est formée par l'écriture en base b de q suivi de r , lorsque q et r sont le quotient et le reste de la division euclidienne de n par b ($n = b * q + r$ avec $0 \leq r < b$).

On écrit la fonction récursive `ecritur` qui renvoie la liste des chiffres de n en base b :

```
ecritur(n,b):={
//n est en base 10 et b<=10, ecritur est recursive
//renvoie la liste des caracteres de l'écriture de n en base b
if (n>=b)
return(concat(ecritur(iquo(n,b),b),irem(n,b)));
else
return([n]);
}
```

La base b est inférieure ou égale à 36

On choisit 36 symboles pour écrire un nombre : les 10 chiffres 0,1..9 et les 26 lettres majuscules A, B, \dots, Z .

On transforme tout nombre positif ou nul n ($n < b$) en un caractère : ce caractère est soit un chiffre (si $n < 10$) soit une lettre ($A, B \dots Z$) (si $9 < n < 36$).

```
chiffre(n,b):={
//transforme n (0<=n<b) en son caractere ds la base b
if (n>9)
n:=char(n+55);
else
n:=char(48+n);
}
```



```
return(n);
}
```

On obtient alors la fonction itérative `ecritu` :

```
ecritu(n,b):={
//n est en base 10 et b<=36, escritu est une fonction itérative
//renvoie la liste des caracteres de l'écriture de n en base b
local L,r,rc;
L:=[];
while (n>=b){
r:=irem(n,b);
rc:=chiffre(r,b);
L:=concat([rc],L);
n:=iquo(n,b);
}
n:=chiffre(n,b);
L:=concat([n],L);
return(L);
}
```

- Version recursive

```
ecriture(n,b):={
//n est en base 10 et b<=36, ecriture est une fonction recursive
//renvoie la liste des caracteres de l'écriture de n en base b
local r,rc;
if (n>=b){
r:=irem(n,b);
rc:=chiffre(r,b);
return(append(ecriture(iquo(n,b),b),rc));
}
else {
return([chiffre(n,b)]);
}
}
```

En utilisant la notion de séquence on peut aussi écrire :

```
ecrit(n,b):= {
//renvoie la sequence des chiffres de n dans la base b
local m,u,cu;
m:=(NULL);
while(n!=0){
u:=(irem(n,b));
if (u>9) {
cu:=(char(u+55));
}
else {
cu:=(char(u+48));
}
```

```

    };
    m:=(cu,m);
    n:=(iquo(n,b));
  };
  return(m);
}

```

3.13.2 Passage de l'écriture en base b de n à l'entier n

Il faut convertir ici chaque caractère en sa valeur (on convertit le caractère contenu dans m en le nombre nm).

Si $m = (c0, c1, c2, c3)$ alors $n = c0 * b^3 + c1 * b^2 + c2 * b + c3$.

On calcule n en se servant de l'algorithme de Hörner (cf 3.14). En effet le calcul de $n = c0 * b^3 + c1 * b^2 + c2 * b + c3$ revient à calculer la valeur du polynôme $P(x) = c0 * x^3 + c1 * x^2 + c2 * x + c3$ pour $x = b$.

n va contenir successivement :

0 ($n := 0$) puis

$c0$ ($n := n * b + c0$) puis

$c0 * b + c1$ ($n := n * b + c1$) puis

$c0 * b^2 + c1 * b + c2 = (c0 * b + c1) * b + c2$ ($n := n * b + c2$) et enfin

$c0 * b^3 + c1 * b^2 + c2 * b + c3$ ($n := n * b + c3$).

On écrit donc la fonction nombre dans xcas :

```

nombre(m,b):={
local s,k,am,nm,n;
  s:=(size(m));
  n:=(0);
  k:=(0);
  if (s!=0) {
    while(k<s) {
      am:=(asc(m[k])[0]);
      if (am>64) {
        nm:=(am-55);
      }
      else {
        nm:=(am-48);
      };
      if (nm>(b-1)) {
        return("erreur");
      }
      n:=(n*b+nm);
      k:=(k+1);
    };
  }
  return(n);
}

```

3.14 Traduction xcas de l'algorithme de Hörner

Soit un polynôme P donné sous la forme d'une liste l formée par les coefficients de P selon les puissances décroissantes.

`hornerl(l,a)` renvoie une liste formée par la valeur `val` du polynôme en $x = a$ et par la liste `lq` des coefficients selon les puissances décroissantes du quotient $Q(x)$ de $P(x)$ par $(x - a)$.

On a :

$$P(a) = l[0] * a^p + l[1] * a^{p-1} + \dots + l[p] =$$

$$l[p] + a * (l[p-1] + a * (l[p-2] + \dots + a * (l[1] + a * l[0])))$$

$$P(x) = l[0] * x^p + l[1] * x^{p-1} + \dots + l[p] =$$

$$(x - a) * (lq[0] * x^{p-1} + \dots lq[p-1]) + val$$

donc `val = P(a)` et `p=s-1` si `s` est la longueur de la liste `l` donc :

$$lq[0] = l[0]$$

$$lq[1] = a * lq[0] + l[1]$$

$$lq[j] = a * lq[j-1] + l[j]$$

....

$$val = a * lq[p-1] + l[p]$$

```
hornerl(l,a):={
local s,val,lq,j;
s:=size(l);
//on traite les polys constants (de degre=0)
if (s==1) {return [l[0],[0]]};
// si s>1
lq:=[];
val:=0;
for (j:=0;j<s-1;j++) {
val:=val*a+l[j];
lq:=append(lq,val);
}
val:=val*a+l[s-1];
return([val,lq]);
};
```

On tape :

```
hornerl([1,2,4],12)
```

On obtient :

```
[172,[1,14]]
```

ce qui veut dire que :

$$x^2 + 2x + 4 = (x + 14)(x - 12) + 172$$

Si le polynôme est donné avec son écriture habituelle.

Pour utiliser la fonction précédente on a alors besoin des deux fonctions :

`symb2poly` qui transforme un polynôme en la liste de ses coefficients selon les puissances décroissantes.

`poly2symb` qui transforme une liste en l'écriture habituelle du polynôme ayant cette pour coefficients selon les puissances décroissantes.

```
hornerp(p,a,x):={
```

```
//ne marche pas pour les polys constants (de degre=0)
local l, val, lh;
l:=symb2poly(p, x);
lh:=hornerl(l, a);
p:=poly2symb(lh[1], x);
val:=lh[0];
return([val, p]);
};
```

On tape :

```
hornerp(x^2+2x+4, 12, x)
```

On obtient :

```
172, x+14
```

On tape :

```
hornerp(y^2+2y+4, 12, y)
```

On obtient :

```
172, y+14
```

Dans *xcas*, il existe la fonction *horner* qui calcule selon la méthode de Hörner la valeur d'un polynôme (donné sous forme de liste ou par son expression) en un point :

On tape :

```
horner(x^2+2x+4, 12)
```

On obtient :

```
172
```

On tape :

```
horner(y^2+2y+4, 12, y)
```

On obtient :

```
172
```

On tape :

```
horner([1, 2, 4], 12)
```

On obtient :

```
172
```

3.15 Affichage d'un nombre en une chaîne comprenant des espaces

3.15.1 Affichage d'un nombre entier par tranches de p chiffres

Pour rendre plus facile la lecture d'un grand nombre entier, on veut l'afficher par tranches, c'est à dire selon une chaîne de caractères constituées par les p premiers chiffres du nombre et d'un espace, puis les p suivants etc... On écrit le programme qui va afficher le nombre n par tranches de p chiffres :

```
affichen(n, p) := {
local reste, result, s;
result:="";
while (n>10^p) {
//on transforme irem(n, 10^p) en une chaîne
reste:=cat(irem(n, 10^p), " ");
```

3.15. AFFICHAGE D'UN NOMBRE EN UNE CHAÎNE COMPRENANT DES ESPACES 69

```
s:=size(reste);
//on ajoute l'espace et les zeros qui manquent
reste:=cat(" ",op(newList(p-s)),reste);
n:=iquo(n,10^p);
//on concatene reste avec result
result:=cat(reste,result);
}
reste:=cat(n);
return cat(reste,result);
};
```

On tape :

```
affichen(1234567,3) On obtient :
"1 234 567"
```

3.15.2 Transformation d'un affichage par tranches en un nombre entier

Pour avoir la transformation inverse, on va transformer une chaîne comportant des chiffres et un autre caractère (par exemple un espace) en un nombre entier.

On écrit le programme :

```
enleve(chn, ch) := {
local l, s;
s:=length(chn)-1;
//on transforme chn en une liste de ces lettres
//puis, on enleve le caractere ch de cette liste
l:=remove(x->(ord(x)==ord(ch)),seq(chn[k],k,0,s));
//on transforme la liste en chaîne
return expr(char(ord(l)));
};
```

On peut aussi remplacer la dernière ligne :

```
return char(ord(l))
```

(ord(l) transforme la liste de caractères en la liste de leurs codes ascii et char transforme la liste des codes ascii en une chaîne).

par :

```
return cat(op(l))
```

car op(l) transforme la liste en une séquence et cat concatène les éléments de cette séquence en une chaîne. On tape :

```
enleve("1 234 567", " ") On obtient :
1234567
```

3.15.3 Affichage d'un nombre décimal de $[0,1[$ par tranches de p chiffres

Pour rendre plus facile la lecture d'un nombre décimal de $[0,1[$, on veut l'afficher par tranches, c'est à dire selon une chaîne de caractères constituées par les p premières décimales du nombre et d'un espace, puis les p suivants etc... On suppose que l'écriture de d comporte un point (.) suivi des décimales et ne comporte pas d'exposant (pas de $e4$)

On écrit le programme qui va afficher le nombre d par tranches de p chiffres :

```
affiche(d,p):={
  local deb,result;
  //on suppose 0<=d<1
  d:=cat(d,"");
  if (d[0]=="0") {d:=tail(d);}
  if (expr(tail(d))<10^p){return d;}
  deb:=mid(d,0,p+1);
  result:=cat(deb," ");
  d:=mid(d,p+1);
  while (expr(d)>10^p) {
    deb:=mid(d,0,p);
    result:=cat(result,deb," ");
    d:=mid(d,p);
  }
  return cat(result,d);
};
```

On tape :

`affiche(0.1234567,3)` On obtient :

`".123 456 7"` **Remarque**

La commande `enleve(affiche(d,3)," ")` permet encore de retrouver d .

```
enleve(ch,chn):={
  local l,s;
  s:=length(chn)-1;
  //on transforme chn en une liste de ces lettres
  //puis, on enleve le caractere ch de cette liste
  l:=remove(x->(ord(x)==ord(ch)),seq(chn[k],k,0,s));
  //on transforme la liste en chaine
  return expr(char(ord(l)));
};
```

3.15.4 Affichage d'un nombre décimal par tranches de p chiffres

Pour rendre plus facile la lecture d'un nombre décimal, on veut l'afficher par tranches, c'est à dire selon une chaîne de caractères constituées par sa partie entière écrite par tranches de p chiffres, puis ses p premières décimales du nombre et d'un espace, puis les p suivants etc...

Ici, le nombre f peut comporter un exposant à la fin de son écriture.

On écrit le programme qui va afficher le nombre décimal f par tranches de p chiffres :

```
//pour les flottants f utiliser affichef
// appelle affichen et affiche
//par exemple affichef(1234.12345,3)
affichef(f,p):={
  local deb,result,s,indicep,fn,fd,indicee;
```

```
//on suppose f>1
f:=cat(f);
s:=size(f)-1;
indicep:=member(".",seq(f[k],k,0,s));
indicee:=member("e",seq(f[k],k,0,s));
if (indicep!=0) {
fn:=mid(f,0,indicep-1);
fd:=mid(f,indicep-1);
if (indicee!=0) {
return affichen(expr(fn),p)+affiched(expr(mid(fd,0,
indicee-1)),p)+mid(fd,indicee-1);}
return affichen(expr(fn),p)+affiched(expr(fd),p)
}
return affichen(expr(f),p);
};
```

On tape :

```
affichef(1234567.1234567,3)
```

On obtient (pour 12 chiffres significatifs) :

```
"1 234 567.123 46"
```

On obtient (pour 14 chiffres significatifs) :

```
"1 234 567.123 456 7"
```

On obtient (pour 15 chiffres significatifs) :

```
"0.123 456 712 345 670 0*e7"
```

Remarque

La commande `enleve(affichef(q,3)," ")` permet encore de retrouver q .

```
enleve(chn,ch):={
local l,s;
s:=length(chn)-1;
//on transforme chn en une liste de ces lettres
//puis, on enleve le caractere ch de cette liste
l:=remove(x->(ord(x)==ord(ch)),seq(chn[k],k,0,s));
//on transforme la liste en chaine
return expr(char(ord(l)));
};
```

3.16 Écriture décimale d'un nombre rationnel

3.16.1 Algorithme de la potence

Pour obtenir la partie entière et le développement décimal de $\frac{a}{b}$, on va construire deux listes : L1 la liste des restes et L2 la liste des quotients obtenus par l'algorithme de la potence .

On met le quotient q dans L1 et le reste r dans L2.

On a ainsi, la partie entière de $\frac{a}{b}$ dans L1 et comme $\frac{a}{b} = q + \frac{r}{b}$ on cherche la partie entière de $\frac{10 * r}{b}$ qui va rallonger L1 etc...

Si on veut, par exemple, le développement décimal de $\frac{278}{31}$ on cherche : le quotient $q = 8$ et le reste $r = 30$ de la division euclidienne de 278 par 31.

La partie entière est donc 8 et, on met 8L1. Pour avoir la partie décimale de $\frac{278}{31}$, on fait comme à la main l'algorithme de la puissance : on multiplie le reste trouvé par 10, on trouve 300 puis on le divise par 31 : le quotient trouvé 9 est rajouté à L1 et le reste est rajouté à L2 etc...

On écrit la fonction puissance qui renvoie dans la première liste la partie entière puis les n décimales de $\frac{a}{b}$ et dans la deuxième liste les restes successifs obtenus.

```

potence(a,b,n):={
  local L1,L2,k;
  b0:=b;
  b:=iquo(a,b0);
  a:=irem(a,b0);
  L1:=[b];
  L2:=[a];
  for (k:=1;k<=n and a!=0;k++){
    b:=iquo(a*10,b0);
    a:=irem(a*10,b0);
    L2:=append(L2,a);
    L1:=append(L1,b);
  };
  return([L1,L2]);
};

```

En exécutant `potence(278,31,20)`, on lit la partie entière de $\frac{278}{31}$ et les chiffres de sa partie décimale dans la première liste et, la suite des restes dans la deuxième liste.

Exercice

Écrire la partie entière et le développement décimal de :

$$a = \frac{11}{7}, b = \frac{15}{14} \text{ et } c = \frac{17}{28}.$$

Calculer $a - b$ et $a - c$ et donner leur partie entière et leur développement décimal.

Que remarquez-vous ?

Exercice Comment modifier L1 et L2 pour que les chiffres de la partie décimale de $\frac{a}{b}$ se lisent par paquet de trois chiffres dans L1.

Avec l'exemple $\frac{278}{31}$ on veut obtenir : $L1 = [8, 967, 741, 935 \dots]$

Tester votre modification pour $\frac{349}{1332}$.

Que remarquez vous ?

3.16.2 Avec un programme

`division(a,b,n,t)` donne la partie entière suivie de n paquets de t décimales (i.e. des $n * t$ premières décimales) de $\frac{a}{b}$.

```

division(a,b,n,t):={

```



```

local L1,L2,p,q,r,k;
L1:=[iquo(a,b)];
r:=irem(a,b);
for (k:=1;k<=n and r!=0;k++) {
q:=iquo(r*10^t,b);
//10^(p-1)<= q <10^p
if (q==0) {p:=1} else {p:=floor(ln(q)/ln(10)+1)};
//on complete par des zeros pour avoir un paquet de t decimales
for (j:=p+1;j<=t;j++){
L1:=append(L1,0);
}
L1:=append(L1,q);
r:=irem(r*10^t,b);
}
return(L1,r);
};

```

On tape pour avoir $5 \times 6 = 30$ decimales :

```
division(2669,201,6,5)
```

On obtient :

```
[13,27860,69651,74129,35323,38308,45771],29
```

3.16.3 Construction d'un rationnel

Trouver un nombre rationnel qui s'écrit :

0.123123123123... se terminant par une suite illimitée de 123.

Trouver un nombre rationnel qui s'écrit :

0.120123123123... se terminant par une suite illimitée de 123.

Écrire un programme qui permet de trouver un nombre rationnel à partir d'un développement décimal périodique.

Réponse :

On écrit la fonction `rationnel` qui a comme le paramètre deux listes `l1` et `l2` :

- `l1` désigne la partie non périodique de ce développement et `l1[0]` désigne la partie entière.

- `l2` représente un développement décimal périodique.

```

rationnel(l1,l2):={
//l1 et l2 sont non vides
local pui,s1,s2,n,p,np,pui,k;
pui:=10;
s2:=size(l2);
n:=l2[0];
for (k:=1;k<s2;k++){
pui:=pui*10;
n:=n*10+l2[k];
}
// 0.123123...=123/999
p:=n/(pui-1);

```

```
//np partie non periodique
np:=l1[0];
s1:=size(l1);
pui:=1;
for (k:=1;k<s1;k++) {
pui:=pui*10;
np:=np+l1[k]/pui;
}
//pu=10^(s1-1)
return(np+p/pui);
};
```

3.17 Développement en fraction continue

3.17.1 Développement en fraction continue d'un rationnel

Les définitions

Théorème 1 Si a et b sont des entiers naturels premiers entre eux, alors il existe des entiers naturels a_0, a_1, \dots, a_n ($0 \leq n$) tels que :

$$\frac{a}{b} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\dots a_{n-2} + \frac{1}{a_{n-1} + \frac{1}{a_n}}}}}$$

Si $b \leq a$ les a_j sont non nuls et, si $a < b$ alors $a_0 = 0$ et les autres a_j sont non nuls.

Définition On pose alors $\frac{a}{b} = (a_0, a_1, \dots, a_n)$ et on dit que (a_0, a_1, \dots, a_n) est une fraction continue : c'est le développement en fraction continue de $\frac{a}{b}$.

Remarque si $b \leq a$ et si $\frac{a}{b} = (a_0, a_1, \dots, a_n)$ alors $\frac{b}{a} = (0, a_0, a_1, \dots, a_n)$.

Réduite et reste On dit que la fraction $\frac{P_p}{Q_p}$ égale à la fraction continue (a_0, a_1, \dots, a_p) ,

où $p \leq n$, est la réduite de rang p de $\frac{a}{b}$ ou que c'est le développement en fraction continue d'ordre p de $\frac{a}{b}$.

On dit que $r = (0, a_{p+1}, \dots, a_n)$ est le reste du développement d'ordre p ($r < 1$) et on a $\frac{a}{b} = (a_0, a_1, \dots, a_p + r) = (a_0, a_1, \dots, a_p, 1/r)$,

$$\frac{a}{b} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\dots a_{p-3} + \frac{1}{a_{p-2} + \frac{1}{a_p + r}}}}}$$

Propriétés des réduites

Si $\frac{P_p}{Q_p}$ égale à la fraction continue (a_0, a_1, \dots, a_p) , où $p \leq n$, est la réduite de rang p de $\frac{a}{b} = (a_0, a_1, \dots, a_n)$, on a :

$P_0 = a_0$

$$Q_0 = 1$$

$$P_1 = a_0 * a_1 + 1$$

$$Q_1 = a_1$$

$$P_p = P_{p-1} * a_p + P_{p-2}$$

$$Q_p = Q_{p-1} * a_p + Q_{p-2}$$

En effet on le montre par récurrence :

$$P_2/Q_2 = a_0 + a_2/(a_1a_2 + 1) \text{ donc}$$

$$P_2 = a_2(a_0 + a_1 + 1) + a_0 = a_2P_1 + P_0 \text{ et}$$

$$Q_2 = a_2a_1 + 1 = a_2Q_1 + Q_0$$

$$(a_0, a_1, \dots, a_p + 1/a_{p+1}) = \frac{P_{p+1}}{Q_{p+1}} \text{ donc}$$

$$P_{p+1}/Q_{p+1} = ((a_p + 1/a_{p+1})P_{p-1} + P_{p-2})/((a_p + 1/a_{p+1})Q_{p-1} + Q_{p-2})$$

$$P_{p+1} = a_{p+1}(a_pP_{p-1} + P_{p-2}) + P_{p-1} = a_{p+1}P_p + P_{p-1} \text{ et}$$

$$Q_{p+1} = a_{p+1}(a_pQ_{p-1} + Q_{p-2}) + Q_{p-1} = a_{p+1}Q_p + Q_{p-1}$$

Les programmes

Le programme f2dfc :

On veut transformer une fraction en son développement en fraction continue :

$$\text{f2dfc}(a/b) = (a_0, a_1, \dots, a_n).$$

Pour obtenir le développement en fraction continue de a/b , on cherche le quotient q et le reste r de la division euclidienne de a par b . On a : $q = a_0$ et $a/b = a_0 + r/b = a_0 + 1/(b/r)$ et, on continue en cherchant la partie entière de b/r qui sera a_1 On reconnaît l'algorithme d'Euclide : la suite (a_0, a_1, \dots, a_n) est donc la suite des quotients de l'algorithme d'Euclide.

On écrit le programme :

```
f2dfc(fract):={
local r,q,l,lres,a,b;
l:=f2nd(fract);
a:=l[0];
b:=l[1];
lres:=[];
while (b>0) {
q:=iquo(a,b)
lres:=concat(lres,q);
r:=irem(a,b);
a:=b;
b:=r;
}
return lres;
}
```

On tape :

```
f2dfc(2599/357)
```

On obtient :

```
[7,3,1,1,3,14]
```

Le programme f2reduites : On veut obtenir la suite des réduites de a/b .

L'algorithme pour obtenir les réduites ressemble beaucoup à l'algorithme que l'on utilise pour obtenir les coefficients u et v de l'identité de Bézout (cf 3.3.5).

En effet on a :

$$P_0 = a_0 = a_0 * 1 + 0 \text{ alors que } v_0 = 0$$

$$Q_0 = 1 = a_0 * 0 + 1 \text{ alors que } u_0 = 1$$

$$P_1 = a_0 a_1 + 1 = P_0 * a_1 + 1 \text{ alors que } v_1 = 1$$

$$Q_1 = a_1 = a_1 * Q_0 + 0 \text{ alors que } u_1 = 0$$

$$P_p = P_{p-1} * a_p + P_{p-2} \text{ alors que } v_p = v_{p-2} - a_{p-2} v_{p-1}$$

$$Q_p = Q_{p-1} * a_p + Q_{p-2} \text{ alors que } u_p = u_{p-2} - a_{p-2} u_{p-1}$$

Ainsi :

$$P_0 = 0 + a_0 * 1 = v_0 - a_0 * v_1 = -v_2$$

$$P_1 = 1 + P_0 * a_1 = v_1 - v_2 * a_1 = v_3$$

$$P_2 = P_0 + P_1 * a_2 = -v_2 + v_3 * a_2 = -(v_2 - v_3 * a_2) = -v_4$$

On a donc pour tout $p \geq 0$, si a_p est la suite des quotients de l'algorithme d'Euclide :

$$Q_p = Q_{p-1} * a_p + Q_{p-2} \text{ avec } Q_{-2} = 1 \text{ et } P_{-1} = 0 \text{ et,}$$

$$P_p = P_{p-1} * a_p + P_{p-2} \text{ avec } P_{-2} = 0 \text{ et } P_{-1} = 1$$

Donc la suite Q_j est donc la suite des $|u|$ et la suite P_j est la suite des $|v|$. Il faut une valeur absolue car $u_p = u_{p-2} - u_{p-1} * a_{p-2}$ donc $Q_0 = u_2$ mais $Q_1 = -u_3$ et donc $Q_2 = u_4$ etc... $Q_n = (-1)^n u_{n+2}$ et,

$$P_0 = -v_2 \text{ mais } P_1 = v_3 \text{ et donc } P_2 = -v_4 \text{ etc... } P_n = (-1)^{n+1} v_{n+2}.$$

On écrit le programme (calqué sur le programme `Bezout` avec les listes) qui transforme une fraction en la suite de ces réduites :

```
f2reduites(fract):={
local lr,q,l,lres,la,lb;
l:=f2nd(fract);
//a:=l[0];b:=l[1];
la:=l[0,1];
lb:=l[1,1];
lres:=[];
while (lb[2]>0) {
q:=iquo(la[2],lb[2]);
lr:=la-q*lb;
lres:=concat(lres,abs(lr[1])/abs(lr[0]));
la:=lb;
lb:=lr;
}
return lres;
}
```

Remarque :

On peut aussi remplacer :

$$lr := la - q * lb;$$

$$lres := concat(lres, abs(lr[1]) / abs(lr[0]));$$

par :

$$lr := la + q * lb;$$

$$lres := concat(lres, lr[1] / lr[0]);$$

On tape :

$$f2reduites(2599/357)$$

On obtient :

[7, 22/3, 29/4, 51/7, 182/25, 2599/357]

Le programme dfc2reduites : On veut obtenir la suite des réduites d'une liste l (qui sera par exemple le développement en fraction continue de a/b) On écrit le programme (calqué sur le programme Bezout sans les listes) qui transforme une liste $[a_0, a_1, \dots, a_n]$ en la liste $[a_0, a_0 + 1/a_1, \dots, (a_0 + 1/a_1 + 1/\dots + 1/a_{n-1} + 1/a_n)]$:

```
dfc2reduites(l) := {
  local s, p0, q0, p1, q1, p, q, lres, j;
  s := size(l);
  lres := [];
  p0 := 0;
  p1 := 1;
  q0 := 1;
  q1 := 0;
  for (j := 0; j < s; j++) {
    p := p0 + l[j] * p1;
    q := q0 + l[j] * q1;
    lres := concat(lres, p/q);
    p0 := p1;
    q0 := q1;
    p1 := p;
    q1 := q;
  }
  return lres;
}
```

On remarquera que :

- la suite des P est initialisée par p_0 et p_1 , puis, quand $j = 0$, on fait le calcul de P_0 qui est mis dans p , puis, quand $j = 1$ on fait le calcul de P_1 qui est mis dans p , etc... et que

- la suite des Q est initialisée par : q_0 et q_1 , puis, quand $j = 0$ on fait le calcul de Q_0 qui est mis dans q , quand $j = 1$, on fait le calcul de Q_1 est mis dans q , etc...

On tape :

```
dfc2reduites([7, 3, 1, 1, 3, 14])
```

On obtient :

[7, 22/3, 29/4, 51/7, 182/25, 2599/357]

3.17.2 Développement en fraction continue d'un réel quelconque

Théorème2 Si α est un nombre réel non rationnel, alors il existe des entiers naturels non nuls a_0, a_1, \dots, a_n et un réel $\beta < 1$ tels que :

$$\alpha = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\dots a_{n-2} + \frac{1}{a_{n-1} + \frac{1}{a_n + \beta}}}}}$$

On dit que (a_0, a_1, \dots, a_n) est le développement en fraction continue d'ordre $n + 1$ de α et que β est le reste de ce développement.

Un rationnel a un développement en fraction continue fini et réciproquement, un développement en fraction continue fini représente un rationnel.

Un réel non rationnel a un développement en fraction continue infini.

Si α est un nombre quadratique (i.e. α est racine d'une équation du second degré), α a un développement en fraction continue périodique et réciproquement, un développement en fraction continue périodique représente un nombre quadratique.

3.17.3 Les programmes

On va écrire deux fonctions `r2dfc` et `dfc2r`.

`r2dfc`

`r2dfc(alpha, n)` qui transforme un réel `alpha` en son développement en fraction continue et qui renvoie deux listes, soit :

- `[a0, a1...ap], []` avec $p \leq n$ où les a_j sont des entiers, la deuxième liste est vide et la première liste est le développement en fraction continue de `alpha` (les a_j sont des entiers et donc `alpha` est une fraction)

- `[a0, a1...an-1, b], []`, la deuxième liste est vide et la première liste est le développement en fraction continue d'ordre $n - 1$ de `alpha` suivi de $b > 1$ (le reste est égal à $1/b$), où les a_j sont des entiers et b est un réel plus grand que 1, soit,

- `[a0, a1...ap], [ar, ..ap]` avec $r \leq p < n$

où les a_j sont des entiers, la première liste est le développement en fraction continue d'ordre p de `alpha` et la deuxième liste représente la période de développement en fraction continue (les a_j sont des entiers et donc `alpha` est un nombre quadratique)

.

On remarquera dans le programme ci-dessous que :

`a0 = floor(alpha) = q` remplace `q := iquo(a, b)` lorsque `alpha=a/b`

et que `r=alpha-q` remplace `irem(a, b)/b` lorsque `alpha=a/b`

et donc que si `r=alpha-q`, `a1 = floor(1/r)` etc...

Le problème ici est de pouvoir comparer `alpha` et `q` c'est à dire savoir si `r==0` et pour cela on est obligé de faire les calculs avec beaucoup de décimales c'est à dire d'augmenter le nombre de digits (on tape par exemple `DIGITS :=30`). Il faut bien sûr repérer la période, pour cela on forme la liste `lr` des restes successifs. La liste `lq` des parties entières successives forme le début du développement en fraction continue.

```
r2dfc(alpha, n) := {
  local r, q, lq, lr, p, j;
  q:=floor(alpha);
  r:=normal(alpha-q);
  lq:=[];
  lr:=[];
  for (j:=1; j<=n; j:=j+1) {
    lq:=concat(lq, q);
    if (r==0){return (lq, []);}
    p:=member(r, lr);
    if (p) {return (lq, mid(lq, p));}
    lr:=concat(lr, r);
    alpha:=normal(1/r);
```

```

q:=floor(alpha);
r:=normal(alpha-q);
}
return (concat(lq,alpha),[]);
};

```

On tape :

dfc2r(sqrt(2),1) On obtient :

```
([1,sqrt(2)+1],[])
```

On tape :

dfc2r(sqrt(2),2) On obtient :

```
([1,2],[2])
```

On tape :

dfc2r(pi),6 On obtient :

```
([3,7,15,1,292,1,(-33102*pi+103993)/(33215*pi-104348)],[])
```

Remarque Le premier argument de doit être un nombre exact, car sinon les calculs sont faits en mode approché et le test $r=0$ n'est jamais réalisé...

dfc2r

On écrit la fonction réciproque de $r2dfc$ qui à partir d'un développement en fraction continue et d'un reste éventuel ou d'un développement en fraction continue et d'une période éventuelle renvoie un réel.

$dfc2r(d,t)$ transforme en un réel, la liste d représente un développement en fraction continue et la liste t représente la période.

On remarquera que lorsque la liste t n'est pas vide il faut déterminer le nombre $0 < y < 1$ qui admet cette liste périodique comme développement en fraction continue et pour ce faire résoudre l'équation :

$y = (0, t_0, \dots, t_{st-1} + y)$ le reste est alors $y + d_{s-1}$ ($s := \text{size}(d)$).

On écrit le programme :

```

dfc2r(d,t):={
local s,st,alpha,l,ap,k;
s:=size(d);
alpha:=d[s-1];
for (k:=s-2;k>=0;k:=k-1) {alpha:=normal(d[k]+1/alpha);}
if (t==[]) {return normal(alpha);}
st:=size(t);
purge(y);
ap:=t[st-1]+y;
for (k:=st-2;k>=0;k:=k-1) {ap:=normal(t[k]+1/ap);}
l:=solve(y=1/ap,y);
if (l[0]>0){y:=normal(l[0]);}else{y:=normal(l[1]);};
alpha:=d[s-1]+y;
for (k:=s-2;k>=0;k:=k-1) {alpha:=normal(d[k]+1/alpha);}
return(normal(alpha));
};

```

ou avec une écriture plus concise :

```

dfc2r(d,t):={
local s,st,alpha,l,ap,k;
s:=size(d);
st:=size(t);
if (st==0)
  {y:=0;}
  else
  {purge(y);
  ap:=t[st-1]+y;
  for (k:=st-2;k>=0;k:=k-1) {ap:=normal(t[k]+1/ap);}
  l:=solve(y=1/ap,y);
  if (l[0]>0){y:=normal(l[0]);}else{y:=normal(l[1]);};
  }
alpha:=d[s-1]+y;
for (k:=s-2;k>=0;k:=k-1) {alpha:=normal(d[k]+1/alpha);}
return(normal(alpha));
};

```

3.17.4 Exemples

1/ Développement en fraction continue de : $\frac{1393}{972}$, $1 + \sqrt{13}$ et $1 - \sqrt{13}$.

On a :

$r2dfc(1393/972,3)=[1,2,3,130/31], []$

$r2dfc(1393/972,7)=[1,2,3,4,5,6], []$

et on a bien :

$r2dfc(130/31,3)=[4,5,6], []$

$r2dfc(31/130,4)=[0,4,5,6], []$

On peut vérifier que :

$dfc2r([1,2,3,4,5,6], [])=1393/972$

$dfc2r([1,2,3+31/130], [])=dfc2r([1,2,3,130/31], [])=1393/972$

On a :

$r2dfc(1+\sqrt{13},3)=[4,1,1,(\sqrt{13}+2)/3], []$

$r2dfc(1+\sqrt{13},6)=[4,1,1,1,1,6], [1,1,1,1,6]$

$r2dfc(1-\sqrt{13},7)=[-3,2,1,1,6,1,1], [1,1,6,1,1]$

2/ Trouver les réels qui ont comme développement en fraction continue :

$[2,4,4,4,4,4,\dots]$ (suite illimitée de 4) et

$[1,1,1,1,1,1,\dots]$ (suite illimitée de 1).

On a :

$dfc2r([2,4],[4])=\sqrt{5}$ ou encore $dfc2r([2],[4])=\sqrt{5}$ On

a :

$dfc2r([1],[1])=(\sqrt{5}+1)/2$

3.17.5 Suite des réduites successives d'un réel

Si α a comme développement en fraction continue $(a_0, a_1, \dots, a_n, \dots)$, la suite des réduites est la suite des nombres rationnels ayant comme développement en fraction continue : $(a_0), (a_0, a_1), \dots, (a_0, a_1, \dots, a_n), \dots$

On écrit le programme permettant d'obtenir les p premières réduites de α .

On écrit le programme naïf `reduiten` (on recalcule les réduites sans se servir des relations de récurrence) :

```
reduiten(alpha,p):={
local l,k,ld,lt,st,s,q,lred,relu;
ld:=r2dfc(alpha,p);
l:=ld[0];
s:= size(l);
if (s<p) {
  lt:=ld[1];
  st:=size(lt);
  if (st!=0){
    q:=iquo(p-s,st);
    for (j:=0;j<=q; j++){
      l:= concat(l,lt)
    }
  }
  else {
    p:=s;
  }
}
lred:=[];
for (k:=1;k<=p;k++){
  relu:=dfc2r(mid(l,0,k),[]);
  lred:=append(lred,relu);
}
return (lred);
};
```

```
reduiten(sqrt(53),5)
```

On obtient :

```
[7, 22/3, 29/4, 51/7, 182/25]
```

On écrit maintenant le programme `reduite` permettant d'obtenir les p premières réduites de α , en se servant de la fonction `dfc2reduites` écrite auparavant et qui utilise les relations de récurrence.

```
reduite(alpha,p):={
local l,ld,lt,st,s,q,lred;
ld:=r2dfc(alpha,p);
l:=ld[0];
s:= size(l);
if (s<p) {
  lt:=ld[1];
  st:=size(lt);
  if (st!=0){
    q:=iquo(p-s,st);
    for (j:=0;j<=q; j++){
      l:= concat(l,lt)
    }
  }
}
```

```

    }
  }
}
l:= mid(l,0,p);
lred:=dfc2reduites(l);
return lred;
}

```

On tape :

```
reduite(sqrt(53),5)
```

On obtient :

```
[7,22/3,29/4,51/7,182/25]
```

On tape :

```
reduite(11/3,2)
```

On obtient :

```
[3,4]
```

3.17.6 Suite des réduites "plus 1" successives d'un réel

Si α a comme développement en fraction continue $(a_0, a_1, \dots, a_n, \dots)$, la suite des réduites "plus 1" est la suite des nombres rationnels ayant comme développement en fraction continue : $(a_0 + 1), (a_0, a_1 + 1), \dots, (a_0, a_1, \dots, a_n + 1), \dots$. On écrit le programme permettant d'obtenir les p premières réduites "plus 1" de α .

```

reduite1(alpha,p):={
local l,ld,lt,st,s,q,lred;
ld:=r2dfc(alpha,p);
l:=ld[0];
s:= size(l);
if (s<p) {
  lt:=ld[1];
  st:=size(lt);
  if (st!=0) {
    q:=iquo(p-s,st);
    for (j:=0;j<=q;j++){
      l:= concat(l,lt)
    }
  }
}
l:= mid(l,0,p)+1;
lred:=dfc2reduites(l);
return lred;
}

```

3.17.7 Propriété des réduites

Propriété des réduites de α :

Une réduite p/q approche α à moins de $1/q^2$ et si s/t est la réduite plus 1 de même rang n on a :

- $|p/q - s/t| < 1/q^2$
- si n est pair $p/q \leq \alpha \leq r/s$
- si n est impair $r/s \leq \alpha \leq p/q$
- les réduites de rang pair et les réduites de rang impair forment deux suites adjacentes qui convergent vers α
- les réduites plus 1 de rang pair et les réduites plus 1 de rang impair forment deux suites adjacentes qui convergent vers α

Donc, si on pose :

$lred := reduite(\alpha, 10)$ et $lred1 := reduite1(\alpha, 10)$, ces deux suites $lred$ et $lred1$ fournissent un encadrement de α plus précisément on a :

$$lred[0] \leq lred1[1] \leq \dots \leq lred[2p] \leq lred1[2p+1] < \alpha < lred[2p+1] \leq lred1[2p] \leq \dots \leq lred[1]$$

c'est à dire que l'encadrement fait avec 2 réduites successives de rang $p-1$ et p est moins bon que l'encadrement fait avec la réduite de rang p et la réduite plus 1 de rang p . Exemple

On a :

$$r2dfc(\sqrt{53}, 5) = [7, 3, 1, 1, 3, \sqrt{53}+7], []$$

$$dfc2r([7, 3, 1, 1, 3], []) = 182/25$$

$$reduite(\sqrt{53}, 5)[4] = 182/25 = 7.28$$

$$reduite1(\sqrt{53}, 5)[4] = 233/32 = 7.28125$$

$$reduite(182/25, 5)[4] = 182/25 = 7.28$$

$$reduite1(182/25, 5)[4] = 233/32 = 7.28125$$

$$\text{et donc } 7.28 < \sqrt{53} < 7.28125$$

$$r2dfc(\sqrt{53}, 6) = [7, 3, 1, 1, 3, 14], [3, 1, 1, 3, 14]$$

$$dfc2r([7, 3, 1, 1, 3, 14], []) = 2599/357$$

$$reduite(\sqrt{53}, 6)[5] = 2599/357 = 7.28011204482$$

$$reduite1(\sqrt{53}, 6)[5] = 2781/382 = 7.28010471204$$

$$reduite(2599/357, 5)[4] = 2599/357 = 7.28011204482$$

$$reduite1(2599/357, 5)[4] = 2781/382 = 7.28010471204$$

$$\text{et donc } 7.28010471204 < \sqrt{53} < 7.28011204482$$

$$\text{On a } 1/357^2 = 7.84627576521e-06 \text{ et } 1/382^2 = 6.8528823223e-06$$

3.18 Suite de Hamming

3.18.1 La définition

La suite de Hamming est la suite des nombres entiers qui n'ont pour diviseurs premiers que 2, 3 et 5.

Cette suite commence par : [2,3,4,5,6,8,9,10,12,15,16,18,20,24,25...]

3.18.2 L'algorithme à l'aide d'un crible

On écrit tous les nombres de Hamming de 0 à $n > 0$ et on barre les nombres qui sont de la forme $2^a * 3^b * 5^c$ avec a, b, c variant de 0 à un nombre tel que $2^a * 3^b * 5^c \leq n$: les nombres barrés (excepté 1) sont les nombres de Hamming inférieurs à $n > 0$.

Voici la fonction `hamming(n)` écrite en `xcas` pour obtenir les nombres de Hamming inférieurs à $n > 0$.

```
hamming(n) := {
  local H, L, a, b, c, j, d;
  L := makelist(x -> x, 0, n);
  // les nbres de Hamming sont  $2^a * 3^b * 5^c$ 
  c := 0; b := 0; a := 0;
  d := 1;
  while (d <= n) {
    while (d <= n) {
      while (d <= n) {
        L[d] := 0;
        // d := 5 * d
        c := c + 1;
        d := 2^a * 3^b * 5^c;
      }
      c := 0;
      b := b + 1;
      // d := 2^a * 3^b * 5^c
      d := 2^a * 3^b;
    }
    // c := 0;
    b := 0;
    a := a + 1;
    // d := 2^a * 3^b * 5^c
    d := 2^a;
  }
  H := [];
  for (j := 2; j <= n; j++) {
    if (L[j] == 0) H := append(H, j);
  }
  return H;
}
```

ou encore en supprimant la variable `c` :

```
hamming(n) := {
  local H, L, a, b, j, d;
  L := makelist(x -> x, 0, n);
  // les nbres de Hamming sont  $2^a * 3^b * 5^c$ 
  a := 0;
  d := 1;
  while (d <= n) {
    b := 0;
    while (d <= n) {
      while (d <= n) {
        L[d] := 0;
        d := 5 * d;
      }
    }
  }
```

```

    }
    b:=b+1;
    d:=2^a*3^b;
  }
  a:=a+1;
  d:=2^a;
}
H:=[];
for (j:=2;j<=n;j++) {
  if (L[j]==0) H:=append(H,j);
}
return H;
}

```

ou encore en supprimant a,b,c et en preservant la valeur de d avant les while :

```

hamming(n):={
  local H,L,d,j,k;
  L:=makelist(x->x,0,n);
  //les nbres de Hamming sont 2^a*3^b*5^c
  d:=1;
  while (d<=n) {
    j:=d;
    while (j<=n){
      k:=j;
      while (k<=n) {
L[k]:=0;
k:=5*k;
      }
      j:=3*j;
    }
    d:=2*d;
  }
  H:=[];
  for (j:=2;j<=n;j++) {
    if (L[j]==0) H:=append(H,j);
  }
  return H;
}

```

On tape :

hamming(20)

On obtient :

[2,3,4,5,6,8,9,10,12,15,16,18,20] **On tape :**

hamming(40)

On obtient :

[2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,40]

3.18.3 L'algorithme sans faire un crible

Supposons que l'on ait trouvé les premiers éléments de cette suite par exemple : 2,3,4,5.

L'élément suivant est obtenu en multipliant une des cases précédentes par 2, 3 ou 5.

Le problème c'est d'avoir les éléments suivants dans l'ordre....

Comment trouver l'élément suivant de $H := [2, 3, 4, 5]$:

on a déjà multiplié $H[0] = 2$ par 2 pour obtenir 4 donc

on peut multiplier $H[1] = 3$ par 2 pour obtenir $m=6$ ou

multiplier $H[0] = 2$ par 3 pour obtenir $p=6$ ou

multiplier $H[0] = 2$ par 5 pour obtenir $q=10$.

L'élément suivant est donc $6 = \min(6, 6, 10)$ et $H := [2, 3, 4, 5, 6]$.

Maintenant, on a déjà multiplié $H[0] = 2$ par 2 et par 3 pour obtenir 4 et 6 et

on a déjà multiplié $H[1] = 3$ par 2 pour obtenir 6 donc
donc

on peut multiplier $H[2] = 4$ par 2 pour obtenir $m=8$ ou

multiplier $H[1] = 3$ par 3 pour obtenir $p=9$ ou

multiplier $H[0] = 2$ par 5 pour obtenir $q=10$.

L'élément suivant est donc $8 = \min(8, 9, 10)$ et $H := [2, 3, 4, 5, 6, 8]$.

Pour que chaque terme de la suite soit multiplié par 2, par 3 et par 5, il faut donc prévoir 3 indices :

k_0 qui sera l'indice de l'élément qu'il faut multiplier par 2,

k_1 qui sera l'indice de l'élément qu'il faut multiplier par 3,

k_2 qui sera l'indice de l'élément qu'il faut multiplier par 5.

Cela signifie que :

pour tout $r < k_0$ les $2 * H[r]$ ont déjà été rajoutés,

pour tout $r < k_1$ les $3 * H[r]$ ont déjà été rajoutés,

pour tout $r < k_2$ les $5 * H[r]$ ont déjà été rajoutés,

Naturellement $k_0 \geq k_1 \geq k_2$.

Les 3 candidats pour être l'élément suivant sont donc :

$2 * H[k_0]$, $3 * H[k_1]$, $5 * H[k_2]$

l'un de ces éléments est plus petit que les autres et on le rajoute à la suite. Il faut alors augmenter l'indice correspondant de 1 : par exemple si c'est $3 * H[k_1]$ qui est le minimum il faut augmenter k_1 de 1 et si $3 * H[k_1] = 5 * H[k_2]$ est le minimum, il faut augmenter k_1 et k_2 de 1.

3.18.4 La traduction de l'algorithme avec xcas

`hamming(n)` va renvoyer les n premiers éléments de la suite de Hamming.

L'indice j sert simplement à compter les éléments de H .

k est une suite qui contient les indices k_0, k_1, k_2 .

On peut initialiser H à $[2, 3, 4, 5]$ donc j à 4, et k à $[1, 0, 0]$ (car $H[0] = 2$ a été multiplié par 2, mais pas par 3, ni par 5) mais cela suppose $n > 3$.

On peut aussi initialiser H à $[1]$, k à $[0, 0, 0]$ ($H[0] = 1$ n'a pas été multiplié par 2, ni par 3, ni par 5) et j à 0 puis enlever 1 de H à la fin car 1 n'est pas un terme de la suite.

Voici la fonction `hamming(n)` écrite en `xcas` pour $n > 3$.

```
//pour n>3
hamming(n):={
  local H,j,k,m,p,q,mi;
  H:=[2,3,4,5];
  j:=4;
  k:=[1,0,0];
  while (j<n) {
    m:=2*H[k[0]];
    p:=3*H[k[1]];
    q:=5*H[k[2]];
    mi:=min(m,p,q);
    H:=append(H,mi);
    j:=j+1;
    if (mi==m) {k[0]:=k[0]+1};
    if (mi==p) {k[1]:=k[1]+1};
    if (mi==q) {k[2]:=k[2]+1};
  }
  return H;
}
```

Voici la fonction hamming(n) écrite en xcas pour n>0.

```
//pour n>0
hamming(n):={
  local H,j,k,m,p,q,mi;
  H:=[1];
  j:=0;
  k:=[0,0,0];
  while (j<n) {
    m:=2*H[k[0]];
    p:=3*H[k[1]];
    q:=5*H[k[2]];
    mi:=min(m,p,q);
    H:=append(H,mi);
    j:=j+1;
    if (mi==m) {k[0]:=k[0]+1};
    if (mi==p) {k[1]:=k[1]+1};
    if (mi==q) {k[2]:=k[2]+1};
  }
  return tail(H);
}
```

On tape :

hamming(20)

On obtient :

[2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,40]

Chapitre 4

codage

4.1 Codage de Jules Cesar

4.1.1 Introduction

Le principe est simple : on écrit un message en n'utilisant que les 26 lettres de l'alphabet et on le code en remplaçant une lettre par une autre lettre. Ceci peut être considéré comme une application f de l'ensemble des lettres $\{A,B,C,...X,Y,Z\}$ dans lui-même.

Pour pouvoir décoder, il faut que l'application f ci-dessus soit bijective !

Il paraît que Jules César utilisait cette méthode pour communiquer ses ordres.

4.1.2 Codage par symétrie point ou par rotation d'angle π

FIG. 4.1 – Codage par symétrie point

Une façon simple de coder est la suivante : on écrit les lettres sur un cercle (de façon équirépartie) et on remplace chaque lettre par la lettre symétrique par rapport au centre du cercle (voir figure 4.1).

Le décodage s'obtient de la même façon car ici $f \circ f = Id$

4.1.3 Avec les élèves

On explique que l'on va coder un message en remplaçant une lettre par une autre (on suppose que le message n'utilise que les 26 lettres de l'alphabet et que l'on n'écrit pas les espaces).

Pour cela on distribue une feuille sur laquelle figure quatre cercles divisés en 26 parties égales.

On écrit sur ce cercle les 26 lettres de l'alphabet et le codage consiste à remplacer chaque lettre du message par la lettre diamétralement opposée sur le cercle.

Par exemple voici le message à coder selon cette méthode :

"BONJOURLESAMIS"

Le message à décoder est donc :

"OBAWBHEYRFNZVF"

Quel sont les éléments pertinents qu'il faut transmettre pour que le décodage soit possible ?

Parmi les réponses il est apparu qu'il fallait ajouter +13 pour décoder.

Puis chaque élève invente un codage et écrit un message selon son codage et le donne à décoder à son voisin, par écrit avec les explications nécessaires pour le décoder.

Voici quelques codages obtenus :

- codage obtenu en remplaçant chaque lettre par celle qui la suit dans l'alphabet,
- codage obtenu en remplaçant chaque lettre par celle qui est obtenue en avançant de +4 sur la roue (ou en reculant de 3 etc...),
- codage obtenu en remplaçant chaque lettre par celle qui est obtenue par symétrie par rapport à la droite $[A,N]$ (verticale sur le dessin), ou par symétrie par rapport à la droite horizontale sur le dessin,
- codage par symétrie par rapport au centre du cercle mais où l'ordre des lettres sur le cercle n'est pas respecté,
- d'autres codages comme de remplacer le message par une suite de nombres (intéressant mais cela ne répond pas à la question posée),
- codage qui dépend de la position de la lettre dans le message. Ce codage n'est pas une application puisque une même lettre peut avoir des codages différents.

Combien y-a-t-il de codages (i.e de bijections) possibles ?

Il a fallu parler de bijections :

un étudiant a dit qu'il fallait que deux lettres différentes soient codées par des lettres différentes pour que le décodage soit possible (injection).

Ceci entraîne que toutes les lettres sont le codage d'une autre lettre (surjection).

Pour simplifier on a préféré parler de permutations des lettres avec comme exemple : trouver tous les codages possibles si on suppose que l'alphabet utilisé ne comporte que les trois lettres A, B, C.

Donnez un ordre de grandeur de 26 !

A supposer que vous vouliez écrire les 26 ! codages possibles sur un cahier et que votre rythme est de 1 codage par seconde (vous êtes super rapide !!!... félicitations !!!) combien de temps (réponse en heures, en mois, en années... ?) vous faut-il ?

Combien de cahiers de 10000 lignes (200 pages de 50 lignes) vous faut-il ? Donner la longueur occupée par ces cahiers dans une bibliothèque, si chaque cahier occupe 1 cm. Combien y-a-t-il de codages involutifs possibles qui sont sans point double (c'est à dire de bijections f vérifiant $f = f^{-1}$ et $f(x) \neq x$ pour tout x) ?

Comment faire pour que la clé du décodage soit simple ?

4.1.4 Travail dans $\mathbb{Z}/26\mathbb{Z}$

Le codage de Jules César consiste à faire une symétrie point ou encore une rotation d'angle π .

Si on numérote les lettres de 0 à 25, le codage consiste donc à faire une addition de 13 (modulo 26). (voir figure 4.1).

Exemple :

"BONJOUR" sera codé par "OBAWBHE"

4.1.5 Codage par rotation d'angle $k * \pi/13$

Le principe est le même :

on écrit les lettres sur un cercle (de façon équirépartie) et on remplace chaque lettre par la lettre obtenue par rotation d'angle $k * \pi/13$ ($0 \leq k \leq 25$) (k est un entier). Si on numérote les lettres de 0 à 25, le codage consiste donc à faire subir à chaque lettre un décalage de k c'est à dire à faire subir à son numéro une addition de k (modulo 26) (voir figure 4.2).

On remarquera que si le paramètre de codage par rotation est k , le décodage sera un codage par rotation de paramètre $-k$ ou encore $26-k$

FIG. 4.2 – Codage par rotation de $5 * \pi/13$

4.2 Écriture des programmes correspondants

4.2.1 Passage d'une lettre à un entier entre 0 et 25

À chaque lettre on peut faire correspondre son code ASCII.

Avec `xcas`, `asc("A")=[65]` et `asc("BON")=[66,79,78]`.

Donc pour avoir un entier entre 0 et 25 il suffit de retrancher 65 :

`asc("A")-65(=0)` ou `asc("BON")-[65,65,65](=[1,14,13])`.

On écrit donc la procédure `c2n` qui transforme une chaîne de caractères `m` en une liste d'entiers `l=c2n(m)` entre 0 et 25 (le 2 de `c2n` veut dire "to" ou "vers" en français).

Il faut créer une liste formée des nombres 65 et de même longueur que le message avec `makelist(65,1,size(m))`.

On écrit :

```
c2n(m):={
return(asc(m)-makelist(65,1,size(m)));
}
```

Exemple :

`c2n("BONJOUR")=[1,14,13,9,14,20,17]`

4.2.2 Passage d'un entier entre 0 et 25 à une lettre

À chaque entier n compris entre 0 et 25, on fait correspondre la $(n-1)^{ieme}$ lettre en majuscule de l'alphabet (à 0 correspond "A", à 1 correspond "B" etc...).

Avec `xcas`, `char(65)="A"` et `char([66,79,78])="BON"`.

On écrit donc la procédure `n2c` qui transforme une liste d'entiers `l` entre 0 et 25 en une chaîne de caractères `m`.

Il faut penser à ajouter 65 à tous les éléments de la liste `l` (on forme une liste formée de 65 avec la fonction `makelist`: `l+makelist(65,1,size(l))`).

On écrit :

```
n2c(l):={
return(char(l+makelist(65,1,size(l))));
}
```

Exemple :

```
n2c([1,14,13,9,14,20,17])="BONJOUR"
```

4.2.3 Passage d'un entier k entre 0 et 25 à l'entier $n+k \bmod 26$

On écrit donc la procédure `decal` de paramètres n et l qui transforme une liste l d'entiers k entre 0 et 25 en la liste d'entiers $n + k \bmod 26$.

On écrit :

```
decal(n,l):={
return(irem(l+makelist(n,1,size(l)),26));
}
```

Exemple :

```
decal(13,[1,14,13,9,14,20,17])=[14,1,0,22,1,7,4]
```

4.2.4 Codage d'un message selon Jules César

On écrit la procédure finale `CESAR` qui a deux paramètres : l'entier n de décalage et le message m .

On écrit :

```
cesar(n,m):={
return(n2c(decal(n,c2n(m))));
}
```

Exemple :

```
cesar(13,"BONJOUR")="OBAWBHE"
```

4.3 Codage en utilisant une symétrie par rapport à un axe

FIG. 4.3 – Codage par symétrie par rapport à Ox

4.3.1 Passage d'un entier k entre 0 et 25 à l'entier $n-k \bmod 26$

On reprend les procédures `c2n` et `n2c` vues précédemment :

`c2n` transforme une chaîne de caractères m en une liste d'entiers entre 0 et 25 et `n2c` transforme une liste l d'entiers entre 0 et 25 en une chaîne de caractères m .

On écrit ensuite la procédure `sym` de paramètres n et l qui transforme une liste l d'entiers k entre 0 et 25 en la liste d'entiers $n - k \bmod 26$. On peut considérer que le paramètre n détermine le diamètre D perpendiculaire à la corde $[0, n]$ (joignant A à la $(n-1)$ ième lettre).

La procédure `sym` de paramètre n est donc une symétrie par rapport à la droite D .

On écrit :

```
sym(n,l):={
return(irem(makelist(n,1,size(l))-1,26));
}
```

4.3.2 Codage d'un message selon une symétrie droite D

On écrit la procédure finale `cesarsym` qui a deux paramètres : l'entier n (définissant la corde $[0, n]$ normale au diamètre D) et le message m . On écrit :

```
cesarsym(n,m) := {
  return(n2c(sym(n, c2n(m)))) ;
}
```

Exemple :

Si on prend $n=13$ on réalise une symétrie par rapport à la droite Ox (cf figure 4.3).
`cesarsym(13, "BONJOUR") = "MZA EZTW"`

4.4 Codage en utilisant une application affine

Comme précédemment, on écrit la procédure `c2n` qui transforme une chaîne de caractères m en une liste d'entiers entre 0 et 25 et on écrit la procédure `n2c` qui transforme une liste l d'entiers entre 0 et 25 en une chaîne de caractères m .

On écrit ensuite la procédure `affine` de paramètre a, b, l qui transforme une liste l d'entiers k entre 0 et 25 en la liste d'entiers $a * k + b \bmod 26$.

On écrit :

```
affine(a,b,l) := {
  return(irem((a*l+makelist(b,l,size(l))),26)) ;
}
```

On écrit ensuite :

```
cesaraffine(a,b,m) := {
  return(n2c(affine(a,b,c2n(m)))) ;
}
```

Question :

Pour quelles valeurs de a et b le codage obtenu par `cesaraffine` peut-il être décodé ?

4.5 Codage en utilisant un groupement de deux lettres

On écrit la procédure `c2n2` qui transforme une chaîne de caractères m en une liste l d'entiers entre 0 et $26^2 - 1 = 675$:

On fait des groupements de deux lettres (quitte à terminer le message par la lettre "F" pour avoir un nombre pair de lettres), chaque groupement est considéré comme l'écriture en base 26 d'un entier en utilisant comme "chiffre" les lettres majuscules. Ainsi, "BC" est l'écriture en base 26 de 28 ($28=1*26+2$).

On écrit :

```
c2n2(m) := {
  local s,lr,l,n;
  s:=size(m) ;
```

```

if (irem(s,2)==1) {
m:=append(m, "F");
s:=s+1;
}
lr:=[];
l:=asc(m);
for (k:=0;k<s;k:=k+2) {
n:=l[k]*26+l[k+1];
lr:=append(lr,n);
}
return(lr);
}

```

On écrit ensuite la procédure `n2c2` qui transforme une liste d'entiers entre 0 et 675 ($675=25*26+25=26*26-1$) en une chaîne de caractères `m` :

chaque entier étant écrit en base 26 avec comme "symboles" les 26 lettres majuscules.

On écrit :

```

n2c2(l):={
local s,n,m;
s:=size(l);
m:=" ";
for (k:=0;k<s;k++){
n:=l[k];
m:=append(m, char(iquo(n,26)+65));
m:=append(m, char(irem(n,26)+65));
}
return(m);
}

```

On écrit ensuite la fonction `affin2` de paramètre `a, b, l` qui transforme une liste `l` d'entiers `k` entre 0 et 675 en la liste d'entiers $a * k + b \bmod 676$ (entiers encore compris entre 0 et 675).

On écrit :

```

affin2(a,b,l):={
local s;
s:=size(l);
for (k:=0;k<s;k++){
l[k]:=irem(a*l[k]+b,676);
}
return(l);
}

```

On écrit ensuite la fonction `cesar2` qui réalise le codage par groupement de 2 lettres utilisant l'application affine `affin2` :

```

cesar2(a,b,m):={
return(n2c2(affin2(a,b,c2n2(m)))));
}

```

Question :

Pour quelles valeurs $a1$ de a et $b1$ de b , le codage obtenu par `cesara` affine peut-il être décodé ?

Réponse :

On doit avoir $a1 * (a * n + b) + b1 = a1 * a * n + a1 * b + b1 = n$.

Il suffit donc de prendre : $b1 = -a1 * b \bmod 676$ et $a1 * a = 1 \bmod 676$

4.6 Le codage Jules César et le codage linéaire

4.6.1 Les caractères et leurs codes

Ici, on ne se borne plus aux 26 lettres de l'alphabet, mais on veut pouvoir utiliser les 101 caractères de la table ci-dessous.

Dans cette table, on a le code du caractère, puis, le caractère : ainsi "5" a pour code 21 et "R" a pour code 50 et l'espace " " a pour code 0.

0									
1 !	2 "	3 #	4 \$	5 %	6 &	7 '	8 (9)	10 *
11 +	12 ,	13 -	14 .	15 /	16 0	17 1	18 2	19 3	20 4
21 5	22 6	23 7	24 8	25 9	26 :	27 ;	28 <	29 =	30 >
31 ?	32 @	33 A	34 B	35 C	36 D	37 E	38 F	39 G	40 H
41 I	42 J	43 K	44 L	45 M	46 N	47 O	48 P	49 Q	50 R
51 S	52 T	53 U	54 V	55 W	56 X	57 Y	58 Z	59 [60 \
61]	62 ^	63 _	64 '	65 a	66 b	67 c	68 d	69 e	70 f
71 g	72 h	73 i	74 j	75 k	76 l	77 m	78 n	79 o	80 p
81 q	82 r	83 s	84 t	85 u	86 v	87 w	88 x	89 y	90 z
91 {	92	93 }	94 ~	95 ê	96 ù	97 ç	98 à	99 è	100 é

4.6.2 Les différentes étapes du codage

Voici les différentes étapes du codage (elles seront programmées avec `xcas` dans le paragraphe suivant) :

1. À chaque lettre ou symbole, on associe un nombre, comme dans la table 4.6.1. Un texte devient ainsi une suite de nombres : par exemple `ê` sera codée par 95 et `BONJOUR` par 34 47 46 42 47 53 50.

La fonction `code_c2n` réalise cette étape : elle transforme un caractère en un nombre n ($0 \leq n < 101$), selon la table 4.6.1.

2. On effectue une ou plusieurs opérations sur ces nombres :

- pour le codage de Jules César, on ajoute à ces nombres un nombre fixé appelé clef de chiffrement (par exemple 17), puis on prend le reste de la division par 101 des nombres obtenus. Ainsi avec ce codage, `ê` est transformé 95 qui est transformé en 11 ($95+17=112=101+11$, ou encore, 11 est le reste de la division de 112 par 101).

Avec `xcas`, on effectue la transformation de n avec la commande `irem(n+clef,101)`, où `clef` est une variable qui contient la clef de chiffrement.

- pour le codage linéaire on multiplie ces nombres par un nombre fixé appelé clef de chiffrement (par exemple 17) puis on prend le reste de la division par 101 des nombres obtenus. Ainsi avec ce codage \hat{e} est transformé 95 qui est transformé en 100 ($95 \cdot 17 = 1615 = 15 \cdot 101 + 100$, ou encore, 100 est le reste de la division de 1615 par 101).

Avec `xcas`, on effectue la transformation de `n` avec la commande `irem(n*clef, 101)` si `clef` est une variable qui contient la clef de chiffrement.

3. On transforme ensuite cette suite de nombre en une suite de caractères, `c` est le message crypté. Ainsi, avec le codage de Jules César de clef 17, \hat{e} devient $+$ et BONJOUR devient `S'_|'fc` et avec le codage linéaire de clef 17, \hat{e} devient \acute{e} et BONJOUR devient `i|k'|}J`.

La fonction `coden2c` réalise cette étape : elle transforme un nombre n ($0 \leq n < 101$) en un caractère c , selon la table 4.6.1.

4. Le décryptage nécessite d'inverser les opérations 3, 2 et 1. Dans les exemples :
 - avec le codage de Jules César, il faut enlever 17 ou rajouter 84 et prendre le reste de la division par 101, (84 est la clef de déchiffrement car $84 + 17 = 101$). Ainsi, 11 est décrypté par 95 puisque $11 + 84 = 95$.
 - avec le codage linéaire, il faut multiplier par 6 et prendre le reste de la division par 101, (6 est la clef de déchiffrement car $6 \cdot 17 = 102 = 101 + 1$). Ainsi, 100 est décrypté par 95 puisque $100 \cdot 6 = 5 \cdot 101 + 95$.

Le calcul de la clef de déchiffrement à partir de la clef de chiffrement fait intervenir l'arithmétique des entiers :

- savoir trouver l'opposé u d'un élément n de $\mathbb{Z}/101\mathbb{Z}$ pour le codage de Jules César ($u = 101 - n$ car $u + n = 101$).
- savoir utiliser l'identité de Bézout pour trouver l'inverse u d'un élément de n de $\mathbb{Z}/101\mathbb{Z}$ pour le codage linéaire ($u = \text{iegcd}(n, 101)[0]$ car $u \cdot n + v \cdot 101 = 1$).

4.6.3 Le programme `xcas`

```

codec2n(c) := {
if (c=="é") return 100;
if (c=="è") return 99;
if (c=="à") return 98;
if (c=="ç") return 97;
if (c=="ù") return 96;
if (c=="ê") return 95;
return(asc(c)-32);
};

coden2c(k) := {
if (k== 100) return "é";
if (k==99) return "è";
if (k==98) return "à";
if (k==97) return "ç";
if (k==96) return "ù";
if (k==95) return "ê";
return(char(k+32));
};

```



```
};

jules_cesar(message,clef):={
local s,j,messcode;
s:=size(message);
messcode:=" ";
for (j:=0;j<s;j++) {
messcode:=append(messcode,coden2c(irem(clef+codec2n(message[j]),101)));
}
return (messcode);
};

lineaire(message,clef):={
local s,j,messcode;
s:=size(message);
messcode:=" ";
for (j:=0;j<s;j++) {
messcode:=messcode+coden2c(irem(clef*codec2n(message[j]),101));
}
return (messcode);
};
```

codec2n transforme un caractère c "autorisé" en un entier n , $0 \leq n < 101$ selon la table 4.6.1.

coden2c transforme un entier n , $0 \leq n < 101$ en un caractère c selon la table 4.6.1.

jules_cesar (respectivement lineaire) code le message selon la clé choisie.

On remarquera que les deux fonctions ne diffèrent que par l'opération effectuée :

$clef+codec2n(message[j])$ pour la fonction jules_cesar et

$clef*codec2n(message[j])$ pour la fonction lineaire.

Pour décoder, il suffit d'employer le même programme en utilisant la clef de décodage associée à la clef de codage et à la méthode utilisée : par exemple, pour jules_cesar de clef de codage 99 la clé de décodage associée est 2 ($2 + 99 = 101 = 0 \bmod 101$) et

pour lineaire de clef de codage 99 la clef de décodage associée est 50 ($99*50 = -2*50 = -100 = 1 \bmod 101$).

4.6.4 Le programme C++

```
//#include <iostream>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

//using namespace std;

int codec2n(char c){
    int i=c;
    switch (c){
```

```

        case 'é':
            i=100;
            break;
        case 'è':
            i=99;
            break;
        case 'à':
            i=98;
            break;
        case 'ç':
            i=97;
            break;
        case 'ù':
            i=96;
            break;
        case 'ê':
            i=95;
            break;
        default:
            i -= 32;
    }
    return i;
}

char coden2c(int i){
    if (i<95)
        return i+32;
    switch (i){
        case 95:
            return 'ê';
        case 96:
            return 'ù';
        case 97:
            return 'ç';
        case 98:
            return 'à';
        case 99:
            return 'è';
        case 100:
            return 'é';
    }
}

int main(int argc, char ** argv){
    char * s=0, ch;
    size_t n=0;
    int i,d,fois;
    if (!strcmp(argv[0], "./table")){

```

```

    for (i=0;i<101;++i){
        ch=coden2c(i);
        printf("%d:%c ",i,ch);
        if (i%10==0)
printf("\n");
    }
    return 0;
}
if (!strcmp(argv[0], "./coden2c")) {
    for (i=1;i<argc;++i){
        d=atoi(argv[i]);
        ch=coden2c(d);
        printf("%c",ch);
    }
    printf("\n");
    return 0;
}
if (argc==3)
    fois=atoi(argv[2]);
else
    fois=0;
if (argc>=2){
    s=argv[1];
    n=strlen(s);
}
else {
    printf("Entrez un message à numériser\n");
    getline(&s,&n,stdin);
    n=strlen(s)-1;
}
for (i=0;i<n;++i){
    d=codec2n(s[i]);
    if (fois)
        printf("%c",coden2c(d*fois % 101));
    else
        printf("%d ",d);
}
printf("\n");
return 0;
}

```

4.6.5 Exercices de décodage

Avec le codage Jules César

message 1, codage Jules César clef 10

ùÿê}*kvvoé*qkqxo|*êx*MN

message 2, codage Jules César clef 10

v k * } y v ê ~ s y x * x l o } ~ * z k } *) ù s n o x ~ o

message 3, codage Jules César clef 23

(ù (| 7 è | % 7 è ! ~ ù z ù | è % 7 è ù y \$ | %

message 4, codage Jules César clef 23

\$ ù | 7 | 7 % | \$ & 7 { | 7 z ! ' \$ ù \$

message 5, codage Jules César clef 35

é . . # * - , 1 C 3 , C ! & é 2 C 3 , C ! & é 2

message 6, codage Jules César clef 35

* é C 0 B . - , 1 # C 0 # * A 4 # C " 3 C " B \$ '

message 7, codage Jules César clef 99

f _ ' g j c r à q è a _ j a s j _ r m g p c q è m ' j g e _ r m g p c q

message 8, codage Jules César clef 99

g j è d _ g r è ' c _ s è c r è a f _ s b

message 9, codage Jules César clef 51

: Z C B 7 : 7 A / B 7 = < S 2 3 S 1 3 S 1 = 2 / 5 3 S 3 A B S 2 7 A 1 C B / 0 : 3

message 10, codage Jules César clef 51

< = C A S D = C : = < A S R D 7 B 3 @ S : / S 1 = < 4 C A 7 = <

message 11, codage Jules César clef 45

* -) = + 7 = 8 M , - M < :) >) 1 4 M 8 7 = : M : 1 - 6

message 12, codage Jules César clef 45
 ;+1-6+-M;) 6; M+76; +1-6+-M6T-; <M9=-M:=16-M, -M4T) 5-

Avec le codage linéaire

message 1, codage linéaire clef 10
 TsJ6 LUUt| #L#it, Ji OY

message 2, codage linéaire clef 10
 UL 6sUJ@7si ift6@ }L6 {T7jti@t

message 3, codage linéaire clef 23
 [_[h ?h{ ?é1_:_h?{ ?_#dh{

message 4, codage linéaire clef 23
 d_hm mh {hd- Qh :éDd_d

message 5, codage linéaire clef 35
 Uii|BF#m N# 6ùU+ N# 6ùU+

message 6, codage linéaire clef 35
 BU JbiF#m| J|B?q| YN Yb:>

message 7, codage linéaire clef 99
 ZhfXR'B"D dhRd@RhBLXF'D LfRX\hBLXF'D

message 8, codage linéaire clef 99
 XR ^hXB f'h@ 'B dZh@b

message 9, codage linéaire clef 51
 FV}JwFw|sJwzG Bu tu tzBsvu u|J Bw|t}JsAFu

message 10, codage linéaire clef 51

Gz } | Kz } FzG | RKwJuI Fs tzGC } | wzG

message 11, codage linéaire clef 45

Ikçxv4xa >k KVç@çUw a4xV VUkl

message 12, codage linéaire clef 45

èvUklvk èçlè v4lèvUklvk l, kèK)xk VxUlk >k w, ç?k

4.6.6 Solutions des exercices de décodage Jules César et linéaire

- 1 : vous allez gagner un CD
- 2 : la solution n'est pas évidente
- 3 : vive les logiciels libres
- 4 : rien ne sert de courir
- 5 : appelons un chat un chat
- 6 : la réponse relève du défi
- 7 : habiletés calculatoires obligatoires
- 8 : il fait beau et chaud
- 9 : l'utilisation de ce codage est discutable
- 10 : nous voulons éviter la confusion
- 11 : beaucoup de travail pour rien
- 12 : science sans conscience n'est que ruine de l'ame

4.7 Chiffrement affine : premier algorithme

4.7.1 L'algorithme

Pour écrire le message on ne se restreint plus aux 26 lettres de l'alphabet. On suppose que le message à coder n'utilise que les caractères dont le code ASCII va de 32 à 127 (avant 32, les caractères ne sont pas imprimables, et après 127 il s'agit de caractères spéciaux...).

On choisit dans ce premier algorithme de coder chaque lettre : cela à l'inconvénient de décrypter facilement le message en analysant la fréquence des lettres du message crypté, c'est pourquoi dans le deuxième algorithme, on code des groupements de trois lettres.

Il y a alors trois choses à faire qui sont les trois instructions de la fonction `cod1` et qui code un caractère par un autre caractère :

- on transforme chaque caractère en un entier n de 0 à 95 (en enlevant 32 à son code ASCII)

- puis, on applique à cet entier n le chiffrement affine :

$$f(n) = a \times n + b \pmod{96} \text{ avec } f(n) \in [0..95].$$

Pour que cette application soit bijective il faut et il suffit que a soit premier avec 96

(car d'après l'identité de Bézout il existe u et v tels que :

$$a \times u + 96 \times v = 1 \text{ donc } a \times u = 1 \pmod{96}.$$

On a donc $f^{-1}(m) = u.(m - b) \pmod{96}$

- on transforme le nombre trouvé $f(n)$ en un caractère de code $f(n) + 32$.

Pour coder le message, il suffit ensuite de coder chaque caractère, c'est ce que fait la fonction `codm1`.

Pour décoder, il suffit de remplacer la valeur de a par $a1 = u \pmod{96}$ si

$$a \times u + 96 \times v = 1$$

et la valeur de b par $b1 = -a1 \times b \pmod{96}$

car alors on a $n = a1 \times f(n) + b1 \pmod{96}$

Les fonctions de décodage et de codage sont donc les mêmes, seuls les paramètres sont différents !

Exemple

$$a = 85 \quad b = 2$$

On a par l'identité de Bézout :

$$85 \times 61 - 96 \times 54 = 1 \text{ et } -2 \times 61 = -122 = 70 \pmod{96}$$

donc on obtient :

$$a1 = 61 \quad b1 = 70$$

4.7.2 Traduction Algorithmique

On note `char` la fonction qui à un nombre n associe le caractère de code ASCII n et `asc` la fonction qui à un caractère associe son code ASCII.

Voici le codage d'une lettre c par la fonction `cod1` (a, b sont les paramètres du chiffrement affine) :

```
fonction cod1(c, a, b)
local n
asc(c)-32 -> n
a.n+b mod 96 -> n
résultat char(n+32)
ffonction
```

On suppose que l'on a accès au k -ième caractère du mot m en mettant $m[k]$.

On suppose que la concaténation de deux mots se fait avec `concat`.

Voici le codage du message m par la fonction `codal` (a, b sont les paramètres du chiffrement affine) :

```
fonction codm1(m, a, b)
local r, k, n
"" -> r
k->0 longueur_mot(m)->s tantque k<s
m[k]->c
k+1->k
concat(r, cod1(c, a, b)) -> r
ftantque
retourne r
ffonction
```

4.7.3 Traduction xcas

On dispose de :

- `char` la fonction qui à un nombre n associe le caractère de code ASCII n et, qui a une liste de nombres associe la chaîne des caractères dont les codes correspondent aux nombres de la liste.

- `asc` la fonction qui à une chaîne de caractères associe la liste des codes ASCII des caractères composant la chaîne.

Attention `asc("A")=[65]` et donc `(asc("A"))[0]=65`.

Voici le codage d'une lettre par la fonction `cod1` :

```
cod1(c,a,b):={
  local n;
  n:=(asc(c))[0]-32;
  n:=irem(a*n+b,96);
  return(char(n+32));
}
```

Voici le codage du message par la fonction `codm1` :

```
codm1(m,a,b):={
  local r,c,s;
  r:=" ";
  s:=size(m);
  for (k:=0;k<s;k++){
    c:=m[k];
    r:=concat(r,cod1(c,a,b));
  }
  return(r);
}
```

On peut aussi coder directement le message `mess` : en effet avec xcas les fonctions `asc` et `char` gèrent les chaînes et les listes.

On transforme donc le message (i.e une chaîne de caractères) en une liste `l` de nombres avec `asc(mess)`, puis on transforme cette liste de nombres par l'application affine $f(n) = a \times n + b \pmod{96}$ en la liste `mc`, puis on transforme la liste `lc` des nombres ainsi obtenus en une chaîne de caractères (avec `char(lc)`, c'est ce que fait la fonction `codm`).

Dans ce qui suit on a écrit les fonctions :

`bonpara(para)` (avec `para=[a,b]`) renvoie la liste des paramètres de décodage si le paramètre `a` est premier avec 96. Cette fonction utilise la fonction `decopara(para)` qui calcule les paramètres de décodage.

```
decopara(para):={
  //decopara permet de trouver les parametres de decodage
  local bez,a,b;
  a:=para[0];
  b:=para[1];
  bez:=bezout(a,96);
  a:=irem(bez[0],96);
```



```

if (a<0) a:=a+96;
b:=irem(-b*a,96);
if (b<0) b:=b+96;
return([a,b]);
};
bonpara(para):={
//teste si a est premier avec 96
if (pgcd(para[0],96)==1) return(decopara(para)); else return(false);
};
codm(mess,para):={
//codage par chiffrement affine de parametres para=[a,b] mod 96
//codm code le message mess ("....") avec para=[a,b]
local l,lc,sl,a,b,c;
a:=para[0];
b:=para[1];
l:=asc(mess);
sl:=size(l);
for (k:=0;k<sl;k++){
//les caracteres de code compris entre 0 et 31 ne sont pas lisibles
l[k]:=l[k]-32;
}
lc=[];
for (j:=0;j<sl;j++){
c:=irem(a*l[j]+b,96);
lc:=concat(lc,32+c);
}
return(char(lc));
}

```

4.8 Chiffrement affine : deuxième algorithme

4.8.1 L'algorithme

On peut aussi choisir de coder le message en le découpant par paquets de 3 lettres que l'on appelle mot et en rajoutant, éventuellement, des espaces à la fin du message pour que le nombre de caractères du message soit un multiple de 3.

Comme précédemment, à chaque caractère on fait correspondre un nombre entier de l'intervalle $[0..95]$.

On considère alors un mot de 3 lettres comme l'écriture dans la base 96 d'un nombre entier n :

Exemple

le mot BAL est la représentation de $n = 34 \times 96^2 + 33 \times 96 + 44 = 316556$.

En effet B est codé par 34 puisque son code ASCII est 66 ($66-32=34$), A est codé par 33 et L est codé par 44.

On code les mots de 3 lettres par un autre mot, puis on en déduit le codage du message tout entier. Le programme `mot2n` transforme un mot m de en un nombre entier n dont l'écriture en base 96 est m . On a donc, si m a 3 lettres $n < 96^3$.

Par exemple `mot2n("BAL")=316559`.

Le programme `codaff` transforme n selon le chiffrement affine :

$$f(n) = a \times n + b \bmod p.$$

- il faut choisir $p \geq 96^3$; si $p > 96^3$, le nombre ($f(n)$) obtenu après transformation affine de n , peut avoir une représentation de plus de 3 lettres dans la base 96. Mais, au décodage tous les mots auront exactement 3 lettres. Pour les calculateurs qui limitent la représentation d'un entier à 12 chiffres il faut choisir $p \leq 10^6$ pour que $a \times n + b < 10^{12}$

- pour que f soit inversible il faut que a et p soient premiers entre eux (cf p 102) .

Exemple $a = 567$ $b = 2$ $p = 10^6$

On obtient par Bézout :

$$567 \times 664903 + 10^6 \times 377 = 1$$

$$\text{et } -2 \times 664903 = 670164 \bmod 10^6$$

donc $a1 = 664903$ et $b1 = 670164$

Le programme `n2mot` fait l'opération inverse et transforme un nombre entier n en un mot m (d'au moins 3 symboles) qui est la représentation de n dans la base 96.

Il faut faire attention aux espaces en début de mot !!! En effet, l'espace est codé par 0 et il risque de disparaître si on ne fait pas attention, au décodage !!!

Exemple

On a `n2mot(34) = " B"` (c'est à dire la chaîne formée par 2 espaces et B).

Le programme `codmot3` code les mots d'au moins 3 lettres en un mot d'au moins 3 lettres à l'aide du chiffrement affine. En changeant les paramètres a et b `codmot3` décode les mots codés avec `codmot3`.

Le programme `codmess3` code les messages à l'aide du chiffrement affine. Pour décoder, il suffit d'utiliser la fonction `codmess3` en changeant les paramètres a et b .

4.8.2 Traduction Algorithmique

Voici la transformation d'un mot m en un nombre entier n par la fonction

```

mot2n:
fonction mot2n(mo)
local k,p,n
0->n
0->k
tantque k<longueur_mot(mo) ≠ ""
asc(mo[k])-32->p
k+1->k
n*96+p->n
ftantque
retourne n
ffonction

```

Voici la transformation d'un nombre entier n en son écriture en base 96 (c'est à dire en un mot m d'au moins 3 lettres) par la fonction `n2mot` :

```

fonction n2mot(n)
local m,r,i
""->m
0->i

```

```

tantque n > 0 ou i < 3
char((n mod 96)+32)->r
int(n/96)->n
r+m->m
i+1->i
ftantque
retourne m
ffonction
Voici le codage d'un mot d'au moins 3 lettres par la fonction codmot3 :
fonction codmot3(m,a,b,p)
local n
mot2n(m)->n
a.n+b mod p ->n
n2mot(n)->m
retourne m
ffonction
Voici le codage d'un message par la fonction codmess :
fonction codmess3(m,a,b,p)
local n,i,r,d
int(dim(m)/3)+1->n
{}->r
1->i
tantque i<n
debut(m,3)->d
fin(m,4)->m
codmot3(d,a,b,p)->r[i]
i+1->i
ftantque
si dim(m)=2 alors
m + " " ->m
codmot3(m,a,b,p)->r[i]
sinon
si dim(m)=1 alors
m + " " ->m
codmot3(m,a,b,p)->r[i]
sinon
fin(r,i-1)->r
fsi
fsi
retourne r
ffonction

```

4.8.3 Traduction xcas

Voici la fonction `decopara3(para)` qui donne les paramètres de décodage quand les paramètres sont corrects.

On prend comme chiffrement affine $a * n + b \bmod 96^3$ car on veut mettre le message codé dans une chaîne et donc transformer un paquet de 3 lettres en un

paquet d'exactly 3 lettres.

```
decopara3(para):={
//=le parametrage de decodage du parametrage para (liste).
local a,b,l;
a:=para[0];
b:=para[1];
l:=bezout(a,96^3);
if (l[2]!=1) return(false);
a:=l[0];
if (a<0) a:=a+96^3;
b:=-irem(b*a,96^3)+96^3;
return([a,b]);
}
```

Voici la transformation d'un mot m d'au moins 3 lettres en un nombre entier par la fonction `mot2n` :

```
mot2n(s):={
//transforme un mot s de 3 lettres en n d'écriture s en base 96
local l,n;
l:=asc(s);
n:=(l[0]-32)*96^2+(l[1]-32)*96+l[2]-32;
return(n);
}
```

Voici la transformation d'un nombre entier n en son écriture en base 96 (c'est à dire en un mot d'au moins 3 lettres) par la fonction `n2mot` : cette fonction utilise la fonction `ecritu96` qui écrit n dans la base 96 comme un mot de 1,2,3 etc caractères. Pour obtenir un mot d'au moins 3 lettres il suffit de rajouter des espaces devant le mot puisque le code ASCII de l'espace vaut 32, cela revient à rajouter des zéros devant l'écriture de n .

```
ecritu96(n):={
//transforme l'entier n en la chaine s=écriture de n en base 96
local s,r;
//n est un entier et b=96, writu96 est une fonction iterative
//ecritu96(n)=le mot de caracteres l'écriture de n en base 96
s:=" ";
while (n>=96){
r:=irem(n,96);
r:=char(r+32);
s:=r+s;
n:=iquo(n,96);
}
n:=char(n+32);
s:=n+s;
return(s);
};
```

```

n2mot(n):={
local mot,s;
mot:=ecritu96(n);
s:=size(mot);
//on suppose  $n < 96^3$  on transforme n en un mot de 3 caracteres
//on rajoute des espaces si le mot n'a pas 3 lettres
if (s==2) {mot:=" "+mot;}
else {
if (s==1) {mot:="  "+mot;}
}
return(mot);
}

```

Voici le codage d'un mot d'au moins 3 lettres par la fonction `codmot3` : en prenant toujours $p = 96^3$

```

codmot3(mot,para):={
//codage d'un mot de 3 lettres avec le parametrage para=[a,b]
local n,m,a,b;
//para:[569,2] mod  $96^3$ 
//decopara3=[674825, 419822]
a:=para[0];
b:=para[1];
n:=mot2n(mot);
m:=irem(a*n+b,96^3);
return(n2mot(m));
}

```

Le décodage d'un mot codé avec `codmot3` se fait aussi avec la fonction `codmot3`.

Voici le codage d'un message par la fonction `codmess3` :

```

codmess3(mess,para):={
//code le message mess,parametrage para et paquet de 3 lettres
local s,messcod,mess3;
s:=size(mess);
if (irem(s,3)==2) {
mess:=mess+" ";
s:=s+1;
}
else {
if (irem(s,3)==1) {
mess:=mess+"  ";
s:=s+2;
}
}
messcod:="";
for (k:=0;k<s;k:=k+3) {
mess3:=mess[k..k+2];
mess3:=codmot3(mess3,para);
messcod:=messcod+mess3;
}
}

```

```

    }
    return(messcod);
}

```

Le décodage du message se fait aussi par la fonction `codmess3`

4.9 Devoir à la maison

On écrit un message plus ou moins long selon le nombre d'élèves de la classe. Puis on le partage en groupement de 8 lettres.

Chaque groupe de 8 lettres est ensuite codé (lettre par lettre) à l'aide d'un chiffrement affine de paramètres différents selon les élèves.

Le chiffrement affine lettre à lettre est déterminé par la donnée de 3 paramètres : a, b, p qui transforme l'entier n en $m = a * n + b \pmod{p}$.

Pour avoir une fonction de décodage il faut et il suffit que a soit inversible dans $\mathbb{Z}/p\mathbb{Z}$ c'est à dire que a et p soient premiers entre eux.

La fonction de décodage est alors :

$a_1 * m + b_1$ avec,

a_1 inverse de a dans $\mathbb{Z}/p\mathbb{Z}$ ($a_1 = u \pmod{p}$ si $a * u + p * v = 1$ (identité de Bézout)) et $b_1 = -b * a_1 \pmod{p}$.

Pour ce chiffrement affine, on peut utiliser tous les caractères dont les codes ASCII vont de 32 à 127 (les caractères de code 0 à 31 ne sont pas imprimables et au delà de 127 ils ne sont pas standards).

Étant donnés $a, b, p = 96$, comment coder ?

À chaque caractère (de code ASCII compris entre 32 et 127) on fait correspondre un entier n entre 0 et 95 ; n est égal à : (code ASCII du caractère) - 32.

Par exemple, à B on fait correspondre 34 (66-32).

Puis, on calcule $m = a * n + b \pmod{96}$:

pour $a = 55, b = 79$ et $n = 34$ on obtient $m = 29$.

Puis, on cherche le caractère de code ASCII $m+32$. Le caractère qui a comme code ASCII $29+32=61$ est le signe $=$: c'est ce caractère qui sera donc choisi comme codage de la lettre B.

Exemple :

On va coder la phrase :

BEAUCOUP DE TRAVAIL POUR RIEN ! ?

en la coupant en trois morceaux (le caractère espace termine les deux premiers morceaux).

Par exemple :

BEAUCOUP sera codé avec $a = 55, b = 79$ et $p = 96$

DE TRAVAIL sera codé avec $a = 49, b = 25$ et $p = 96$

POUR RIEN ! ? sera codé avec $a = 73, b = 48$ et $p = 96$

On code BEAUCOUP avec $a = 55, b = 79$ et $p = 96$.

On obtient :

`"f2th2?o`

Le devoir du premier élève est donc :

avec les paramètres de codage $a = 55, b = 79$ et $p = 96$, décoder `"f2th2?o`

L'élève doit :

- trouver les paramètres de décodage (ici $a_1 = 7$ $b_1 = 23$) :

en effet l'inverse a_1 de a (mod 96) est obtenu en écrivant l'identité de Bézout pour a et p :

$$55 * 7 - 96 * 4 = 1 \text{ et } b_1 = 79 * 7 \pmod{96} = 23$$

- puis décoder à l'aide d'une table de code ASCII :

le code ASCII du caractère = est 61 donc :

$$m = 61 - 32 = 29$$

$$a_1 * n + b_1 \pmod{96} = 7 * 29 + 23 \pmod{96} = 226 \pmod{96} = 34$$

La lettre de code $34+32 = 66$ est B

4.9.1 Le code Ascii

Voici pour `xcas`, la table des codes ASCII compris entre 30 et 127.

	0	1	2	3	4	5	6	7	8	9
30	^^	^_		!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~	^?		
130										

4.10 Codage RSA

On suppose que l'on dispose d'un annuaire de clés (comme un annuaire de téléphone) qui permet d'envoyer à quelqu'un (par exemple à Tartanpion) un message que l'on code en se servant de la clé de Tartanpion mais seul Tartanpion pourra décoder les messages qu'il reçoit.

Autrement dit tout le monde sait comment il faut coder les messages pour Tartanpion : la fonction f de codage est connue mais la fonction g de décodage n'est connue que de Tartanpion car il existe des fonctions inversibles f dont l'inverse g est difficile à trouver.

4.10.1 Le cryptage des nombres avec la méthode RSA

Tartanpion choisit deux grand nombres premiers p et q et pose $n = pq$, puis il choisit m un nombre premier avec $(p-1)(q-1)$ (par exemple il prend pour m un nombre premier plus grand que $(p-1)/2$ et que $(q-1)/2$).

Il calcule l'entier u pour que $um = 1 \pmod{(p-1)(q-1)}$ (d'après l'identité de Bézout il existe des entiers u et v tel que $um = 1 + v(p-1)(q-1)$). Puis il met dans l'annuaire les nombres u et n (quand n est grand p et q sont difficiles à obtenir à partir de n), le couple (u, n) est la clé publique alors que (m, n) est la clé

secrète qui va servir à décoder le message : bien sûr p et q restent secrets, car sinon n'importe qui peut calculer m en fonction de u avec l'identité de Bézout.

La fonction de codage f est la suivante :

à un entier a inférieur à $n = pq$ f fait correspondre $a^u \bmod n$.

La fonction de décodage g est la suivante :

à un entier b , g on fait correspondre $b^m \bmod n$.

Pour montrer que $g(f(a)) = a$, on utilise le petit théorème de Fermat amélioré :

si p et q sont premiers, si $n = pq$ si a est inférieur à $n = pq$ alors :

$$a^{k(p-1)(q-1)+1} = a \bmod n$$

On peut appliquer ce théorème ici car : - si a est premier avec n , p et q sont premiers, $n = pq$ et donc a est premier avec p et est premier avec q donc :

$$a^{v(p-1)(q-1)} = 1^v = 1 \bmod n \text{ (d'après le petit théorème de Fermat) } a^{p-1} =$$

$$1 \bmod p \text{ et } a^{q-1} = 1 \bmod q \text{ donc}$$

$$a^{v(p-1)(q-1)} = 1^v = 1 \bmod p \text{ et } a^{v(p-1)(q-1)} = 1^v = 1 \bmod q$$

donc

$$a^{v(p-1)(q-1)} = 1^v = 1 \bmod n.$$

- si a n'est pas premier avec n , c'est que a est soit un multiple de p soit un multiple de q (puisque $a < n = pq$). Supposons que a soit un multiple de p , a est donc premier avec q puisque $a < p * q$ et on a :

$$a = 0 \bmod p \text{ donc } a^{k(p-1)(q-1)+1} = a = 0 \bmod p$$

$$a^{q-1} = 1 \bmod q \text{ car } q \text{ est premier et } a \text{ est premier avec } q \text{ (th de Fermat), donc}$$

$$a^{k(p-1)(q-1)+1} = a \bmod q$$

Donc $a^{k(p-1)(q-1)+1} - a$ est un multiple de p et de q donc est un multiple de $n = pq$ (car p et q sont premiers).

on a donc bien :

$$g(f(a)) = g(a^u) \bmod n = a^{um} \bmod n = a^{v(p-1)(q-1)+1} \bmod n = a$$

Un exemple :

$$p = 123456791 \text{ et } q = 1234567891$$

$$: n = p * q = 152415790094497781$$

$$\varphi = (p-1)(q-1) = 152415788736473100$$

$$m = 12345701 \text{ (} m \text{ est un nombre premier et } m \text{ ne divise pas } \varphi \text{)}$$

On cherche u en tapant `inv(m%φ)` on trouve :

$$(-36645934363466299) \% 152415788736473100 \text{ et on a :}$$

$$u = -36645934363466299 + \varphi = 115769854373006801$$

Pour coder, on utilise la clé publique u et n et pour décoder, on utilise la clé secrète m et n .

Remarque

Pour passer d'un message à une suite de nombres, on groupe plusieurs caractères (car sinon on pourrait décrypter le message en utilisant des statistiques de fréquence des caractères en fonction de la langue), le groupement est l'écriture d'un nombre en une base donnée (256 ici correspondant au codage ASCII d'un caractère), par exemple, puisque `asc("BONJOUR")=[66,79,78,74,79,85,82]`, si on groupe par 3, BONJOUR devient les nombres $(66*256+79)*256+78=4345678$, $(74*256+79)*256+85=4869973$, $(82*256+0)*256+0=5373952$ qui seront transformés par f en 156330358492191937 , 126697584810299952 , 50295601528998788 car `powmod(4345678,u,n)=15633035849219193` etc...

Pour décoder on applique à ces nombres la fonction g on a :

$\text{powmod}(15633035849219193, m, n) = 4345678$ etc...

4.10.2 La fonction de codage

Tartanpion choisit 2 grands nombres premiers p et q et met dans l'annuaire le nombre $n = p * q$. Il est facile d'obtenir n à partir de p et q mais par contre p et q sont difficiles à obtenir à partir de n car la décomposition en facteurs premiers de grands nombres est longue et presque impossible si n a plus de 130 chiffres.

Première étape

On découpe le message en tranche ayant $ncara$ caractères. On choisit $ncara$ en fonction des nombres p et q pour que 256^{ncara} soit inférieur à p et à q . En effet, on considère que la tranche du message que l'on veut coder est l'écriture en base 256 d'un nombre : par exemple si $ncara = 5$ "BABAR" est le nombre $a = 66 * 256^4 + 65 * 256^3 + 66 * 256^2 + 65 * 256 + 82 = 284562702674$ et on verra dans la section suivante que ces nombres a doivent être premiers avec $n = p * q$, donc par exemple être inférieurs à p et q (ce qui est vérifié si $256^{ncara} < p$ et $256^{ncara} < q$ puisque $a < 256^{ncara}$).

Deuxième étape : un peu de maths

Le petit théorème de Fermat dit que :

si n est premier et si a est premier avec n alors $a^{n-1} = 1 \mod n$.

Une généralisation (simple) du petit théorème de Fermat est :

si p et q sont premiers, si a est quelconque, si $n = p * q$ et si k est un entier, alors : $a^{k(p-1)(q-1)+1} = a \mod p * q$.

Montrons cette généralisation simple :

- si a est un multiple de n c'est évident puisque

$a = 0 \mod n$ donc $a^{k(p-1)(q-1)+1} = a = 0 \mod n$ - si a est premier avec n alors a est premier avec q (car q est un diviseur de n),

puisque a est premier avec q on a :

$a^{q-1} = 1 \mod q$ (application du petit théorème de Fermat) et

donc $a^{k(q-1)(p-1)} = 1 \mod q$

puisque a est premier avec p on a :

$a^{p-1} = 1 \mod p$ (application du petit théorème de Fermat)

donc $a^{k(p-1)(q-1)} = 1 \mod p$.

On en déduit donc que :

$a^{k(p-1)(q-1)} - 1 = 0 \mod p$ et $a^{k(p-1)(q-1)} - 1 = 0 \mod q$ c'est à dire que :

$a^{k(p-1)(q-1)} - 1$ est un multiple de p et de q donc de $n = p * q$ puisque p et q sont premiers.

donc $a^{k(q-1)(p-1)} = 1 \mod n$ et donc

$a^{k(q-1)(p-1)+1} = a \mod n$

- si a n'est pas premier avec n et si $a < n$, c'est que a est soit un multiple de p soit un multiple de q (puisque $a < n = pq$).

Supposons que a soit un multiple de p :

$a = 0 \mod p$ donc $a^{k(p-1)(q-1)+1} = a = 0 \mod p$

$a^{q-1} = 1 \mod q$ car q est premier et a est premier avec q (th de Fermat), donc

$a^{k(p-1)(q-1)+1} = a \mod q$

Donc $a^{k(p-1)(q-1)+1} - a$ est un multiple de p et de q donc est un multiple de $n = pq$ (car p et q sont premiers).

Donc si $n = p * q$ avec p et q premiers quelque soit a et quelque soit k entier on a :
 $a^{k(p-1)(q-1)+1} = a \pmod n$.

Revenons au codage.

Soit m un nombre premier avec $(p-1) * (q-1)$ (par exemple on peut choisir pour m un nombre premier assez grand).

D'après l'identité de Bézout il existe deux entiers u et v tels que :

$$u * m + v * (p-1) * (q-1) = 1$$

donc :

$$a^{u*m+v*(p-1)*(q-1)} = a^1 \text{ et comme } a^{(p-1)*(q-1)} = 1 \pmod n,$$

$$a^{u*m} = a \pmod n$$

La fonction f de codage sera alors :

$$a \mapsto a^u \pmod n \text{ pour } a < p \text{ et } a < q \text{ (pour avoir } \text{pgcd}(a, n) = 1).$$

La fonction g de décodage sera alors :

$$b \mapsto b^m \pmod n.$$

et on a bien $g(f(a)) = a^{u*m} = a \pmod n$ ou encore $g(f(a)) = a$ car $a < n$

La clé publique se trouvant dans l'annuaire sera (u, n) ,

la clé secrète sera (m, n) , mais bien sûr, p et q devront rester secrets.

Remarque : u et m jouent un rôle symétrique, par exemple u est aussi premier avec $(p-1)(q-1)$ et donc si on connaît u , p et q il sera aisé de retrouver m avec l'identité de Bézout ($u * m + v * (p-1) * (q-1) = 1$).

Troisième étape : le choix des clés

Comment vais-je choisir ma clé publique et ma clé secrète ?

Si on tape :

`p := nextprime(123456789)` (p est un grand nombre premier),

`q := nextprime(1234567890)` (q est un grand nombre premier),

`n := p*q`

`m := nextprime(12345678)` (m est un nombre premier),

`phi := (p-1) * (q-1)`

On vérifie que m est premier avec phi en tapant :

`gcd(m, (p-1) * (q-1))`, on obtient bien 1.

On obtient :

$p = 123456791$, $q = 1234567891$, $m = 12345701$, $phi = 152415788736473100$
 et on a $n = 152415790094497781$ (n a 18 chiffres).

On cherche u et v en tapant : `iegcd(m, phi)` ($u*m+v*phi=1$)
 on obtient : `[-36645934363466299, 2968326, 1]`

On tape `u := -36645934363466299+phi` donc $u = 115769854373006801$.

Donc, ma clé publique qui se trouvera dans l'annuaire sera (u, n) ,

ma clé secrète sera (m, n) p et q devront rester secrets. Avec ce choix de p et de q , on va choisir de découper le message en tranches de 3 caractères ($ncara = 3$) car $256^3 = 16777216 < p < q$ et ainsi tout nombre inférieur à 256^3 sera premier avec

n

Quatrième étape : le codage

Vous voulez m'envoyer le message "BABAR". Dans l'annuaire, vous trouvez en face de mon nom :

$u = 115769854373006801$ et $n = 152415790094497781$

Grâce à la première étape le mot "BABAR" est transformé en la liste de nombres

$l = [4342082, 16722]$ car

$\text{chaine2n}(\text{"BAB"}) = 4342082$ et $\text{chaine2n}(\text{"AR"}) = 16722$.

Vous calculez : $f(a) = a^u \bmod n$ grâce à la commande `powmod(a, u, n)`

Vous obtenez :

$f(4342082) = 4342082^{115769854373006801} = 6243987715571440 \bmod n$ car

$\text{powmod}(4342082, u, n) = 6243987715571440$.

$f(16722) = 16722^{115769854373006801} = 70206283680955159 \bmod n$ car

$\text{powmod}(16722, u, n) = 70206283680955159$.

Le message codé est donc :

$l = [6243987715571440, 70206283680955159]$ et c'est cette liste de nombres

que vous m'envoyez. Remarque : On ne transforme pas cette liste de nombres en

un message de caractères car on risque d'avoir des caractères non imprimables.

Le codage transforme donc le message en une suite de nombres.

Cinquième étape : le décodage

Le décodage transforme une suite de nombres en un message.

Je reçois $l = [6243987715571440, 70206283680955159]$ pour le décoder

je calcule pour chaque élément b de la liste :

$g(b) = b^m \bmod n$ grâce à la commande `powmod(b, m, n)`

$g(6243987715571440) = 6243987715571440^m = 4342082 \bmod n$ car

$\text{powmod}(6243987715571440, m, n) = 4342082$.

$g(70206283680955159) = 70206283680955159^m = 16722 \bmod n$ car

$\text{powmod}(70206283680955159, m, n) = 16722$.

Il suffit maintenant de traduire le nombre $a = 4342082$ en écrivant ce nombre dans la base 256 les symboles pour écrire $0 < k < 256$ étant le caractère de code ASCII k .

Je tape :

`irem(a, 256) = 66`

`a := iquo(a, 256) = 16961`

`puis irem(a, 256) = 65`

`a := iquo(a, 256) = 66`

`irem(a, 256) = 66`

`a := iquo(a, 256) = 0`

on obtient la liste $l = [66, 65, 66]$ qui correspond à "BAB"

puis pour $a = 16722$

Je tape :

`irem(a, 256) = 82`

`a := iquo(a, 256) = 65`

`puis irem(a, 256) = 65`

`a := iquo(a, 256) = 0`
 on obtient la liste `l=[65,82]` qui correspond à "AR" c'est ce que fait la fonction `ecritu256(a)` (cf 4.11), on a :
`ecritu256(4342082) = "BAB"` et `ecritu256(16722) = "AR"`

4.11 Les programmes correspondants au codage et décodage RSA

les programmes qui suivent se trouvent dans le fichier `rsa.xws` du menu Exemples → arit.

La première et la dernière étape

```

chaine2n(m) := {
//chaine2n(m) transforme la chaine m en l'entier n
//m est l'écriture de n dans la base 256
local l, n, s;
s := size(m);
l := asc(m);
n := 0;
for (k:=0; k<s; k++) {
n := n*256 + l[k];
}
return(n);
};

ecritu256(n) := {
//transforme l'entier n en son écriture en base 256
local s, r;
//n est un entier et b=256, escritu256 est une fonction itérative
//ecritu256(n)=le mot de caracteres l'écriture de n en base 256
s := "";
while (n >= 256) {
r := irem(n, 256);
r := char(r);
s := r + s;
n := iquo(n, 256);
}
n := char(n);
s := n + s;
return(s);
};

```

Le codage

En principe les valeurs de p et q sont beaucoup plus grandes et donc $ncara$ le nombre de caractères par tranche peut être choisi plus grand que 3, il suffira alors

4.11. LES PROGRAMMES CORRESPONDANTS AU CODAGE ET DÉCODAGE RSA117

dans le programme qui suit de d'initialiser `ncara` par la valeur de *ncara* que l'on a choisie (ou de rajouter le paramètre *ncara* et remplacer tous les 3 par *ncara*).

```
//mess est une chaine u:=115769854373006801 n:=152415790094497781
codrsa(mess,u,n):={
local s,j,j3,l,mot,ncara;
s:=size(mess);
j:=0;
ncara:=3;
j3:=ncara;
l:=[];
//j est le nombre de paquets de 3 lettres
while (j3<s) {
mot:="";
for (k:=j;k<j3;k++){
mot:=mot+mess[k];
}
//on code le mot
a:=chaine2n(mot);
l:=append(l,powmod(a,u,n));
j:=j3;
j3:=j+ncara;
}
mot:="";
for (k:=j;k<s;k++){
mot:=mot+mess[k];
}
a:=chaine2n(mot);
l:=append(l,powmod(a,u,n));
return(l);
};
```

Le décodage

```
//l=codrsa(mess,u,n) m:=12345701 n:=152415790094497781
decodrsa(l,m,n):={
local mess,s,a,j,b;
s:=size(l);
mess:="";
for (j:=0;j<s;j++){
b:=l[j];
a:=powmod(b,m,n);
mess:=mess+ecritu256(a);
}
return(mess);
};
```

4.11.1 Exercices de décodage RSA avec différents paramètres

message 1, clefs $u=115769854373006801$, $n=15241579009449778$

[19997497666981017, 33496307064035264, 7220821078918523
38104201170888279, 56130089351291689, 10872985337522715
136817903768324205, 81458241359929537]

message 2, clefs $u=115769854373006801$, $n=152415790094497781$

[58349203435709531, 75028631000317890, 1019566127374431
110175082548593487, 118493016617194452, 106538423823370
65709918090014837, 24509343625849117, 77226207180242279
113156355216337045]

message 3, clefs $u=115769854373006801$, $n=152415790094497781$

[33980482988235109, 116825916853291998, 113895924367530
95775057248987857, 24977608335450648, 13496814948248933
76334436210349648, 98925075877640635, 20555288284806828

message 4, clefs $u=115769854373006801$, $n=152415790094497781$

[99352887245994702, 7452725220300033, 99097515262143188
35018957119694836, 76149403346897851, 17052742903948315
34401001263323402, 146933964211893603]

message 5, clefs $u=115769854373006801$, $n=152415790094497781$

4.11. LES PROGRAMMES CORRESPONDANTS AU CODAGE ET DÉCODAGE RSA119

[69885005423074530, 71482680640174902, 76566059181695815,
136817903768324205, 34106973474155998, 33620404767752971,
12729299234654259, 20531594133810598]

message 6, clefs u=115769854373006801, n=152415790094497781

[58349203435709531, 68614189698168613, 95894768844660062,
130069211243087116, 130376871729616040, 74306178317514482,
125801418681172709, 128922533769427856, 138902168479749054]

message 7, clefs u=115769854373006801, n=152415790094497781

[82887698188763362, 147317794362550340, 11063280558757506,
62347560564639831, 66591192455199994, 108687460796034362,
68698456418704171, 113895924367530643, 97840742991040396,
103130061177405851, 21744064573167843, 81810384887704706]

message 8, clefs u=115769854373006801, n=152415790094497781

[42711799087786740, 134878744490172482, 149439358926120238,
25442479184362935, 1828072730594369, 28742122827339904,
77333486723748758]

message 9, clefs u=115769854373006801, n=152415790094497781

[143623866399748045, 7966012486327335, 82446555671577207,
59363718845705744, 116869540684493084, 27219079163512489,
27219079163512489, 115256914037599394, 81123177371824181,
99166826446083588, 90648282883057820, 28314425697650614,
147744966399483701, 24903506954684046]

message 10, clefs $u=115769854373006801$, $n=152415790094497781$

[21958089817862266, 123349109967966870, 519278453155556
95894768844660062, 24509343625849117, 31027419256533256
125503703895953175, 33160330760344892, 6104036142271832
9544287545681754, 61022858667046639]

message 11, clefs $u=115769854373006801$, $n=152415790094497781$

[62981976688200842, 64536302600310087, 6131084051694421
7486629931368896, 17472057692137769, 130815067487053887
53351663594984181, 144381092812007128, 4125125881631510
114751369092267608]

message 12, clefs $u=115769854373006801$, $n=152415790094497781$

[123477331568497546, 46498884798484169, 990975152621431
7837029050945492, 17052742903948315, 106657774868209674
48690166899675267, 79883234756846796, 11849301661719445
10653842382337002, 128514412154198539, 1425792960093434
28886262313123302, 76149403346897851, 12346636557859071
123115348956556877]

4.11.2 Solutions des exercices de décodage

- 1 : vous allez gagner un CD
- 2 : la solution n'est pas évidente
- 3 : vive les logiciels libres
- 4 : rien ne sert de courir
- 5 : appelons un chat un chat
- 6 : la réponse relève du défi
- 7 : habiletés calculatoires obligatoires
- 8 : il fait beau et chaud
- 9 : l'utilisation de ce codage est discutable
- 10 : nous voulons éviter la confusion
- 11 : beaucoup de travail pour rien
- 12 : science sans conscience n'est que ruine de l'âme

4.12 Codage RSA avec signature

Avec des codages à clé publique comme RSA, n'importe qui peut vous envoyer un message codé. La question qui se pose est : comment être sûr de l'identité de l'envoyeur ?

Avec le codage RSA, c'est assez facile car si Tartanpion m'envoie un message codé avec signature, il va coder et signer le message en utilisant **ma** clé publique et **sa** clé secrète.

Voici par exemple, les clés de codage et de décodage de Tartanpion.

```
ptar :=nextprime(223456789)
qtar :=nextprime(823456789)
mtar :=nextprime(32345678)
phitar :=(ptar-1)*(qtar-1)
ntar :=ptar*qtar
```

On obtient :

```
ptar=223456811,
qtar= 823456811,
mtar= 32345689 et
phitar=184007031935376100
ntar=184007032982289721 (ntar a 18 chiffres)
```

et on vérifie que `mtar` et `phitar` en tapant :

```
gcd(mtar, (ptar-1)*(qtar-1)) on obtient bien 1.
```

On cherche `utar` et `vtar` en tapant : `egcd(mtar, phitar)`

on obtient : `[-44971265178398091,7905277,1]`.

On tape `utar :=-44971265178398091+phitar` donc

```
utar=139035766756978009.
```

Donc, la clé publique de Tartanpion (celle qui se trouve dans l'annuaire) sera `(utar, ntar)`,

sa clé secrète sera `(mtar, ntar)` mais `ptar` et `qtar` devront rester secrets.

Tartanpion va coder le message `mess` qu'il veut m'envoyer selon un programme analogue à `codrsa(mess,u,n)` mais avant de mettre les nombres `b` dans la liste `l` il va utiliser sa fonction de décodage selon sa clé secrète et mettera dans `l` les nombres `powmod(b,mtar,ntar)`.

Il m'envoie donc `codrsas(mess,u,n,mtar,ntar)` (voir le programme ci-dessous).

Voici le détail du programme de codage avec signature `codrsas` (les programmes qui suivent se trouvent dans le fichier `rsas`) :

```
//mess=chaine
//u:=115769854373006801 n:=152415790094497781 (ma cle publique)
//ntar:=184007032982289721 et mtar:=32345689 (cle secrete de Tar)
codrsas(mess,u,n,mtar,ntar):={
local s,j,j3,l,mot,a,b,ncara;
s:=size(mess);
j:=0;
ncara:=3
j3:=ncara;
l:=[];
//j est l'indice du premier \el\ement d'un paquet de 3 lettres
```

```

while (j3<=s) {
mot:="";
for (k:=j;k<j3;k++){
mot:=mot+mess[k];
}
//on code le mot
a:=chaine2n(mot);
b:=powmod(a,u,n);
//fct de codage selon la cle publique (u,n) du receveur puis
//fct de decodage selon la cle secrete de l'envoyeur (mtar,ntar)
l:=append(l,powmod(b,mtar,ntar));
j:=j3;
j3:=j+ncara;
}
//on code la derniere tranche du message
mot:="";
for (k:=j;k<s;k++){
mot:=mot+mess[k];
}
a:=chaine2n(mot);
b:=powmod(a,u,n);
l:=append(l,powmod(b,mtar,ntar));
return(l);
};

```

Pour décoder il me suffira de coder les nombres b de la liste l en utilisant la clé publique de celui qui a signé le message ($a := \text{powmod}(b, \text{utar}, \text{ntar})$) puis, de décoder a en utilisant ma clé secrète ($b := \text{powmod}(a, u, n)$).

```

//l=codrsas(mess,u,n,mtar,ntar)
// m:=12345701 n:=152415790094497781 ma cle secrete (receveur)
//ntar:=184007032982289721 utar:=139035766756978009 cle pub de T
decodrsas(l,m,n,utar,ntar):={
local mess,s,a,j,b;
s:=size(l);
mess:="";
for (j:=0;j<s;j++){
b:=l[j];
//codage selon la cle publique (utar,ntar) de l'envoyeur (T)
a:=powmod(b,utar,ntar);
//decodage selon la cle secrete du receveur (m,n) (moi)
b:=powmod(a,m,n);
mess:=mess+ecritu256(b);
}
return(mess);
};

```

Je reçois un message l signé de Tartanpion : je le décote en utilisant sa clé publique et ma clé secrète en tapant :

```

decodrsas(l,m,n,utar,ntar)
Voici le détail avec mess := "demain 10 heures gare de Grenoble".
l := codrsas(mess,u,n,mtar,ntar)
l := [137370234628529043,113626149789068692,125222577739438308,
33473651820936779,42708525589347295,23751805405519257,
66289870504591745]
decodrsas(l,m,n,utar,ntar) = "demain 10 heures gare de Grenoble"

```

4.12.1 Quelques précautions

Lorsqu'on envoie un message, il ne faut utiliser que les caractères dont les codes ASCII vont de 32 à 127. Il faut par exemple se méfier des caractères accentués qui n'ont pas toujours le même code... Pour un codage avec signature on a des problèmes si on tape :

```

messcs := decodrsa(codrsa(mess,u,n),mtar,ntar) car alors dans
messcs il figure très certainement des caractères qui ont des codes inférieurs à 32
ou des codes supérieurs à 127.

```

Pour mémoire :

```

decodrsa(codrsa(mess,u,n),m,n) = mess car mess n'a que des caractères
qui ont des codes compris entre 32 et 127.

```


Chapitre 5

Algorithmes sur les suites et les séries

L'objectif ici est de traduire les algorithmes en l'écriture de programmes. On écrit ici des programmes permettant d'avoir les termes d'une suite ou d'une série et de trouver des valeurs approchées de leur limites.

Mais, pour étudier les suites et les séries, on peut aussi utiliser le tableur ce qui est souvent plus facile que d'écrire un programme.

5.1 Les suites

Soit u_n une suite de réels définie soit par $u_n = f(n)$, soit par une relation de récurrence $u_n = f(u_{n-m}, \dots, u_{n-1})$ et la donnée de ses premiers termes. On veut ici, calculer les valeurs de u_n . Pour les fonctions qui suivent, il suffira de rajouter la fonction `evalf` dans le `return` pour avoir une valeur approchée de u_n : par exemple `return evalf(uk)`.

5.1.1 Les suites $u_n = f(n)$

Pour avoir le n -ième terme u_n il suffit :
de définir la fonction f et de taper `f(n)`.
On peut aussi mettre f comme paramètre et taper :

`u(f, n) := f(n)`

Ainsi `u(sq, 3)` vaut 9 et `u(sqrt, 3)` vaut `sqrt(3)`.

On remarquera qu'il est souvent préférable de simplifier l'écriture de `u(f, n)` avec la commande `normal` : mettre plutôt `normal` dans la définition de f .

Par exemple on définit : `f(x) := normal(x/sqrt(3)+sqrt(3))`.

On tape :

`u(f, 3)`

On obtient :

`2*sqrt(3)`

On peut aussi considérer qu'il n'y a qu'un paramètre l qui est la séquence f, n et définir `u` par :

`u(l) := l[0](l[1])`

Pour avoir la suite des termes u_k , pour k allant de k_0 à n , on écrit :

```
utermes(f, k0, n) := {
  local k, lres;
  lres := NULL;
  for (k := k0; k <= n; k++) {
    lres := lres, f(k);
  }
  return lres;
}
```

On a choisit de mettre tous les termes cherchés dans une séquence.

On a : `lres := NULL` ; initialise la séquence à vide.

Par exemple, avec la fonction :

`f(x) := normal(x/sqrt(3)+sqrt(3))`.

On tape :

`utermes(f, 0, 5)`

On obtient :

`sqrt(3), 4*sqrt(3)/3, 5*sqrt(3)/3, 2*sqrt(3), 7*sqrt(3)/3, 8*sqrt(3)/3`

5.1.2 Les suites récurrentes

Commençons par un exemple : la suite de Fibonacci définie par :

$$u_0 = a$$

$$u_1 = b$$

$$u_n = u_{n-1} + u_{n-2} \text{ pour } n \geq 2$$

On écrit pour avoir u_n :

```
fibon(n, a, b) := {
  local k, uk;
  for (k := 2; k <= n; k++) {
    uk := a + b;
    a := b;
    b := uk;
  }
  return uk;
}
```

On écrit pour avoir $u_0, u_1 \dots u_n$:

```
fibona(n, a, b) := {
  local k, uk, res;
  res := a, b;
  for (k := 2; k <= n; k++) {
    uk := a + b;
    a := b;
    b := uk;
    res := res, uk;
  }
}
```

```
return res;
}
```

On écrit pour avoir $u_d, u_1 \dots u_n$ pour $d \leq 0$:

```
fibonac(c,n,a,b):={
local k,uk,res;
for (k:=2;k<c;k++) {
    uk:=a+b;
    a:=b;
    b:=uk
};
if c>1 res:=NULL else
    if c==0 {res:=a,b;c:=2;} else
        if c==1 {res:=b;c:=2};
for (k:=c;k<=n;k++) {
    uk:=a+b;
    a:=b;
    b:=uk
    res:=res,uk;
}
return res;
}
```

On suppose maintenant que la suite est définie par une relation de récurrence définie par une fonction de m variables f : pour définir la suite on se donne les m premiers termes :

u_0, u_1, \dots, u_{m-1} et la relation :

$u_n = f(u_{n-m}, u_{n-m+1}, \dots, u_{n-1})$ pour $n \geq m$.

On veut calculer u_n , et on suppose que les valeurs de u_0, u_1, \dots, u_{m-1} sont dans la liste `l0`.

On écrit :

```
urec(f,n,l0):={
local s,k,uk;
s:=size(l0);
l0:=op(l0);
for (k:=s;k<=n;k++) {
    uk:=f(l0);
    l0:=tail(l0),uk;
}
return uk;
}
```

On utilise `op` au début, pour transformer la liste `l0` en une séquence et `tail(l0)` pour enlever le premier élément et ainsi `l0 := tail(l0)`, `uk` est une séquence qui a toujours s éléments.

On peut aussi considérer que le paramètre l contient toutes les variables à savoir $l = f, n, u_0, \dots, u_{m-1}$. On écrit mais c'est inutilement compliqué (!) :

```

urecs(l) := {
  local f, n, s, k, uk;
  f := l[0];
  n := l[1];
  l := tail(tail(l));
  s := size(l);
  //f est une fonction de s variables
  for (k:=s; k<=n; k++) {
    uk := f(l);
    l := tail(l), uk;
  }
  return uk;
}

```

Pour avoir tous les termes u_k de la suite pour k allant de 0 à n , On considère que le paramètre l contient toutes les variables à savoir $l = f, n, u_0, \dots, u_{m-1}$. On écrit :

```

urec_termes(l) := {
  local f, n, s, k, uk, lres;
  f := l[0];
  n := l[2];
  l := tail(tail(tail(l)));
  s := size(l);
  //f est une fonction de s variables
  lres := l;
  for (k:=s; k<=n; k++) {
    uk := f(l);
    lres := lres, uk;
    l := tail(l), uk;
  }
  return lres;
}

```

Par exemple on définit :

$f(x, y) := \text{normal}(x+y)$

On tape :

`urec_termes(f, 5, 1, 1)`

On obtient la suite de Fibonacci :

1, 1, 2, 3, 5, 8

On tape :

`urec_termes(f, 5, 1, (sqrt(5)+1)/2)`

On obtient :

1, (sqrt(5)+1)/2, (sqrt(5)+3)/2, sqrt(5)+2, (3*sqrt(5)+7)/2, (5*sqrt(5)+11)/2

On tape, pour vérifier que l'on a obtenu la suite géométrique de raison $(\sqrt{5}+1)/2$:

`seq(normal(((sqrt(5)+1)/2)^k), k=0..5)`

On obtient :

1, (sqrt(5)+1)/2, (sqrt(5)+3)/2, sqrt(5)+2, (3*sqrt(5)+7)/2, (5*sqrt(5)+11)/2

Pour avoir tous les termes u_k de la suite pour k allant de k_0 à n , On considère que le paramètre l contient toutes les variables à savoir $l = f, k_0, n, u_0, \dots, u_{m-1}$.
On écrit :

```
urec_termekn(l):={
local f,n,s,k,uk,k0,lres;
f:=l[0];
k0:=l[1];
n:=l[2];
l:=tail(tail(tail(l)));
s:=size(l);
//f est une fonction de s variables
for (k:=s;k<k0;k++) {
    uk:=f(l);
    l:=tail(l),uk;
};
if k0>1 res:=NULL else
    if k0==0 {res:=a,b;k0:=2;} else
        if k0==1 {res:=b;k0:=2};
for (k:=k0;k<=n;k++) {
    uk:=f(l);
    lres:=lres,uk;
    l:=tail(l),uk;
}
return lres;
}
```

Par exemple on définit :

$f(x,y) := \text{normal}(x+y)$

On tape :

`urec_termekn(f,5,10,1,1)`

On obtient la suite de Fibonacci :

8, 13, 21, 34, 55, 89

On tape :

`urec_termes(f,5,9,1,(sqrt(5)+1)/2)`

On obtient :

$5*\sqrt{5}+11)/2, 4*\sqrt{5}+9, (13*\sqrt{5}+29)/2, (21*\sqrt{5}+47)/2, 17*\sqrt{5}+38$

5.2 Les séries

Soit u_n une suite de réels telle que la série $\sum_{k=0}^{\infty} u_k$ converge vers S . On veut ici, calculer une valeur approchée de cette somme. Si la série converge rapidement, il suffit de calculer $\sum_{k=0}^n u_k$ pour n assez grand, sinon il faut procéder à une accélération de convergence, en construisant une série de même somme et convergeant plus rapidement.

5.2.1 Les sommes partielles

On écrit :

```
sum_serie(f,n0,n):={
  local s,k;
  //un=f(n) ou f est une fonction de 1 variable
  s:=0;
  for (k:=n0;k<=n;k++) {
    s:=s+evalf(f(k));
  }
  return s;
}
```

Il est plus précis de faire le calcul de la somme en commençant par les plus petits termes, on écrit :

```
serie_sum(f,n0,n):={
  local s,k;
  //un=f(n) ou f est une fonction de 1 variable
  s:=0;
  for (k:=n;k>=n0;k--) {
    s:=s+evalf(f(k));
  }
  return s;
}
```

On peut avoir aussi besoin de la suite des sommes partielles : par exemple pour les séries alternées deux sommes partielles successives encadrent la somme de la série.

On écrit en utilisant un paramètre supplémentaire `alt` pour repérer les séries alternées de la forme $u_n = \text{alt}^n * f(n)$:

```
sums_serie(f,n0,n,alt):={
  local ls,s,k;
  //un=(alt)^n*f(n) ou f est une fonction de 1 variable
  s:=0;
  ls:=[];
  if (alt<0){
    if (irem(n0,2)==0) {alt:=-alt;}
    for (k:=n0;k<=n;k++) {
      s:=s+evalf(alt*f(k));
      alt:=-alt;
      ls:=concat(ls,s);
    }
  }
  else {
    for (k:=n0;k<=n;k++) {
      s:=s+evalf(alt*f(k));
      ls:=concat(ls,s);
    }
  }
}
```

```

    }
}
return ls;
}

```

5.2.2 Exemple d'accélération de convergence des séries à termes positifs

On suppose que $u_n = f(n)$ et que $f(n)$ admet un développement limité à tous les ordres par rapport à $1/n$.

On suppose que $u_k \sim a/k^p$ et on pose :

$$v_k = u_k - \frac{a}{(k+1)(k+2)\dots(k+p)}$$

On a alors, $v_k = O(\frac{1}{k^{p+1}})$ et on connaît :

$$\sum_{k=0}^{\infty} \frac{a}{(k+1)(k+2)\dots(k+p)}$$

En effet :

$$\frac{a}{(k+1)(k+2)\dots(k+p)} = \frac{a}{p-1} \left(\frac{1}{(k+1)(k+2)\dots(k+p-1)} - \frac{a}{(k+2)(k+3)\dots(k+p)} \right)$$

donc

$$\sum_{k=0}^{\infty} \frac{a}{(k+1)(k+2)\dots(k+p)} = \frac{a}{p-1} \left(\frac{1}{1 \cdot 2 \cdot \dots \cdot (p-1)} \right) = \frac{a}{(p-1)(p-1)!} \text{ et,}$$

$$\sum_{k=k_0}^{\infty} \frac{a}{(k+1)(k+2)\dots(k+p)} = \frac{a}{(p-1)(k_0+1)(k_0+2)\dots(k_0+p-1)}$$

On a :

$$\sum_{k=0}^{\infty} u_k = \frac{a}{(p-1)(p-1)!} + \sum_{k=0}^{\infty} v_k$$

On peut ensuite continuer à appliquer la même méthode à v_k .

Exercice

Utiliser cette méthode pour calculer numériquement : $\sum_{k=0}^{\infty} \frac{1}{(k+1)^2}$.

On va faire "à la main " trois accélérations successives.

On pose :

$$u_k = \frac{1}{(k+1)^2}$$

- 1-ière accélération :

$$v_k = u_k - \frac{1}{(k+1)(k+2)}, \text{ et donc}$$

$$v_k = \frac{1}{(k+1)^2(k+2)}$$

$$\sum_{k=0}^{\infty} u_k = 1 + \sum_{k=0}^{\infty} v_k$$

- 2-ième accélération :

$$w_k = v_k - \frac{1}{2(k+1)(k+2)(k+3)}, \text{ et donc}$$

$$w_k = \frac{1}{(k+1)^2(k+2)(k+3)}$$

$$\sum_{k=0}^{\infty} u_k = 1 + \frac{1}{2 \cdot 2!} + \sum_{k=0}^{\infty} w_k$$

– 3-ième accélération :

$$t_k = w_k - \frac{2}{(k+1)(k+2)(k+3)(k+4)}, \text{ et donc}$$

$$t_k = \frac{2}{(k+1)^2(k+2)(k+3)(k+4)}$$

$$\sum_{k=0}^{\infty} u_k = 1 + \frac{1}{2 \cdot 2!} + \frac{2}{3 \cdot 3!} + \sum_{k=0}^{\infty} t_k$$

On tape :

`u(k) := 1 / (k+1) ^ 2`

On tape :

`v(k) := 1 / ((k+1) ^ 2 * (k+2))`

On tape :

`w(k) := 2 / ((k+1) ^ 2 * (k+2) * (k+3))`

On tape :

`t(k) := 6 / ((k+1) ^ 2 * (k+2) * (k+3) * (k+4))`

On compare $\frac{\pi^2}{6}$ et les valeurs obtenues pour $n = 200$, car on sait que :

$$S = \sum_{k=0}^{\infty} \frac{1}{(k+1)^2} = \frac{\pi^2}{6} \simeq 1.64493406685$$

On tape :

`serie_sum(u, 0, 200)`

On obtient S à $5 * 10^{-3}$ près (1 décimale exacte) :

1.63997129788

On tape :

`1+serie_sum(v, 0, 200)`

On obtient S à $1.25 * 10^{-5}$ près (4 décimales exactes) :

1.64492179293

On tape :

`1+1/4+serie_sum(w, 0, 200)`

On obtient S à $8.3 * 10^{-8}$ près (5 décimales exactes) :

1.64493398626

On tape :

`1+1/4+1/9+serie_sum(t, 0, 200)`

On obtient S à $9.2 * 10^{-10}$ près (8 décimales exactes) :

1.64493406596

Les erreurs

Si on compare la somme $\sum_{k=n+1}^{\infty} 1/(k+1)^2$ à une intégrale on a :

$$\sum_{k=n+1}^{\infty} \frac{1}{(k+1)^2} < \int_n^{\infty} \frac{1}{(x+1)^2} dx = \frac{1}{n+1}$$

Ou encore, on peut aussi remarquer que :

$$\sum_{k=n+1}^{\infty} \frac{1}{(k+1)^2} < \sum_{k=n+1}^{\infty} \frac{1}{k(k+1)} = \frac{1}{n+1}$$

puisque $\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$.

Au bout de la p-ième accélération on calcule la somme de :

$$u_k^{(p)} = \frac{p!}{(k+1)^2(k+2)\dots(k+p+1)} \text{ et on a :}$$

$$\sum_{k=n+1}^{\infty} \frac{1}{(k+1)^2(k+2)\dots(k+p+1)} < \sum_{k=n+1}^{\infty} \frac{1}{k(k+1)(k+2)\dots(k+p+1)}$$

Et puisque :

$$\frac{p+1}{k(k+1)(k+2)\dots(k+p+1)} = \frac{1}{k(k+1)(k+2)\dots(k+p)} - \frac{1}{(k+1)(k+2)\dots(k+p+1)}$$

On a :

$$\sum_{k=n+1}^{\infty} u_k^{(p)} = \sum_{k=n+1}^{\infty} \frac{p!}{(k+1)^2(k+2)\dots(k+p+1)} < \frac{p!}{(p+1)(n+1)(n+2)\dots(n+p+1)}$$

Donc

$$\sum_{k=n+1}^{\infty} u_k^{(p)} < \frac{p!}{(p+1)(n+1)^{p+1}}$$

On vérifie ($\frac{\pi^2}{6} \simeq 1.64493406685$) :

$$1.63997129788 < \frac{\pi^2}{6} < 1.63997129788 + 1/201 = 1.64494642226$$

$$1.64492179293 < \frac{\pi^2}{6} < 1.64492179293 + 1/(2 * 201^2) = 1.64493416886$$

$$1.64493398626 < \frac{\pi^2}{6} < 1.64493398626 + 2/(3 * 201^3) = 1.64493406836$$

$$1.64493406596 < \frac{\pi^2}{6} < 1.64493406596 + 6/(4 * 201^4) = 1.64493406688$$

Le programme

On peut écrire un programme qui va demander le nombre d'accélération pour

calculer $\sum_{k=0}^{\infty} 1/(k+1)^2$

```
serie_sumacc(n,acc):={
local p,l,j,k,ls,sf,sg,gk,fact;
ls:=[];
//calcul sans acceleration
sf:=0.0;
for (k:=n;k>=0;k--) {
    sf:=sf+1/(k+1)^2;
}
ls:=[sf];
```

```

sf:=0.0;
fact:=1;
for (p:=1;p<=acc;p++){
  //calcul de 1+1/4+...+1/p^2, le terme a rajouter
  sf:=sf+evalf(1/p^2);
  //calcul de p!
  fact:=fact*(p);
  //calcul de sg, somme(de 0 a n) de la serie acceleree p fois
  sg:=0.0;
  for (k:=0;k<=n;k++){
    gk:=1/(k+1)^2;
    //calcul du k-ieme terme gk de la serie acceleree p fois (sans p!)
    for (j:=1;j<=p;j++){
      gk:=evalf(gk/(k+j+1));
    }
    sg:=sg+gk;
  }
  ls:=concat(ls,sf+fact*sg);
}
return(ls);
}

```

5.3 Méthodes d'accélération de convergence des séries alternées

5.3.1 Un exemple d'accélération de convergence des séries alternées

Un premier exemple

On suppose que $u_k = (-1)^k f(k)$ avec $f(k)$ tend vers zéro quand k tend vers $+\infty$ et f décroissante de \mathbb{R}^+ dans \mathbb{R}^+ .

On pose :

$g(x) = \frac{1}{2}(f(x) - f(x+1))$ donc

$$v_k = (-1)^k \frac{f(k) - f(k+1)}{2} = \frac{u_k + u_{k+1}}{2} = (-1)^k g(k)$$

On a :

$$\sum_{k=0}^n v_k = \frac{1}{2} \left(\sum_{k=0}^n u_k + \sum_{k=0}^n u_{k+1} \right)$$

donc,

$$\sum_{k=0}^n v_k = \frac{1}{2} \left(\sum_{k=0}^n u_k + \sum_{k=1}^{n+1} u_k \right)$$

donc,

$$\sum_{k=0}^n v_k = \frac{u_0}{2} + \frac{u_{n+1}}{2} + \sum_{k=0}^n u_k$$

Puisque $f(k)$ tend vers zéro quand k tend vers $+\infty$, $g(k) = \frac{1}{2}(f(k) - f(k+1))$ tend aussi vers zéro quand k tend vers $+\infty$.

Si la fonction f est convexe ($f''(x) > 0$), la série $\sum_{k=0}^{\infty} v_k$ vérifie aussi le théorème des séries alternées.

En effet, pour $x > 0$ on a :

$$g(x) = \frac{1}{2}(f(x) - f(x+1)) \geq 0 \text{ puisque } f \text{ décroissante sur } \mathbb{R}^+$$

$$g'(x) = \frac{1}{2}(f'(x) - f'(x+1)) < 0 \text{ puisque } f''(x) > 0, f' \text{ est négative et croissante sur } \mathbb{R}^+$$

donc g est décroissante de \mathbb{R}^+ dans \mathbb{R}^+ et $g(k)$ tend vers zéro quand k tend vers $+\infty$.

Conclusion : La série $\sum_{k=0}^{\infty} v_k$ est une série alternée de somme $S + \frac{u_0}{2}$.

Si de plus, $f'(x)/f(x)$ tend vers zéro quand x tend vers l'infini, la série $\sum_{k=0}^{\infty} v_k$ converge plus rapidement que $\sum_{k=0}^{\infty} u_k$, puisque il existe $c, x < c < x+1$ d'après le th des accroissements finis tel que :

$$0 < g(x) = \frac{1}{2}(f(x) - f(x+1)) = \frac{-1}{2}f'(c)$$

on a donc, puisque f' est négative et croissante :

$$0 < g(x) < \frac{-1}{2}f'(x) = o(f(x)).$$

Un exercice

Utiliser cette méthode pour calculer numériquement : $\sum_{k=0}^{\infty} \frac{(-1)^k}{k+1}$.

Toutes les dérivées de $f(x) = 1/(x+1)$ ont un signe constant sur $[0; +\infty[$ et tendent vers zéro à l'infini, ces dérivées sont donc monotones et on peut donc faire plusieurs accélérations successives.

On va faire "à la main " trois accélérations successives.

On pose :

$$u_k = \frac{(-1)^k}{(k+1)}$$

– 1-ière accélération :

$$v_k = (-1)^k \left(\frac{1}{2(k+1)} - \frac{1}{2(k+2)} \right), \text{ et donc}$$

$$v_k = (-1)^k \left(\frac{1}{2(k+1)(k+2)} \right)$$

$$\sum_{k=0}^{\infty} u_k = \frac{1}{2} + \sum_{k=0}^{\infty} v_k$$

– 2-ième accélération :

$$w_k = (-1)^k \left(\frac{1}{4(k+1)(k+2)} - \frac{1}{4(k+2)(k+3)} \right), \text{ et donc}$$

$$w_k = (-1)^k \left(\frac{1}{2(k+1)(k+2)(k+3)} \right)$$

et comme $\frac{v_0}{2} = \frac{1}{8}$ on a :

$$\sum_{k=0}^{\infty} u_k = \frac{1}{2} + \frac{1}{8} + \sum_{k=0}^{\infty} w_k$$

– 3-ième accélération :

$$t_k = (-1)^k \left(\frac{1}{4(k+1)(k+2)(k+3)} - \frac{1}{4(k+2)(k+3)(k+4)} \right), \text{ et donc}$$

$$t_k = (-1)^k \left(\frac{1}{4(k+1)(k+2)(k+3)(k+4)} \right)$$

et comme $\frac{w_0}{2} = \frac{1}{24}$ on a :

$$\sum_{k=0}^{\infty} u_k = \frac{1}{2} + \frac{1}{8} + \frac{1}{24} + \sum_{k=0}^{\infty} t_k$$

On tape :

$$u(k) := (-1)^k / (k+1)$$

On tape :

$$v(k) := (-1)^k / (2 * (k+1) * (k+2))$$

On tape :

$$w(k) := (-1)^k / (2 * (k+1) * (k+2) * (k+3))$$

On tape :

$$t(k) := (-1)^k * 3 / (4 * (k+1) * (k+2) * (k+3) * (k+4))$$

On compare $\ln(2)$ et les valeurs obtenues pour $n = 200$, car on sait que :

$$S = \sum_{k=0}^{\infty} (-1)^k \frac{1}{(k+1)} = \ln(2) \simeq 0.69314718056$$

On tape :

$$\text{serie_sum}(u, 0, 200)$$

On obtient S à $5 * 10^{-3}$ près (2 décimales exactes) :

$$0.69562855486$$

On tape :

$$1/2 + \text{serie_sum}(v, 0, 200)$$

On obtient S à $1.23 * 10^{-5}$ près (4 décimales exactes) :

$$0.693153307335$$

On tape :

$$1/2 + 1/8 + \text{serie_sum}(w, 0, 200)$$

On obtient S à $6.1 * 10^{-8}$ près (8 décimales exactes) :

$$0.693147210666$$

On tape :

$$1/2 + 1/8 + 1/24 + \text{serie_sum}(t, 0, 200)$$

On obtient S à $4.6 * 10^{-10}$ près (10 décimales exactes) :

$$0.693147180781$$

Les erreurs

Le reste d'une série alternée est du signe de son premier terme et la valeur absolue du reste est inférieure à la valeur absolue de son premier terme :

$$\left| \sum_{k=n+1}^{\infty} (-1)^k \frac{1}{(k+1)} \right| < \frac{1}{(n+2)}$$

Au bout de la p -ième accélération on calcule la somme de :

$$u_k^{(p)} = (-1)^k \frac{p!}{2^p (k+1)(k+2)\dots(k+p+1)} \text{ et on a :}$$

$$\left| \sum_{k=n+1}^{\infty} \frac{(-1)^k p!}{2^p (k+1)\dots(k+p+1)} \right| < \frac{p!}{2^p (n+2)\dots(n+p+2)} < \frac{p!}{2^p (n+2)^{p+1}}$$

On vérifie ($\ln(2) \simeq 0.69314718055995$) :

$$0.69562855486 < \ln(2) < 0.69562855486 + 1/202 = 0.70057904991$$

$$0.693153307335 < \ln(2) < 0.693153307335 + 1/(2 * 202^2) = 0.693165561036$$

$$0.693147210666 < \ln(2) < 0.693147210666 + 2/(4 * 202^3) = 0.693147271328$$

$$0.693147180781 < \ln(2) < 0.693147180781 + 6/(8 * 202^4) = 0.693147181231.$$

Le programme

On peut écrire un programme qui va demander le nombre p d'accéléérations.

Si $u_k^{(p)}$ désigne le k -ième terme de la série accélérée p fois, on a :

$$\sum_{k=0}^{\infty} (-1)^k / (k+1) = \sum_{k=0}^p \frac{u_0^{(k-1)}}{2} + \sum_{k=0}^{\infty} u_k^{(p)}$$

$$u_k^{(p)} = \frac{(-1)^k p!}{2^p (k+1) \dots (k+p+1)}$$

On choisit de multiplier seulement à la fin par $\frac{p!}{2^p}$ et de ne calculer que la somme des n premiers termes :

$$\sum_{k=0}^n \frac{(-1)^k}{(k+1) \dots (k+p+1)}$$

On met cette somme dans la variable `sg`, pour cela on calcule $\frac{(-1)^k}{(k+1) \dots (k+p+1)}$ que l'on met dans la variable `gk` :

au début `sg=0` et `gk=` $\frac{1}{(p+1)!}$ (c'est la valeur pour $k=0$)

puis, on ajoute `gk` à la somme `sg`, ensuite on calcule $\frac{(-1)^1 1!}{(p+2)!}$ que l'on met dans `gk` (c'est la valeur pour $k=1$) etc...

La variable `sf` sert au début à calculer $\sum_{k=0}^n (-1)^k / (k+1)$ puis,

`sf` sert à calculer la somme à rajouter $\sum_{k=0}^p \frac{u_0^{(k-1)}}{2}$ (qui vaut $1/2 + 1/8 + 1/24$ pour $p=3$ accélérations).

Dans le programme, on utilise la variable `fact` pour calculer $(p+1)!$ et la variable `fact2` pour calculer $p!/2^p$.

On écrit :

```
seriealt_sumacc(n,acc):={
local l,j,k,ls,sf,sg,gk,fact,fact2,alt,t0,p;
//calcul sans acceleration
sf:=0.0;
alt:=1;
for (k:=n;k>=0;k--) {
sf:=sf+alt/(k+1);
alt:=-alt;
}
if (alt==1) {
ls:=[-sf];}
else {
ls:=[sf];}
}
t0:=0.5;
// sf maintenant est la somme a rajouter
```

```

sf:=0.0;
fact:=1;fact2:=1;
for (p:=1;p<=acc;p++){
    sf:=sf+fact2*t0;
    //calcul de p+1! et de p!/2^p
    fact:=fact*(p+1);
    fact2:=fact2*p/2;
//sg, somme(de k=0 a n) de la serie gk acceleree p fois
    sg:=0.0;
//terme d'indice 0 (ds gk) de la serie acceleree p fois
//(sans p!/2^p=fact2)
    gk:=1/fact;
//on conserve gk/2 dans t0 car il faut rajouter t0
//au prochain sf
    t0:=gk/2;
    sg:=sg+gk;
    alt:=-1;
    for (k:=1;k<=n;k++){
        gk:=1/(k+1);
//terme d'indice k (ds gk) de la serie acceleree p fois
//(sans p!/2^p=fact2)
        for (j:=1;j<=p;j++){
            gk:=evalf(gk/(k+j+1));
        }
        sg:=sg+alt*gk;
        alt:=-alt;
    }
ls:=concat(ls,sf+fact2*sg);
}
return(ls);
}

```

On met ce programme dans un éditeur de programmes (que l'on ouvre avec Alt+p), puis on le teste et le valide avec OK et on tape dans une ligne de commandes :

```
seriealt_sumacc(200,3)
```

On obtient :

```
[0.69562855486,0.693153307335,0.693147210666,0.693147180781]
```

On tape :

```
seriealt_sumacc(100,4)
```

On obtient :

```
[0.698073169409,0.693171208625,0.693147412699,
0.693147183892,0.693147180623]
```

5.3.2 La transformation d'Euler pour les series alternées

La transformation d'Euler

On cherche une approximation de :

$\sum_{n=0}^{\infty} (-1)^n u(n) = \text{sum}((-1)^n u(n), n, 0, \text{infinity})$ lorsque $u(n)$ tend

vers 0 en décroissant.

On pose : $\Delta(u)(n) = u(n+1) - u(n)$ et

$\delta(u, p, n) = (\Delta^p(u))(n)$

On a :

$\delta(u, 2, n) = u(n+2) - 2u(n+1) + u(n)$

$\delta(u, 3, n) = u(n+3) - 3u(n+2) + 3u(n+1) - u(n)$

$\delta(u, p, N) = u(n+p) - \text{comb}(p, 1)u(n+p-1) + \text{comb}(p, 2)u(n+p-2) + \dots + (-1)^p u(n)$

c'est à dire :

$\delta(u, p, n) = \sum_{j=0}^p (-1)^{p-j} \text{comb}(p, j) u(n+j)$

La transformation d'Euler consiste à écrire :

$\sum_{n=N}^{\infty} (-1)^n u(n)$

sous la forme :

$(-1)^N \sum_{p=0}^{\infty} \delta(u, p, N) / 2^{p+1}$

Pour prouver cette égalité il suffit de développer la dernière expression et de chercher le coefficient de $u(N+k)$ dans la somme :

$$\sum_{p=0}^{\infty} (-1)^p * \frac{\delta(u, p, N)}{2^{p+1}}$$

Le coefficient de $u(N+k)$ est :

$s(k) = (-1)^k \sum_{p=0}^{\infty} \text{comb}(k+p, p) / 2^{k+p+1}$

et cette somme vaut $(-1)^k$ quelque soit k entier.

En effet par récurrence :

pour $k=0$, $\text{comb}(k+p, p) = 1$ et

$\sum_{p=0}^{\infty} 1/2^{p+1} = 1/2 + 1/4 + \dots = 1$

On a de plus :

- pour $p=0$, $\text{comb}(k+p, p) = \text{comb}(k+1+p, p) = 1$

- pour $p>0$, $\text{comb}(k+p, p) = \text{comb}(k+1+p, p) - \text{comb}(k+1+p-1, p-1)$

donc

$s(k) = (-1)^k \sum_{p=0}^{\infty} \text{comb}(k+1+p, p) / 2^{k+p+1} -$

$(-1)^k \sum_{p=1}^{\infty} \text{comb}(k+1+p-1, p-1) / 2^{k+p-1+1} =$

$-2s(k+1) -$

$(-1)^k \sum_{p=0}^{\infty} \text{comb}(k+1+p, p) / 2^{k+p+1} =$

$-2s(k+1) + s(k+1) = -s(k+1).$

donc si $s(k) = (-1)^k$ alors $s(k+1) = (-1)^{k+1}$.

La transformation d'Euler permet une accélération de convergence car la série :

$\sum_{p=0}^{\infty} (-1)^p \delta(u, p, N) / 2^{p+1}$

converge plus rapidement.

Le programme

On définit, tout d'abord, la fonction `delta` :

```
delta(u, p, n) := {
  local val, k, s;
  val := 0;
  s := 1;
  for (k:=p; k>=0; k--) {
    val := val + comb(p, k) * u(n+k) * s;
```

```

    s:=s*-1;
  }
  return val;
};

```

On écrit la transformation d'Euler :

`trans_euler(u, N, M)` qui approche $\text{sum}((-1)^n u(n), n, 0, \text{infinity})$ et vaut :

$$\text{sum}((-1)^n u(n), n, 0, N-1) + (-1)^N \text{sum}((-1/2)^p \text{delta}(u, p, N)/2, p, 0, M).$$

```

trans_euler(u, N, M) := {
  local S, T, k, s;
  S:=0;
  s:=1;
  for (k:=0; k<N; k++) {
    S:=S+u(k)*s;
    s:=s*-1;
  }
  T:=0;
  s:=s*1/2;
  for (k:=0; k<=M; k++) {
    T:=T+delta(u, k, N)*s;
    s:=s*-1/2;
  }
  return evalf(normal(S+T));
};

```

Par exemple pour $u(n) = 1/(n+1)$ avec 20 digits, on tape :

```

u(n) :=1/(n+1) ;
DIGITS :=20 ;
trans_euler(u, 10, 20) ;

```

On obtient :

```
0.693147180559945056511
```

```
trans_euler(u, 9, 21) ;
```

On obtient :

```
0.693147180559945594072
```

On remarque que l'on a 16 décimales exactes car on a :

```
evalf(ln(2))=0.693147180559945309415
```

5.3.3 Autre approximation d'une série alternée

La méthode présentée dans cette section est très largement inspirée par le texte “Somme de séries alternées” de l'épreuve de modélisation de l'agrégation de mathématiques (session 2006).

Le problème

On veut évaluer la somme S de la série alternée : $S = \sum_{n=0}^{\infty} (-1)^n a_n$ avec $(a_n)_{n \geq 0}$ est une suite de nombres positifs qui tend vers 0 en décroissant.

On suppose que l'on a pour $n \geq 0$:

$$a - n = \int_0^1 x^n d\mu$$

où μ est une mesure positive sur $[0,1]$.

C'est en particulier le cas si $a_n = A(n)$ avec A fonction indéfiniment dérivable pour laquelle les k ème dérivées $A^{(k)}$ sont telles que $(-1)^k * A^{(k)}(x)$ soit positif pour $x \geq 0$ pour tout $k \geq 0$.

Le théorème

Théorème :

Soit P_n une suite de polynômes de degré n vérifiant $P_n(-1) \neq 0$.

À P_n , on associe les coefficients $c_{n,k}$ pour $0 \leq k < n$ définis par :

$$\frac{P_n(-1) - P_n(x)}{1+x} = \sum_{k=0}^{n-1} c_{n,k} x^k$$

et le coefficient d_n défini par :

$$d_n = P_n(-1)$$

$$\text{Soient } S = \sum_{k=0}^{\infty} (-1)^k a_k \text{ et } S_n = \frac{1}{d_n} \sum_{k=0}^{n-1} c_{n,k} a^k$$

Alors :

$$|S - S_n| \geq \frac{\sup_{x \in [0,1]} |P_n(x)|}{|d_n|} S$$

On a, en effet, avec l'hypothèse faite sur les a_k :

$$S = \int_0^1 \frac{1}{1+x} d\mu \text{ et}$$

$$S - S_n = \int_0^1 \frac{P_n(x)}{d_n(1+x)} d\mu$$

Le choix des polynômes P_n

Pour calculer S il reste à choisir la suite des polynômes P_n .

On peut choisir :

$$- P_n(x) = (1-x)^n$$

on aura une convergence en 2^{-n} car $d_n = 2^n$ et $\sup_{x \in [0,1]} |P_n(x)| = 1$.

On a :

$$P_0(x) = 1$$

$$P_1(x) = 1 - x$$

$$P_{n+1}(x) = P_n(x) * (1-x) \text{ si } n \geq 0$$

$$d_n = 2^n$$

et la formule explicite de P_n :

$$P_n(x) = \sum_{k=0}^n (-1)^k C_n^k x^k = \sum_{k=0}^n p_{n,k} x^k \text{ si } n \geq 0$$

donc les coefficients $p_{n,k}$ vérifient :

$$p_{n,0} = 1$$

$$p_{n,k} = p_{n,k-1} * (k-1-n)/k \text{ pour } 1 \leq k < n$$

On a :

$$d_n - P_n(x) = (1+x) \sum_{k=0}^{n-1} c_{n,k} x^k = c_{n,0} \sum_{k=1}^n (c_{n,k-1} + c_{n,k}) x^k$$

donc

- $c_{n,0} = d_n - p_{n,0} = d_n - 1$
 $c_{n,k} = -c_{n,k-1} - p_{n,k}$ pour $1 \leq k < n$
 - $P_n(x) = x^q(1-x)^{2q}$ et $n = 3 * q$
 on aura une convergence en 3^{-n} car
 $|d_n| = 2^{2q}$ et
 $\sup_{x \in [0,1]} |P_n(x)| = P_n(1/3) = 2^{2q} * 3^{-n}$.
 On a :
 $P_0(x) = 1$
 $P_3(x) = x(1-x)^2$
 $P_{n+3}(x) = P_n(x) * x * (1-x)^2$ si $n \geq 0$
 $d_n = (-1)^q * 2^{2q}$
 et la formule explicite de P_n :
 $P_n(x) = \sum_{k=0}^{2q} (-1)^k C_{2q}^k x^{k+q} = \sum_{k=q}^n (-1)^{k-q} C_{2q}^{k-q} x^k =$
 $\sum_{k=0}^n p_{n,k} x^k$ si $n \geq 0$
 donc les coefficients $p_{n,k}$ vérifient :
 $p_{n,k} = 0$ si $k < q$
 $p_{n,q} = 1$
 et comme $C_n^p = C_n^{p-1} * (n-p+1)/p$
 $p_{n,k} = (-1)^{k-q} C_{2q}^{k-q}$ si $q \leq k \leq n$
 $p_{n,k} = -p_{n,k-1} * (k-1-n)/(k-q)$ si $q < k \leq n$
 On a :
 $d_n - P_n(x) = (1+x) \sum_{k=0}^{n-1} c_{n,k} x^k = c_{n,0} + \sum_{k=1}^n (c_{n,k-1} + c_{n,k}) x^k$
 donc
 $c_{n,0} = d_n - p_{n,0} = d_n - 1$
 $c_{n,k} = -c_{n,k-1} - p_{n,k}$ pour $1 \leq k < n$
 - Si le polynôme P_n défini $P_n(\sin(t)^2) = \cos(2nt)$
 P_n est défini à partir du polynôme T_n de Chebyshev ($T_n(\cos(t)) = \cos(nt)$) :
 $P_n(\sin(t)^2) = \cos(2nt) T_n(\cos(2t))$
 on a donc puisque $1 - 2 \sin(t)^2 = \cos(2t)$:
 $T_n(1 - 2 \sin(t)^2) = \cos(2nt)$ et
 $P_n(x) = T_n(1 - 2x)$
 on aura une convergence meilleure que dans les cas précédents car la convergence est en :
 $2/((3 + \sqrt{8})^n + (3 - \sqrt{8})^n) \simeq 2/((3 + \sqrt{8})^n) \simeq 2/(5.8)^n$
 car
 $d_n = ((3 + \sqrt{8})^n + (3 - \sqrt{8})^n)/2$ et $\sup_{x \in [0,1]} |P_n(x)| = 1$.
 Donc :
 $P_0(x) = 1$
 $P_1(x) = 1 - 2x$
 $P_{n+2}(x) = 2(1-2x)P_{n+1}(x) - P_n(x)$ si $n \geq 0$
 $d_n = ((3 + \sqrt{8})^n + (3 - \sqrt{8})^n)/2$
 et la formule explicite de P_n :
 $P_n(x) = \sum_{k=0}^n (-1)^k \frac{n}{n+k} C_{n+k}^{2k} 2^{2k} x^k =$
 $\sum_{k=0}^n p_{n,k} x^k$ si $n \geq 0$
 donc les coefficients $p_{n,k}$ vérifient :
 $p_{n,0} = 1$ $p_{n,k} = p_{k-1,n} (k-1+n)(k-1-n)/((k-1/2)(k))$ pour $1 \leq k < n$
 On a :

$$d_n - P_n(x) = (1+x) \sum_{k=0}^{n-1} c_{n,k} x^k = c_{n,0} \sum_{k=1}^n (c_{n,k-1} + c_{n,k}) x^k \text{ donc}$$

$$c_{n,0} = d_n - p_{0,n} \quad c_{n,k} = -c_{n,k-1} - p_{n,k} \text{ pour } 1 \leq k < n$$

Les formules de récurrences et le programme pour le polynôme Chebyshev

– Les formules de récurrences

On va calculer les coefficients $c_{n,k}$ de proche en proche pour n fixé.

On pose :

$p := 1$;

$d := ((3 + \sqrt{8})^n + (3 - \sqrt{8})^n) / 2$;

$c := d - p$;

Le premier terme de S_n :

$S := a(0) * c$;

puis, pour $k := 1$ jusque $k := n-1$ on calcule $p_{n,k}$ et $c_{n,k}$:

$p := p * (k+n-1) * (k-n-1) / (k-1/2) / k$;

$c := -p - c$;

On ajoute le k ième terme de S_n :

$S := S + a(k) * c$;

– Le programme

```
//n=nombre de termes et a fonction definissant a(n)
//S_n(P_n) =seriealt(n,a)
//S_n(P_n) approche sum((-1)^k*a(k),k,0,+infinity)
//avec P_n=poly de chebyshev
seriealt1(n,a):={
local k,d,c,p,S;
d:=((3+sqrt(8))^n+(3-sqrt(8))^n)/2;
p:=1;
c:=d-p;
S:=a(0)*c;
for (k:=1;k<n;k++) {
p:=p*(k+n-1)*(k-n-1)/(k-1/2)/k;
c:=-p-c;
S:=S+a(k)*c;
}
return evalf(S/d);
};
```

Les formules et le programme pour le polynôme $P_n(x) = (1-x)^n$

– Les formules de récurrences

On va calculer les coefficients $c_{n,k}$ de proche en proche pour n fixé.

On pose :

$p := 1$;

$d := 2^n$;

$c := d - p$;

Le premier terme de S_n :

```

S := a(0) * c ;
puis, pour k := 1 jusqu'à k := n-1 on calcule  $p_{n,k}$  et  $c_{n,k}$  :
p := p * (k-n-1) / k ;
c := -p - c ;
On ajoute le  $k$ ième terme de  $S_n$  :
S := S + a(k) * c ;

```

– **Le programme**

```

//n=nombre de termes et a fonction définissant a(n)
//S_n(P_n) = seriealt(n,a)
//S_n(P_n) approche sum((-1)^k*a(k),k,0,+infinity)
//avec P_n(x)=poly (1-x)^n
seriealt2(n,a):={
local k,d,c,p,S;
d:=2^n;
p:=1;
c:=d-p;
S:=a(0)*c;
for (k:=1;k<n;k++) {
p:=p*(k-n-1)/k;
c:=-p-c;
S:=S+a(k)*c;
}
return evalf(S/d);
};

```

Les formules et le programme pour le polynôme $P_{3q}(x) = x^q(1-x)^{2q}$

– **Les formules de récurrences**

On va calculer les coefficients $c_{n,k}$ de proche en proche pour n fixé.

On pose :

```

p := 0 ; si  $0 \leq k < q$ 
p := 1 ; si  $k = q$ 
d :=  $(-1)^q \cdot 2^{2q}$  ;
c := d - p ;

```

Le premier terme de S_n :

```

S := a(0) * c ;
pour k := 1 jusqu'à k := q-1 on a  $p_{n,k} = 0$  et on calcule  $c_{n,k}$  ( $c := -p - c$ ) ;
et on ajoute le  $k$ ième terme de  $S_n$  :
S := S + a(k) * c ;
puis, pour k := q on a  $p_{n,q} = 1$  et on calcule  $c_{n,q}$  ( $c := -p - c$ ) et on
ajoute le  $q$ ième terme de  $S_n$  :
S := S + a(q) * c ;
puis, pour k := q+1 jusqu'à k := n-1 on calcule  $p_{n,k}$  et  $c_{n,k}$  :
p := p * (k-n-1) / k ;
c := -p - c ;
On ajoute le  $k$ ième terme de  $S_n$  :
S := S + a(k) * c ;

```


– **Le programme**

```
//n=nombre de termes et a fonction definissant a(n)
//S_n(P_n) =seriealt(n,a)
//S_n(P_n) approche sum((-1)^k*a(k),k,0,+infinity)
//avec P_n=poly de chebyshev
seriealt3(n,a):={
local k,d,c,p,q,S;
q:=ceil(n/3);
n:=3*q;
d:=(-1)^q*2^(2*q);
p:=0;
c:=d-p;
S:=a(0)*c;
for (k:=1;k<q;k++) {
c:=-p-c;
S:=S+a(k)*c;
}
p:=1;
c:=-c-p;
S:=S+a(q)*c;
for (k:=q+1;k<n;k++) {
p:=p*(k-n-1)/(k-q);
c:=-p-c;
S:=S+a(k)*c;
}
return evalf(S/d);
};
```

Les essais

On choisit $n=20$.

On tape :

```
evalf(2/(3+sqrt(8))^20,2^-20,3^-21)=
9.77243031253e-16,9.53674316406e-07,9.55990663597e-11
```

On a donc pour $n=20$ une approximation en 10^{-15} pour Chebyshev, en 10^{-6} pour $(1-x)^{20}$ et en 10^{-10} pour $x^7(1-x)^{14}$:

On choisit dans la suite `Digits :=20`

Pour calculer une approximation de $\pi/4$.

On a :

```
sum((-1)^n/(2*n+1),n,0,+infinity)=pi/4
```

On tape :

```
b(n) :=1/(2*n+1)
seriealt1(20,b);evalf(pi/4)
```

On obtient :

```
0.785398163397448309926, 0.785398163397448309615
```

On tape :

```
seriealt2(20,b);seriealt3(20,b);
```

On obtient :

```
0.785397981918786731599, 0.785398163413201025973
```

Pour calculer une approximation de $\ln(2)$.

On a :

```
sum((-1)^n/(n+1),n,0,+infinity)=ln(2)
```

On tape :

```
a(n) :=1/(n+1)
```

```
seriealt1(30,a);evalf(ln(2))
```

On obtient :

```
0.693147180559945311245, 0.693147180559945309415
```

On tape :

```
seriealt2(20,a);seriealt3(20,a);
```

On obtient :

```
0.693147137051028936275, 0.693147180577738915258
```

5.3.4 Transformation d'une série en série alternée

On a l'identité formelle :

$$\sum_{n \geq 1} a_n = \sum_{m \geq 1} (-1)^{m-1} b_m \text{ avec}$$

$$b_m = \sum_{k \geq 0} 2^k a_{2^k m}.$$

En effet, si n_0 est un entier il existe un entier p_0 et un entier impair m_0 uniques vérifiant $n_0 = 2^{p_0} * m_0$. Dans la somme $\sum_{m \geq 1} (-1)^{m-1} \sum_{k \geq 0} 2^k a_{2^k m}$ on cherche le coefficient de a_{n_0} , on a soit :

$k = 0$ et $m = n_0 = m_0 * 2^{p_0}$, soit

$k = 1$ et $m = m_0 * 2^{p_0-1}$, soit

..... soit

$k = p_0$ et $m = m_0$.

On remarquera que toutes les valeurs, sauf la dernière, de m sont paires, donc les différentes valeurs de $(-1)^{m-1}$ sont (-1) sauf la dernière qui vaut $+1$.

$$\sum_{m \geq 1} \sum_{k \geq 0} (-1)^{m-1} 2^k a_{2^k m} =$$

$$\sum_{n_0 \geq 1} a_{n_0} * (\sum_{k=0}^{p_0-1} (-1) * 2^k + 2^{p_0}) =$$

$$\sum_{n \geq 1} a_n \text{ puisque } 2^{p_0} - \sum_{k=0}^{p_0-1} 2^k = 1$$

Prenons comme exemple la série de terme général $a_n = \frac{1}{n^s}$ avec $s > 1$: pour $s=2$

si $a(n) = 1/n^2$

on a :

$$2^k * a(2^k * m) = 1 / (2^k * m^2)$$

$$b(m) = 1/m^2 * \sum (1/2^k, k, 0, +infinity) = 2/m^2$$

si

$$a(n) = 1/n^s, \quad 2^k * a(2^k * m) = 1 / (2^k * (s-1) * m^s)$$

$$b(m) = 1/m^s * \sum ((1/2^{(s-1)})^k, k, 0, +infinity)$$

Donc :

$$b(m) = 2^{(s-1)} / ((2^{(s-1)} - 1) * m^s)$$

pour $s=2$

$$b(m) := 2 / (m^2)$$

pour $s=4$

$$b(m) := 8 / (7 * m^4)$$

On a :

```
sum((-1)^(m-1)*b(m),1,+infinity)=
sum((-1)^(m)*b(m+1),0,+infinity)
```

On choisit encore Digits :=20

pour s=2, $\sum_{n=1}^{\infty} 1/n^2 = \pi^2/6$

On tape :

```
t2(m) :=2/(m+1)^2
seriealt1(20,t2),evalf(pi^2/6)
```

On obtient :

```
1.64493406684822645248, 1.64493406684822643645
```

On tape :

```
seriealt2(20,t2);seriealt3(20,t2);
```

On obtient :

```
1.64493374613777534516, 1.64493406688805599300
```

pour s=4, $\sum_{n=1}^{\infty} 1/n^4 = \pi^4/90$

On tape :

```
t4(m) :=8/(7*(m+1)^4)
seriealt(20,t4);evalf(pi^4/90)
```

On obtient :

```
1.08232323371113822384, 1.08232323371113819149
```

On tape :

```
seriealt2(20,t4);seriealt3(20,t4);
```

On obtient :

```
1.08232265198912440013, 1.08232323371697925335
```

pour calculer une approximation de la constante d'Euler=-psi(1).

On a :

```
-psi(1)=sum((-1)^n*ln(n)/n,n,1,+infinity)/ln(2)+ln(2)/2
et
```

```
sum((-1)^n*ln(n)/n,n,1,+infinity)=
```

```
-sum((-1)^n*ln(n+1)/(n+1),n,0,+infinity)
```

```
c(n) :=log(n+1)/(n+1)
```

```
-seriealt(20,c)/ln(2)+ln(2)/2;-evalf(psi(1),0)
```

On obtient :

```
0.577215664901532859864, 0.57721566490153
```

On tape :

```
-seriealt2(20,c)/ln(2)+ln(2)/2,-seriealt3(20,c)/ln(2)+ln(2)/2
```

On obtient :

```
0.577215550220266823551, 0.577215664918305723256
```

5.4 Solution de $f(x) = 0$ par la méthode de Newton

Dans xcas, il existe déjà une fonction qui calcule la valeur approchée d'une solution de $f(x) = 0$ par la méthode de Newton, qui est : newton.

5.4.1 La méthode de Newton

Soit f deux fois dérivable ayant un zéro et un seul r dans l'intervalle $[a ; b]$. Supposons de plus que f' et f'' ont un signe constant sur $[a ; b]$. La méthode de Newton consiste à approcher r par l'abscisse x_1 du point commun à Ox et à la tangente en un point M_0 du graphe de f . Si M_0 a pour coordonnées $(x_0, f(x_0))$ ($x_0 \in [a ; b]$), la tangente en M_0 a pour équation :
 $y = f(x_0) + f'(x_0) * (x - x_0)$ et donc on a :

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

On peut alors réitérer le processus, et on obtient une suite x_n qui converge vers r soit par valeurs supérieures, si $f' * f'' > 0$ sur $[a ; b]$ (i.e. si $f'(r) > 0$ et si f est convexe ($f'' > 0$ sur $[a ; b]$) ou si $f'(r) < 0$ et si f est concave ($f'' < 0$ sur $[a ; b]$)) soit par valeurs inférieures, si $f' * f'' < 0$ sur $[a ; b]$ (i.e. si $f'(r) < 0$ et si f est convexe ($f'' > 0$ sur $[a ; b]$) ou si $f'(r) > 0$ et si f est concave ($f'' < 0$ sur $[a ; b]$)).

On fait le dessin en tapant :

```
f:=sq-2;
x0:=5/2;
G:=plotfunc(f(x));
T0:=tangent(G,x0);
Ox:=droite(0,1);
M1:=inter(T0,Ox)[0];
x1:=affiche(M1)
segment(point(x1,0),point(x1,f(x1)));
T1:=tangent(G,x1);
M2:=inter(T1,droite(0,1))[0];
x2:=affiche(M2);
segment(point(x2,0),point(x2,f(x2)));
```

Puis, on écrit la fonction `newton_rac` qui renvoie la valeur approchée à ϵ près de la racine de $f(x) = 0$ on commençant l'itération avec x_0 .

On remarquera que le paramètre f est une fonction et donc, que sa dérivée est la fonction `g := fonction_diff(f)`.

On cherche une valeur approchée donc il faut écrire :

```
x0 :=evalf(x0-f(x0)/g(x0))
car si on ne met pas evalf, les calculs de l'itération se feront exactement et
seront vite compliqués.
```

```
newton_rac(f,x0,eps):={
local x1,h,g;
g:=fonction\_diff(f)
x0:=evalf(x0-f(x0)/g(x0));
x1:=x0-f(x0)/g(x0);
if (x1>x0) {h:=eps;} else {h:=-eps;}
while (f(x1)*f(x1+h)>0) {
x1:=x1-f(x1)/g(x1);
}
```

```
return x1;
}
```

5.4.2 La méthode de Newton avec préfacteur

Lorsqu'on part d'une valeur x_0 trop éloignée de la racine de $f(x)$ (si par exemple $|f(x_0)|$ est grand), on a intérêt à utiliser un préfacteur pour se rapprocher plus vite de la solution de $f(x) = 0$.

Posons $n(x) = -\frac{f(x)}{f'(x)}$, on a alors :

$$\lim_{h \rightarrow 0} \frac{(f(x_0 + h * n(x_0)) - f(x_0))}{h * n(x_0)} = f'(x_0)$$

donc

$$\lim_{h \rightarrow 0} \frac{(f(x_0 + h * n(x_0)) - f(x_0))}{h} = n(x_0) * f'(x_0) = -f(x_0)$$

ce qui veut dire que :

$f(x_0 + h * n(x_0)) = f(x_0)(1 - h) + h \cdot \epsilon(h)$ avec $\epsilon(h)$ tend vers 0 quand h tend vers 0.

Donc, il existe h_0 vérifiant :

$$|f(x_0 + h_0 * n(x_0))| < |f(x_0)|$$

Remarque : Il faut minimiser $|f(x_0 + h_0 * n(x_0))|$. or plus h_0 est proche de 1 et plus $|f(x_0) * (1 - h_0)|$ sera petit. Par exemple, on prendra le plus grand h_0 , dans la liste $[1, 3/4, (3/4)^2, \dots]$ qui vérifie $|f(x_0 + h_0 * n(x_0))| < |f(x_0)|$

Pour cette valeur de h_0 , $x_0 + h_0 * n(x_0)$ est probablement plus proche de la racine que x_0 : on dit que h_0 est le préfacteur de la méthode de Newton.

On va choisir par exemple au début $h_0 = 1$, et on regarde si $|f(x_0 + n(x_0))| < |f(x_0)|$, si ce n'est pas le cas on prend $h_0 = (3/4)$ et on regarde si $|f(x_0 + 3/4 * n(x_0))| < |f(x_0)|$, si ce n'est pas le cas on prend $h_0 = (3/4)^2$ etc...

On change de préfacteurs à chaque étape jusqu'à ce que : $abs(f(x_1)) - abs(f(x_0)) < 0$ sans préfacteur, on continue alors l'itération sans préfacteur, c'est à dire avec la méthode de Newton normale. On écrit donc :

```
newton_prefacts(f, x0, eps) := {
  local x1, h, h0, prefact, niter;
  //prefact est egal par ex a 3/4
  h0:=1.0;
  niter:=0;
  prefact:=0.75;
  x1:=x0-h0*f(x0)/function_diff(f)(x0);
  while (abs(f(x1))-abs(f(x0))>0) {
    h0:=h0*prefact;
    x1:=x0-h0*f(x0)/function_diff(f)(x0);
  }
  h:=eps;
  while (h0!=1 and niter<100){
    x0:=x1;
```

```

x1:=x1-h0*f(x1)/function_diff(f)(x1);
while (abs(f(x1))-abs(f(x0))>0) {
    h0:=h0*prefact;
    x1:=x0-h0*f(x0)/function_diff(f)(x0);
}
while (abs(f(x1))-abs(f(x0))<0 and h0!=1) {
    h0:=h0/prefact;
    x1:=x0-h0*f(x0)/function_diff(f)(x0);
}
niter:=niter+1;
}
while (f(x1-h)*f(x1+h)>0 and niter<200){
    x0:=x1;
    x1:=x1-f(x1)/function_diff(f)(x1);
    niter:=niter+1;
}

if (niter<200) {return x1;} else {return "pas trouve";}
}

```

On définit la fonction f par $f(x) := x^2 - 2$ et on met ce programme dans un éditeur de programmes (que l'on ouvre avec **Alt+p**), puis on le teste et le valide avec **OK**.

On tape :

```
newton_prefacts(f,100,1e-10)
```

On obtient :

```
1.41421356237
```

On tape :

```
newton_prefacts(f,3,1e-5)
```

On obtient :

```
1.41421378005
```

5.5 Trouver un encadrement de la valeur pour laquelle une fonction est minimum

Soit f une fonction définie sur $[a; b]$. On suppose que f est unimodale sur $[a; b]$, c'est à dire que f a un seul extremum sur $[a; b]$. On suppose de plus que cet extremum est un minimum (sinon on remplacera f par $-f$.) On se propose de trouver un encadrement à eps près de la valeur pour laquelle f est minimum.

5.5.1 Description du principe de la méthode

On partage $[a; b]$ en trois morceaux en considérant c et d vérifiant : $a < c < d < b$.

On calcule $f(c)$ et $f(d)$ et on les compare.

Puisque f a un seul minimum sur $[a; b]$ elle décroît, passe par son minimum, puis f croît. Selon les trois cas possibles on a :

- $f(c) < f(d)$

dans ce cas, la valeur pour laquelle f est minimum n'appartient pas à $[d; b]$

5.5. TROUVER UN ENCADREMENT DE LA VALEUR POUR LAQUELLE UNE FONCTION EST MINIMUM

- $f(c) > f(d)$
dans ce cas, la valeur pour laquelle f est minimum n'appartient pas à $[a; c]$
- $f(c) = f(d)$
dans ce cas, la valeur pour laquelle f est minimum n'appartient pas à $[d; b]$
ni à $[a; c]$

Ainsi, l'intervalle de recherche a diminué et on peut recommencer le processus. Pour que l'algorithme soit performant, on veut que l'intervalle de recherche diminue rapidement et que le nombre de valeurs de f à calculer soit le plus petit possible. Pour cela comment doit-on choisir c et d ?

5.5.2 Description de 2 méthodes

On fait presque une dichotomie

On choisit c et d proche de $\frac{a+b}{2}$ par exemple :
 $c = \frac{a+b-eps}{2}$ et $d = \frac{a+b+eps}{2}$ pour eps donné. Dans ce cas, à chaque étape l'intervalle diminue presque de moitié mais on doit calculer, à chaque étape, deux valeurs de f .

On utilise la suite de Fibonacci

Comment faire pour que l'une des valeurs de f déjà calculée serve à l'étape suivante ?

La solution se trouve dans la suite de Fibonacci, suite définie par :

$u_0 = 1, u_1 = 2, u_n = u_{n-2} + u_{n-1}$ dont les premiers termes sont : 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

Par exemple si on partage $[a; b]$ en 89 parties égales si $l = (b - a)/89$, on choisit $c = a + 34 * l$ et $d = a + 55 * l$ et ainsi on a :

$c - a = 34 * l, d - c = 21 * l, b - d = 34 * l$ (car $89 = 55 + 34$ et $34 + 21 = 55$ puisque 21, 34, 55, 89 sont des termes consécutifs de la suite de Fibonacci).

On calcule $f(c)$ et $f(d)$ puis on réduit l'intervalle en un intervalle de longueur $(b - a) * 55/89$, par exemple si l'intervalle suivant est $[a; d]$ et, si on recommence le processus, le point c est le futur point d .

Donc à chaque étape il suffit de calculer une seule valeur de f pour passer de l'intervalle $[a; b]$ (proportionnel à u_n) à l'intervalle $[a; d]$ ou $[c; b]$ (proportionnel à u_{n-1}). Il y a bien sûr le cas $f(c) = f(d)$ où il faut à l'étape suivante calculer deux valeurs de f , mais dans ce cas on gagne 3 étapes car on passe de l'intervalle $[a; b]$ (proportionnel à u_n) à l'intervalle $[c; d]$ (proportionnel à u_{n-3}).

Selon la valeur eps de la longueur de l'encadrement, on calcule $k := \text{ceil}(2 * (b - a)/eps)$; et la première valeur $t = u_n$ de la suite de Fibonacci supérieure à k . il faut alors diviser l'intervalle $[a; b]$ en t parties égales. On applique alors plusieurs fois le processus et on s'arrête quand $n = 1$, c'est à dire quand l'intervalle a été réduit à un intervalle de longueur $2 * (b - a)/t$ qui est, grace au choix de t ($t > k > 2 * (b - a)/eps$) inférieur à eps .

5.5.3 Traduction Xcas de l'algorithme avec Fibonacci

```
// f(x) := 2*x^4 - 10*x^3 + 4*x^2 + 100
// fibomin(f, 1, 5, 0.000001)
// g(x) := 2*x^4 - 10*x^3 + 4*x^2 + 100
```

```

//fibomin(g,1,5,1e-20)
//calcul la valeur du min d'une fonction ayant un seul extrema sur [a
fibomin(f,a,b,eps):={
    local c,d,F,k,n,t,g,h,l,fc,fd;
    if (a>b) {c:=a;a:=b;b:=c;}
    k:=ceil(2*(b-a)/eps);
    F:=1,2;
    n:=1;
    g:=1;
    t:=2;
    //construction de F=la suite de Fibonacci
    //h,g,t sont 3 termes consecutifs de F
    while (t<k) {
        n:=n+1;
        h:=g;
        g:=t;
        t:=h+g;
        F:=F,t;
    }
    l:=(b-a)/t;
    c:=a+h*l;
    d:=a+g*l;
    fc:=f(c);
    fd:=f(d);
    //on itere le processus et on s'arrete qd n=1
    while (n>1) {
        if (fc>fd) {
            a:=a+h*l;
            fc:=fd;
            t:=h;
            h:=g-h;
            g:=t;
            fd:=f(a+g*l);
            n:=n-1;
        }else{
            if (fc<fd) {
b:=a+g*l;
t:=h;
h:=g-h;
g:=t;
fd:=fc;
fc:=f(a+h*l);
n:=n-1;
            }else{
a:=a+h*l;
b:=b-h*l;
t:=g-h;
g:=h-t;

```


5.5. TROUVER UN ENCADREMENT DE LA VALEUR POUR LAQUELLE UNE FONCTION EST MINIMUM

```
h:=t-g;
fc:=f(a+h*1);
fd:=f(a+g*1);
n:=n-3;
    }
  }
}
return [a,b];
}
```

On tape :

```
f(x) :=x^4-10
fibomin(f,-1,1,1e-10)
```

On obtient :

```
[(-1)/53316291173,1/53316291173]
```

On tape :

```
g(x) :=2*x^4-10*x^3-4*x^2+100
fibomin(g,1,5,1e-10)
```

On obtient :

```
[86267571271/21566892818,86267571273/21566892818] On tape :
```

```
h(x) :=2*x^4-10*x^3+4*x^2+100
fibomin(h,1,5,1e-10)
```

On obtient :

```
[74644573011/21566892818,74644573013/21566892818]
```


Chapitre 6

Algorithmes d'algèbre

6.1 Méthode pour résoudre des systèmes linéaires

Dans `xcas`, il existe déjà les fonctions qui correspondent aux algorithmes qui suivent, ce sont : `ref`, `rref`, `ker`, `pivot`

6.1.1 Le pivot de Gauss quand A est de rang maximum

Étant donné un système d'équations noté $AX = b$, on cherche à le remplacer par un système équivalent et triangulaire inférieur.

À un système d'équations $AX = b$, on associe la matrice M formée par A que l'on borde avec b comme dernière colonne.

La méthode de Gauss (resp Gauss-Jordan) consiste à multiplier A et b (donc M) par des matrices inversibles, afin de rendre A triangulaire inférieure (resp diagonale). Cette transformation, qui se fera au coup par coup en traitant toutes les colonnes de A (donc toutes les colonnes de M , sauf la dernière), consiste par des combinaisons de lignes de M à mettre des zéros sous (resp de part et d'autre) la diagonale de A . La fonction `gauss_redi` ci-dessous, transforme M par la méthode de Gauss, la variable `pivo` (car `pivot` est une fonction de `xcas`) sert à mettre le pivot choisi. Pour $j = 0..p-2$ ($p-2$ car on ne traite pas la dernière colonne de M), dans chaque colonne j , on cherche ce qui va faire office de pivot à partir de la diagonale : dans le programme ci-dessous on choisit le premier élément non nul, puis par un échange de lignes, on met le pivot sur la diagonale ($pivo = M[j, j]$). Il ne reste plus qu'à former, pour chaque ligne k ($k > j$) et pour $a = M[k, j]$, la combinaison : $pivo * ligne_k - a * ligne_j$ (et ainsi $M[k, j]$ devient nul).

On écrit pour réaliser cette combinaison :

```
a := M[k, j] ;  
for (l := j ; l < nc ; l++) {  
  M[k, l] := M[k, l] * pivo - M[j, l] * a ; }
```

On remarquera qu'il suffit que l parte de j car :

pour tout $l < j$, on a déjà obtenu, par le traitement des colonnes $l = 0..j-1$, $M[k, l] = 0$.

Le programme ci-dessous ne sera utile que si on trouve un pivot pour chaque colonne : c'est à dire si la matrice A est de rang maximum. En effet, dans ce programme, si on ne trouve pas de pivot (i. e. si tous les éléments d'une colonne sont nuls sur et sous la diagonale), on continue comme si de rien était...

```

gauss_redi(M) := {
  local pivo, j, k, nl, nc, temp, l, n, a;
  nl := nrow(M);
  nc := ncol(M);
  n := min(nl, nc - 1);
  // on met des 0 sous la diagonale du carre n*n
  for (j := 0; j < n; j++) {
    // choix du pivot mis ds pivo
    k := j;
    while (M[k, j] == 0 and k < nl - 1) {k := k + 1;}
    // on ne fait la suite que si on a pivo != 0
    if (M[k, j] != 0) {
      pivo := M[k, j];
      // echange de la ligne j et de la ligne k
      for (l := j; l < nc; l++) {
        temp := M[j, l];
        M[j, l] := M[k, l];
        M[k, l] := temp;
      }
      // fin du choix du pivot qui est M[j, j]
      for (k := j + 1; k < nl; k++) {
        a := M[k, j];
        for (l := j; l < nc; l++) {
          M[k, l] := M[k, l] * pivo - M[j, l] * a;
        }
      }
    }
  }
  return M;
}

```

On tape :

```

M0 := [[1, 2, 3, 6], [2, 3, 1, 6], [3, 2, 1, 6]]
gauss_redi(M0)

```

On obtient :

```

[[1, 2, 3, 6], [0, -1, -5, -6], [0, 0, -12, -12]]

```

On tape :

```

M1 := [[1, 2, 3, 4], [0, 0, 1, 2], [0, 0, 5, 1]]
gauss_redi(M1)

```

On obtient :

```

[[1, 2, 3, 4], [0, 0, 1, 2], [0, 0, 5, 1]]

```

On tape :

```

M2 := [[1, 2, 3, 4], [0, 0, 1, 2], [0, 0, 5, 1], [0, 0, 3, 2], [0, 0, -1, 1]]
gauss_redi(M2)
On obtient :
[[1, 2, 3, 4], [0, 0, 1, 2], [0, 0, 5, 1], [0, 0, 0, 7], [0, 0, 0, 6]]

```

c'est à dire :

$$\text{gauss_redi} \left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 5 & 1 \\ 0 & 0 & 3 & 2 \\ 0 & 0 & -1 & 1 \end{bmatrix} \right) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 5 & 1 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

6.1.2 Le pivot de Gauss pour A quelconque

On cherche un programme valable lorsque A est quelconque : on veut, dans ce cas, mettre des zéros sous la "fausse diagonale" de A (ce qu'on appelle "fausse diagonale" c'est la diagonale obtenue en ne tenant pas compte des colonnes pour lesquelles la recherche du pivot a été infructueuse : un peu comme si ces colonnes étaient rejetées à la fin de la matrice).

Donc, dans ce programme, si on ne trouve pas de pivot pour la colonne d'indice jc (i. e. si tous les éléments de la colonne jc sont nuls sur et sous la diagonale), on continue en cherchant, pour la colonne suivante (celle d'indice $jc + 1$), un pivot à partir de l'élément situé à la ligne d'indice jc (et non comme précédemment à partir de $jc + 1$), pour mettre sur la colonne $jc + 1$, des zéros sur les lignes $jc + 1, \dots, nl - 1$. On est donc obligé, d'avoir 2 indices jl et jc pour repérer les indices de la "fausse diagonale".

```
gauss_red(M) := {
  local pivo, jc, jl, k, nl, nc, temp, l, a;
  nl := nrows(M);
  nc := ncols(M);
  //on met des 0 sous la fausse diagonale d'indice jl, jc
  jc := 0;
  jl := 0;
  //on traite chaque colonne (indice jc)
  while (jc < nc - 1 and jl < nl - 1) {
    //choix du pivot que l'on veut mettre en M[jl, jc]
    k := jl;
    while (M[k, jc] == 0 and k < nl - 1) {k := k + 1;}
    //on ne fait la suite que si on a M[k, jc] (=pivo) != 0
    if (M[k, jc] != 0) {
      pivo := M[k, jc];
      //echange de la ligne jl et de la ligne k
      for (l := jc; l < nc; l++) {
        temp := M[jl, l];
        M[jl, l] := M[k, l];
        M[k, l] := temp;
      }
    }
    //fin du choix du pivot qui est M[jl, jc]
    //on met des 0 sous la fausse diagonale de la colonne jc
    for (k := jl + 1; k < nl; k++) {
      a := M[k, jc];
      for (l := jc; l < nc; l++) {
```

```

        M[k,l]:=M[k,l]*pivo-M[jl,l]*a;
    }
}
//on a 1 pivot,l'indice-ligne de la fausse diag augmente de 1
jl:=jl+1;
} //fin du if (M[k,jc]!=0)
//colonne suivante,l'indice-colonne de la fausse diag augmente de
jc:=jc+1;
} //fin du while
return M;
}

```

On tape :

```
M0 := [[1,2,3,6],[2,3,1,6],[3,2,1,6]]
```

```
gauss_red(M0)
```

On obtient :

```
[[1,2,3,6],[0,-1,-5,-6],[0,0,-12,-12]]
```

On tape :

```
M1 := [[1,2,3,4],[0,0,1,2],[0,0,5,1]]
```

```
gauss_red(M1)
```

On obtient :

```
[[1,2,3,4],[0,0,1,2],[0,0,0,-9]]
```

On tape :

```
M2 := [[1,2,3,4],[0,0,1,2],[0,0,5,1],[0,0,3,2],[0,0,-1,1]]
```

```
gauss_red(M2)
```

On obtient :

```
[[1,2,3,4],[0,0,1,2],[0,0,0,-9],[0,0,0,-4],[0,0,0,3]]
```

$$\text{gauss_red} \left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 5 & 1 \\ 0 & 0 & 3 & 2 \\ 0 & 0 & -1 & 1 \end{bmatrix} \right) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & -9 \\ 0 & 0 & 0 & -4 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

6.1.3 La méthode de Gauss-Jordan

Étant donnée un système d'équations, on cherche à le remplacer par un système diagonale équivalent. À un système d'équations $AX = b$, on associe la matrice M formée de A , bordée par b comme dernière colonne.

La fonction `gaussjordan_redi` transforme M par la méthode de Gauss-Jordan.

On cherche dans chaque colonne j , ($j = 0..nc - 2$) à partir de la diagonale, ce qui va faire office de pivot : dans le programme ci-dessous on choisit le premier élément non nul, puis par un échange de lignes, on met le pivot sur la diagonale ($pivo = M[j, j]$). Il ne reste plus qu'à former, pour chaque ligne k ($k \neq j$) et pour $a = M[k, j]$, la combinaison :

$pivo * ligne_k - a * ligne_j$ (et ainsi $M[k, j]$ devient nul).

On écrit pour réaliser cette combinaison :

lorsque $k < j$

```
a :=M[k,j] ;
```

```
for (l := 0 ; l < j ; l++) {
  M[k, l] := M[k, l] * pivo - M[j, l] * a ;}
```

et lorsque $k > j$

```
a := M[k, j] ;
for (l := j ; l < nc ; l++) {
  M[k, l] := M[k, l] * pivo - M[j, l] * a ;}
```

On remarquera que l part soit de 0 soit de j car pour $l < j$, on a $M[k, l] = 0$ seulement si $k > j$.

Si on ne trouve pas de pivot, on continue comme si de rien n'était : on obtiendra donc des zéros au dessus de la diagonale que si on a trouvé un pivot pour chaque colonne.

```
gaussjordan_redi(M) := {
  local pivo, j, k, nl, nc, temp, l, n, a ;
  nl := nrow(M) ;
  nc := ncol(M) ;
  n := min(nl, nc - 1) ;
  // on met des 0 sous et au dessus de la diagonale du carre n*n
  for (j := 0 ; j < n ; j++) {
    // choix du pivot
    k := j ;
    while (M[k, j] == 0 and k < nl - 1) {k := k + 1 ;}
    // on ne fait la suite que si on a pivo != 0
    if (M[k, j] != 0) {
      pivo := M[k, j] ;
      // echange de la ligne j et de la ligne k
      for (l := j ; l < nc ; l++) {
        temp := M[j, l] ;
        M[j, l] := M[k, l] ;
        M[k, l] := temp ;
      }
      // fin du choix du pivot qui est M[j, j]
      // on met des zeros au dessus de la diag
      for (k := 0 ; k < j ; k++) {
        a := M[k, j] ;
        for (l := 0 ; l < nc ; l++) {
          M[k, l] := M[k, l] * pivo - M[j, l] * a ;
        }
      }
      // on met des zeros au dessous de la diag
      for (k := j + 1 ; k < nl ; k++) {
        a := M[k, j] ;
        for (l := j ; l < nc ; l++) {
          M[k, l] := M[k, l] * pivo - M[j, l] * a ;
        }
      }
    }
  }
  return M ;}
```

```
}
```

De la même façon que pour la méthode de Gauss, on va mettre des zéros sous la "fausse diagonale" et au dessus de cette "fausse diagonale" (on ne pourra pas bien sûr, mettre des zéros au dessus de cette "fausse diagonale", pour les colonnes sans pivot !!)

```
gaussjordan_red(M) := {
  local pivo, jc, jl, k, nl, nc, temp, l, a;
  nl := nrows(M);
  nc := ncols(M);
  //on met des 0 sous la fausse diagonale
  jc := 0;
  jl := 0;
  //on doit traiter toutes les colonnes sauf le dernière et toutes les
  while (jc < nc-1 and jl < nl) {
    //choix du pivot que l'on veut mettre en M[jl, jc]
    k := jl;
    while (M[k, jc] == 0 and k < nl-1) {k := k+1;}
    //on ne fait la suite que si on a pivo != 0
    if (M[k, jc] != 0) {
      pivo := M[k, jc];
      //echange de la ligne jl et de la ligne k
      for (l := jc; l < nc; l++) {
        temp := M[jl, l];
        M[jl, l] := M[k, l];
        M[k, l] := temp;
      }
    }
    //fin du choix du pivot qui est M[jl, jc]
    //on met des 0 au dessus la fausse diagonale de la colonne jc
    for (k := 0; k < jl; k++) {
      a := M[k, jc];
      for (l := 0; l < nc; l++) {
        M[k, l] := M[k, l] * pivo - M[jl, l] * a;
      }
    }
    //on met des 0 sous la fausse diagonale de la colonne jc
    for (k := jl+1; k < nl; k++) {
      a := M[k, jc];
      for (l := jc; l < nc; l++) {
        M[k, l] := M[k, l] * pivo - M[jl, l] * a;
      }
    }
    //on a un pivot, le numero de ligne de la fausse diag augmente de
    jl := jl+1;
  }
  //ds tous les cas, le numero de colonne de la fausse diag augmente
  jc := jc+1;
}
```



```
return M;
}
```

On tape :

```
M0 := [[1,2,3,6],[2,3,1,6],[3,2,1,6]]
gaussjordan_red(M0)
```

On obtient :

```
[[12,0,0,12],[0,12,0,12],[0,0,-12,-12]]
```

On tape :

```
M1 := [[1,2,3,4],[0,0,1,2],[0,0,5,1]]
gaussjordan_red(M1)
```

On obtient :

```
[[1,2,0,-2],[0,0,1,2],[0,0,0,-9]]
```

On tape :

```
M2 := [[1,2,3,4],[0,0,1,2],[0,0,5,1],[0,0,3,2],[0,0,-1,1]]
gaussjordan_red(M2)
```

On obtient :

```
[[1,2,0,-2],[0,0,1,2],[0,0,0,-9],[0,0,0,-4],[0,0,0,3]]
```

$$\text{gaussjordan_red} \left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 5 & 1 \\ 0 & 0 & 3 & 2 \\ 0 & 0 & -1 & 1 \end{bmatrix} \right) = \begin{bmatrix} 1 & 2 & 0 & -2 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & -9 \\ 0 & 0 & 0 & -4 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

6.1.4 La méthode de Gauss et de Gauss-Jordan avec recherche du pivot

On peut bien sûr modifier la recherche du pivot.

Pour les méthodes numériques il est recommandé de normaliser les équations (on divise chaque ligne k par $\max_j |a_{k,j}|$) et de choisir le pivot qui a la plus grande valeur absolue.

En calcul formel, on prend l'expression exacte la plus simple possible. Ici on normalise les équations à chaque étape et on choisit le pivot qui a la plus grande valeur absolue : c'est ce que font, avec ce choix du pivot les programmes (cf le répertoire exemples), `gauss_reducni` (analogue à `gauss_redi`), `gauss_reducn` (analogue à `gauss_red`), `gaussjordan_reducni` (analogue à `gaussjordan_redi`) et `gaussjordan_reducn` (analogue à `gaussjordan_red`) :

```
gaussjordan_reducn(M) := {
local pivo, j, jc, jl, k, nl, nc, temp, l, a, piv, kpiv, maxi;
nl:=nrows(M);
nc:=ncols(M);
//on met des 0 sous la fausse diagonale
jc:=0;
jl:=0;
while (jc<nc-1 and jl<nl) {
  //on normalise les lignes
```

```

for (jj:=0;jj<nl;jj++) {
    maxi:=max(abs(seq(M[jj,kk], kk=0..nc-1)));
    for (kk:=0;kk<nc;kk++) {
        M[jj,kk]:=M[jj,kk]/maxi;
    }
}
//choix du pivot que l'on veut mettre en M[jl,jc]
kpiv:=jl;
piv:=abs(M[kpiv,jc]);
for (k:=jl+1;k<nl;k++){
    if (abs(M[k,jc])>piv) {piv:=abs(M[k,jc]);kpiv:=k;}
}
//on ne fait la suite que si on a piv!=0
if (piv!=0) {
    pivo:=M[kpiv,jc];
    k:=kpiv;
    //echange de la ligne jl et de la ligne k
    for (l:=jc;l<nc;l++){
        temp:=M[jl,l];
        M[jl,l] := M[k,l];
        M[k,l]:=temp;
    }
    //fin du choix du pivot qui est M[jl,jc]
    //on met des 0 au dessus la fausse diagonale de la colonne jc
    for (k:=0;k<jl;k++) {
        a:=M[k,jc];
        for (l:=0;l<nc;l++){
            M[k,l]:=M[k,l]*pivo-M[jl,l]*a;
        }
    }
    //on met des 0 sous la fausse diagonale de la colonne jc
    for (k:=jl+1;k<nl;k++) {
        a:=M[k,jc];
        for (l:=jc;l<nc;l++){
            M[k,l]:=M[k,l]*pivo-M[jl,l]*a;
        }
    }
    //on a un pivot, le numero de ligne de la fausse diag augmente de
    jl:=jl+1;
}
//ds tous les cas, le numero de colonne de la fausse diag augmente
jc:=jc+1;
}
return M;
}

```

On tape :

```
M0 := [[1,2,3,6],[2,3,1,6],[3,2,1,6]]
```

```
gaussjordan_reducn(M0)
```

On obtient :

```
[[5/6,0,0,5/6],[0,5/6,0,5/6],[0,0,1,1]]
```

On tape :

```
M1 := [[1,2,3,4],[0,0,1,2],[0,0,5,1]]
```

```
gaussjordan_reducn(M1)
```

On obtient :

```
[[1/4,1/2,0,17/20],[0,0,1,1/5],[0,0,0,9/10]]
```

On tape :

```
M2 := [[1,2,3,4],[0,0,1,2],[0,0,5,1],[0,0,3,2],[0,0,-1,1]]
```

```
gaussjordan_reducn(M2)
```

On obtient :

```
[[1/4,1/2,0,17/20],[0,0,1,1/5],[0,0,0,9/10],[0,0,0,7/15],[0,0,0,6/5]]
```

$$\text{gaussjordan_reducn} \left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 5 & 1 \\ 0 & 0 & 3 & 2 \\ 0 & 0 & -1 & 1 \end{bmatrix} \right) = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & 0 & \frac{17}{20} \\ 0 & 0 & 1 & \frac{1}{5} \\ 0 & 0 & 0 & \frac{9}{10} \\ 0 & 0 & 0 & \frac{7}{15} \\ 0 & 0 & 0 & \frac{6}{5} \end{bmatrix}$$

6.1.5 Application : recherche du noyau grâce à Gauss-Jordan

Soit une application linéaire f de \mathbb{R}^n dans \mathbb{R}^p , de matrice A dans la base canonique de \mathbb{R}^n et de \mathbb{R}^p .

Trouver le noyau de f revient à résoudre $f(X) = 0$ ou encore à résoudre $AX = 0$ pour $X \in \mathbb{R}^n$.

Pour cela, on va utiliser la méthode de Gauss-Jordan en rajoutant des lignes de 0 aux endroits où l'on n'a pas trouvé de pivot de façon à ce que les pivots suivants se trouvent sur la diagonale (et non sur la "fausse diagonale").

Plus précisément :

- quand on a trouvé un pivot pour une colonne, à l'aide de la méthode habituelle de réduction de Gauss-Jordan, on met sur cette colonne :

1 sur la diagonale et

0 de part et d'autre du 1.

- quand on n'a pas trouvé de pivot pour la colonne j , on rajoute une ligne de 0 : la ligne j devient une ligne de 0 et les autres lignes sont décalées.

- on rajoute éventuellement à la fin, des lignes de 0, pour traiter toutes les colonnes de A et avoir une matrice carrée.

- on enlève éventuellement à la fin, des lignes de 0, pour avoir une matrice carrée.

Remarque : on ne change pas le nombre de colonnes.

Ainsi, si la colonne C_j a un 0 sur sa diagonale, si e_j est le j -ième vecteur de la base canonique de \mathbb{R}^n , alors $N_j = C_j - e_j$ est un vecteur du noyau. En effet, on a $A(e_j) = C_j$; si on a $C_j = a_1 e_1 + \dots + a_{j-1} e_{j-1}$, pour $k < j$, et pour $a_k \neq 0$, on a $A(e_k) = e_k$ c'est à dire $A(C_j) = C_j = A(e_j)$, soit $A(C_j - e_j) = A(N_j) = 0$.

Voici le programme correspondant en conservant la variable `j1` afin de faciliter la lisibilité du programme :

```
gaussjordan_noyau(M) := {
```

```

local pivo,jc,jl,k,j,nl,nc,temp,l,a,noyau;
nl:=nrows(M);
nc:=ncols(M);
//on met des 0 sous la diagonale
jc:=0;
jl:=0;
// on traite toutes les colonnes
while (jc<nc and jl<nl) {
  //choix du pivot que l'on veut mettre en M[jl,jc]
  k:=jl;
  while (M[k,jc]==0 and k<nl-1) {k:=k+1;}
  //on ne fait la suite que si on a pivo!=0
  if (M[k,jc]!=0) {
    pivo:=M[k,jc];
    //echange de la ligne jl et de la ligne k
    for (l:=jc;l<nc;l++){
      temp:=M[jl,l];
      M[jl,l] := M[k,l];
      M[k,l] :=temp;
    }
    //fin du choix du pivot qui est M[jl,jc]
    //on met 1 sur la diagonale de la colonne jc
    for (l:=0;l<nc;l++) {
      M[jl,l]:=M[jl,l]/pivo;
    }
    //on met des 0 au dessus de la diagonale de la colonne jc
    for (k:=0;k<jl;k++) {
      a:=M[k,jc];
      for (l:=0;l<nc;l++){
        M[k,l]:=M[k,l]-M[jl,l]*a;
      }
    }
    //on met des 0 sous la diagonale de la colonne jc
    for (k:=jl+1;k<nl;k++) {
      a:=M[k,jc];
      for (l:=jc;l<nc;l++){
        M[k,l]:=M[k,l]-M[jl,l]*a;
      }
    }
  }
  else{
    //on ajoute une ligne de 0 si ce n'est pas le dernier zero
    if (jl<nc-1){
      for (j:=nl;j>jl;j--){
        M[j]:=M[j-1];
      }
      M[jl]:=makelist(0,1,nc);
      nl:=nl+1;
    }
  }
}

```

```

    }
  }
  //ds tous les cas, le numero de colonne et de ligne augmente de 1
  jc:=jc+1;  jl:=jl+1;
  //il faut faire toutes les colonnes
  if (jl==nl and jl<nc) { M[nl]:=makelist(0,1,nc);nl:=nl+1;}
}
noyau:=[];
//on enleve les lignes en trop pour avoir une matrice carree de dim nc
//on retranche la matrice identite
M:=M[0..nc-1]-idn(nc);
for(j:=0;j<nc;j++){
  if (M[j,j]==-1) {noyau:=append(noyau,M[0..nc-1,j]);}
}
return noyau;
}

```

On peut écrire le même programme correspondant en supprimant la variable jl puisque $jl = jc$ (on met jc à la place de jl et on supprime $jl := 0$ et $jl := jl + 1$). On met ce programme dans un éditeur de programmes (que l'on ouvre avec Alt+p), puis on le teste et le valide avec OK.

On tape :

```
gaussjordan_noyau([ [1,2,3], [1,3,6], [2,5,9] ])
```

On obtient :

```
[ [-3,3,-1] ]
```

```
On tape : gaussjordan_noyau([ [1,2,3,4], [1,3,6,6], [2,5,9,10] ])
```

On obtient :

```
[ [-3,3,-1,0], [0,2,0,-1] ]
```

6.2 Résolution d'un système linéaire

6.2.1 Résolution d'un système d'équations linéaires

L'algorithme

On associe à un système d'équations linéaires, une matrice A constituée de la matrice du système augmentée d'une colonne formée par l'opposé du second membre.

Par exemple au système : $[x + y = 4, x - y = 2]$ d'inconnues $[x, y]$ on associe la matrice :

$$A = \begin{bmatrix} 1 & 1 & -4 \\ 1 & -1 & -2 \end{bmatrix}.$$

Puis on réduit avec la méthode de Gauss-Jordan la matrice A pour obtenir une matrice B . Pour chaque ligne de B :

- si il n'y a que des zéro on regarde la ligne suivante,
- si il y a des zéro sauf en dernière position, il n'y a pas de solution,
- dans les autres cas on obtient la valeur de la variable de même indice que le premier élément non nul rencontré sur la ligne,
- les valeurs arbitraires correspondent aux zéros de la diagonale de B et en général

il y a des valeurs non nulles au dessus de ces zéros, c'est pourquoi il faut initialisé la solution au vecteur des variables.

Le programme

Voici le programme de résolution d'un système linéaire :

```
//Veq vecteur des equations
//v vecteur de variables
//renvoie le vecteur solution
linsolv(Veq,v):={
  local A,B,j,k,l,deq,d,res,ll,rep;
  d:=size(v);
  deq:=size(Veq);
  //A est la matrice du systeme +le 2nd membre
  A:=syst2mat(Veq,v);
  //B matrice reduite de Gauss-jordan
  B:=rref(A);
  res:=v;
  //ll ligne l de B
  ll:=makelist(0,0,d);
  for (l:=0; l<deq;l++){
    for (k:=0;k<d+1;k++){
      ll[k]:=B[l][k];
    }
    j:=l;
    while (ll[j]==0 && j<d){
      j:=j+1;
    }
    //si (j==d and ll[d]==0)
    //ll=ligne de zeros on ne fait rien
    if (j==d and ll[d]!=0){
      // pas de sol
      return [];
    }
    else { //la sol res[j] vaut rep/ll[j]
      if (j<d) {
        rep:=-ll[d];
        for (k:=j+1;k<d;k++) {
          rep:=rep-ll[k]*v[k];
        }
        res[j]:=rep/ll[j];
      }
    }
  }
  return res;
}
```

Autre algorithme**6.2.2 Résolution de $MX = b$ donné sous forme matricielle****L'algorithme**

On transforme la résolution de $MX = b$ en $MX - b = AY = 0$ où A est la matrice constituée de la matrice M du système augmentée d'une colonne formée par l'opposé du second membre b .

Y est donc un vecteur du noyau de A ayant comme dernière composante 1.

D'après l'algorithme de recherche du noyau, seul le dernier vecteur a comme dernière composante -1. Si `-ker(A)` renvoie n vecteurs, la solution Y est donc une combinaison arbitraire des $n-1$ premiers vecteurs du noyau plus le n -ième vecteur.

Le programme

Voici le programme de résolution de $AX = b$:

```
//M*res=b res renvoie le vecteur solution
//M:=[[1,1],[1,-1]]; b:=[4,2]
//M:=[[1,1,1],[1,1,1],[1,1,1]]; b:=[0,0,0]
//M:=[[1,2,1],[1,2,5],[1,2,1]]; b:=[0,1,0]
linsolv(M,b):={
  local A,B,N,n,k,d,res;
  d:=ncols(M);
  //A est la matrice du systeme +le 2nd membre
  A:=border(M,-b);
  //N contient une base du noyau
  N:=-ker(A);
  n:=size(N);
  //res a d+1 composante (la derniere=1)
  res:=makelist(0,0,d);
  //C_(k) designe les constantes arbitraires
  for (k:=0;k<n-1;k++){
    res:=res+N[k]*C_(k);
  }
  res:=res+N[n-1];
  res:=suppress(res,d);
  return res;
}
```

6.3 La décomposition LU d'une matrice

C'est l'interprétation matricielle de la méthode de Gauss.

Si A est une matrice carrée d'ordre n , il existe une matrice L triangulaire supérieure, une matrice L triangulaire inférieure, et une matrice P de permutation telles que :

$$P * A = L * U.$$

Supposons tout d'abord que l'on peut faire la méthode de Gauss sans échanger des lignes. Mettre des zéros sous la diagonale de la 1-ère colonne (d'indice 0) de A ,

reviens à multiplier A par la matrice E_0 qui a des 1 sur sa diagonale, comme première colonne :

$[1, -A[1, 0]/A[0, 0], \dots - A[n-1, 0]/A[0, 0]]$ et des zéros ailleurs.

Puis si $A1 = E1 * A$, mettre des zéros sous la diagonale de la 2-ième colonne (d'indice 1) de $A1$, reviens à multiplier $A1$ par la matrice E_1 qui a des 1 sur sa diagonale, comme deuxième colonne :

$[0, 1, -A1[2, 1]/A[1, 1], \dots - A[n-1, 1]/A[1, 1]]$ et des zéros ailleurs.

On continue ainsi jusqu'à mettre des zéros sous la diagonale de la colonne d'indice $n-2$, et à la fin la matrice $U = E_{n-2} * \dots * E_1 * E_0 * A$ est triangulaire supérieure et on a $L = inv(E_{n-2} * \dots * E_1 * E_0)$.

Le calcul de $inv(L)$ est simple car on a :

- $inv(E_0)$ des 1 sur sa diagonale, comme colonne d'indice 0 $[1, +A[1, 0]/A[0, 0], \dots + A[n-1, 0]/A[0, 0]]$ et des zéros ailleurs de même $inv(E_k)$ est obtenue à partir de E_k "en changeant les moins en plus".

- la colonne d'indice k de $inv(E_0) * inv(E_1) \dots inv(E_{n-2})$ est égale à la colonne d'indice k de E_k .

Lorsqu'il y a à faire une permutation de lignes, il faut répercuter cette permutation sur L et sur U : pour faciliter la programmation on va conserver les valeurs de L et de U dans une seule matrice R que l'on séparera à la fin : U sera la partie supérieure et la diagonale de R L sera la partie inférieure de R , avec des 1 sur sa diagonale.

Voici le programme de séparation :

```
splitmat(R) := {
    local L, U, n, k, j;
    n := size(R);
    L := idn(n);
    U := makemat(0, n, n);
    for (k:=0; k<n; k++) {
        for (j:=k; j<n; j++) {
            U[k, j] := R[k, j];
        }
    }
    for (k:=1; k<n; k++) {
        for (j:=0; j<k; j++) {
            L[k, j] := R[k, j];
        }
    }
    return (L, U);
};
```

Le programme ci-dessous, `decomplu(A)`, renvoie la permutation p que l'on a fait sur les lignes, L et U et on a : $P * A = L * U$.

Voici le programme de décomposition LU qui utilise `splitmat` ci-dessus :

```
//A:=[ [5,2,1], [5,2,2], [-4,2,1] ]
//A:=[ [5,2,1], [5,-6,2], [-4,2,1] ]
// utilise splitmat
decomplu(A) := {
```



```

local B,R,L,U,n,j,k,l,temp,p;
n:=size(A);
p:=seq(k,k,0,n-1);
R:=makemat(0,n,n);
B:=A;
l:=0;
//on traite toutes les colonnes
while (l<n-1) {
  if (A[l,l]!=0){
    //pas de permutations
    //on recopie dans R la ligne l de A
    //a partir de la diagonale
    for (j:=1;j<n;j++){R[l,j]:=A[l,j];}
    //on met des zeros sous la diagonale
    //dans la colonne l
    for (k:=l+1;k<n;k++){
for (j:=l+1;j<n;j++){
  A[k,j]:=A[k,j]-A[l,j]*A[k,l]/A[l,l];
  R[k,j]:=A[k,j];
}
R[k,l]:=A[k,l]/A[l,l];
A[k,l]:=0;
    }
    l:=l+1;
  }
  else {
    k:=l;
    while ((k<n-1) and (A[k,l]==0)) {
      k:=k+1;
    }
    //si (A[k,l]==0) A est non inversible,
    //on passe a la colonne suivante
    if (A[k,l]==0) {
l:=l+1;
    }
    else {
      //A[k,l]!=0) on echange la ligne l et k ds A et R
      for (j:=1;j<n;j++){
        temp:=A[k,j];
        A[k,j]:=A[l,j];
        A[l,j]:=temp;
      };
      for (j:=0;j<n;j++){
        temp:=R[k,j];
        R[k,j]:=R[l,j];
        R[l,j]:=temp;
      }
      //on note cet echange dans p

```

```

        temp:=p[k];
        p[k]:=p[l];
        p[l]:=temp;
    }//fin du if (A[k,l]==0)
}//fin du if (A[l,l]!=0)
}//fin du while
L,U:=splitmat(R);
return(p,L,U);
};

```

On tape :

```

A :=[[5,2,1],[5,2,2],[-4,2,1]]
decomplu(A)

```

On obtient :

```

[0,2,1],[1,0,0],[-4/5,1,0],[1,0,1]],
[5,2,1],[0,18/5,9/5],[0,0,1]]

```

On verifie, on tape (car $\text{permu2mat}(p)=P=\text{inv}(P)$) :

```

[[1,0,0],[0,0,1],[0,1,0]]*[[1,0,0],[-4/5,1,0],[1,0,1]]*
[[5,2,1],[0,18/5,9/5],[0,0,1]]

```

On obtient bien $[[5,2,1],[5,2,2],[-4,2,1]]$

On tape :

```

B :=[[5,2,1],[5,-6,2],[-4,2,1]]
decomplu(B)

```

On obtient :

```

[0,1,2],[1,0,0],[1,1,0],[-4/5,-9/20,1]],
[5,2,1],[0,-8,1],[0,0,9/4]]

```

On verifie, on tape :

```

[[1,0,0],[1,1,0],[-4/5,-9/20,1]]*[[5,2,1],[0,-8,1],[0,0,9/4]]

```

On obtient bien $[[5,2,1],[5,-6,2],[-4,2,1]]$

6.4 La décomposition de Cholesky d'une matrice symétrique définie positive

6.4.1 Les méthodes

Lorsque la matrice A est la matrice associée à une forme bilinéaire définie positive, on lui associe la matrice symétrique $B=1/2*(A+\text{tran}(A))$ qui est la matrice de la forme quadratique associée.

Avec ses hypothèses, il existe une matrice triangulaire inférieure unique C telle que $B=C*\text{tran}(C)$.

Pour déterminer C on présente ici deux méthodes :

La méthode utilisant la décomposition LU

Si on décompose B selon la méthode LU, on n'est pas obligé de faire des échanges de lignes car les sous-matrices principales B_k d'ordre k (obtenues en prenant les k premières lignes et colonnes de B) sont des matrices inversibles car ce sont des matrices de formes définies positives.

On a donc :

$p, L, U := \text{decomplu}(B)$ avec $p = [0, 1 \dots n-1]$ si A est d'ordre n . Posons D la matrice diagonale ayant comme diagonale la racine carrée de la diagonale de U .

On a alors $C = L * D$ et $\text{tran}(C) = \text{inv}(D) * U$.

Pour le montrer on utilise le théorème :

Si $B = C * F$ avec B symétrique, C triangulaire inférieure et F triangulaire supérieure de même diagonale que C alors $F = \text{tran}(C)$.

En effet on a :

$B = \text{tran}(B)$ donc $C * F = \text{tran}(C * F) = \text{tran}(F) * \text{tran}(C)$.

On en déduit l'égalité des 2 matrices :

$\text{inv}(\text{tran}(F)) * C = \text{tran}(C) * \text{inv}(F)$

la première est triangulaire inférieure, et la deuxième est triangulaire supérieure donc ces matrices sont diagonales et leur diagonale n'a que des 1 ces 2 matrices sont donc égales à la matrice unité.

La méthode par identification

On peut aussi déterminer C par identification en utilisant l'égalité $B = C * \text{tran}(C)$ et C triangulaire inférieure.

On trouve pour tout entier j compris entre 0 et $n - 1$:

- pour la diagonale :

$$(C[j, j])^2 = B[j, j] - \sum_{k=0}^{j-1} (C[j, k])^2$$

- pour les termes subdiagonaux c'est à dire pour $j + 1 \leq l \leq n - 1$:

$$C[l, j] = 1/C[j, j] * (B[l, j] - \sum_{k=0}^{j-1} C[l, k] * C[j, k])$$

On peut donc calculer les coefficients de C par colonnes, en effet, pour avoir la colonne j :

- on calcule le terme diagonal $C[j, j]$ qui fait intervenir les termes de la ligne j des colonnes précédentes (avec un test vérifiant que le terme $B[j, j] - \sum_{k=0}^{j-1} (C[j, k])^2$ est positif), puis,

- on calcule les termes subdiagonaux $C[l, j]$ pour $j + 1 \leq l \leq n - 1$ qui font intervenir les termes des colonnes précédentes des lignes précédentes. Mais cet algorithme n'est pas très bon car les termes de la matrice C sont obtenus à chaque étape avec une multiplication ou une division de racine carrée : il serait préférable d'introduire les racines carrées qu'à la fin du calcul comme dans la méthode LU.

On va donc calculer une matrice CC (sans utiliser de racines carrées) et une matrice diagonale D (qui aura des racines carrées sur sa diagonale) de tel sorte que $C = CC * D$ (la colonne k de $CC * D$ est la colonne k de CC multiplié par $D[k, k]$).

Par exemple : $C[0, 0] * C[0, 0] = B[0, 0]$ on pose :

$CC[0, 0] = B[0, 0] = a$ et $D[0, 0] = 1/\text{sqrt}(a)$ ainsi

$C[0, 0] = CC[0, 0] / \text{sqrt}(a) = \text{sqrt}(a)$ et on a bien :

$C[0, 0] * C[0, 0] = a = B[0, 0]$

On aura donc $C[l, 0] = CC[l, 0] / \text{sqrt}(a)$ c'est à dire :

$CC[l, 0] = B[l, 0]$ pour $0 \leq l < n$

Puis :

$C[1, 1] * C[1, 1] = B[1, 1] - C[1, 0] * C[1, 0] = B[1, 1] - CC[1, 0] * CC[1, 0] / a = b$

on pose :

$CC[1, 1] = B[1, 1] - CC[1, 0] * CC[1, 0] / a = b$ et

$D[1, 1] = 1/\text{sqrt}(b)$

On aura donc pour $2 \leq l < n$;

$$C[1,1] = CC[1,1] / \text{sqrt}(b) = 1 / \text{sqrt}(b) * (B[1,1] - C[1,0] * C[1,0])$$

c'est à dire pour $2 \leq l < n$:

$$CC[1,1] = B[1,1] - C[1,0] * C[1,0] = B[1,1] - CC[1,0] * CC[1,0] / a$$

etc.....

Les formules de récurrences pour le calcul de CC sont :

- pour la diagonale :

$$CC[j,j] = B[j,j] - \sum_{k=0}^{j-1} (CC[j,k])^2 / CC[k,k]$$

- pour les termes subdiagonaux c'est à dire pour $j+1 \leq l \leq n-1$:

$$CC[1,j] = B[1,j] - \sum_{k=0}^{j-1} CC[1,k] * CC[j,k] / CC[k,k]$$

avec $D[j,j] = 1 / \text{sqrt}(CC[j,j])$.

6.4.2 Le programme de factorisation de Cholesky avec LU

```
//utilise decompilu et splitmat ci-dessus
//A:=[ [1,0,-1],[0,2,4],[-1,4,11]]
//A:=[ [1,1,1],[1,2,4],[1,4,11]]
//A:=[ [1,0,-2],[0,2,6],[0,2,11]]
//A:=[ [1,-2,4],[-2,13,-11],[4,-11,21]]
//A:=[ [-1,-2,4],[-2,13,-11],[4,-11,21]] (pas def pos)
//A:=[ [24,66,13],[66,230,-11],[13,-11,210]]
choles(A):={
  local j,n,p,L,U,D,p0;
  n:=size(A);
  A:=1/2*(A+tran(A));
  (p,L,U):=decompilu(A);
  p0:=makelist(x->x,0,n-1);
  if (p!=p0) {return "pas definie positive ";}
  D:=makemat(0,n,n);
  for (j:=0;j<n;j++) {
    //if (U[j,j]<0) {return "pas def positive";}
    D[j,j]:=sqrt(U[j,j]);
  }
  return normal(L*D);
}
```

6.4.3 Le programme de factorisation de Cholesky par identification

```
//A:=[ [1,0,-1],[0,2,4],[-1,4,11]]
//A:=[ [1,1,1],[1,2,4],[1,4,11]]
//A:=[ [1,0,-2],[0,2,6],[0,2,11]]
//A:=[ [1,-2,4],[-2,13,-11],[4,-11,21]]
//A:=[ [-1,-2,4],[-2,13,-11],[4,-11,21]] (pas def pos)
//A:=[ [24,66,13],[66,230,-11],[13,-11,210]]
cholesi(A):={
  local j,n,l,k,C,c2,s;
  n:=size(A);
  A:=1/2*(A+tran(A));
```

```

C:=makemat(0,n,n);;
for (j:=0;j<n;j++) {
  s:=0;
  for (k:=0;k<j;k++) {
    s:=s+(C[j,k])^2;
  }
  c2:=A[j,j]-s;
  if (c2<=0) {return "pas definie positive "};
  C[j,j]:=normal(sqrt(c2));
  for (l:=j+1;l<n;l++) {
    s:=0;
    for (k:=0;k<j;k++) {
      s:=s+C[l,k]*C[j,k];
    }
    C[l,j]:=normal(1/sqrt(c2)*(A[l,j]-s));
  }
}
return C;
}

```

Ce programme n'est pas très bon car les termes de la matrice C sont obtenus avec une multiplication ou une division de racine carrée...

On se pourra se reporter ci-dessous pour avoir le programme choleski optimisé.

6.4.4 Le programme optimisé de factorisation de Cholesky par identification

```

//A:=[[1,0,-1],[0,2,4],[-1,4,11]]
//A:=[[1,1,1],[1,2,4],[1,4,11]]
//A:=[[1,0,-2],[0,2,6],[0,2,11]]
//A:=[[1,-2,4],[-2,13,-11],[4,-11,21]]
//A:=[[-1,-2,4],[-2,13,-11],[4,-11,21]] (pas def pos)
//A:=[[24,66,13],[66,230,-11],[13,-11,210]]
choleski(A):={
  local j,n,l,k,CC,D,c2,s;
  n:=size(A);
  A:=1/2*(A+tran(A));
  CC:=makemat(0,n,n);
  D:=makemat(0,n,n);
  for (j:=0;j<n;j++) {
    s:=0;
    for (k:=0;k<j;k++) {
      s:=s+(CC[j,k])^2/CC[k,k];
    }
    c2:=normal(A[j,j]-s);
    //if (c2<=0) {return "pas definie positive "};
    CC[j,j]:=c2;
    D[j,j]:=normal(1/sqrt(c2));
  }
}

```

```

        for (l:=j+1;l<n;l++) {
            s:=0;
            for (k:=0;k<j;k++) {
s:=s+CC[l,k]*CC[j,k]/CC[k,k];
            }
            CC[l,j]:=normal(A[l,j]-s);
        }
    }
return normal(CC*D);
}

```

Avec cette méthode, pour obtenir les coefficients diagonaux on utilise la même relation de récurrence que pour les autres coefficients. De plus on peut effectuer directement et facilement la multiplication par D sans avoir à définir D en multipliant les colonnes de CC par la même valeur $1/\sqrt{c}$ avec $c=CC[k,k]$... donc on écrit :

```

choleskii(A):={
    local j,n,l,k,CC,c,s;
    n:=size(A);
    A:=1/2*(A+tran(A));
    CC:=makemat(0,n,n);
    for (j:=0;j<n;j:=j+1) {
        for (l:=j;l<n;l++) {
            s:=0;
            for (k:=0;k<j;k++) {
//if (CC[k,k]<=0) {return "pas definie positive ";}
if (CC[k,k]==0) {return "pas definie";}
s:=s+CC[l,k]*CC[j,k]/CC[k,k];
            }
            CC[l,j]:=A[l,j]-s;
        }
    }
    for (k:=0;k<n;k++) {
        c:=CC[k,k];
        for (j:=k;j<n;j++) {
            CC[j,k]:=normal(CC[j,k]/sqrt(c));
        }
    }
    return CC;
}

```

Avec la traduction pour avoir une fonction interne à xcas :

```

cholesky(_args)={
gen args=(_args+mtran(_args))/2;
matrice &A=*args._VECTptr;
int n=A.size(),j,k,l;
vector<vecteur> C(n,vecteur(n)), D(n,vecteur(n));

```

```

for (j=0; j<n; j++) {
  gen s;
  for (k=0; k<j; k++)
    s=s+pow(C[j][k], 2)/C[k][k];
  gen c2=A[j][j]-s;
  if (is_strictly_positive(-c2))
    setsizeerr("Not a positive definite matrice");
  C[j][j]=c2;
  D[j][j]=normal(1/sqrt(c2));
  for (l=j+1; l<n; l++) {
    s=0;
    for (k=0; k<j; k++)
      s=s+C[l][k]*C[j][k]/C[k][k];
    C[l][j]=A[l][j]-s;
  }
}
matrice Cmat, Dmat;
vector_of_vecteur2matrice(C, Cmat);
vector_of_vecteur2matrice(D, Dmat);
return Cmat*Dmat;
}

```

6.5 Réduction de Hessenberg

6.5.1 La méthode

Une matrice de Hessenberg est une matrice qui a des zéros sous la "deuxième diagonale inférieure".

Soit A une matrice. On va chercher B une matrice de Hessenberg semblable à A . Pour cela on va mettre des zéros sous cette diagonale en utilisant la méthode de Gauss mais en prenant les pivots sur la "deuxième diagonale inférieure" encore appelée "sous-diagonale" : cela revient à multiplier A par $Q = R^{-1}$ et cela permet de conserver les zéros lorsque l'on multiplie à chaque étape le résultat par R pour obtenir une matrice semblable à A . Si on est obligé de faire un échange de lignes (correspondant à la multiplication à droite par E) il faudra faire aussi un échange de colonnes (correspondant à la multiplication à gauche par E).

Par exemple si on a :

$$A := \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Pour mettre des zéros dans la première colonne en dessous de a_{10} , on va multiplier A à gauche par Q et à droite par $R = Q^{-1}$ avec si on suppose $a_{10} \neq 0$ c'est à dire

si on peut choisir comme pivot a_{10} : $Q :=$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -a_{20}/a_{10} & 1 & 0 & 0 \\ 0 & -a_{30}/a_{10} & 0 & 1 & 0 \\ 0 & -a_{40}/a_{10} & 0 & 0 & 1 \end{bmatrix}$$

$$R := \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & +a_{20}/a_{10} & 1 & 0 & 0 \\ 0 & +a_{30}/a_{10} & 0 & 1 & 0 \\ 0 & +a_{40}/a_{10} & 0 & 0 & 1 \end{bmatrix}$$

On tape alors :

$A := [[a_{00}, a_{01}, a_{02}, a_{03}, a_{04}], [a_{10}, a_{11}, a_{12}, a_{13}, a_{14}],$
 $[a_{20}, a_{21}, a_{22}, a_{23}, a_{24}], [a_{30}, a_{31}, a_{32}, a_{33}, a_{34}], [a_{40}, a_{41}, a_{42}, a_{43}, a_{44}]]$
 $Q := [[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, (-a_{20})/a_{10}, 1, 0, 0],$
 $[0, (-a_{30})/a_{10}, 0, 1, 0], [0, (-a_{40})/a_{10}, 0, 0, 1]]$
 $R := [[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, (a_{20})/a_{10}, 1, 0, 0],$
 $[0, (a_{30})/a_{10}, 0, 1, 0], [0, (a_{40})/a_{10}, 0, 0, 1]]$

On obtient la matrice $B1$:

$$B1 = Q * A * R = R^{-1} * A * R = \begin{bmatrix} a_{00} & \dots & a_{02} & a_{03} & a_{04} \\ a_{10} & \dots & a_{12} & a_{13} & a_{14} \\ 0 & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots \end{bmatrix} \text{ où les } \dots \text{ sont}$$

des expressions des coefficients de A .

On va faire cette transformation successivement sur la deuxième colonne de la matrice $B1$ pour obtenir $B2$ etc...

On appellera B toutes les matrices obtenues successivement.

Si on doit échanger les deux lignes k et j pour avoir un pivot, cela revient à multiplier à gauche la matrice B par une matrice E égale à la matrice identité ayant subie l'échange des deux lignes k et j . Il faudra alors, aussi multiplier à droite la matrice B par E c'est à dire échanger les deux colonnes k et j de B

6.5.2 Le programme de réduction de Hessenberg

```
//A est mise sous forme de hessenberg B avec
//P matrice de passage
//p indique si il y a eu une permutation de lignes
//a chaque etape la matrice inv(R)
// met des zeros sous la "sous diagonale"
//B=P^-1AP ex A:=[[5,2,1],[5,2,2],[-4,2,1]]
//A:=[[5,2,1,1],[5,2,2,1],[5,2,2,1],[-4,2,1,1]]
hessenbg(A):={
    local B,R,P,n,j,k,l,temp,p;
    n:=size(A);
    P:=idn(n);
    p:=seq(k,k,0,n-1);
    B:=A;
    l:=1;
    while (l<n) {
```



```

        R:=idn(n);
        if (B[l,l-1]!=0){
            for (k:=l+1;k<n;k++){
R[k,l]:=B[k,l-1]/B[l,l-1];
//on multiplie B a droite par inv(R)
                for (j:=l;j<n;j++){
                    B[k,j]:=B[k,j]-B[l,j]*B[k,l-1]/B[l,l-1];
                }
            B[k,l-1]:=0;
        }
        //on multiplie B et P a gauche par R
        B:=B*R;
        P:=P*R;
        l:=l+1;
    }
    else {
        k:=l;
        while ((k<n-1) and (B[k,l-1]==0)) {
            k:=k+1;
        }

        if (B[k,l-1]==0) {l:=l+1;}
        else{
            //B[k,l]!=0) on echange ligne l et k ds B
            for (j:=l-1;j<n;j++){
                temp:=B[k,j];
                B[k,j]:=B[l,j];
                B[l,j]:=temp;
            };
            //A[k,l]!=0) on echange colonne l et k ds B et P
            for (j:=0;j<n;j++){
                temp:=B[j,k];
                B[j,k]:=B[j,l];
                B[j,l]:=temp;
            };
            for (j:=0;j<n;j++){
                temp:=P[j,k];
                P[j,k]:=P[j,l];
                P[j,l]:=temp;
            };
            temp:=p[k];
            p[k]:=p[l];
            p[l]:=temp;
        }
    }
    return(p,P,B);
};

```

p nous dit les échanges effectués et on a $B = P^{-1} * A * P$.

Attention !! si A est symétrique, cette transformation détruit la symétrie et on n'a donc pas une tridiagonalisation d'une matrice symétrisée par cette méthode.

6.6 Tridiagonalisation des matrices symétriques avec des rotations

On a le théorème : Pour toute matrice symétrique A d'ordre n , il existe une matrice P , produit de $n - 2$ matrices de rotations, telle que $B = {}^t P * A * P$ soit tridiagonale : c'est la réduction de Givens.

6.6.1 Matrice de rotation associée à e_p, e_q

Dans \mathbb{R}^n , on appelle rotation associée à e_p, e_q , une rotation d'angle t du plan dirigé par e_p, e_q où e_k désigne le $k + 1$ -ième vecteur de la base canonique de \mathbb{R}^n (la base canonique est $e_0 = [1, 0, \dots, 0]$, $e_1 = [0, 1, 0, \dots, 0]$ etc...).

Si \mathbb{R}^n est rapporté à la base canonique, à cette rotation est associée une matrice $G(n, p, q, t)$ dont le terme général est :

si $k \notin \{p, q\}$, $g_{k,k} = 1$

$g_{p,p} = g_{q,q} = \cos(t)$

$g_{p,q} = -g_{q,p} = -\sin(t)$

Voici une matrice de rotation associée à e_1, e_3 (deuxième et quatrième vecteur de base) de \mathbb{R}^5 :

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos(t) & 0 & -\sin(t) & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & \sin(t) & 0 & \cos(t) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

et le programme qui construit une telle matrice :

```
rota(n,p,q,t):={
local G;
G:=idn(n);
G[p,p]:=cos(t);
G[q,q]:=cos(t);
G[p,q]:=-sin(t);
G[q,p]:=sin(t);
return G;
}
```

6.6.2 Réduction de Givens

Soit A une matrice symétrique. On va chercher G_1 une matrice de rotation associée à e_1, e_2 pour annuler les coefficients 2, 0 et 0, 2 de ${}^t G_1 * A * G_1$.

Regardons un exemple :

```
G1 := [[1,0,0,0,0], [0,cos(t),-sin(t),0,0], [0,sin(t),cos(t),0,0],
[0,0,0,1,0], [0,0,0,0,1]]
```

$A := \begin{bmatrix} a & b & c & d & e \\ b & f & g & h & j \\ c & g & k & l & m \\ d & h & l & n & o \\ e & j & m & o & r \end{bmatrix}$

On obtient :

$${}^tG1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos(t) & \sin(t) & 0 & 0 \\ 0 & -\sin(t) & \cos(t) & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, A = \begin{bmatrix} a & b & c & d & e \\ b & f & g & h & j \\ c & g & k & l & m \\ d & h & l & n & o \\ e & j & m & o & r \end{bmatrix}$$

On a :

$${}^tG1 * A = \begin{bmatrix} a & b & c & d & e \\ b \cos(t) + c \sin(t) & f \cos(t) + g \sin(t) & .. & .. & .. \\ -b \sin(t) + c \cos(t) & -f \sin(t) + g \cos(t) & .. & .. & .. \\ d & h & l & n & o \\ e & j & m & o & r \end{bmatrix}$$

On choisit t pour que $-b \sin(t) + c \cos(t) = 0$ par exemple :

$\cos(t) = b/\sqrt{(b^2 + c^2)}$ et $\sin(t) = c/\sqrt{(b^2 + c^2)}$.

On a :

$$G1 := \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos(t) & -\sin(t) & 0 & 0 \\ 0 & \sin(t) & \cos(t) & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

et donc puisqu'on a choisit $-b \sin(t) + c \cos(t) = 0$, on a bien annulé les coefficients 2, 0 et 0, 2 de :

$${}^tG1 * A * G1 = \begin{bmatrix} a & b \cos(t) + c \sin(t) & -b \sin(t) + c \cos(t) & d & e \\ b \cos(t) + c \sin(t) & .. & .. & .. & .. \\ -b \sin(t) + c \cos(t) & .. & .. & .. & .. \\ d & .. & .. & n & o \\ e & .. & .. & o & r \end{bmatrix}$$

Puis on annule les coefficients 0, 3 et 3, 0 de la matrice $A1$ obtenue en formant

${}^tG2 * A1 * G2$ avec :

$$G2 := \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos(t) & 0 & -\sin(t) & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & \sin(t) & 0 & \cos(t) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \text{ en choisissant correctement } t \text{ etc...}$$

6.6.3 Le programme de tridiagonalisation par la méthode de Givens

```
//A:= [[1,-1,2,1],[-1,1,-1,1],[2,-1,1,-1],[1,1,-1,-1]]
//tran(R)*A*R=B si tridiagivens(A)=R,B
//pour annuler le terme 2,0 on multiplie A par TG
//TG[1,1]=cos(t)=TG[2,2],TG[1,2]=sin(t)=-TG[2,1],
//avec -sin(t)A[1,0]+cos(t)A[2,0]=0
//TG*A multiplie par tran(TG) annule les termes 2,0 et 0,2
//pour annuler le terme q,p (q>p+1) on multiplie A par TG
//TG[p+1,p+1]=cos(t)=TG[q,q],TG[p+1,q]=sin(t)=-TG[q,p+1],
//avec sin(t)A[p+1,p]=cos(t)A[q,p]
//donc sin(t)=A[q,p]/r et cos(t)=A[p+1,p]/r
//avec r:=sqrt((A[p+1,p])^2+(A[q,p])^2)
```

```

tridiagivens(A):={
  local n,p,q,r,TG,R,c,s;
  n:=size(A);
  R:=idn(n);
  for (p:=0;p<n-2;p++) {
    for (q:=p+2;q<n;q++) {
      r:=sqrt((A[p+1,p])^2+(A[q,p])^2);
      if (r!=0) {
        c:=normal(A[p+1,p]/r);
        s:=normal(A[q,p]/r);
        TG:=idn(n);
        TG[p+1,p+1]:=c;
        TG[q,q]:=c;
        TG[p+1,q]:=s;
        TG[q,p+1]:=-s;
        TG:=normal(TG);
        A:=normal(TG*A);
        A:=normal(A*tran(TG));
        A:=normal(A);
        R:=normal(R*tran(TG));
      }
    }
  }
  return (R,A);
}

```

On tape :

```

A :=[[1,-1,2,1],[-1,1,-1,1],[2,-1,1,-1],[1,1,-1,-1]]
tridiagivens(A)

```

On obtient :

```

[[1,0,0,0],[0,(-(sqrt(6)))/6,(-(sqrt(4838400000)))/201600,
(-(sqrt(2962400000)))/64400],[0,sqrt(6)/3,sqrt(4838400000)/252000,
(-(sqrt(26661600000)))/322000],[0,sqrt(6)/6,
(-(26*sqrt(210)))/420,12*sqrt(35)/420]],
[[1,sqrt(6),0,0],[sqrt(6),1/3,sqrt(275990400000)/266400,0],
[0,sqrt(209026944000)/231840,73/105,8*sqrt(6)/35],[0,0,8*sqrt(6)/35,1

```

On tape :

```

A :=[[1,-1,2,0],[-1,1,-1,1],[2,-1,1,-1],[1,1,-1,-1]]
tridiagivens(A)

```

On obtient :

```

[[1,0,0,0],[0,(-(sqrt(6)))/6,(-(sqrt(4838400000)))/201600,(-(sqrt(296

```

6.7 Tridiagonalisation des matrices symétriques avec Householder

On a le théorème : Pour toute matrice symétrique A d'ordre n , il existe une matrice P , produit de $n - 2$ matrice de Householder (donc $P^{-1} = {}^tP$), telle que $B = {}^tP * A * P$ soit tridiagonale.

Si A n'est pas symétrique, il existe une matrice P , produit de $n - 2$ matrice de Householder, telle que $B = {}^tP * A * P$ soit une matrice de Hessenberg c'est à dire une matrice dont les coefficients sous-tridiagonaux sont nuls : c'est la réduction de Householder.

On va dans cette section écrire un programme qui renverra P et B .

6.7.1 Matrice de Householder associée à v

Soit un hyperplan défini par son vecteur normal v .

On appelle matrice de Householder, la matrice d'une symétrie orthogonale h_v par rapport à cet hyperplan. on a :

$$h_v(u) := u - 2(\text{dot}(v, u)) / \text{norm}(v)^2 * v$$

et la matrice de Householder associée est :

$$H_v = \text{idn}(n) - 2 * \text{tran}(v) * [v] / (v * v).$$

On a $H_v = \text{tran}(H_v) = \text{inv}(H_v)$. Par convention la matrice unité est une matrice de Householder.

On écrit le programme householder qui à un vecteur v renvoie la matrice de Householder associé à v .

```
householder(v) := {
    local n, w, nv;
    n := size(v);
    nv := v * v;
    w := tran(v);
    return normal(idn(n) - 2 * w * [v] / nv);
};
```

6.7.2 Matrice de Householder annulant les dernières composantes de a

Soit $H(v)$ la matrice de Householder associée à v .

Etant donné un vecteur a non nul, il existe deux vecteurs v tels que le vecteur $b = H(v) * \text{tran}(a)$ a ses $n - 1$ dernières composantes nulles ($H(v)$ étant la matrice de Householder associé à v).

Plus précisément, si $e_0 = [1, 0, \dots, 0]$, les vecteurs v sont :

$$v_1 = a + \text{norm}(a) * e_0 \text{ et } v_2 = a - \text{norm}(a) * e_0.$$

Plus généralement, il existe deux vecteurs v tels que le vecteur $b = H(v) * \text{tran}(a)$ a ses composantes à partir de la $(l + 1)$ -ième nulles.

Plus précisément, si $e_l = [0, \dots, 0, 1, 0, \dots, 0]$ et si $a_l = [0, \dots, 0, a[l], \dots, a[n - 1]]$, les vecteurs v sont :

$$v_1 = a_l + \text{norm}(a_l) * e_l \text{ et } v_2 = a_l - \text{norm}(a_l) * e_l.$$

On écrit le programme qui étant donné a et l renvoie si a est nul renvoie la matrice identité et si a est non nul renvoie la matrice de Householder associée à $v = a_l + \text{eps} * \text{norm}(a_l) * e_l$ en choisissant eps égal au signe de a_l si a_l est réelle et $\text{eps} = 1$ sinon.

```
//renvoie une matrice hh= H tel que b:=H*tran(a)
//a ses n-l-1 dernieres comp=0
//ou encore b[l+1]...b[n-1]=0
```

```

make_house(a,l):={
  local n,na,el,v;
  n:=size(a);
  for (k:=0;k<l;k++) a[k]:=0;
  na:=normal(norm(a)^2);
  na:=sqrt(na);
  if (na==0) return idn(n);
  el:=makelist(0,1,n);
  el[l]:=1;
  if (im(a[l])==0 and a[l]<0) v:=normal(a-na*el);
  else v:=normal(a+na*el);
  return householder(v);
};

```

6.7.3 Réduction de Householder

Le programme `tridiaghouse(A)` renvoie H, B tel que $\text{normal}(H^*A^*\text{tran}(H))=B$ (ou encore $\text{normal}(\text{tran}(H)^*B^*H)=A$) avec H est le produit de matrice de householder ($\text{tran}(H)=\text{inv}(H)$) et B est une matrice de hessenberg (ou bien B est une matrice tridiagonale si $\text{tran}(A)=A$).

```

tridiaghouse(A):={
  local n, a, H,Ha;
  B:=A;
  n:=size(A);
  H:=idn(n);
  for (l:=0;l<n-2;l++) {
    a:=col(B,l);
    Ha:=make_house(a,l+1);
    B:=normal(Ha*B*Ha);
    H:=normal(Ha*H);
  }
  //normal(H*A*tran(H))=B et B de hessenberg
  //(ou tridiagonale si tran(A)=A)
  return(H,B);
};

```

On tape :

```

A :=[[3,2,2,2,2],[2,1,2,-1,-1],[2,2,1,-1,1],
[2,-1,-1,3,1],[2,-1,1,1,2]]
H,B :tridiaghouse(A)

```

On obtient :

```

[[1,0,0,0,0],[0,1/-2,1/-2,1/-2,1/-2],[0,5*sqrt(11)/22,
(-(3*sqrt(11)))/22,sqrt(11)/22,(-(3*sqrt(11)))/22],
[0,0,22*sqrt(2)/44,0,(-(22*sqrt(2)))/44],[0,22*sqrt(22)/242,
22*sqrt(22)/484,(-(22*sqrt(22)))/121,22*sqrt(22)/484]],
[[3,-4,0,0,0],[-4,9/4,sqrt(11)/4,0,0],[0,sqrt(11)/4,3/4,
44*sqrt(22)/121,0],[0,0,44*sqrt(22)/121,1/2,286*sqrt(11)/484],
[0,0,0,286*sqrt(11)/484,7/2]]

```

On vérifie et on tape :

```
normal(H*A*tran(H))
```

On tape :

```
A := [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
```

```
H, B := tridiaghouse(A)
```

On obtient :

```
[[1, 0, 0], [0, -(2*sqrt(13))/13, -(3*sqrt(13))/13],
 [0, -(3*sqrt(13))/13, (2*sqrt(13))/13]],
 [[1, -(sqrt(13)), 0], [-(sqrt(13)), 105/13, 8/13], [0, 8/13, (-1)/13]]
```

On vérifie et on tape :

```
normal(H*A*tran(H))
```

6.8 La méthode des trapèzes et du point milieu pour calculer une aire

Dans `xcas`, il existe déjà une fonction qui calcule la valeur approchée d'une intégrale (en accélérant la méthode des trapèzes par l'algorithme de Romberg) qui est : `romberg`

Soit une fonction définie et continue sur l'intervalle $[a ; b]$.

On sait que $\int_a^b f(t)dt$ peut être approchée par l'aire, soit :

- du trapèze de sommets $a, b, b + i * f(b), a + i * f(a)$ soit,
- du rectangle de sommets $a, b, b + i * f((a + b)/2), a + i * f((a + b)/2)$.

La méthode des trapèzes (resp point milieu) consiste à partager l'intervalle $[a ; b]$ en n parties égales et à faire l'approximation de l'intégrale sur chaque sous-intervalle par l'aire des trapèzes (resp rectangles) ainsi définis.

6.8.1 La méthode des trapèzes

On partage $[a ; b]$ en n parties égales et `trapeze` renvoie la somme des aires des n trapèzes déterminés par la courbe représentative de f et la subdivision de $[a ; b]$.

```
trapeze(f, a, b, n) := {
  local s, k;
  s := evalf((f(a) + f(b))/2);
  for (k:=1; k<n; k++) {
    s := s + f(a + k*(b-a)/n);
  }
  return s/n*(b-a);
}
```

On partage $[a ; b]$ en 2^p parties égales et `trapezel` renvoie la liste de la somme des aires des $n = 2^k$ trapèzes déterminés par la courbe représentative de f et la subdivision de $[a ; b]$, liste de longueur $p + 1$, obtenue en faisant varier k de 0 à p .

On pourra ensuite, appliquer à cette liste une accélération de convergence.

On remarquera qu'à chaque étape, on ajoute des "nouveaux" points à la subdivision, et que le calcul utilise la somme s précédente en lui ajoutant la contribution s_1 des "nouveaux" points de la subdivision.

```

trapezel(f,a,b,p):={
local s,n,k,lt,s1,j;
s:=evalf((f(a)+f(b))/2);
n:=1;
lt:=[s*(b-a)];
for (k:=1;k<=p;k++) {
s1:=0;
for (j:=0;j<n;j++) {
s1:=s1+f(a+(2*j+1)*(b-a)/(2*n));
}
s:=s+s1;
n:=2*n;
lt:=concat(lt,s/n*(b-a));
}
return lt;
}

```

On met ce programme dans un éditeur de programmes (que l'on ouvre avec Alt+p), puis on le teste et le valide avec OK.

On tape (on partage $[0;1]$ en $2^6=64$ parties égales) :

```
trapezel(x->x^2+1,0,1,6)
```

On obtient :

```
[1.5,1.375,1.34375,1.3359375,1.333984375,1.33349609375,1.33337402344]
```

On sait que $\int (x^2+1, x, 0, 1) = \frac{4}{3} = 1.333333333$

On tape (on partage $[0;1]$ en $2^6=64$ parties égales) :

```
trapezel(exp,0,1,6)
```

On obtient :

```
[1.85914091423,1.75393109246,1.72722190456,1.72051859216,1.7188411285]
```

On sait que $\int (\exp(x), x, 0, 1) = e-1 = 1.71828182846$

6.8.2 La méthode du point milieu

On partage $[a; b]$ en n parties égales et `ptmilieu` renvoie la somme des aires des n rectangles déterminés par la courbe représentative de f et les milieux des segments de la subdivision de $[a; b]$.

```

ptmilieu(f,a,b,n):={
local s,k;
s:=0.0;
for (k:=0;k<n;k++) {
s:=s+f(a+(b-a)/(2*n)+k*(b-a)/n);
}
return s/n*(b-a);
}

```

On partage $[a; b]$ en 3^p parties égales et `ptmilieul` renvoie la liste de la somme des aires des $n = 3^k$ rectangles déterminés par la courbe représentative de f et les milieux de la subdivision de $[a; b]$, liste de longueur $p + 1$, obtenue en faisant varier k de 0 à p .

On peut ensuite appliquer à cette liste une accélération de convergence.

6.9. ACCÉLÉRATION DE CONVERGENCE : MÉTHODE DE RICHARDSON ET ROMBERG 185

On remarquera qu'à chaque étape, on ajoute des "nouveaux" points à la subdivision, et, que le calcul utilise la somme s précédente en lui ajoutant la contribution s_1 des "nouveaux" points de la subdivision.

```
ptmilieul(f,a,b,p):={
  local s,n,k,lt,s1,j;
  s:=evalf(f((a+b)/2));
  n:=1;
  lt:=[s*(b-a)];
  for (k:=1;k<=p;k++) {
    s1:=0.0;
    for (j:=0;j<n;j++) {
      s1:=s1+f(a+(6*j+1)*(b-a)/(6*n))+f(a+(6*j+5)*(b-a)/(6*n));
    }
    s:=s+s1;
    n:=3*n;
    lt:=concat(lt,s*(b-a)/n);
  }
  return lt;
}
```

On met ce programme dans un éditeur de programmes (que l'on ouvre avec Alt+p), puis on le teste et le valide avec OK.

On tape (on partage $[0;1]$ en $3^4=81$ parties égales) :

```
ptmilieul(x->x^2+1,0,1,4)
```

On obtient :

```
[1.25,1.32407407407,1.33230452675,1.33321902149,1.33332063202]
```

On sait que $\int (x^2+1, x, 0, 1) = \frac{4}{3} = 1.3333333333$

On tape (on partage $[0;1]$ en $3^4=81$ parties égales) :

```
ptmilieul(exp,0,1,4)
```

On obtient :

```
[1.6487212707,1.71035252482,1.7173982568,1.71818362241,1.71827091629]
```

On sait que $\int (\exp(x), x, 0, 1) = e-1 = 1.71828182846$

6.9 Accélération de convergence : méthode de Richardson et Romberg

Hypothèse : soit v une fonction qui admet un développement limité au voisinage de zéro à l'ordre n :

$$v(h) = v(0) + c_1 \cdot h + c_2 \cdot h^2 + \dots + c_n \cdot h^n + O(h^{n+1})$$

On veut accélérer la convergence de v vers $v(0)$ quand h tend vers 0.

6.9.1 La méthode de Richardson

Le principe

La méthode de Richardson consiste à faire disparaître le terme en $c_1 \cdot h$ en faisant par exemple la combinaison $2 \cdot v(h/2) - v(h)$.

On a en effet :

$$v(h) = v(0) + c_1 \cdot h + c_2 \cdot h^2 + \dots + c_n \cdot h^n + O(h^{n+1})$$

$$v\left(\frac{h}{2}\right) = v(0) + c_1 \cdot \frac{h}{2} + c_2 \cdot \frac{h^2}{4} + \dots + c_n \cdot \frac{h^n}{2^n} + O(h^{n+1})$$

Posons :

$$u(h) = 2 \cdot v(h/2) - v(h)$$

$$\text{On a donc } u(h) = v(0) - c_2 \cdot \frac{h^2}{2} + \dots - c_n \cdot h^n \cdot \frac{2^{n-1} - 1}{2^{n-1}} + O(h^{n+1})$$

u tend donc plus vite que v vers $v(0)$ quand h tend vers 0.

On peut continuer et faire subir la même chose à u .

L'algorithme général

On considère $r \in]0; 1[$ (on peut prendre $r = \frac{1}{2}$) et on pose :

$$v_{0,0}(h) = v(h) = v(0) + c_1 \cdot h + c_2 \cdot h^2 + \dots + c_n \cdot h^n + O(h^{n+1})$$

$$v_{1,0}(h) = v(r \cdot h) = v(0) + c_1 \cdot r \cdot h + c_2 \cdot r^2 \cdot h^2 + \dots + c_n \cdot r^n \cdot h^n + O(h^{n+1})$$

$$v_{2,0}(h) = v(r^2 \cdot h) = v(0) + c_1 \cdot r^2 \cdot h + c_2 \cdot r^4 \cdot h^2 + \dots + c_n \cdot r^{2n} \cdot h^n + O(h^{n+1})$$

Soit :

$$v_{1,1}(h) = \frac{1}{1-r} (v(r \cdot h) - r \cdot v(h)) = \frac{1}{1-r} (v_{1,0}(h) - r \cdot v_{0,0}(h))$$

$$v_{1,1}(h) = v(0) + c_2 \cdot r \cdot h^2 + \dots + c_n \cdot r \cdot (r^n - 1)/(r - 1) \cdot h^n + O(h^{n+1})$$

on n'a pas de terme en h dans $v_{1,1}(h)$ et de la même façon en posant :

$$v_{2,1}(h) = v_{1,1}(r \cdot h) = \frac{1}{1-r} (v(r^2 \cdot h) - r \cdot v_{1,0}(h)) = \frac{1}{1-r} (v_{2,0}(h) - r \cdot v_{1,0}(h))$$

$$v_{2,1}(h) = v(0) + c_2 \cdot r^3 \cdot h^2 + \dots + O(h^{n+1})$$

on n'a pas de terme en h dans $v_{2,1}(h)$

On obtient la suite des fonctions $v_{k,1} = v_{1,1}(r^{k-1} \cdot h)$ ($k > 1$) qui n'ont pas de terme en h dans leur développements limités et qui convergent vers $v(0)$.

On peut faire subir à la suite $v_{k,1}$ le même sort qu'à la suite $v_{k,0}$ et obtenir la suite $v_{k,2}$ ($k > 2$) :

$$\text{on pose } v_{2,2}(h) = \frac{1}{1-r^2} (v_{2,1}(h) - r^2 \cdot v_{1,1}(h))$$

et on n'a pas de terme en h^2 dans $v_{2,2}(h)$ etc....

On obtient ainsi les formules de récurrence :

$$v_{k,0}(h) = v(r^k \cdot h)$$

$$v_{k,p}(h) = \frac{1}{1-r^p} (v_{k,p-1}(h) - r^p \cdot v_{k-1,p-1}(h))$$

On a alors :

$$v_{k,p}(h) = v(0) + O(h^{p+1})$$

6.9.2 Application au calcul de $S = \sum_{k=1}^{\infty} \frac{1}{k^2}$

On a :

$$R_p = S - \sum_{k=1}^p \frac{1}{k^2} = \sum_{k=p+1}^{\infty} \frac{1}{k^2}$$

D'après la comparaison avec une intégrale on a :

$$\int_{p+1}^{\infty} \frac{1}{x^2} dx < R_p < \int_p^{\infty} \frac{1}{x^2} dx$$

Donc

$$\frac{1}{p+1} < R_p < \frac{1}{p}$$

6.9. ACCÉLÉRATION DE CONVERGENCE : MÉTHODE DE RICHARDSON ET ROMBERG 187

donc $S_{0,p} = S_p = S - R_p = S - \frac{1}{p} + O(\frac{1}{p^2})$

On forme pour $p = 1..2^{n-1}$:

$S_{1,p} = S_{1p} = 2 * S_{2p} - S_p$ puis,

$S_{2,p} = S_{2p} = (2^2 * S_{12p} - S_{1p}) / (2^2 - 1)$ puis,

$S_{k,p} = (2^k * S_{k-1,2p} - S_{k-1,p}) / (2^k - 1)$

A la main :

$S_{0,1} = 1,$

$S_{0,2} = 5/4 = 1.25,$

$S_{0,3} = 49/36 = 1.36111111,$

$S_{0,4} = 205/144 = 1.4236111111$

$S_{1,1} = 2 * S_{0,2} - S_{0,1} = 3/2 = 1.5$

$S_{1,2} = 2 * S_{0,4} - S_{0,2} = 1.5972222222$

$S_{2,2} = (4 * S_{1,2} - S_{1,1}) / 3 = 1.62962962963$

Dans la liste S on met $1, 1 + 1/4, 1 + 1/4 + 1/9, \dots, 1 + 1/4 + \dots + 1/2^{2n},$

S a 2^n termes d'indices 0 à $2^n - 1,$

puis on forme

$S1 = 1.5, 1 + 1/4 + 2/9 + 2/16, \dots, S[2^n - 1] - S[2^{n-1} - 1]$

(on doit mettre -1 car les indices de S commencent à 0)

$S1$ a 2^{n-1} termes d'indices 0 à $2^{n-1} - 1$

puis on continue avec une suite $S2$ de termes pour $k = 1..2^{n-2}$:

$(4 * S1[2 * k - 1] - S1[k - 1]) / 3$ etc...

On écrit le programme suivant :

```
ridchardson(n) := {
local s0,s1,k,j,st,S,puiss;
s0:=[];
st:=0.0;
for (k:=1;k<=2^n;k++) {
st:=st+1/k^2;
s0:=concat(s0,st);
}
//attention s0=S a 2^n termes d'indices 0 2^n-1
S:=s0;
for (j:=1;j<=n;j++){
s1:=[];
puiss:=2^j;
//j-ieme acceleration s1 a 2^(n-j) termes d'indices 0 2^(n-j)-1
for (k:=1;k<=2^(n-j);k++) {
st:=(puiss*s0[2*k-1]-s0[k-1])/(puiss-1);
s1:=concat(s1,st);
}
s0:=s1;
}
return S[2^n-1],s1[0];
}
```

La première valeur est la somme des 2^n premiers termes, calculée sans accélération, la deuxième valeur a été obtenue après avoir accéléré cette somme n fois.

On tape :

ridchardson(6)

On obtient :

1.62943050141, 1.64493406732

On a du calculer $2^6 = 64$ termes (1 et 8 décimales exactes).

On tape :

ridchardson(8)

On obtient avec plus de décimales :

1.6410354363087, 1.6449340668481 (2 et 12 décimales exactes)

On a du calculer $2^8 = 256$ termes.

On sait que $S = \sum_{k=1}^{\infty} \frac{1}{k^2} = \pi^2/6 \simeq 1.6449340668482$

6.9.3 La méthode de Romberg

C'est l'application de la méthode de Richardson à la formule des trapèzes dans le calcul d'une intégrale.

La formule d'Euler Mac Laurin

La formule à l'ordre 4 On a :

$$\int_0^1 g(t) dt = \frac{1}{2}(g(0)+g(1)) + \text{bernoulli}(2)(g'(0)-g'(1)) + \frac{1}{4!}\text{bernoulli}(4)(g^{(3)}(0) - g^{(3)}(1)) + \int_0^1 P_4(t)g^{(4)}(t)dt$$

où $P_4(t) = \frac{1}{4!}B_4(t)$ avec

$B_n(0) = \text{bernoulli}(n)$ et où B_n est le n -ième polynôme de Bernoulli.

La suite P_n est définie par :

$P_0 = 1$, et pour $k \geq 1$ $P'_k = P_{k-1}$ et $\int_0^1 P_k(u)du = 0$.

Pour la démonstration voir le fascicule Tableur, statistique à la section : Suites adjac-

centes et convergence de $\sum_{k=0}^n \frac{(-1)^k}{2k+1}$.

Théorème pour la formule des trapèzes

Si $f \in \mathcal{C}^{2p+2}([a, b])$, il existe des constantes c_{2k} pour $k = 0..p$ telles que :

$$I = \int_a^b f(x)dx = T(h) + c_1 h^2 + ..c_p h^{2p} + O(h^{2p+2})$$

avec

$$h = \frac{b-a}{n} \text{ et } T(h) = \frac{h}{2}(f(a) + f(b)) + h \sum_{k=1}^{n-1} f(a + k \cdot h)$$

On aura reconnu que $T(h)$ est la formule des trapèzes pour le calcul de $\int_a^b f(x)dx$ avec $h = \frac{b-a}{n}$.

Application de ce théorème pour la formule du point milieu

Soit un intervalle $[a; a+h]$ de longueur h si on applique la formule du point milieu à $I = \int_a^{a+h} f(t)dt$ on a $m(h) = h * f((a+a+h)/2)$ et si on applique la formule des trapèzes aux intervalles $[a; a+h/2]$ et $[a+h/2, a+h]$ on a $t(h/2) = h * (f(a) + f(a+h))/4 + h * f((a+a+h)/2)/2 = t(h)/2 + m(h)/2$ donc quand on coupe $[a; b]$ en n intervalles de longueur h si on note $M(h)$ la formule obtenue pour le point milieu et $T(h)$ celle obtenue pour les trapèzes, on a :

$$M(h) = 2 * T(h/2) - T(h)$$

D'après la formule d'Euler Mac Laurin :

$T(h) = I - c_1 h^2 - c_2 * h^4 - \dots c_p * h^{2p} + O(h^{2p+2})$ et donc

$$M(h) = I + c_1/2 * h^2 + c_2 * h^4 * (2^3 - 1)/2^3 + \dots c_p * h^{2p} * (2^{2p-1} - 1)/2^{2p-1} + O(h^{2p+2})$$

On en déduit que les termes de ces deux développements sont de signes contraires et donc si on fait le même nombre d'accélération de convergence à $T(h)$ et à $M(h)$ on obtiendra un encadrement de I .

L'algorithme de Romberg

On applique l'algorithme de Richardson à $T(h)$ avec $r = \frac{1}{2}$.
On pose :

$$T_{n,0} = T\left(\frac{b-a}{2^n}\right)$$

$$T_{n,1} = \frac{4T_{n,0} - T_{n-1,0}}{3}$$

$$T_{n,k} = \frac{2^{2k}T_{n,k-1} - T_{n-1,k-1}}{2^{2k} - 1}$$

Théorème :

On a :

$$T_{n,p} = I + O\left(\frac{1}{2^{2np}}\right)$$

On partage successivement l'intervalle $[a; b]$ en $1 = 2^0, 2 = 2^1, 4 = 2^2, \dots, 2^n$ et on calcule la formule des trapèzes correspondante : $T_{0,0}, T_{1,0}, \dots, T_{n,0}$ c'est ce que fait le programme `trapezel` fait en 6.8.1 que je recopie ci-dessous.

```
trapezel(f, a, b, n) := {
  local s, puiss2, k, lt, sl, j;
  s := evalf((f(a) + f(b))/2);
  puiss2 := 1;
  lt := [s * (b-a)];
  for (k:=1; k<=n; k++) {
    sl := 0;
    for (j:=0; j<puiss2; j++) {
      sl := sl + f(a + (2*j+1) * (b-a) / (2*puiss2));
    }
    s := s + sl;
    puiss2 := 2*puiss2;
    lt := concat(lt, s * (b-a) / puiss2);
  }
  return lt;
}
```

On va travailler tout d'abord avec deux listes : $l0$ et $l1$ au début $l0 = [T_{0,0}]$ et $l1 = [T_{1,0}]$, on calcule $T_{1,1}$ et $l1 = [T_{1,0}, T_{1,1}]$, puis on n'a plus besoin de $l0$ donc on met $l1$ dans $l0$ et on recommence avec $l1 = [T_{2,0}]$, on calcule $T_{2,1}$ et $T_{2,2}$, et $l1 = [T_{2,0}, T_{2,1}, T_{2,2}]$ puis on met $l1$ dans $l0$ et on recommence avec.....pour enfin avoir $T_{n,0}, T_{n,1}, \dots, T_{n,n}$ dans $l1$.

Les programmes

On applique l'algorithme de Romberg à $T_{k,0} = l[k]$ où $l = \text{trapezel}(f, a, b, n)$.

```
intt_romberg(f,a,b,n):={
local l,l0,l1,puis,k,j;
l:=trapezel(f,a,b,n);
//debut de l'acceleration de Romberg
l0:=[l[0]];
//on fait n accelerations
for (k:=1;k<=n;k++) {
  l1:=[l[k]];
  //calcul des T_{k,j} (j=1..k) dans l1
  for (j:=1;j<=k;j++) {
    puis:=2^(2*j);
    l1[j]:=(puis*l1[j-1]-l0[j-1])/(puis-1);
  }
  l0:=l1;
}
return l1;
}
```

On applique l'algorithme de Romberg à $M_{k,0} = l[k]$ où $l = \text{ptmilieul}(f, a, b, n)$ que je rappelle ci-dessous.

```
ptmilieul(f,a,b,n):={
local s,puiss3,k,lt,s1,j;
s:=evalf(f((a+b)/2));
puiss3:=1;
lt:=[s*(b-a)];
for (k:=1;k<=n;k++) {
  s1:=0.0;
  for (j:=0;j<puiss3;j++) {
    s1:=s1+f(a+(6*j+1)*(b-a)/(6*puiss3))+f(a+(6*j+5)*(b-a)/(6*puiss3));
  }
  s:=s+s1;
  puiss3:=3*puiss3;
  lt:=concat(lt,s*(b-a)/puiss3);
}
return lt;
}
```

On procède comme précédemment en remplaçant les puissances de 2 par des puissances de 3 et le calcul de $T(h)$ par celui de $M(h)$.

```
intm_romberg(f,a,b,n):={
local l,l0,l1,puis,k,j;
l:=milieul(f,a,b,n);
//debut de l'acceleration de Romberg
l0:=[l[0]];
}
```

6.9. ACCÉLÉRATION DE CONVERGENCE : MÉTHODE DE RICHARDSON ET ROMBERG 191

```
//on fait n accelerations
for (k:=1;k<=n;k++) {
    l1:=[l[k]];
    //calcul des M_{k,j} (j=1..k) dans l1
    for (j:=1;j<=k;j++) {
        puis:=3^(2*j);
        l1[j]:=(puis*l1[j-1]-l0[j-1])/(puis-1);
    }
    l0:=l1;
}
return l1;
}
```

On peut raffiner en calculant $\text{puis} = 2^{(2 * j)}$ dans la boucle (en rajoutant `puis :=1 ; avant for (j :=1 ; j<=k ; j++) .. et en remplaçant $\text{puis} := 2^{(2 * j)}$;` par `puis :=puis*4 ;`) et aussi en n'utilisant qu'une seule liste...

On met ces programmes successivement dans un éditeur de programmes (que l'on ouvre avec Alt+p), puis on le teste et le valide avec OK.

On tape (on partage $[0;1]$ en $2^3=8$ parties égales) :

```
intt_romberg(x->x^2+1,0,1,3)
```

On obtient :

```
[1.3359375,1.33333333333,1.33333333333,1.33333333333]
```

On sait que $\text{int}(x^2+1, x, 0, 1) = \frac{4}{3} = 1.3333333333$

On tape (on partage $[0;1]$ en $2^4=16$ parties égales) :

```
intt_romberg(exp,0,1,4)
```

On obtient :

```
1.71884112858,1.71828197405,1.71828182868,1.71828182846,1.71828182846]
```

On sait que $\text{int}(\exp(x), x, 0, 1) = e-1 = 1.71828182846$.

Dans la pratique, l'utilisateur ne connaît pas la valeur de n qui donnera un résultat correct sans faire trop de calculs. C'est pourquoi, on va faire le calcul de la méthode des trapèzes au fur et à mesure et changer le test d'arrêt : on s'arrête quand la différence de deux termes consécutifs est en valeur absolue plus petit que ϵ .

```
intrap_romberg(f,a,b,epsi):={
local l0,l1,puis,puiss2,k,j,s,s1,test;
//initialisation on a 1 intervalle
s:=evalf((f(a)+f(b))/2);
puiss2:=1;
l0:=[s*(b-a)];
k:=1;
test:=1;
while (test>epsi) {
    //calcul de la methode des trapezes avec 2^k intervalles
    s1:=0;
    for (j:=0;j<puiss2;j++) {
        s1:=s1+f(a+(2*j+1)*(b-a)/(2*puiss2));
    }
    s:=s+s1;
    puiss2:=2*puiss2;
}
```

```

l1:=[s*(b-a)/puiss2];
//debut de l'acceleration de Romberg
//calcul des T_{k,j} (j=1..k) dans l1
j:=1;
while ((test>epsi) and (j<=k)) {
    puis:=2^(2*j);
    l1[j]:=(puis*l1[j-1]-l0[j-1])/(puis-1);
    test:=abs(l1[j]-l1[j-1]);
    j:=j+1;
}
l0:=l1;
k:=k+1;
}
return [k-1,j-1,l1[j-1]];
}

```

On renvoie la liste $[p, q, val]$ où $val = T_{q,p}$ (p =le nombre d'accéléérations).

On tape :

```
intrap_romberg(sq,0,1,1e-12)
```

On obtient :

```
[2,2,0.333333333333]
```

(on a donc dû partager $[0;1]$ en $2^2=4$ parties égales et on a fait 2 accéléérations)

On tape :

```
intrap_romberg(exp,0,1,1e-12)
```

On obtient :

```
[5,4,1.71828182846]
```

(on a donc dû partager $[0;1]$ en $2^5=32$ parties égales et on a fait 4 accéléérations)

On peut aussi appliquer l'algorithme de Romberg à $M(h)$ on le même programme en remplaçant les puissances de 2 par des puissances de 3 et le calcul de $T(h)$ par celui de $M(h)$.

```

intmili_romberg(f,a,b,epsi):={
local l0,l1,puis,puiss3,k,j,s,s1,test;
//initialisation on a 1 intervalle
s:=evalf(f((a+b)/2));
puiss3:=1;
l0:=[s*(b-a)];
k:=1;
test:=1;
while (test>epsi) {
    //calcul de la methode du point milieu avec 3^k intervalles
    s1:=0;
    for (j:=0;j<puiss3;j++) {
        s1:=s1+f(a+(6*j+1)*(b-a)/(6*puiss3))+f(a+(6*j+5)*(b-a)/(6*puiss3));
    }
    s:=s+s1;
    puiss3:=3*puiss3;
    l1:=[s*(b-a)/puiss3];
}

```


6.9. ACCÉLÉRATION DE CONVERGENCE : MÉTHODE DE RICHARDSON ET ROMBERG 193

```
//debut de l'acceleration de Romberg
//calcul des T_{k,j} (j=1..k) dans l1
j:=1;
while ((test>epsi) and (j<=k)) {
    puis:=3^(2*j);
    l1[j]:=(puis*l1[j-1]-l0[j-1])/(puis-1);
    test:=abs(l1[j]-l1[j-1]);
    j:=j+1;
}
l0:=l1;
k:=k+1;
}
return [k-1,j-1,l1[j-1]];
}
```

On tape :

```
inmili_romberg(sq,0,1,1e-12)
```

On obtient :

```
[2,2,0.33333333333333]
```

(on a donc dû partager $[0;1]$ en $3^2=9$ parties égales et on a fait 2 accélérations)

On tape :

```
intmili_romberg(exp,0,1,1e-12)
```

On obtient :

```
[4,3,1.71828182846]
```

(on a donc dû partager $[0;1]$ en $3^4=81$ parties égales et on a fait 3 accélérations)

Application au calcul de $\sum_{k=1}^{\infty} f(k)$

On peut aussi reprendre le programme sur les séries $\sum_{k=1}^{\infty} f(k)$ et écrire pour faire n accélérations :

```
serie_romberg(f,n):={
local l,l0,l1,puis,k,j,t,p;
// calcul des sommes s1,s2,s4,s8,...s2^n que l'on met ds l
l:=[f(1)];
p:=1;
for (k:=1;k<=n;k++) {
    t:=0.0
    for (j:=p+1;j<=2*p;j++){
        t:=t+f(j)
    }
    t:=l[k-1]+t;
    l:= concat(l,t);
    p:=2*p;
}
//debut de l'acceleration de Richardson
l0:=l[0];
for (k:=1;k<=n;k++) {
```

```

l1:=[l[k]];
//calcul des S_{k,j} (j=1..k) dans l1
for (j:=1;j<=k;j++) {
    puis:=2^(j);
    l1[j]:=(puis*l1[j-1]-l0[j-1])/(puis-1);
}
l0:=l1;
}
return l1;
}

```

On met ce programme dans un éditeur de programmes (que l'on ouvre avec Alt+p), puis on le teste et le valide avec OK.

On définit f :

On tape :

$f(x) := 1/x^2$

On tape (on calcule $\sum_{k=1}^6 4f(k)$) :

serie_romberg(f,6)

On obtient cette somme après avoir subit 0,1,..6 accélérations :

[1.62943050141,1.64469373999,1.64492898925,1.64493403752,
1.64493409536,1.64493407424,1.64493406732]

Avec le programme richardson on avait :

1.62943050141, 1.64493406732

On tape (on calcule $\sum_{k=1}^2 56f(k)$) :

serie_romberg(f,8)

On obtient cette somme après avoir subit 0,1,..8 accélérations avec plus de décimales :

[1.6410354363087,1.6449188676629,1.6449339873839,1.6449340668192,
1.6449340668791,1.6449340668489,1.6449340668477,1.644934066848,
1.6449340668481]

On sait que $\pi^2/6 = 1.6449340668482$

6.9.4 Deux approximations de l'intégrale

On calcule en même temps l'accélération de convergence pour la méthode des trapèzes et pour la méthode du point milieu.

On remarquera qu'ici on découpe l'intervalle $[a;b]$ en 2 puis en $2^2 \dots 2^k$ morceaux et que le calcul de sm (somme des valeurs de f aux "points milieux") sert dans le calcul du st suivant (somme des valeurs de f aux "points de subdivision" + $(f(a)+f(b))/2$) comme contribution des nouveaux points.

```

inttm_romberg(f,a,b,epsi):={
local lt0,lt1,lm0,lm1,puis,puiss2,k,j,st,sm,s1,test;
//initialisation on a 1 intervalle
st:=evalf((f(a)+f(b))/2);
sm:=evalf(f((a+b)/2));
puiss2:=1;
lt0:=[st*(b-a)];
lm0:=[sm*(b-a)];

```

6.10. LES MÉTHODES NUMÉRIQUES POUR RÉSOUDRE $Y' = F(X, Y)$ 195

```

k:=1;
test:=1;
while (test>epsi) {
    //calcul de la methode des trapezes avec 2^k intervalles
    st:=st+sm;
    //calcul de la methode des milieux avec 2^k intervalles
    puiss2:=2*puiss2;
    s1:=0.0;
    for (j:=0; j<puiss2; j++) {
        s1:=s1+f(a+(2*j+1)*(b-a)/(2*puiss2));
    }

    sm:=s1;
    lm1:=[sm*(b-a)/puiss2];
    lt1:=[st*(b-a)/puiss2];
    //debut de l'acceleration de Romberg
    //calcul des T_{k,j} (j=1..k) dans lt1
    //calcul des M_{k,j} (j=1..k) dans lm1
    j:=1;
    while ((test>epsi) and (j<=k)) {
        puis:=2^(2*j);
        lt1[j]:=(puis*lt1[j-1]-lt0[j-1])/(puis-1);
        lm1[j]:=(puis*lm1[j-1]-lm0[j-1])/(puis-1);
        test:=abs(lt1[j]-lm1[j]);
        j:=j+1;
    }
    lt0:=lt1;
    lm0:=lm1;
    k:=k+1;
}
return [k-1, j-1, lt1[j-1], lm1[j-1]];
}

```

6.10 Les méthodes numériques pour résoudre $y' = f(x, y)$

Dans `xcas`, il existe déjà les fonctions qui tracent les solutions de $y' = f(x, y)$, ce sont : `plotode`, `interactive_plotode` et une fonction `odesolve` qui calcule la valeur numérique en un point d'une solution de $y' = f(x, y)$ et $y(t_0) = y_0$. Soit f une fonction continue de $[a; b] \times \mathbb{R}$ dans \mathbb{R} .

On considère l'équation différentielle :

$$y(t_0) = y_0$$

$$y'(t) = f(t, y(t))$$

Problème de Cauchy Soit U un ouvert de \mathbb{R}^2 et f une application continue de U dans \mathbb{R} . On appelle solution du problème de Cauchy (E) :

$$y(t_0) = y_0$$

$$y'(t) = f(t, y(t))$$

tout couple (I, g) où I est un intervalle contenant t_0 et g est une I -solution de (E) c'est à dire une fonction de classe $C^1(I, \mathbb{R})$ vérifiant $g(t_0) = y_0$ et $g'(t) = f(t, g(t))$ pour $t \in I$.

Théorème de Cauchy-Lipschitz faible Soit f , une fonction continue de $[a; b] \times \mathbb{R}$ dans \mathbb{R} , lipschitzienne par rapport à la seconde variable c'est à dire :

il existe $K > 0$ tel que pour tout $t \in [a; b]$ et pour tout $(y_1, y_2) \in \mathbb{R}^2$, $|f(t, y_1) - f(t, y_2)| \leq K|y_1 - y_2|$.

Alors, quel que soit $(t_0, y_0) \in [a; b] \times \mathbb{R}$, il existe une $[a; b]$ -solution unique au problème de Cauchy (E) que l'on appellera y .

6.10.1 La méthode d'Euler

La méthode d'Euler consiste à approcher la solution de cette équation différentielle au voisinage de t_0 par sa tangente en ce point. Cette tangente a comme pente $y'(t_0) = f(t_0, y(t_0)) = f(t_0, y_0)$.

On a donc pour t proche de t_0 par exemple pour $t \in [t_0 - h; t_0 + h]$:

$y(t) \simeq y_0 + f(t_0, y_0) \cdot (t - t_0)$: on dit que h est le pas de l'approximation.

En principe plus h est petit, meilleure est l'approximation.

Soit h un pas, on pose $t_1 = t_0 + h$, et $y_1 = y_0 + f(t_0, y_0) \cdot (t_1 - t_0) = y_0 + f(t_0, y_0) \cdot h$ on a l'approximation :

$y(t_1) = y(t_0 + h) \simeq y_1$

et on réitère cette approximation, donc si $t_2 = t_1 + h$:

$y(t_2) = y(t_1 + h) \simeq y_1 + f(t_1, y_1) \cdot (t_2 - t_1) \simeq y_1 + f(t_1, y_1) \cdot h$.

On écrit donc la fonction `euler_f`, qui réalise une seule étape de cette méthode et qui permet de passer d'un point d'abscisse t_0 au point d'abscisse $t_0 + h$, ces points étant situés sur le graphe de la solution approchée par cette méthode.

```
euler_f(f, t0, y0, h) := {
  local t1, y1;
  t1 := t0 + h;
  y1 := y0 + h * f(t0, y0);
  return (t1, y1);
}
```

Pour tracer le graphe de la solution approchée sur $[t_0; t_1]$, on écrit :

```
segment(point(t0, y0), point(euler_f(f, t0, y0, h)))
```

Pour trouver une solution approchée sur $[a; b]$ de l'équation différentielle :

$y'(t) = f(t, y(t))$, $y(t_0) = y_0$, avec $t_0 \in [a; b]$,

il suffit de partager $[a; b]$ en parties égales de longueur h et d'appliquer plusieurs fois la fonction `euler_f` avec le pas h puis avec le pas $-h$.

Voici le programme qui trace la solution sur $[a; b]$ dans l'écran `DispG` et qui utilise la fonction `euler_f`.

Les derniers segments servent à s'arrêter exactement en a et en b .

```
trace_sol(f, t0, y0, h, a, b) := {
  local t1, y1, td0, yd0, l1, j;
  td0 := t0;
```

6.10. LES MÉTHODES NUMÉRIQUES POUR RÉSOUDRE $Y' = F(X, Y)$ 197

```

yd0:=y0;
h:=abs(h);
while (t0<b-h){
  l1:=euler_f(f,t0,y0,h);
  t1:=l1[0];
  y1:=l1[1];
  segment(t0+i*y0,t1+i*y1);
  t0:= t1;
  y0:=y1;
}
segment(t0+i*y0,b+i*(y0+(b-t0)*f(t0,y0)));
//on trace avec -h
t0:=td0;
y0:=yd0;
while ( t0>a+h){
  l1:=euler_f(f,t0,y0,-h);
  t1:=l1[0];
  y1:=l1[1];
  segment(t0+i*y0,t1+i*y1);
  t0:= t1;
  y0:=y1;
}
segment(t0+i*y0,a+i*(y0+(t0-a)*f(t0,y0)));
}

```

ou encore pour avoir le dessin dans l'écran graphique obtenu comme réponse, on écrit une fonction qui renvoie la séquence des segments à dessiner :

```

trace_euler(f,t0,y0,h,a,b):={
local td0,yd0,l1,j,ls;
td0:=t0;
yd0:=y0;
h:=abs(h);
ls:=[];
while (t0<b-h){
  l1:=euler_f(f,t0,y0,h);
  ls:=ls,segment(t0+i*y0,point(l1));
  (t0,y0):=l1;
}
ls:=ls,segment(t0+i*y0,point(euler_f(f,t0,y0,b-t0)));
//on trace avec -h en partant de td0 et de yd0 (les valeurs du debut)
t0:=td0;
y0:=yd0;
while ( t0>a+h){
  l1:=euler_f(f,t0,y0,-h);
  ls:=ls,segment(t0+i*y0,point(l1));
  (t0,y0):=l1;
}
}

```

```

ls:=ls,segment(t0+i*y0,point(euler_f(f,t0,y0,a-t0)));
return ls;
}

```

6.10.2 La méthode du point milieu

La méthode du point milieu consiste à approcher, au voisinage de t_0 , la solution de l'équation différentielle $y'(t) = f(t, y(t))$, $y(t_0) = y_0$ $t_0 \in [a; b]$, par une parallèle à la tangente au "point milieu de la solution obtenue par la méthode d'Euler" :

Le "point milieu de la solution obtenue par la méthode d'Euler" est le point de coordonnées :

$$(t_0 + h/2, y_0 + h/2 f(t_0, y_0))$$

et sa tangente a donc comme pente :

$$\alpha = f(t_0 + h/2, y_0 + h/2 f(t_0, y_0))$$

La méthode du point milieu fait passer du point (t_0, y_0) au point (t_1, y_1) avec :

$$t_1 = t_0 + h \text{ et } y_1 = y_0 + h * \alpha = y_0 + h * f(t_0 + h/2, y_0 + h/2 * f(t_0, y_0)).$$

On remarquera que :

$$\text{euler_f}(f, t_0, y_0, h/2) = (t_0 + h/2, y_0 + h/2 * f(t_0, y_0))$$

On va donc écrire la fonction :

$$\text{ptmilieu_f}(f, t_0, y_0, h) = (t_0 + h, y_0 + h * f(t_0 + h/2, y_0 + h/2 * f(t_0, y_0)))$$

qui réalise une seule étape de cette méthode et qui permet de passer d'un point d'abscisse t_0 au point d'abscisse $t_0 + h$, ces points étant situés sur le graphe de la solution approchée par cette méthode.

```

ptmilieu_f(f,t0,y0,h):={
local t1,y1;
t1:=t0+h;
y1:=y0+h*f(t0+h/2,y0+h/2*f(t0,y0));
return (t1,y1);
}

```

Il reste à écrire une fonction qui renvoie la séquence des segments obtenus par cette méthode, pour avoir le dessin dans l'écran graphique obtenu comme réponse.

On peut écrire la même fonction que précédemment en remplaçant simplement tous les appels à `euler_f` par `ptmilieu_f`.

Mais on va plutôt rajouter un paramètre supplémentaire `methode` qui sera soit la fonction `euler_f` soit la fonction `ptmilieu_f`. On écrit :

```

trace_methode(methode,f,t0,y0,h,a,b)
où methode qui est une fonction des variables f,t0,y0,h

```

```

trace_methode(methode,f,t0,y0,h,a,b):={
local td0,yd0,l1,j,ls;
td0:=t0;
yd0:=y0;
h:=abs(h);
ls:=[];
while (t0<b-h){
l1:=methode(f,t0,y0,h);

```

6.10. LES MÉTHODES NUMÉRIQUES POUR RÉSOUDRE $Y' = F(X, Y)$ 199

```

ls:=ls, segment(t0+i*y0, point(l1));
(t0, y0) := l1;
}
ls:=ls, segment(t0+i*y0, point(methode(f, t0, y0, b-t0)));
//on trace avec -h en partant de td0 et de yd0 (les valeurs du debut)
t0:=td0;
y0:=yd0;
while ( t0>a+h){
    l1:=methode(f, t0, y0, -h);
    ls:=ls, segment(t0+i*y0, point(l1));
    (t0, y0) := l1;
}
ls:=ls, segment(t0+i*y0, point(methode(f, t0, y0, a-t0)));
return ls;
}

```

6.10.3 La méthode de Heun

La méthode de Heun consiste à approcher, au voisinage de t_0 , la solution de l'équation différentielle $y'(t) = f(t, y(t))$, $y(t_0) = y_0$ $t_0 \in [a; b]$, par une parallèle à la droite de pente $1/2 * (f(t_0, y_0) + f(t_0 + h, y_0 + h * f(t_0, y_0)))$ c'est à dire, par une pente égale à la moyenne des pentes des tangentes :

- de la solution au point (t_0, y_0) (la pente de la tangente en ce point vaut $f(t_0, y_0)$) et

- de la solution obtenue par la méthode d'Euler au point d'abscisse $t_0 + h$ (point de coordonnées $(t_0 + h, y_0 + h * f(t_0, y_0))$ et la pente de la tangente en ce point vaut $f(t_0 + h, y_0 + h * f(t_0, y_0))$).

Donc, la méthode de Heun fait passer du point (t_0, y_0) au point (t_1, y_1) avec :

$t_1 = t_0 + h$ et $y_1 = y_0 + h/2 * (f(t_0, y_0) + f(t_0 + h, y_0 + h * f(t_0, y_0)))$.

On remarquera que :

$euler_f(f, t_0, y_0, h) = (t_0 + h, y_0 + h * f(t_0, y_0))$

On va donc écrire la fonction :

$heun_f(f, t_0, y_0, h) = (t_0 + h, y_0 + h/2 * (f(t_0, y_0) + f(t_0 + h, y_0 + h * f(t_0, y_0))))$

qui réalise une seule étape de cette méthode et qui permet de passer d'un point d'abscisse t_0 au point d'abscisse $t_0 + h$, ces points étant situés sur le graphe de la solution approchée par cette méthode.

```

heun_f(f, t0, y0, h) := {
    local t1, y1;
    t1:=t0+h;
    y1:=y0+h/2*(f(t0, y0)+f(t0+h, y0+h*f(t0, y0)));
    return (t1, y1);
}

```

Il reste à écrire une fonction qui renvoie la séquence des segments obtenus par cette méthode, pour avoir le dessin dans l'écran graphique obtenu comme réponse.

On peut écrit la même fonction que précédemment en remplaçant simplement tous les appels à `euler_f` par `heun_f` ou encore utiliser la fonction `trace_methode` avec comme valeur de `methode` le fonction `heun_f`.

On valide les différentes fonctions :

`euler_f`, `ptmilieu_f`, `heun_f` et `trace_methode`.

On tape par exemple pour avoir le tracé de la solution sur $[-1;1]$ de :

$y' = y$ vérifiant $y(0) = 1$ par les 3 méthodes avec un pas de 0.1 :

`f(t,y) := y` `trace_methode(euler_f,f,0,1,0.1,-1,1)` et

`trace_methode(ptmilieu_f,f,0,1,0.1,-1,1)` ou

`trace_methode(heun_f,f,0,1,0.1,-1,1)` ou

Chapitre 7

Les quadriques

7.1 Équation d'une quadrique

Une forme quadratique de 4 variables X, Y, Z, T peut s'interpréter dans l'espace projectif $Oxyz$ comme le premier membre de l'équation d'une quadrique.

Par exemple soit :

$$q(x, y, z) := 4x^2 + y^2 + z^2 - 4xy + 4xz - 2yz + 8x - 4y + 4z + 2$$

$q(x, y, z) = 0$ est l'équation d'une quadrique.

On associe à q la forme quadratique Q :

$$Q(x, y, z, t) := \text{normal}(t^2 \cdot \text{normal}(q(x/t, y/t, z/t)))$$

et la matrice carrée A d'ordre 4 :

$$A := q2a(Q(x, y, z, t), [x, y, z, t])$$

On obtient :

$$A := [[4, -2, 2, 4], [-2, 1, -1, -2], [2, -1, 1, 2], [4, -2, 2, 2]]$$

on retrouve la forme Q à partir de A si :

$$v := [x, y, z, t]$$

la forme quadratique est :

$$Q(\text{op}(v)) := \text{normal}(v^* A \cdot \text{tran}(v)) [0]) \text{ ou plus simplement :}$$

$$Q(\text{op}(v)) := \text{normal}(v^* A \cdot v)$$

on retrouve la forme Q à partir de A si :

$$v1 := [x, y, z, 1]$$

$$q(\text{op}(v1)) := \text{normal}(v1^* A \cdot v1)$$

et $Q(v) = 0$ représente une quadrique Q de l'espace projectif.

Les points doubles de Q vérifie $A \cdot \text{tran}(v) = [0, 0, 0, 0]$ ie $\text{tran}(v)$ est un élément du noyau de A .

Discussion selon le rang r de A : $r := \text{rank}(A)$

- Si $r=4$, Q est une quadrique propre ou non dégénérée sans point double.
- Si $r=3$, Q est un cône et a un point double S qui est le sommet du cône.
- Si $r=2$, q est le produit de 2 formes linéaires indépendantes et Q est formée de 2 plans distincts et secants selon la droite lieu des points doubles de Q .
- Si $r=1$, q est le carré d'une forme linéaire non nulle et Q est un plan de points doubles.

7.2 Équation réduite d'une quadrique

Pour réaliser la réduction de l'équation d'une quadrique, on pose :

$B := A[0..2, 0..2]$;

$C := A[0..2, 3]$;

$d := A[3, 3]$;

Les directions propres de B sont les directions principales de la quadrique Q .

Comme une matrice symétrique réelle est diagonalisable dans un repère orthonormé, il existe un repère orthonormé (O, u, v, w) dans lequel la quadrique a comme équation :

$s_1 * x_1^2 + s_2 * y_1^2 + s_3 * z_1^2 + 2c_1 * x_1 + 2 * c_2 * y_1 + 2 * c_3 * z_1 + d_1 = 0$
où s_1, s_2, s_3 sont les valeurs propres de B et u, v, w sont les vecteurs propres associés, choisis normés et orthogonaux.

Remarques

- Si les 3 valeurs propres de B sont différentes : les vecteurs propres sont orthogonaux et il suffira de les normer.
- Si il y a une valeur propre double non nulle il faudra rendre les vecteurs propres orthogonaux et les normer.
- Si il y a une valeur propre double nulle il faut rendre les vecteurs propres orthogonaux et les normer.

Voici les différents cas :

- 1ier cas $s_1 * s_2 * s_3 \neq 0$ supposons que $\text{signe}(s_1) = \text{signe}(s_2)$

l'équation s'écrit :

$$s_1 * (x_1 + c_1/s_1)^2 + s_2 * (y_1 + c_2/s_2)^2 + s_3 * (z_1 + c_3/s_3)^2 + d_2 = 0$$

en faisant un changement d'origine O_1 avec :

$$OO_1 = -c_1/s_1 * u - c_2/s_2 * v - c_3/s_3 * w$$

l'équation s'écrit :

$$s_1 * X^2 + s_2 * Y^2 + s_3 * Z^2 + d_2 = 0$$

quitte à multiplier par -1 on obtient une équation de la forme :

$$X^2/a^2 + Y^2/b^2 + s * Z^2/c^2 + d_3 = 0$$

avec $s = 1$ ou $s = -1$ et $d_3 = 1$ ou $d_3 = -1$ ou $d_3 = 0$.

La quadrique est donc de révolution et est engendrée par la rotation autour de l'axe des Z de la conique d'équation $Y = 0, X^2/a^2 + s * Z^2/c^2 + d_3 = 0$.

On a donc :

- $s = 1, d_3 = 1$ on a un ellipsoïde imaginaire,
- $s = 1, d_3 = 0$ on a un cône imaginaire,
- $s = 1, d_3 = -1$ on a un ellipsoïde réelle,
- $s = -1, d_3 = 1$ on a un hyperboloïde à 2 nappes, engendré par la rotation d'une hyperbole autour de son axe transverse (c'est l'axe focal, celui qui coupe l'hyperbole),
- $s = -1, d_3 = 0$ on a un cône réel,
- $s = -1, d_3 = -1$ on a un hyperboloïde à 1 nappe, engendré par la rotation d'une hyperbole autour de son axe non transverse (celui qui ne coupe pas l'hyperbole),

O_1 est centre de symétrie et les plans (resp les axes) de coordonnées du repère (O_1, u, v, w) sont des plans (resp des axes) de symétrie pour Q .

- 2ième cas $s_1 * s_2 = 0$ et $s_3 \neq 0$

l'équation s'écrit :

- $s1 * (x1 + c1/s1)^2 + s2 * (y1 + c2/s2)^2 + 2 * c3 * z1 + d2 = 0$
 – Si $c3 \neq 0$ l'équation s'écrit :
 $s1 * (x1 + c1/s1)^2 + s2 * (y1 + c2/s2)^2 + 2 * c3 * (z1 + d2/(2 * c3)) = 0$
 en faisant un changement d'origine $O1$ avec :
 $OO1 = -c1/s1 * u - c2/s2 * v - d2/(2 * c3) * w$
 selon le signe de $s1 * s2$ l'équation s'écrit :
 si $s1 * s2 > 0$ l'équation s'écrit :
 $X^2/a^2 + Y^2/b^2 - 2 * s * Z = 0$ avec $s = 1$ ou $s = -1$
 on a un paraboloïde elliptique
 si $s1 * s2 < 0$
 l'équation s'écrit : $X^2/a^2 - Y^2/b^2 - 2 * s * Z = 0$ avec $s = 1$ ou $s = -1$
 on a un paraboloïde hyperbolique
 – Si $c3 = 0$ l'équation s'écrit :
 $s1 * (x1 + c1/s1)^2 + s2 * (y1 + c2/s2)^2 + d2 = 0$
 en faisant un changement d'origine $O1$ avec :
 $OO1 = -c1/s1 * u - c2/s2 * v$ on obtient
 $s1 * X^2 + s2 * Y^2 + d2 = 0$
 La quadrique Q est soit, un cylindre de génératrices parallèles à w , soit, formée de 2 plans parallèles :
 si $s1 > 0, s2 > 0, d2 > 0$ on a un cylindre elliptique imaginaire de génératrices parallèles à w ,
 si $s1 > 0, s2 > 0, d2 < 0$ on a un cylindre elliptique réel de génératrices parallèles à w ,
 si $s1 > 0, s2 < 0, d2 < 0$ on a un cylindre hyperbolique de génératrices parallèles à w ,
 si $s1 > 0, s2 > 0, d2 = 0$ on a 2 plans imaginaires conjugués non parallèles qui se coupent selon la droite réelle $X = 0, Y = 0$ i.e. selon un cylindre réel ayant comme base un point et de génératrices parallèles à w .
 si $s1 > 0, s2 < 0, d2 = 0$ on a 2 plans réels non parallèles.
 – 3ième cas $s1 = 0, s2 = 0$ et $s3 = 0$
 l'équation s'écrit :
 $s1 * (x1 + c1/s1)^2 + 2 * c2 * y1 + 2 * c3 * z1 + d2 = 0$
 Si $c2 \neq 0$ ou $c3 \neq 0$:
 soit $v1$ le vecteur unitaire de la droite du plan $x1 = 0$ définie par :
 $c2 * y1 + c3 * z1 = 0, v1 = [0, c3, -c2]/\sqrt{c2^2 + c3^2}$ et soit $w1$ le vecteur unitaire de la droite du plan $x1 = 0$ définies par :
 $c3 * y1 - c2 * z1 = 0, w1 = [0, c2, c3]/\sqrt{c2^2 + c3^2}$
 $Y * v1 + Z * w1 = [0, Y * c3 + Z * c2, -Y * c2 + Z * c3]/\sqrt{c2^2 + c3^2}$
 $y1 = (Y * c3 + Z * c2)/\sqrt{c2^2 + c3^2}$
 $z1 = (-Y * c2 + Z * c3)/\sqrt{c2^2 + c3^2}$ donc
 $c2 * y1 + c3 * z1 = Z * \sqrt{c2^2 + c3^2}$
 Si on ne norme pas $v1$ et $w1$ on a $v1 = [0, c3, -c2], w1 = [0, c2, c3],$
 $Y * v1 + Z * w1 = [0, Y * c3 + Z * c2, -Y * c2 + Z * c3], y1 = (Y * c3 + Z * c2),$
 $z1 = (-Y * c2 + Z * c3)$ et $c2 * y1 + c3 * z1 = Z$ donc le nouveau $c3 = 1$.
 Ainsi, $2 * (c2 * y1 + c3 * z1) = 2 * Z * \sqrt{c2^2 + c3^2}$
 en faisant un changement d'origine $O1$ avec :
 $OO1 = -c1/s1 * u$
 Dans le repère $(O1, u, v1, w1)$ l'équation s'écrit :

$s1 * X^2 + 2 * c4 * Z + d2 = 0$ avec $c4 = \sqrt{c2^2 + c3^2} \neq 0$

en faisant un changement d'origine $O2$ avec :

$OO2 = -d2/(2 * c4) * w1$

l'équation s'écrit :

$s1 * X^2 + 2/c4 * Z = 0$

Q est un cylindre parabolique dont les génératrices sont parallèles à $u1$.

Si $c2 = 0$ et $c3 = 0$ l'équation s'écrit :

$s1 * (x1 + c1/s1)^2 + d2 = 0$

on a alors deux plans parallèles imaginaires conjugués ($s1 * d2 > 0$) deux plans parallèles réels ($s1 * d2 < 0$) ou deux plans confondus ($d2 = 0$).

Pour faciliter l'étude des différents cas le programme `jortho` va renvoyer 6 valeurs :

les 3 valeurs propres (en regroupant les valeurs propres égales et non nulles au début et en mettant les valeurs propres nulles à la fin) et les 3 vecteurs propres seront normés et orthogonaux sauf pour les vecteurs propres associés à 0.

Soit les trois valeurs propres sont : $s1, s2, s3$ et les trois vecteurs propres associés sont : u, v, w .

On a soit les zéros déjà à la fin et les valeurs égales non nulles déjà regroupées au début, soit on peut avoir 8 cas ($a \neq 0, b \neq 0, a \neq b$) :

$(0, a, b)$ ou $(0, a, a)$ ou $(0, a, 0)$ ou (b, a, a) ou (a, b, a) ou $(0, 0, a)$ ou $(a, 0, b)$ ou $(a, 0, a)$.

Pour les quatre premiers cas : Si ($s1 == 0$ ou $s2 == s3$) et $s2 \neq 0$) on renvoie $s2, s3, s1$ et v, w, u .

Ainsi on a maintenant :

$(a, b, 0)$ ou $(a, a, 0)$ ou $(a, 0, 0)$ ou (a, a, b) ou (a, b, a) ou $(0, 0, a)$ ou $(a, 0, b)$ ou $(a, 0, a)$.

Pour les quatre derniers cas, on a :

($s2 == 0$ ou $s1 == s3$) et $s3 \neq 0$) on renvoie $s3, s1, s2$ et w, u, v .

Il faut rendre u, v orthogonaux lorsque $s1 == s2$ si $v * u \neq 0$

$v1 := -(v * u) * u + (u * u) * v$ ainsi $v1 * u = 0$ il faut rendre v, w orthogonaux lorsque $s3 == s2 == 0$ si $v * w \neq 0$

puis on norme u, v, w .

Voici les programmes :

`jortho` renvoie les valeurs propres et les vecteurs propres orthonormés.

`quadrique` renvoie la nouvelle origine la matrice de passage et la forme réduite ainsi que la vérification.

```
jortho(A) := {
local P, J, u, v, w, s1, s2, s3, a, b;
(P, J) := jordan(A);
u := P[0..2, 0];
v := P[0..2, 1];
w := P[0..2, 2];
s1 := J[0, 0];
s2 := J[1, 1];
s3 := J[2, 2];
if ((s1 == 0 || s2 == s3) && s2 != 0) {
b := u;
```

```

u:=v;
v:=w;
w:=b;
a:=s1;
s1:=s2;
s2:=s3;
s3:=a;
}
if ((s2==0 || s1==s3) && s3!=0) {
b:=w;
w:=v;
v:=u;
u:=b;
a:=s3;
s3:=s2;
s2:=s1;
s1:=a;
}
//si s1==s2 et si v*u!=0
a:=normal(u*u);
b:=normal(v*u);
if (b!=0) {
v:=-b*u+a*v;
}
//on norme u
u:=u/sqrt(a);
//si s3==s2==0 et si v*w!=0
a:=normal(v*v);
b:=normal(v*w);
if (b!=0) {
w:=-b*v+a*w;
}
//on norme w et v
w:=w/sqrt(normal(w*w));
v:=v/sqrt(a);
return (s1,s2,s3,u,v,w);
};

```

Le programme `Quadrique` renvoie le repère (la nouvelle origine et la matrice de passage) et l'équation réduite dans ce repère :

```

//q0(x,y,z):=4*x^2+y^2+z^2-4*x*y+4*x*z-2*y*z+8*x-4*y+4*z+2
//          (s2=s3=0, c2=c3=0)
//q1(x,y,z):=x^2+3*y^2-3*z^2-8*y*z+2*z*x-4*x*y-1
//          (s3=0, c3=0)
//q2(x,y,z):=5*x^2+y^2+z^2-2*x*y+2*x*z-6*y*z+2*x+4*y-6*z+1
//          (s1,s2,s3!=0)
//Quadrique(q2)=[(-1)/2,-1,1/2],[[0,2/(sqrt(6)),-(1/(sqrt(3)))],
//[1/(sqrt(2)),-(1/(sqrt(6)),-(1/(sqrt(3)))],

```

```

// [1/(sqrt(2)), 1/(sqrt(6)), 1/(sqrt(3))], -2*X^2+6*Y^2+3*Z^2-3
// q3(x,y,z) := 2*x^2+2*y^2+z^2+2*x*z-2*y*z+4*x-2*y-z+3
// (s3=0, c3!=0)
// q4(x,y,z) := 4*x^2+y^2+z^2-4*x*y+4*x*z-2*y*z+8*x-6*y+4*z+2
// ((s2=s3=0, c2!=0 | c3!=0))
// q5(x,y,z) := x^2+2*y^2-z^2-4*x*y+4*x*z+4*y*z+6*x-4*y+2*z+1
// (root of)
// q6(x,y,z) := (x+y)*(y-z)+3*x-5*y
// (s3=0, c3!=0)
// q7(x,y,z) := 7*x^2+4*y^2+4*z^2+4*x*y-4*x*z-2*y*z-4*x+5*y+4*z-18
// (s1,s2,s3!=0)
// q8(x,y,z) := -x^2-3*z^2-4*x*y+4*x*z+4*y*z+6*x-4*y+2*z+1
// q9(x,y,z) := x^2-3*x*y+y^2+x-y-z^2/2+sqrt(10)/5*z
// q est une fct de 3 variables de degre 2
// Quadrique renvoie la nouvelle origine la matrice de passage et
// la forme reduite avec le vecteur de variables ainsi que
// la verification par ex pour q4
// B:=[ [4,-2,2], [-2,1,-1], [2,-1,1] ]; C:=[4,-3,2];
Quadrique(q) := {
  local Q,A,B,C,d,J,r,P,O1,c,c1,c2,c3,c4,s1,s2,s3,u,v,w,v1,w1,N,k;
  Q(x,y,z,t) := normal(t^2*normal(q(x/t,y/t,z/t)));
  A:=q2a(Q(x,y,z,t), [x,y,z,t]);
  r:=rank(A);
  B:=A[0..2,0..2];
  C:=A[0..2,3];
  d:=A[3,3];
  // on rend P orthogonale sauf si il y a des val propres =0
  (s1,s2,s3,u,v,w) := jortho(B);
  P:=tran([u,v,w]);
  // c:=normal(diff(C*P*[x,y,z], [x,y,z]));
  c:=normal(C*P);
  // les vp nulles sont a la fin
  c1:=c[0];
  c2:=c[1];
  c3:=c[2];
  if (s1*s2*s3!=0) {
    O1:=normal(-c1/s1*u-c2/s2*v-c3/s3*w);
    // O1:=solve(B*[x,y,z]+C, [x,y,z])[0];
    d:=q(O1[0],O1[1],O1[2]);
    return (O1,P,s1*X^2+s2*Y^2+s3*Z^2+d, [X,Y,Z], r, normal(q(op(O1+P*[X,Y,Z],
  }
  if (s1*s2!=0) {
    if (c3!=0) {
      O1:=normal(-c1/s1*u-c2/s2*v);
      d:=q(O1[0],O1[1],O1[2]);
      O1:=normal(-c1/s1*u-c2/s2*v-d/(2*c3)*w);
      return (O1,P,s1*X^2+s2*Y^2+2*c3*Z, [X,Y,Z], r,
        normal(q(op(O1+P*[X,Y,Z]))));
    }
  }
}

```

```

}
if (c3==0) {
O1:=normal(P*[-c1/s1,-c2/s2,0]);
d:=q(O1[0],O1[1],O1[2]);
return (O1,P,s1*X^2+s2*Y^2+d,[X,Y,Z],r,
        normal(q(normal(op(O1+P*[X,Y,Z])))));
}
}
if (s1!=0) {
if (c2 !=0 or c3 !=0) {
c4:=sqrt(c2^2+c3^2);
v1:=normal(P*[0,c3,-c2]/c4);
w1:=normal(P*[0,c2,c3]/c4);
O1:=normal(P*[-c1/s1,0,0]);
d:=q(O1[0],O1[1],O1[2]);
P:=P[0..2,0..0];
P:=border(P,v1);
P:=border(P,w1);
//en fait le nouveau c3 ci dessous ==c4 mais ca ne marche pas avec c4 mystere
//c:=normal(diff(C*P*[x,y,z],[x,y,z]));
//c3:=c[2];
//O1:=O1+normal(P*[0,0,-d/(2*c3)]);
O1:=O1+normal(P*[0,0,-d/(2*c4)]);
d:=q(O1[0],O1[1],O1[2]);
return (O1,P,s1*X^2+2*c3*Z+d,[X,Y,Z],r,
        normal(q(normal(op(O1+P*[X,Y,Z])))));
//return (O1,P,s1*X^2+2*c4*Z+d,[X,Y,Z],r,normal(q(op(O1+P*[X,Y,Z]))));
}
if (c2==0 && c3==0) {

//on rend P orthogonale
k:=normal(v*v);
w:=normal(-(w*v)*v+k*w);
w:=normal(w/sqrt(normal(w*w)));
v:=normal(v/sqrt(k));
P:=tran([u,v,w]);
O1:=normal(P*[-c1/s1,0,0]);
d:=q(O1[0],O1[1],O1[2]);

return (O1,P,s1*X^2+d,[X,Y,Z],r,
        normal(q(normal(op(O1+P*[X,Y,Z])))));
}
}
}

```

On tape :

$q_0(x, y, z) := 4x^2 + y^2 + z^2 - 4xy + 4xz - 2yz + 8x - 4y + 4z + 2$
 Quadrique(q0)

On obtient :

$[(-2)/3, 1/3, (-1)/3], [2/(\sqrt{6}), (\sqrt{5})/5, (\sqrt{30})/15],$
 $[-(1/(\sqrt{6})), 0, (\sqrt{30})/6], [1/(\sqrt{6}), (-2\sqrt{5})/5,$
 $(\sqrt{30})/30]], 6x^2-2, [X, Y, Z], 2, 6x^2-2$

On tape :

$q1(x, y, z) := x^2 + 3y^2 - 3z^2 - 8yz + 2zx - 4xy - 1$

Quadrique(q1)

On obtient :

$[0, 0, 0], [0, 1/(\sqrt{6}), 5(-(1/(\sqrt{30})))]], [1/(\sqrt{5}),$
 $2(-(1/(\sqrt{6}))), 2(-(1/(\sqrt{30})))]], [2/(\sqrt{5}),$
 $1/(\sqrt{6}), 1/(\sqrt{30})]]], -5x^2 + 6y^2 - 1, [X, Y, Z], 3,$
 $6y^2 - 5x^2 - 1$

On tape :

$q2(x, y, z) := 5x^2 + y^2 + z^2 - 2xy + 2xz - 6yz + 2x + 4y - 6z + 1$

Quadrique(q2)

On obtient :

$[(-1)/2, -1, 1/2], [0, 2/(\sqrt{6}), -(1/(\sqrt{3}))]]], [1/(\sqrt{2}),$
 $-(1/(\sqrt{6})), -(1/(\sqrt{3}))]]], [1/(\sqrt{2}), 1/(\sqrt{6}), 1/(\sqrt{3})]]],$
 $-2x^2 + 6y^2 + 3z^2 - 3, [X, Y, Z], 4$
 $3z^2 + 6y^2 - 2x^2 - 3$

On tape :

$q3(x, y, z) := 2x^2 + 2y^2 + z^2 + 2xz - 2yz + 4x - 2y - z + 3$

Quadrique(q3)

On obtient :

$[(-113)/144, 41/144, 17/72], [1/(\sqrt{3}), -(1/(\sqrt{2})), 1/(\sqrt{6})],$
 $[-(1/(\sqrt{3})), -(1/(\sqrt{2})), -(1/(\sqrt{6}))], [1/(\sqrt{3}),$
 $0, 2(-(1/(\sqrt{6})))]]], 3x^2 + 2y^2 + (2\sqrt{6}z)/3,$
 $[X, Y, Z], 4, 3x^2 + (4\sqrt{6}z)/3 + 2y^2$

On tape :

$q4(x, y, z) := 4x^2 + y^2 + z^2 - 4xy + 4xz - 2yz + 8x - 6y + 4z + 2$

Quadrique(q4)

On obtient :

$[(-227)/180, (-71)/72, (-227)/360],$
 $[2/(\sqrt{6}), (-\sqrt{5})/5, (-\sqrt{30})/15],$
 $[-(1/(\sqrt{6})), 0, (-\sqrt{30})/6],$
 $[1/(\sqrt{6}), (2\sqrt{5})/5, (-\sqrt{30})/30]],$
 $6x^2 + (2\sqrt{30}z)/6, [X, Y, Z], 3,$
 $(48\sqrt{30}z)/144 + 6x^2$

On tape :

$q5(x, y, z) := x^2 + 2y^2 - z^2 - 4xy + 4xz + 4yz + 6x - 4y + 2z + 1$

Quadrique(q5)

On obtient :

$[(-15)/13, 5/13, (-7)/13], [(-2)/3, (-\sqrt{13}+1)/(\sqrt{-8\sqrt{13}+52}),$
 $(\sqrt{13}+1)/(\sqrt{8\sqrt{13}+52})], [(-1)/3, 4/(\sqrt{-8\sqrt{13}+52}),$
 $4/(\sqrt{8\sqrt{13}+52})], [(-2)/3, (\sqrt{13}-3)/(\sqrt{-8\sqrt{13}+52}),$
 $(-\sqrt{13}-3)/(\sqrt{8\sqrt{13}+52})]]],$
 $2x^2 + \sqrt{13}y^2 + (-\sqrt{13})z^2 + (-49)/13, [X, Y, Z],$
 $4, ((-2\sqrt{13})z^2)/2 + (2\sqrt{13}y^2)/2 + 2x^2 + (-49)/13$

On tape :

$q6(x, y, z) := (x+y) * (y-z) + 3*x - 5*y$

Quadrique(q6)

On obtient :

$[16/9, 8/9, 11/9], [[1/(\sqrt{2}), 1/(\sqrt{6}), -(1/(\sqrt{3}))],$
 $[0, 2/(\sqrt{6}), 1/(\sqrt{3})], [1/(\sqrt{2}), -(1/(\sqrt{6}))],$
 $1/(\sqrt{3})]]], (X^2)/-2 + (3*Y^2)/2 + (2*(-4*\sqrt{3})*Z)/3,$
 $[X, Y, Z], 4, -(X^2)/2 - (8*\sqrt{3}*Z)/3 - (-3*Y^2)/2$

On tape :

$q7(x, y, z) := 7*x^2 + 4*y^2 + 4*z^2 + 4*x*y - 4*x*z - 2*y*z - 4*x + 5*y + 4*z - 18$

Quadrique(q7)

On obtient :

$[11/27, (-26)/27, (-29)/54], [[1/(\sqrt{5}), 54*(-(1/(\sqrt{30})*27))],$
 $2/(\sqrt{6})], [0, 5/(\sqrt{30}), 1/(\sqrt{6})], [2/(\sqrt{5}), 1/(\sqrt{30}),$
 $-(1/(\sqrt{6}))]]], 3*X^2 + 3*Y^2 + 9*Z^2 + (-602)/27,$
 $[X, Y, Z], 4, 3*Y^2 + 3*X^2 + 9*Z^2 + (-602)/27$

On tape :

$q8(x, y, z) := -x^2 - 3*z^2 - 4*x*y + 4*x*z + 4*y*z + 6*x - 4*y + 2*z + 1$

Quadrique(q8)

On obtient (c'est tres long !):

$[(-11)/54, 223/108, 61/54], [[(-\sqrt{13}+1)/(\sqrt{-8*\sqrt{13}+52}),$
 $(\sqrt{13}+1)/(\sqrt{8*\sqrt{13}+52}), (-2)/3], [4/(\sqrt{-8*\sqrt{13}+52}),$
 $4/(\sqrt{8*\sqrt{13}+52}), (-1)/3], [(\sqrt{13}-3)/(\sqrt{-8*\sqrt{13}+52}),$
 $(-\sqrt{13}-3)/(\sqrt{8*\sqrt{13}+52}), (-2)/3]]],$
 $(\sqrt{13}-2)*X^2 + (-\sqrt{13}-2)*Y^2 - 4*Z, [X, Y, Z], 4,$
 $((-4+2*\sqrt{13})*X^2)/2 + ((-4-2*\sqrt{13})*Y^2)/2 - 4*Z$ On tape :

$q9(x, y, z) := x^2 - 3*x*y + y^2 + x - y - z^2/2 + \sqrt{10}/5*z$

Quadrique(q9)

On obtient un cône car r=3 :

$[-1/5, 1/5, (\sqrt{10})/5], [[1/(\sqrt{2}), 1/(\sqrt{2}), 0], [-(1/(\sqrt{2})),$
 $1/(\sqrt{2}), 0], [0, 0, 1]], (5*X^2)/2 + (Y^2)/(-2) + (Z^2)/(-2), [X, Y, Z], 3,$
 $5/2*X^2 + (-1)/2*Y^2 + (-1)/2*Z^2$ On tape :

$q10(x, y, z) := x^2 - 3*x*y + y^2 - x - y - z^2/2 + \sqrt{10}/5*z$

Quadrique(q10)

On obtient :

$[-1, -1, (\sqrt{10})/5], [[1/(\sqrt{2}), 1/(\sqrt{2}), 0], [-(1/(\sqrt{2})),$
 $1/(\sqrt{2}), 0], [0, 0, 1]], (5*X^2)/2 + (Y^2)/(-2) + (Z^2)/(-2) + 6/5, [X, Y, Z], 4,$
 $5/2*X^2 + (-1)/2*Y^2 + (-1)/2*Z^2 + 6/5$

Chapitre 8

Les programmes récursifs

Certains programmes se trouvent dans `examples/recur`.

8.0.1 Une liste de mots

On veut énumérer les éléments d'une liste. Pour cela on écrit le premier élément et on énumère la liste privée de son premier élément. On s'arrête quand la liste est vide.

On écrit :

```
enumere(l) := {
  if (l==[]) return 0;
  print(l[0]);
  enumere(tail(l));
}
```

On tape :

```
enumere(["jean", "paul", "pierre"])
```

On obtient, en écriture bleue, dans la zone des résultats intermédiaires :

```
"jean"
"paul"
"pierre"
```

8.0.2 Les mots

Étant donné un mot de n lettres, on veut écrire n lignes :
la première ligne sera constituée par la première lettre du mot,
la deuxième ligne sera constituée par les deux premières lettres...,
la dernière ligne sera constituée par le mot tout entier.

On écrira : `mots(m)` de façon récursive. On peut se servir de la fonction `size(m)` de `xcas` qui renvoie le nombre de lettres du mot `m`, et de la fonction `suppress(m, k)` de `xcas` qui renvoie le mot `m` privé de sa k -ième lettre ($k=0 \dots \text{size}(m)-1$).

On tape :

```
saufdernier(m) := {
  return suppress(m, size(m)-1);
}
```

puis

```
mots(m) := {
  if (size(m) == 0) return 0;
  mots(saufdernier(m));
  print(m);
}
```

Exercice

Comment modifier le programme précédent pour avoir :

Étant donné un mot de n lettres, on veut écrire n lignes :

la première ligne sera constituée par le mot tout entier,

la deuxième ligne sera constituée par le mot privé de sa première lettre...,

la dernière ligne sera constituée par la première lettre du mot.

Réponse

On peut se servir de la fonction `tail(m)` de `xcas` qui renvoie le mot m privé de sa première lettre.

```
motex(m) := {
  if (size(m) == 0) return 0;
  print(m);
  motex(tail(m));
}
```

8.0.3 Les palindromes

Étant donné une phrase, on veut écrire cette phrase en l'écrivant de droite à gauche. On écrira :

`palindrome(s)` de façon récursive :

il faut rajouter la première lettre de la phrase à la fin du palindrome de la phrase privée de sa première lettre.

On tape :

```
palindrome(ph) := {
  local s;
  if (s == 0) return ph;
  s := size(ph) - 1;
  return concat(palindrome(tail(ph)), ph[0]);
}
```

ou encore :

il faut rajouter la dernière lettre de la phrase devant le palindrome de la phrase privée de sa dernière lettre.

On tape :

```
saufdernier(m) := {
  return suppress(m, size(m) - 1);
}
```

```
palindrome(ph) := {
  local s;
```

```

if (s==0) return ph;
s:=size(ph)-1;
return concat(ph[s],palindrome(saufdernier(ph)));
}

```

8.0.4 Les tours de Hanoï

Une tour de Hanoï est composée de p disques de rayons différents que l'on numérote de 1 à p selon l'ordre croissant des rayons (le plus petit disque a le numéro 1 et le plus gros le numéro p). On dispose de 3 plots numérotés de 1 à 3.

Au départ les disques sont empilés selon l'ordre croissant sur le plot 1.

Le jeu consiste à reconstituer la tour sur le plot 2, en se servant du plot 3 comme plot intermédiaire, en déplaçant les disques un à un, et en posant toujours un disque sur un disque plus petit que lui.

Par exemple, on peut mettre le disque 2 sur le disque 5, mais pas sur le disque 1.

On veut écrire un programme qui imprime ce qu'il faut faire comme manipulations : ce sera `tour(a,b,c,p)`, où p représente le nombre de disques, où a représente le plot de départ, où b représente le plot d'arrivée, et où c représente le plot intermédiaire.

On tapera alors par exemple :

```
tour(1,2,3,4)
```

si on a une tour de 4 disques sur le plot 1, et qu'on veut la reconstituer sur le plot 2 par l'intermédiaire du plot 3.

Les manipulations à faire sont récursives, en voici les étapes :

- il faut arriver à dégager le disque p pour qu'il soit seul sur le plot 1 avec les $p-1$ disques sur le plot 3, le plot 2 étant vide et prêt à recevoir le disque p . Dans cette situation, il y a une tour de $p-1$ disques sur le plot 3. Cela veut dire que l'on est arrivé à reconstituer la tour constituée des $p-1$ premiers disques sur le plot 3, en se servant du plot 2 comme plot intermédiaire.

Avec les notations ci dessus c'est que l'on a effectué :

```
tour(a,c,b,p-1)
```

c'est l'instruction qui permet de reconstituer une tour de $p-1$ disques sur le plot 3 en partant du plot 1 et en se servant du plot 2 comme plot intermédiaire.

- on déplace ensuite le disque p du plot 1 sur le plot 2 :

```
print("deplacer le disque ",p," de ", a , " vers", b).
```

- il reste à reconstituer la tour de $p-1$ disques sur le plot 2 en partant du plot 3 et en se servant du plot 1 comme plot intermédiaire.

Il faut donc effectuer :

```
tour(c,b,a,p-1)
```

- il reste à trouver le test d'arrêt qui est simplement ($p==0$) c'est à dire : quand on a une tour de zéro disque on ne fait rien (on renvoie 0).

On tape dans un éditeur de programmes (que l'on ouvre avec Alt+p), puis on le teste et le valide avec OK :

```

//tour(1,2,3,4) (tour de hanoi)
//deplacement des p disques de a vers b en passant par c
tour(a,b,c,p) :={
    if (p==0) return 0;
    tour(a,c,b,p-1);
}

```

```

    print("deplacer le disque ",p," de ", a , " vers", b);
    tour(c,b,a,p-1);
    return 0;
}

```

8.0.5 Les permutations circulaires

On écrit la fonction `circulaire(l)` qui renvoie la liste obtenue à partir de `l` en renvoyant le début de la liste `l` à la fin de `l`.

```

//l:=[1,2,3]; circulaire(l)
//renvoie la liste l ou la tete est mise a la fin.
circulaire(l):={
return concat(tail(l),l[0]);
};

```

On écrit la fonction `permcir(l)` qui renvoie la liste des permutations circulaires obtenues à partir de `l`. On écrit cette fonction récursivement en remplaçant `l` par `circulaire(l)`. Il faut un test d'arrêt pour ce parcours, pour cela on a besoin d'un paramètre supplémentaire qui sera `ld` : c'est une liste de référence égale à `l` au départ et qui n'est pas modifiée. On s'arrête quand `circulaire(l)==ld`, c'est à dire quand on retrouve la liste de départ. On utilise une variable locale `lr` égale à la liste à renvoyer.

```

// utilise circulaire, l:=[1,2,3];permcir(l,l);
//renvoie les permutations circulaires de l
//variable locale lr la liste resultat
// ld liste reference de depart
permcir(l,ld):={
local lr;
if (circulaire(l)==ld) return [l];
lr:=[l];
lr:= append(lr,op(permcir(circulaire(l),ld)));
return lr;
};

```

On peut supprimer la variable locale `lr` et la fonction `circulaire`.

On écrit alors la fonction `permcc(l)` qui renvoie la liste des permutations circulaires obtenues à partir de `l`.

Ici, on utilise un autre test d'arrêt, on a toujours besoin d'un paramètre supplémentaire qui sera `ld` : c'est une liste de référence égale à `l` au départ et qui est modifiée, sa taille diminue de 1 à chaque appel récursif. On s'arrête quand `ld==[]`, c'est à dire quand on a fait autant d'appels que la taille de `l`.

```

//l:=[1,2,3];permcc(l,l);
//renvoie les permutations circulaires de l
//sans variable locale, ld liste reference de depart
permcc(l,ld):={
if (ld==[]) return [];
return [l,op(permcc(concat(tail(l),l[0]),tail(ld)))];
};

```

Comme il faut 2 paramètres pour écrire la fonction récursive `permcc`, on écrit la fonction finale `permutation_circ` qui utilise `permcc` :

```
//l:=[1,2,3];permutation_circ(l);
//renvoie les permutations circulaires de l
//utilise permcc
permutation_circ(l):={
return permcc(l,l);
};
```

On tape :

```
permutation_circ([1,2,3])
```

On obtient :

```
[[1,2,3],[2,3,1],[3,1,2]]
```

8.0.6 Les permutations

1/ En faisant `n=size(l)` appels récursifs.

Les fonctions que l'on va écrire vont utiliser la fonction `echange`.

```
//echange ds l les elements d'indices j et k
echange(l,j,k):={
local a;
a:=l[j];
l[j]:=l[k];
l[k]:=a;
return l;
};
```

On peut décrire l'arbre des permutations de la liste l :

à partir de la racine on a `n=size(l)` branches. Chaque branche commence respectivement par chacun des éléments de la liste l .

On va donc parcourir cet arbre de la racine (nœud de niveau 0) aux différentes extrémités, en renvoyant la liste des branches parcourues pour arriver à cette extrémité.

On va parcourir cet arbre en parcourant les n branches. On numérote ces branches par $p = 1..n$ et le niveau des nœuds $q = 0..n$.

On aura donc n appels récursifs.

Chaque branche p peut être considérée à leur tour comme un arbre ayant $n - 1$ branches. La branche p aboutit aux permutations qui laissent invariant le p -ième élément de l ($l[p - 1]$). C'est cet élément que l'on va échanger avec $l[0]$ pour que chaque branche p laisse invariant l'élément $l[0]$.

On sait que l'on est arrivé au bout de la branche, quand on se trouve au nœud de niveau $n - 1$, dans ce cas la permutation cherchée est l (c'est la permutation obtenue à partir de l en laissant ces $n - 1$ premiers éléments invariants).

On utilise une variable locale `lr`, égale à la liste à renvoyer et un paramètre k , pour que `permus(l,k)` renvoie toutes les permutations de l qui laissent invariant les k premiers éléments de l . On tape :

```
//utilise echange et la variable locale lr (liste resultat)
```

```
//permus(l,k) laisse invariant les k premiers elements de l
//permus([1,2,3,4],0); renvoie toutes les permutations de l
permus(l,k):={
  local lr;
  if (k==size(l)-1) return [l];
  lr:=[];
  for (j:=k;j<size(l);j++){
    l:=echange(l,k,j);
    lr:=[op(lr),op(permus(l,k+1))];
    l:=echange(l,j,k);
  }
  return lr;
};
```

On n'est pas obligé de remettre la suite l à sa valeur de départ pour recommencer l'itération puisque le premier échange dans l'itération revient à transformer l en la liste où on a mis son j -ième élément en tête ($j = 0..n - 1$).

```
//utilise echage permuts([1,2,3,4],0)
//la 2ieme instruction echage est inutile car on echage
// 0 et 1 ([1,0,2..]) puis 0 et 2 ([2,0,1..]) etc
//avec la 2ieme instruction echage, on echage
//0 et 1 ([1,0,2..]) puis 0 et 2 ([2,1,0..]) etc
permuts(l,k):={
  local lr;
  if (k==size(l)-1) return [l];
  lr:=[];
  for (j:=k;j<size(l);j++){
    l:=echange(l,k,j);
    lr:=[op(lr),op(permuts(l,k+1))];
  }
  return lr;
};
```

Comme il faut 2 paramètres pour écrire la fonction récursive permuts, on écrit la fonction permutation qui utilise permuts :

```
//l:=[1,2,3];permutation(l);
//renvoie toutes les permutations de l
//utilise permuts
permutation(l):={
  return permuts(l,0);
};
```

On tape :

```
permutation([1,2,3])
```

On obtient :

```
[ [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1] ]
```

On peut aussi écrire une autre fonction récursive ayant comme paramètre ld et lf . ld contient les premières valeurs de l qui seront inchangées dans la permutation et

lf contient les valeurs restantes de l, celles qui restent à permuter. On remarquera que l'on utilise le resultat mis dans res est ici une séquence.

```
//au debut ld=[] et lf=l,
//groupe_s([],l) renvoie toutes les permutations de l
groupe_s(ld,lf):={
  local j,n,res;
  n:=size(lf);
  res:=NULL;
  if (n==1)
    return concat(ld,lf);
  for (j:=0;j<n;j++){
    res:=res,groupe_s(append(ld,lf[j]),tail(lf));
    // permutation circulaire
    lf:=append(tail(lf),lf[0]);
  }
  return res;
};
```

Et la fonction groupesyml qui utilise la fonction récursive groupe_s :

```
//utilise groupe_s
//groupesyml(l) renvoie toutes les permutations de l
groupesyml(l):=return(groupe_s([],l));
```

2/ En faisant 2 appels récursifs.

Cet algorithme est surtout fait pour des langages qui n'ont pas de boucle for.

Les fonctions vont utiliser la fonction circulaire (pour plus de clareté), puis on remplacera circulaire(l) par concat(tail(l),l[0]).

```
//l:=[1,2,3]; circulaire(l)
//renvoie la liste l ou la tete est mise a la fin.
circulaire(l):={
  return concat(tail(l),l[0]);
};
```

On peut décrire l'arbre des permutations de la liste l :

à partir de la racine on a $n = \text{size}(l)$ branches. Chaque branche commence par chacun des éléments de la liste l.

On va parcourir cet arbre, en parcourant la première branche, puis en considérant qu'il reste à parcourir un arbre de $n - 1$ branches.

On aura donc 2 appels récursifs.

Pour le parcours de la première branche, il faut connaître la liste des éléments qui nous a permis d'arriver à un nœud donné, c'est cette liste que l'on met dans ldl, l contenant les éléments qu'il faut encore permuter. On s'arrête quand $l = []$, et le résultat est [ldl].

Pour le parcours des $n - 1$ branches restantes, on change pour chaque branche la liste à permuter en circulaire(l).

Il faut un test d'arrêt pour ce parcours, pour cela on a besoin d'un paramètre supplémentaire qui sera ld (liste de référence égale à l au départ) dans permss ou qui sera n (longueur de l au départ) dans permss1.

On écrit permss :

```
// utilise circulaire, l:=[1,2];permss([],l,l);
//ldl=debut de l, l=liste a permuter,
//ld=liste de reference (=l au debut)
permss(ldl,l,ld):={
  if (l==[]) return [ldl];
  if (ld==[]) return ld;
  return [op(permss(concat(ldl,l[0]),tail(l),tail(l))),
          op(permss(ldl,circulaire(l),tail(ld))))];
};
```

On écrit `permss1` qui utilise comme paramètre `n` qui représente la longueur de la liste qui reste à permuter (`n=size(l)` au départ) :

```
//utilise circulaire, l:=[1,2,3,4];permss1([],l,size(l));
//ldl=debut de l, l=liste a permuter, n=size(l) au debut
permss1(ldl,l,n):={
  if (l==[]) return [ldl];
  if (n==0) return [];
  return [op(permss1(concat(ldl,l[0]),tail(l),size(tail(l)))),
          op(permss1(ldl,circulaire(l),n-1))];
};
```

On a aussi écrit la fonction `permss2` contenant une variable locale `lr` qui est la liste à renvoyer et qui donne un algorithme plus lisible.

```
//l:=[1,2];permss2([],l,l);
//ldl=debut de l, l=liste a permuter,
//ld=liste de reference (=l au debut)
// lr liste a renvoyer en variable locale
permss2(ldl,l,ld):={
  local lr;
  if (l==[]) return [ldl];
  if (ld==[]) return [];
  lr:=permss2(concat(ldl,l[0]),tail(l),tail(l));
  lr:=append(lr,op(permss2(ldl,concat(tail(l),l[0]),tail(ld))));
  return lr
};
```

puis la fonction `permute` qui utilise `permss2` :

```
//utilise permss2,
//permute(l) renvoie toutes les permutations de l
permute(l):={
  return permss2([],l,l);
};
```

On tape :

```
permute([1,2,3])
```

On obtient :

```
[ [1,2,3], [1,3,2], [2,3,1], [2,1,3], [3,1,2], [3,2,1] ]
```

Chapitre 9

Récupérer et installer un logiciel

La description qui suit s'applique au navigateur Netscape. Elle s'applique de manière identique pour d'autres navigateurs à quelques détails près.

Lorsqu'un lien mentionne un logiciel que vous voulez télécharger, maintenez la touche Shift appuyée et cliquez avec la souris sur ce lien. Une fenêtre s'ouvre et vous propose de sauvegarder un fichier sur votre disque dur, changez éventuellement le nom de ce fichier et notez l'emplacement du répertoire où il a été sauvegardé.

La méthode d'installation du logiciel dépend de votre système d'exploitation et du type de fichier téléchargé, que l'on détermine en général par son extension, c'est-à-dire par les lettres (en principe 3) qui suivent le caractère point dans le nom de ce fichier.

Sous Windows, vous récupérerez en général un fichier exécutable `exe` il suffit alors de cliquer sur son icône pour installer le logiciel. Ou alors il s'agira d'une archive, il vous faudra alors un logiciel de décompression pour l'installer (l'un des plus populaires s'appelle Winzip et propose une version à l'essai gratuitement). Une fois désarchivé votre logiciel, vous trouverez dans l'archive un fichier donnant des instructions complémentaires ou vous devrez vous référer au site où vous avez téléchargé l'archive.

Sous Linux, les formats les plus courants sont `rpm`, `deb` et `tgz` (ou `tar.gz`).

Les deux premiers correspondent à des distributions Linux (Red Hat, Mandrake, Suse par exemple pour `rpm`, Debian pour `deb`) ils doivent être installés par `root` (tapez `su` dans une fenêtre de commandes pour passer `root`) en utilisant respectivement les commandes :

```
rpm -U nom_de_fichier.rpm pour Red Hat, Mandrake, Suse
```

```
apt-get install nom_de_fichier.deb pour Debian
```

Regardez bien les éventuels messages d'erreurs qui apparaissent. Ils peuvent signaler que vous devez installer d'autres logiciels de votre distribution Linux pour faire fonctionner votre nouveau logiciel. En général ces logiciels non installés se trouvent sur le CD-ROM de votre distribution Linux. Ils s'installent par la même commande, mais vous devrez d'abord vous déplacer dans le répertoire du CD-ROM qui les contient. Pour cela, introduisez le CD-ROM dans le lecteur, attendez quelques secondes, tapez : `ls /mnt/cdrom`

(si rien n'apparaît, tapez : `mount /mnt/cdrom` et recommencez).

Ensuite tapez `cd /mnt/cdrom` puis allez dans le répertoire correct (par exemple `cd Mandrake/RPMS` pour une distribution Mandrake) et lancez les commandes

d'installation des logiciels manquants (`rpm -U ...` ou `apt ...`). Certaines distributions Linux proposent des interfaces plus ou moins conviviales pour installer des logiciels, par exemple `rpmdrake` pour Mandrake, `dselect` pour Debian...

Les archives `tar.gz` et `tgz` s'installent sur toute distribution Linux. Commencez par en regarder la table des matières par la commande :

```
tar tvfz nom_de_fichier.tgz
```

afin de déterminer depuis quel répertoire il faut décompacter le fichier ou lisez la documentation du logiciel disponible en général sur le site où vous avez téléchargé l'archive. Si vous voyez apparaître des chemins tels que `usr/local/bin` ou `usr/bin`, placez vous dans le répertoire racine (`cd /`) puis tapez :

```
tar xvfz nom_du_repertoire/nom_de_fichier.tgz
```

(si vous avez sauvegardé l'archive dans votre répertoire, vous pouvez utiliser `~votre_nom_de_login` comme `nom_de_repertoire`). Sinon, tapez simplement depuis le répertoire où se trouve l'archive :

```
tar xvfz nom_de_fichier.tgz
```

placez-vous dans le répertoire créé et lisez les fichiers `README` ou/et `INSTALL` qui s'y trouvent sans doute.

Remarques :

1/ N'oubliez pas de quitter le compte de `root` avant d'utiliser votre logiciel (en tapant simultanément sur les touches `Ctrl` et `D`).

2/ les archives `.tar.bz2` se traitent de manière identique, à ceci près qu'il faut enlever le `z` de `tvfz` ou `xvfz` et rajouter

`--use-compress-program bunzip2`, par exemple :

```
tar tvf nom_de_fichier.tar.bz2 --use-compress-program bunzip2
```

3/ les archives `.zip` se visualisent par la commande :

```
unzip -v nom_de_fichier.zip
```

et se désarchivent par la commande :

```
unzip nom_de_fichier.zip
```

Sous linux et autres Unix, vous trouverez également des fichiers sources qu'il vous faudra compiler avant de pouvoir utiliser le logiciel correspondant. De nombreux fichiers sources suivent la procédure d'installation du projet GNU. Pour les compiler la procédure est la suivante (après avoir décompressé l'archive) :

```
cd nom_d_archive
```

```
./configure
```

```
make
```

Puis on passe `root` en tapant `su`, puis : `make install-strip`

puis quittez le compte `root` (en tapant simultanément sur les touches `Ctrl` et `D`). Vous pourrez alors utiliser le logiciel nouvellement installé (éventuellement après avoir tapé la commande `rehash` si votre interpréteur de commandes est `tcsh`).

Si cela ne fonctionne pas vous devriez trouver un fichier `README` ou `INSTALL` qui explique la procédure à suivre.

Index

`||`, 22
`„`, 25
`:=`, 10, 15
`;`, 14
`$`, 25
`%`, 33
`&&`, 22

about, 17
and, 22
append, 25
asc, 63, 91
assume, 16, 17
augment, 25

beakpoint, 4

char, 63, 91
concat, 25

debug, 4

for, 20

head, 25
horner, 67

if, 18
input, 12
iquo, 33
irem, 33

local, 10

makelist, 24
mod, 33

nops, 25
not, 22

op, 24, 25
or, 22
ord, 63

poly2symb, 67
prepend, 25
print, 13
purge, 10

read, 3
return, 22
rmbeakpoint, 4
rmwatch, 4

seq, 24, 25
set, 24
size, 25
smod, 33
symb2poly, 67

tail, 25

watch, 4
while, 21

Table des matières

1	Vue d'ensemble de xcas pour le programmeur	3
1.1	Installation de xcas	3
1.2	Éditer, sauver, exécuter un programme avec xcas	3
1.3	Débugger un programme avec xcas	4
1.4	Présentation générale des instructions avec xcas	5
2	Les différentes instructions	9
2.1	Les commentaires	9
2.1.1	Traduction Algorithmique	9
2.1.2	Traduction xcas	9
2.1.3	Traduction MapleV	9
2.1.4	Traduction MuPAD	9
2.1.5	Traduction TI89/92	10
2.2	Les variables	10
2.2.1	Leurs noms	10
2.2.2	Notion de variables locales	10
2.3	Les paramètres	11
2.3.1	Traduction xcas	11
2.3.2	Traduction MapleV	12
2.3.3	Traduction MuPAD	12
2.3.4	Traduction TI89/92	12
2.4	Les Entrées	12
2.4.1	Traduction Algorithmique	12
2.4.2	Traduction xcas	13
2.4.3	Traduction MapleV	13
2.4.4	Traduction MuPAD	13
2.4.5	Traduction TI89/92	13
2.5	Les Sorties	13
2.5.1	Traduction Algorithmique	13
2.5.2	Traduction xcas	13
2.5.3	Traduction MapleV	13
2.5.4	Traduction MuPAD	14
2.5.5	Traduction TI89/92	14
2.6	La séquence d'instructions ou action	14
2.6.1	Traduction xcas	14
2.6.2	Traduction MapleV	15
2.6.3	Traduction MuPAD	15

2.6.4	Traduction TI89/92	15
2.7	L'instruction d'affectation	15
2.7.1	Traduction Algorithmique	15
2.7.2	Traduction xcas	15
2.7.3	Traduction Maple	16
2.7.4	Traduction MuPAD	16
2.7.5	Traduction TI89/92	16
2.8	L'instruction pour faire des hypothèses sur une variable formelle	16
2.8.1	Traduction Algorithmique	16
2.8.2	Traduction xcas	16
2.8.3	Traduction Maple	17
2.8.4	Traduction MuPAD	17
2.8.5	Traduction TI89/92	17
2.9	L'instruction pour connaître les contraintes d'une variable	17
2.9.1	Traduction Algorithmique	17
2.9.2	Traduction xcas	17
2.9.3	Traduction Maple	17
2.9.4	Traduction MuPAD	18
2.9.5	Traduction TI89/92	18
2.10	Les instructions conditionnelles	18
2.10.1	Traduction Algorithmique	18
2.10.2	Traduction xcas	18
2.10.3	Traduction MapleV	19
2.10.4	Traduction MuPAD	19
2.10.5	Traduction TI89/92	20
2.11	Les instructions "Pour"	20
2.11.1	Traduction Algorithmique	20
2.11.2	Traduction xcas	20
2.11.3	Traduction MapleV	20
2.11.4	Traduction MuPAD	21
2.11.5	Traduction TI89 92	21
2.12	L'instruction "Tant que"	21
2.12.1	Traduction Algorithmique	21
2.12.2	Traduction xcas	21
2.12.3	Traduction MapleV	21
2.12.4	Traduction MuPAD	21
2.12.5	Traduction TI89/92	21
2.13	Les conditions ou expressions booléennes	22
2.13.1	Les opérateurs relationnels	22
2.13.2	Les opérateurs logiques	22
2.14	Les fonctions	22
2.14.1	Traduction Algorithmique	23
2.14.2	Traduction xcas	23
2.14.3	Traduction MapleV	23
2.14.4	Traduction MuPAD	23
2.14.5	Traduction TI89 92	23
2.15	Les listes	24
2.15.1	Traduction Algorithmique	24

2.15.2	Traduction xcas	24
2.15.3	Traduction MapleV	26
2.15.4	Traduction MuPAD	26
2.15.5	Traduction TI89/92	27
2.16	Un exemple : le crible d'Eratosthène	27
2.16.1	Description	27
2.16.2	Écriture de l'algorithme	28
2.16.3	Traduction xcas	28
2.16.4	Traduction TI89/92	29
2.17	Un exemple de fonction vraiment récursive	30
2.17.1	La définition	30
2.17.2	Le programme	30
3	Les programmes d'arithmétique	33
3.1	Quotient et reste de la division euclidienne	33
3.1.1	Les fonctions ico, irem et smod de xcas	33
3.1.2	Activité	33
3.2	Calcul du PGCD par l'algorithme d'Euclide	36
3.2.1	Traduction algorithmique	36
3.2.2	Traduction xcas	37
3.2.3	Traduction MapleV	37
3.2.4	Traduction MuPAD	38
3.2.5	Traduction TI89 92	38
3.3	Identité de Bézout par l'algorithme d'Euclide	39
3.3.1	Version itérative sans les listes	39
3.3.2	Version itérative avec les listes	40
3.3.3	Version récursive sans les listes	40
3.3.4	Version récursive avec les listes	41
3.3.5	Traduction xcas	41
3.4	Décomposition en facteurs premiers d'un entier	42
3.4.1	Les algorithmes et leurs traductions algorithmiques	42
3.4.2	Traduction xcas	45
3.5	Décomposition en facteurs premiers en utilisant le crible	45
3.5.1	Traduction Algorithmique	46
3.5.2	Traduction xcas	46
3.6	La liste des diviseurs	47
3.6.1	Les programmes avec les élèves	47
3.6.2	Le nombre de diviseurs d'un entier n	48
3.6.3	L'algorithme sur un exemple	48
3.6.4	Les algorithmes donnant la liste des diviseurs de n	49
3.7	La liste des diviseurs avec la décomposition en facteurs premiers	51
3.7.1	FPDIV	51
3.7.2	CRIBLEDIV	52
3.7.3	Traduction Algorithmique	52
3.8	Calcul de $A^P \bmod N$	53
3.8.1	Traduction Algorithmique	53
3.8.2	Traduction xcas	55
3.9	La fonction "estpremier"	55

3.9.1	Traduction Algorithmique	55
3.9.2	Traduction xcas	57
3.10	La fonction estpremc en utilisant le crible	58
3.10.1	Traduction algorithmique	58
3.10.2	Traduction xcas	58
3.11	Méthode probabiliste de Mr Rabin	58
3.11.1	Traduction Algorithmique	59
3.11.2	Traduction xcas	59
3.12	Méthode probabiliste de Mr Miller-Rabin	60
3.12.1	Un exemple	60
3.12.2	L'algorithme	60
3.12.3	Traduction Algorithmique	61
3.12.4	Traduction xcas	62
3.13	Numération avec xcas	63
3.13.1	Passage de l'écriture en base dix à une écriture en base b	63
3.13.2	Passage de l'écriture en base b de n à l'entier n	66
3.14	Traduction xcas de l'algorithme de Hörner	67
3.15	Affichage d'un nombre en une chaîne comprenant des espaces	68
3.15.1	Affichage d'un nombre entier par tranches de p chiffres	68
3.15.2	Transformation d'un affichage par tranches en un nombre entier	69
3.15.3	Affichage d'un nombre décimal de $[0,1[$ par tranches de p chiffres	69
3.15.4	Affichage d'un nombre décimal par tranches de p chiffres	70
3.16	Écriture décimale d'un nombre rationnel	71
3.16.1	Algorithme de la puissance	71
3.16.2	Avec un programme	72
3.16.3	Construction d'un rationnel	73
3.17	Développement en fraction continue	74
3.17.1	Développement en fraction continue d'un rationnel	74
3.17.2	Développement en fraction continue d'un réel quelconque	77
3.17.3	Les programmes	78
3.17.4	Exemples	80
3.17.5	Suite des réduites successives d'un réel	80
3.17.6	Suite des réduites "plus 1" successives d'un réel	82
3.17.7	Propriété des réduites	82
3.18	Suite de Hamming	83
3.18.1	La définition	83
3.18.2	L'algorithme à l'aide d'un crible	83
3.18.3	L'algorithme sans faire un crible	86
3.18.4	La traduction de l'algorithme avec xcas	86
4	codage	89
4.1	Codage de Jules Cesar	89
4.1.1	Introduction	89
4.1.2	Codage par symétrie point ou par rotation d'angle π	89
4.1.3	Avec les élèves	89
4.1.4	Travail dans $\mathbb{Z}/26\mathbb{Z}$	90

4.1.5	Codage par rotation d'angle $k * \pi/13$	91
4.2	Écriture des programmes correspondants	91
4.2.1	Passage d'une lettre à un entier entre 0 et 25	91
4.2.2	Passage d'un entier entre 0 et 25 à une lettre	91
4.2.3	Passage d'un entier k entre 0 et 25 à l'entier $n+k \bmod 26$	92
4.2.4	Codage d'un message selon Jules César	92
4.3	Codage en utilisant une symétrie par rapport à un axe	92
4.3.1	Passage d'un entier k entre 0 et 25 à l'entier $n-k \bmod 26$	92
4.3.2	Codage d'un message selon une symétrie droite D	93
4.4	Codage en utilisant une application affine	93
4.5	Codage en utilisant un groupement de deux lettres	93
4.6	Le codage Jules César et le codage linéaire	95
4.6.1	Les caractères et leurs codes	95
4.6.2	Les différentes étapes du codage	95
4.6.3	Le programme xcas	96
4.6.4	Le programme C++	97
4.6.5	Exercices de décodage	99
4.6.6	Solutions des exercices de décodage Jules César et linéaire	102
4.7	Chiffrement affine : premier algorithme	102
4.7.1	L'algorithme	102
4.7.2	Traduction Algorithmique	103
4.7.3	Traduction xcas	104
4.8	Chiffrement affine : deuxième algorithme	105
4.8.1	L'algorithme	105
4.8.2	Traduction Algorithmique	106
4.8.3	Traduction xcas	107
4.9	Devoir à la maison	110
4.9.1	Le code Ascii	111
4.10	Codage RSA	111
4.10.1	Le cryptage des nombres avec la méthode RSA	111
4.10.2	La fonction de codage	113
4.11	Les programmes correspondants au codage et décodage RSA	116
4.11.1	Exercices de décodage RSA avec différents paramètres	118
4.11.2	Solutions des exercices de décodage	120
4.12	Codage RSA avec signature	121
4.12.1	Quelques précautions	123
5	Algorithmes sur les suites et les séries	125
5.1	Les suites	125
5.1.1	Les suites $u_n = f(n)$	125
5.1.2	Les suites récurrentes	126
5.2	Les séries	129
5.2.1	Les sommes partielles	130
5.2.2	Exemple d'accélération de convergence des séries à termes positifs	131
5.3	Méthodes d'accélération de convergence des séries alternées	134
5.3.1	Un exemple d'accélération de convergence des séries alternées	134

5.3.2	La transformation d'Euler pour les series alternées	138
5.3.3	Autre approximation d'une série alternée	140
5.3.4	Transformation d'une série en série alternée	146
5.4	Solution de $f(x) = 0$ par la méthode de Newton	147
5.4.1	La méthode de Newton	148
5.4.2	La méthode de Newton avec préfacteur	149
5.5	Trouver un encadrement de la valeur pour laquelle une fonction est minimum	150
5.5.1	Déscription du principe de la méthode	150
5.5.2	Déscription de 2 méthodes	151
5.5.3	Traduction <code>Xcas</code> de l'algorithme avec Fibonacci	151
6	Algorithmes d'algèbre	155
6.1	Méthode pour résoudre des systèmes linéaires	155
6.1.1	Le pivot de Gauss quand A est de rang maximum	155
6.1.2	Le pivot de Gauss pour A quelconque	157
6.1.3	La méthode de Gauss-Jordan	158
6.1.4	La méthode de Gauss et de Gauss-Jordan avec recherche du pivot	161
6.1.5	Application : recherche du noyau grâce à Gauss-Jordan	163
6.2	Résolution d'un système linéaire	165
6.2.1	Résolution d'un système d'équations linéaires	165
6.2.2	Résolution de $MX = b$ donné sous forme matricielle	167
6.3	La décomposition LU d'une matrice	167
6.4	La décomposition de Cholesky d'une matrice symétrique définie positive	170
6.4.1	Les méthodes	170
6.4.2	Le programme de factorisation de Cholesky avec LU	172
6.4.3	Le programme de factorisation de Cholesky par identification	172
6.4.4	Le programme optimisé de factorisation de Cholesky par identification	173
6.5	Réduction de Hessenberg	175
6.5.1	La méthode	175
6.5.2	Le programme de réduction de Hessenberg	176
6.6	Tridiagonalisation des matrices symétriques avec des rotations	178
6.6.1	Matrice de rotation associée à e_p, e_q	178
6.6.2	Réduction de Givens	178
6.6.3	Le programme de tridiagonalisation par la méthode de Givens	179
6.7	Tridiagonalisation des matrices symétriques avec Householder	180
6.7.1	Matrice de Householder associée à v	181
6.7.2	Matrice de Householder annulant les dernières composantes de a	181
6.7.3	Réduction de Householder	182
6.8	La méthode des trapèzes et du point milieu pour calculer une aire	183
6.8.1	La méthode des trapèzes	183
6.8.2	La méthode du point milieu	184
6.9	Accélération de convergence : méthode de Richardson et Romberg	185
6.9.1	La méthode de Richardson	185

6.9.2	Application au calcul de $S = \sum_{k=1}^{\infty} \frac{1}{k^2}$	186
6.9.3	La méthode de Romberg	188
6.9.4	Deux approximations de l'intégrale	194
6.10	Les méthodes numériques pour résoudre $y' = f(x, y)$	195
6.10.1	La méthode d'Euler	196
6.10.2	La méthode du point milieu	198
6.10.3	La méthode de Heun	199
7	Les quadriques	201
7.1	Équation d'une quadrique	201
7.2	Équation réduite d'une quadrique	202
8	Les programmes récursifs	211
8.0.1	Une liste de mots	211
8.0.2	Les mots	211
8.0.3	Les palindromes	212
8.0.4	Les tours de Hanoï	213
8.0.5	Les permutations circulaires	214
8.0.6	Les permutations	215
9	Récupérer et installer un logiciel	219