Questions à Robert Amalberti : DERIVE et la programmation fonctionnelle

Question : *Le logiciel de calcul formel DERIVE, depuis sa version 2.0, ne possède-t-il pas des possibilités de programmation un peu réduites ?*

Réponse : Il est vrai que ces procédures résident en tout et pour tout dans deux constructeurs, l'un itératif - **ITERATE** - et l'autre récursif - **IF** -.

En fait, avec un peu de pratique, et pour peu que l'on veuille bien faire abstraction de ses habitudes (éventuelles) de programmation procédurale, la programmation fonctionnelle que permettent ces deux constructeurs se révèle particulièrement riche, souvent à la fois concise et élégante, très proche de la pensée mathématique. Pour celui qui a connu les listings interminables de Pascal, quel plaisir de programmer en une seule ligne le calcul de la représentation paramétrique de la développée d'une courbe plane ou la décomposition LU d'une matrice !

De plus, on remet ainsi nécessairement en valeur une grande absente de la formation mathématique traditionnelle, à savoir la pensée récursive. Pourquoi, lorsque le langage s'y prête, calculer itérativement un objet mathématique défini récursivement ? La *FAST FOURIER TRANSFORM* de COOLEY et TUKEY a fait, depuis 1964, le tour du monde des laboratoires d'électronique et de mathématiques appliquées à l'industrie ou à la recherche, mais on continue dans l'enseignement à limiter la pensée récursi-

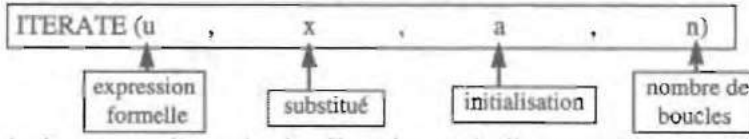
ve à un peu (très peu, dans le secondaire) de démonstration par récurrence : un exemple significatif est celui de la notion de barycentre qui est toujours définie itérativement alors que l'on glisse rapidement sur sa propriété dite d'«associativité», laquelle n'est autre que sa définition récursive, et donc un mode de calcul particulièrement performant !

Un bon exercice consiste d'ailleurs à essayer de programmer récursivement tout ce que l'on a programmé itérativement ou vice-versa, puis à comparer à la fois l'élégance des écritures et les temps de calcul. De ce côté-là, Derive réserve quelques surprises aux détracteurs de la récursivité.

Q : Peux-tu nous décrire, exemples à l'appui, les deux constructeurs de DERIVE ?

R : ITERATE

La syntaxe en est :



Il s'agit, non pas d'une simple affectation, mais d'une procédure combinée de substitution et affectation itérée n fois. Pour n positif, on peut décrire cette procédure en langage algorithmique de type pseudo-Pascal, sous la forme :

```
iterate  (u,x,a,n)
var
   y:variable formelle
   k:réel
début
   y ← a
   k ← 1
   tantque  k ≤ n
faire
      y ← subst (u,x,y)
      k ← k + 1
   fintantque
   sortie  y
fin
```

où subst(u,x,y) représente le résultat de la substitution de la variable formelle y à la variable formelle x dans l'expression u.

Q : Quelles applications peut-on en faire dans nos classes ?

ITERATE est une procédure particulièrement adaptée au calcul du $n^{\text{ième}}$ terme d'une suite définie par récurrence.

Par exemple, ci-contre, la suite $u_{n+1} = 2u_n + 1$, $u_0 = a$:

```
ITERATE(2 x + 1, x, a, 1)
2a + 1
ITERATE(2 x + 1, x, a, 2)
4a + 3
ITERATE(2x + 1, x, a, 100)
126765060022822940149670320537 6 a
+126765060022822940149670320537 5
```

ITERATE est aussi pratique pour, tout simplement, substituer une indéterminée à une autre dans un calcul formel ; par exemple, la primitive ci-dessous permet d'écrire le développement limité en h de $f(x_0 + h)$:

```
8:  DEV_LIM(f,tθ,n):=TAYLOR(ITERATE(f,t,tθ
                              +h,1),h,θ,n)
```

9: DEV_LIM$\left[\frac{\text{LOG(t)}}{\text{t}^2}, 1, 4\right]$

10: $-\frac{77h^4}{12} + \frac{13h^3}{3} - \frac{5h^2}{2} + h$

Moins évident pour qui pense uniquement en termes d'affectation, ITERATE accepte un paramètre n négatif et rend alors pour $n = -1$, la valeur de y qu'il faut substituer à x dans u pour obtenir a comme résultat (itération inverse) :

```
12: ITERATE(2x+1,x,a,-1)
```

13: $\frac{a - 1}{2}$

```
14: ITERATE(2x+1,x, ITERATE(2x+1,x,a,-1),1)
15: a
16: ITERATE(2x+1,x, ITERATE(2x+1,x,a,1), -1)
17: a
```

ITERATE permet donc d'inverser, si cela est possible, une fonction réelle d'une variable réelle :

```
18: "programmation du calcul de l'inverse de
    la fonction définie par u"
27: INVERSE(u,x) := IETARTE(u,x,x,-1)
```

28: INVERSE$\left[\frac{1 + x}{1 - x}, x\right]$

29: $\frac{x - 1}{x + 1}$

Le constructeur IF.
La syntaxe en est :

IF(clause-test, instruction1, instruction2, instruction3)			
(booléen)	(alors)	(sinon)	(non évaluation)

L'instruction 3 s'exécute en cas de non évaluation de la clause-test, généralement lorsque DERIVE compare deux expressions formelles différentes ou une expression formelle à un nombre (voir plus loin la programmation du degré d'un polynôme). On peut l'omettre (ainsi que instruction2) si l'on est sûr du résultat de l'évaluation de clause-test.
L'exemple le plus banal d'utilisation est la programmation récursive "de tête" dite encore "initiale" de la fonction factorielle :

```
31: FACTORIELLE(n):= IF(n=0,1,n FACTORIELLE(n-1))
32: FACTORIELLE(50)
33:  30414093201713378043612608166064768844377641568960512000
       000000000
```

L'appel de la fonction factorielle ainsi programmée génère un processus d'empilage (pile de récursivité) puis de dépilage que l'on peut représenter schématiquement sous la forme suivante : (pour $n = 3$, par exemple)

Factorielle (3)
3 Factorielle (2)
3 (2 Factorielle (1))
3 (2 (1 Factorielle (0))))
3 (2 (1.1))
3 (2.1)
3.2
6

Q : *N'existe-t-il pas des procédés de programmation récursive plus efficaces que d'autres ?*

R : Si, il existe par exemple la **programmation récursive dite "de queue"** (tail recursion), ou encore "terminale", qui s'avère très efficace.

Brièvement, la programmation récursive terminale consiste à définir récursivement un invariant de boucle dont les variables et les fonctions ne sont pas typées est fabuleuse. Il faut bien comprendre, en effet, qu'une fonction récursive peut s'appeler elle-même et fournir un résultat lié à son initialisation sans que la nature de l'objet qu'elle recouvre ait été parfaitement explicitée.

Dans l'exemple de la factorielle, définissons la fonction **Fact_aux(n,k)**

par **Fact_aux(n,k) = Fact_aux(n-1,nk)** pour exprimer que

$$k.[n.(n-1).(n-2) \ldots 1] = k.n.[(n-1).(n-2) \ldots 1]$$

ou encore que le résultat de la procédure que nous allons mettre en œuvre est le même en n étapes à partir de k qu'en $n-1$ étapes à partir de nk (pour calculer $1.2\ldots(n-1).n.k$), il vient :

Fact_aux(n,k) = Fact_aux(n-1,nk) = … = Fact_aux(1.2.3…n.k)

Il suffira donc d'initialiser **Fact(n,k)** en posant :

Si n = 1 alors Fact_aux(n,k) = k

pour pouvoir écrire :

Factorielle(n) = Fact_aux(n,1).

De cette façon, l'appel de factorielle (3) génère le processus schématisé ci-dessous :

Factorielle (3)
Fact_aux(3,1)
Fact_aux(2,3)
Fact_aux(1,6)
6

Et il n'y a plus de dépilage !

Programmation sous DERIVE :

```
35 : FACT_AUX(n, k) := IF(n=1, k, FACT_AUX(n-1, n k)
36 : FACTBIS(n) := FACT_AUX(N, 1)
37 : FACTBIS(50)
38 : 30414093201713378043612608166064768844377641568960512000
       000000000
```

L'avantage de ce prodcédé de programmation récursive n'est pas évident sur cet exemple mais nous verrons par la suite qu'il peut en être différemment sur des exemples plus compliqués.

Q : *Peux-tu nous en donner quelques-uns ?*

R : J'en donnerai deux classiques, celui de la suite de Fibonacci et celui du degré d'un polynôme, qui permettront de comparer les avantages respectifs d'une programmation itérative et d'une programmation récursive.

Suite de Fibonacci

La suite de Fibonacci, célèbre par ses liens avec le nombre d'or et … les petits lapins, est définie par :

$u_0 = 0$, $u_1 = 1$ et, pour n supérieur ou égal à 2, $u_n = u_{n-1} + u_{n-2}$.

Ecrivons une première programmation de type itérative utilisant la notation matricielle :

```
1: FIB1(n) := ITERATE [m . [ 1  1 ], m, [1 , 0], n]
                      [ 1  0 ]

2: FIB1(200)
3: [45397369416530795319729696969741061923382 6,
                    28057117299251014003761193241 3]
```

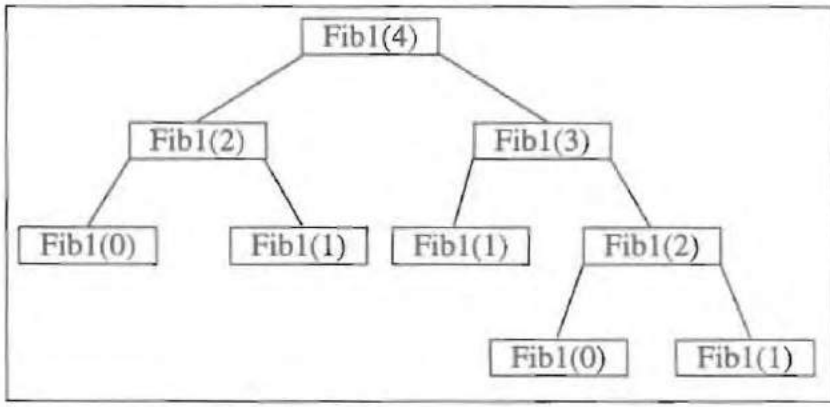
(nota : $(u_n \;\; u_{n-1}) = (u_{n-1} \;\; u_{n-2}) \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$)

`Fib0(n)` fournit le couple (u_{201}, u_{200}) en 2,7 secondes sur un 386 SX 20).

La définition de la suite de Fibonacci étant éminemment récursive, il est naturel de penser à une programmation de même nature, et la première idée qui vient en tête est d'écrire :

```
28: FIB1(N) := IF(n≤1, n, FIB1(n-1) + FIB1((n-2))
29: FIB1(15)
30: 610
```

Catastrophe ! Fib1 est incapable de calculer u_{20} et met environ 16 secondes pour calculer u_{15}. Que s'est-il passé ? En fait, nous avons créé un arbre binaire de récursion qui, à défaut de technique de mémorisation des nœuds (l'option REMEMBER de MAPLE déjà évoquée) a multiplié les calculs redondants selon le schéma suivant (pour $n = 3$) :



La "vectorisation" de la récurrence permet alors de retrouver une pile de récursivité (et non plus un arbre) sous la forme :

```
33: FIB2(n) := IF IF[n=0,[1, 0], FIB2(n-1).[ 1  1 ]]
                                            [ 1  0 ]
34: FIB2(200)
35: [45397369416530795319729696969741061923382 6,
      28057117299251014003761193 2413
```

Fib2(200) étant ici calculé en environ 4 secondes.

Essayons la récursion terminale. L'invariant de boucle peut s'exprimer de la façon suivante : u_n se calcule en n étapes à partir de α ou β ou en $n-1$ étapes à partir de $\alpha+\beta$ et α. Définissons `Fib3_aux(n,α,β)` par :

`Fib3_aux(n,α,β)=Fib3_aux(n-1,α+β,α)`
`Si n = 0 Fib3_aux(n,α,β) = β`

on a :

`Fib3_aux(n,`u_1`,`u_0`) = ..... = Fib3_aux(0,`u_{n+1}`,`u_n`)`

d'où u_n = `Fib3_aux(n,1,0)`.

```
37: FIB3_AUX(n,α,β):= IF(n=0,β,FIB3_AUX(n-1,α+β,α))
38: FIB3(n) := FIB3_AUX(n,1,0)
39: FIB3(200)
40: 28057117299251014003761193241303867718 9525
```

Et bravo pour la *«tail recursion»* puisque le calcul s'est effectué ici, malgré l'appel de deux primitives de programmation extérieures au système en 2,6 secondes !

Degré d'un polynôme

Comment faire calculer à DERIVE le degré d'un polynôme à coefficients formels (bien utile pour programmer ensuite la division euclidienne, pgcd, ppcm, etc.). Récursivement, on peut définir deg P par (P' désignant le polynôme dérivé) :

Si $P' = 0$ alors
  si $P = 0$ deg$P = -$ infini sinon deg$P = 0$
  finsi
sinon deg$P$:= deg$P'+1$
finsi

Le problème étant que DERIVE ne sait pas évaluer la clause-test $a = 0$ lorsque a est une variable formelle libre, ce qui va nécessiter une petite gymnastique de programmation utilisant l'instruction 3 du constructeur IF.

Une première programmation récursive (le mode itératif n'est vraiment

pas de circonstance ici) de type initiale ou de tête donne : (remarquer la double utilisation de l'instruction3 du constructeur IF destinée, à la fin du second IF, à pouvoir conclure que le polynôme constant a est de degré 0 et, à la fin du IF de début, à exécuter la récursion pour les polynômes de degré supérieur ou égal à 2 ou du tye $ax + b$, tandis que le 1 donnant l'instruction précédente traite le cas des polynômes $x+b$)

12: DEGRE(p) := IF$\left[\frac{d}{dx}p = 0,\ IF(p=0,\ -\infty, 0, 0), 1, 1+DEGRE\left[\frac{d}{dx}p\right]\right]$

18: DEGRE($ax^2 + bx + c$)
19: 2
13: DEGRE(ax^{100})
14: 100

En 13 : et 14 :, la réponse est obtenue en 7,5 secondes (toujours sur le même 396 SX 20).

Une deuxième programmation de type récursion de queue peut s'envisager sous la forme suivante qui consiste à créer un invariant de boucle signifiant que la somme du degré de P et d'une variable n est constante si chaque fois que l'on dérive P, on augmente la variable n de 1:

Merci de nouveau à la récursion terminale puisque DEGREBIS calcule le

32: DBIS_AUX(p, n) :=
IF$\left[p=0, n, DBIS_AUX\left[\frac{d}{dx}p, n+1\right], DBIS_AUX\left[\frac{d}{dx}p, n+1\right]\right]$

33: DEGREBIS(p) := DBIS_AUX(p, -1) + IF(p=0, $-\infty$, 0, 0)
34: DEGREBIS(ax^{100})
35: 100

degré de ax^{100} en 4,3 secondes !

Q : *Il existe, je suppose, des exemples moins courants, c'est-à-dire qu'on n'a guère l'occasion de rencontrer dans l'enseignement secondaire, et qui permettent de bien comparer ces deux méthodes de programmation.*

R : Bien sûr, et je ne résisterai pas au plaisir de vous en présenter quelques-uns.

Polynômes de Bernoulli.

La suite $(P_n)_n$ des polynômes de Bernoulli est définie par :
$P_0(X) = 1$ et pour n>1 :

$$P_n'(X) = nP_n(X) \text{ avec } \int_0^1 P_n(t)\,dt = 0$$

L'exploitation des possibilités d'itération inverse du constructeur ITERATE en donne une première programmation récursive initiale sous la forme (noter la déclaration, obligatoire de $P(n)$ comme fonction de n) :

```
7: P(n) :=
9: P(n) := IF[n=0,1,∫ n p(n-1) dx]
          + ITERATE[∫_0^1 (∫ nP(n-1) dx + a) dx, a, 0, -1]
10: VECTOR(P(k), k, 0, 4)
11: [1, x - 1/2, x^2 - x + 1/6, x(2x^2-3x+1)/2, (30x^4 - 60x^3 + 30x^2 - 1)/30]
```

Le vecteur des $p(k)$ pour k inférieur ou égal à 4 est calculé en 4 secondes.

La programmation récursive terminale consiste à définir un invariant qui exprime que le résultat de n étapes du processus à partir du polynôme p est le même qu'au bout de $n-1$ étapes à partir du polynôme

$$\int p\,dx + \text{ITERATE}\left(\int_0^1 \left(\int p\,dx + a\right) dx, a, 0, -1\right)$$

ce qui donne :

```
p:=
BERAUX(n,p):=IF(n=0,p,
          BERAUX(N-1, ∫ np dx+ITERATE(∫_0^1 ∫ np dx+a dx, a, 0, -1)))
BER(n):=BERAUX(n,1)
VECTOR(BER(k),k,0,4)
[1, x-1/2, x^2-x+1/6, x(2x^2-3x+1)/2, (30x^4-60x^3+30x^2-1)/30]
```

Le vecteur des polynômes de Bernoulli, de degré inférieur ou égal à 4 est calculé ici en 2,3 secondes.

Si P_n désigne le polynôme de Bernoulli, de degré n, on définit le n-ième nombre de Bernoulli B_n par $B_n = P_n(0)$.

L'utilisation de la formule d'Euler-Mac Laurin conduit alors à la programmation ci-dessous où Z désigne la fonction zeta de Riemann (les 16 premiers nombres de Bernoulli sont obtenus en 0,4 seconde) :

```
BERNOULLI(n):=IF(n=0,1,IF(n=1,-1/2,-nZ(1-n)))
VECTOR(BETNOULLI(k),k,0,15)
```

$$\left[1,-\frac{1}{2},\frac{1}{6},0,-\frac{1}{30},0,\frac{1}{42},0,-\frac{1}{30},0,\frac{5}{66},0,-\frac{691}{2730},0,\frac{7}{6},0\right]$$

Résolution d'un système d'équations non linéaires et intégration Gaussienne

Programmation récursive de la résolution d'un système d'équations non linéaires :

Considérons un système de n équations non linéaires aux n inconnues α_k :

$$\begin{cases} u_1(\alpha_1, \ldots, \alpha_n) = 0 \\ \ldots\ldots\ldots\ldots \\ u_n(\alpha_1, \ldots, \alpha_n) = 0 \end{cases}$$

Dans certains cas, la simple substitution itérée d'une inconnue exprimée en fonction des autres dans la suite des équations permet d'aboutir à la résolution du système. C'est ce que fait la primitive SOL (programmée récursivement en récursion terminale) ci-dessous quand on l'applique à la matrice

$$\begin{bmatrix} u_1 u_2 \ldots u_n \\ \alpha_1 \alpha_2 \ldots \alpha_n \end{bmatrix}$$

```
DIM(u):=DIMENSION(u)
EL(u,k,1):= ELEMENT(u,k,1)
SOLAUX(m,k):=IF(k=DIM(m'), ITERATE(ELEMENT(m,2),
EL(m,2,DIM(m')), ITERATE(EL(m,1, DIM(m')),
EL(m,2,DIM(m')),0,-1,),1), SOLAUX(ITERATE(m,EL(m,2,k), ITE-
RATE(EL(m,1,k), EL(M,2,k),0,1),1),k+1))
SOL(m):=SOLAUX(m,1)
```

Application à la détermination de formules d'intégration Gaussienne

Considérons la formule d'intégration numérique :

$$(1)\quad \int_0^1 f(t)\,dt = \alpha f(a) + \beta f(b) + E(f) \quad (a \text{ et } b \text{ appartenant à } \mathbb{R})$$

Le problème consiste à déterminer α, β, a et b de telle façon que la formule soit exacte ($E(f) = 0$) pour un polynôme de degré aussi élevé que possible. Une première approche consiste à fixer les points a et b aux deux extrémités de l'intervalle d'intégration et donc, à déterminer seulement α et β de façon que (1) soit exacte pour les polynômes de degré inférieur ou égal à 1.

$$AUX(u) := \int_0^1 u\,dx - \alpha \lim_{x\to 0}(u) - \beta \lim_{x\to 1}(u)$$

$$VECTOR(AUX(x^k), k, 0, 1) \qquad \begin{bmatrix} -\alpha - \beta + 1 & \frac{1}{2} - \beta \end{bmatrix}$$

$$SOL\left(\begin{bmatrix} -\alpha - \beta + 1 & \frac{1}{2} - \beta \\ \alpha & \beta \end{bmatrix}\right) \qquad \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Ceci donne la bien connue formule des trapèzes (réécrite avec les coefficients trouvés sur un intervalle $[x, y]$ quelconque):

$$\int_x^y f(t)\,dt = \frac{y-x}{2}(f(x) + f(y)) + E(f)$$

Une deuxième approche consiste à "libérer" le point b de façon à le situer au mieux pour avoir une formule (1) exacte pour les polynômes de degré inférieur ou égal à 2:

$$AUX(U) := \int_0^1 u\,dx - \alpha \lim_{x\to 0}(u) - \beta \lim_{x\to b}(u)$$

$$VECTOR(AUX(x^k), k, 0, 2) \qquad \begin{bmatrix} -\alpha - \beta + 1, & \frac{1}{2} - b\beta, & \frac{1}{3} - b^2\beta \end{bmatrix}$$

$$SOL\left(\begin{bmatrix} -\alpha - \beta + 1 & \frac{1}{2} - b\beta & \frac{1}{3} - b^2\beta \\ \alpha & \beta & b \end{bmatrix}\right) \qquad \begin{bmatrix} \frac{1}{4} & \frac{3}{4} & \frac{2}{3} \end{bmatrix}$$

On obtient alors les coefficients de la formule dite de Gauss-Radau :

$$\int_x^y f(t)\,dt = \frac{y-x}{4}\left(f(x) + 3f\left(x + \frac{2}{3}(y-x)\right)\right) + E(f)$$

Enfin, on peut "libérer" les deux points a et b de la formule (1) :

$$\text{AUX}(U) := \int_0^1 u\,dx - \alpha \lim_{x\to a}(u) - \beta \lim_{x\to b}(u)$$

$$\text{VECTOR}(\text{AUX}(x^k), k, 0, 3) \quad \left[-\alpha-\beta+1,\ -a\alpha-b\beta+\frac{1}{2},\ -a^2\alpha-b^2\beta+\frac{1}{3},\ -a^3\alpha-b^3\beta+\frac{1}{4}\right]$$

$$\text{SOL}\left(\begin{bmatrix} -\alpha-\beta+1 & -a\alpha-b\beta+\frac{1}{2} & -a^2\alpha-b^2\beta+\frac{1}{3} & -a^3\alpha-b^3\beta+\frac{1}{4} \\ \alpha & \beta & a & b \end{bmatrix}\right) \quad \left[\frac{1}{2}, \frac{1}{2}, \frac{1}{2}-\frac{\sqrt{3}}{6}, \frac{\sqrt{3}}{6}+\frac{1}{2}\right]$$

ce qui donne la bien connue formule de Gauss d'ordre 2 :

$$\int_x^y f(t)\,dt = \frac{y-x}{2}\left[f\left(\frac{x+y}{2} + \frac{y-x}{2\sqrt{3}}\right) + f\left(\frac{x+y}{2} + \frac{y-x}{2\sqrt{3}}\right)\right] + E(f)$$

Mais rien n'empêche de concocter sa propre formule (peut-être moins performante, ce qui justifierait des calculs très intéressants de majoration de la valeur absolue de E(f)). Par exemple :

$$\text{AUX}(u) := \int_0^1 u\,dx - \alpha \lim_{x\to 1/4} - \beta \lim_{x\to b} - \mu \lim_{x\to 1/2}$$

$$\text{SOL}\left(\begin{bmatrix} -\alpha-\beta-\mu+1 & -\frac{\alpha}{4}-b\beta-\frac{3\mu}{4}+\frac{1}{2} & -\frac{\alpha}{16}-b^2\beta-\frac{9\mu}{16}+\frac{1}{3} & -\frac{\alpha}{64}-b^2\beta-\frac{27\mu}{64}+\frac{1}{4} \\ \alpha & \beta & * & b \end{bmatrix}\right)$$

$$\left[\frac{2}{3}, -\frac{1}{3}, \frac{2}{3}, \frac{1}{2}\right]$$

qui correspond à

$$\int_x^y f(t)\,dt = \frac{y-x}{3}\left[2f\left(x + \frac{y-x}{4}\right) - f\left(\frac{x+y}{2}\right) + 2f\left(x + \frac{3(y-x)}{4}\right)\right] + E(f)$$

Q : *Nous ne doutons pas que la programmation récursive se montre performante dans bien d'autres cas. Peux-tu, sans entrer dans tous les détails, nous en donner un aperçu ?*

R : Evidemment, on peut utiliser avec efficacité ce procédé pour le calcul des polynôme interpolateurs de Lagrange, dont voici un exemple de calcul :
L'exemple traité ci-dessous est celui de la fonction

$$f(x) = \frac{1}{1 + 20x^3}$$

sur l'intervalle [– 1 , 1] divisé en 7, 8, 10 parties égales :

LAGRANGE($\frac{1}{(1+20x^3)}$,7)

$$\frac{941190000x^6 - 1660531800x^4 + 866834580x^2 - 146186729}{18218729}$$

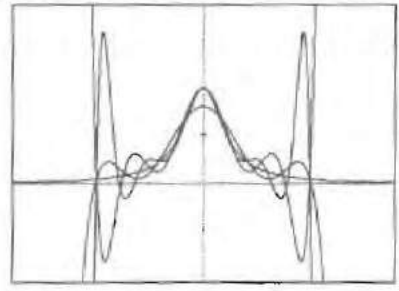
LAGRANGE($\frac{1}{(1+20x^3)}$,8)

$$\frac{1280000x^6 - 2464000x^5 + 1488200x^4 - 330080x^3 + 27783}{27783}$$

LAGRANGE($\frac{1}{(1+20x^3)}$,10)

$$\frac{2000000000x^{10} - 4500000000x^9 + 3498600000x^6 - 1163490000x^4 + 165683620x^3 - 11238301}{11238301}$$

La représentation graphique met en évidence la mauvaise qualité de l'approximation de f ainsi obtenue et de spectaculaires effets de bord décrits par Méray et Runge au début du siècle (phénomène de Runge).



Un autre exemple est pris en algèbre linéaire, avec le procédé d'orthogonalisation de Gram-Schmidt, qui permet de construire à partir d'une suite de vecteurs linéairement indépendants d'un espace vectoriel euclidien E une suite de vecteurs deux à deux orthogonaux, ou même une suite de vecteurs orthonormée.

Programmation itérative

La donnée de r vecteurs se fait sous la forme d'une matrice **A** de type (r,n) dont les vecteurs-lignes sont les coordonnées dans une base orthonormée de $(u_1 ; \ldots u_r)$.

En désignant par dB le nombre de lignes de la matrice **B** (DIMENSION(B) dans Derive), $\mathbf{B}^k$ la $k^{ième}$ ligne de **B** (ELEMENT(B,k) dans Derive), **M** la matrice courante.
La primitive d'orthogonalisation, donnant $(u_1 ; \ldots u_r)$, s'écrit alors :

Gram-Schmidt1(A):=

Itère(dA–1) fois
ajout à M du vecteur $A^{dM+1}-\sum_{i\le dM}(A^{dM+1}|M^i)/(M^i|M^i)M^i$
à partir de
$M=[A^1]$

d'où

$$NORMALISATIONVECTEUR(w):=\frac{w}{\sqrt{w.w}}$$

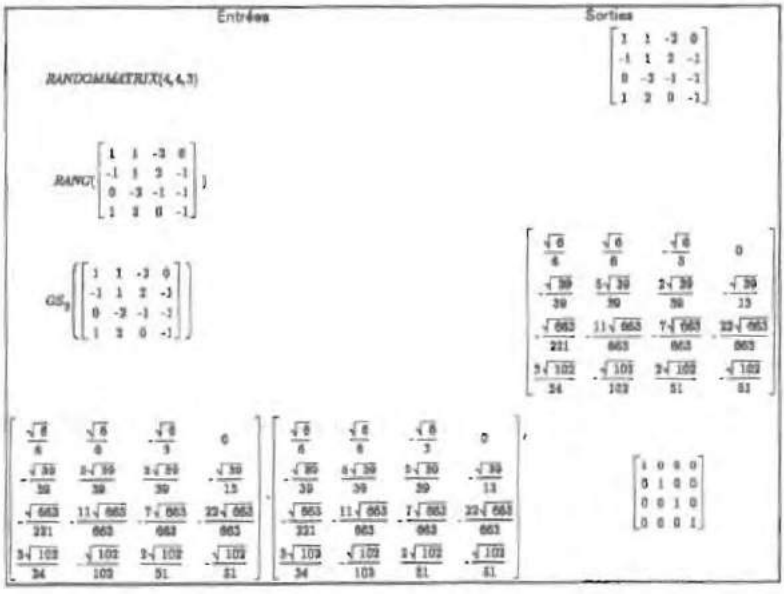
$$NORMALISATIONMATRICE(b):=VECTOR(NORMALISATIONVECTEUR(ELEMENT(b,\overline{k})),\overline{k},1,DIMENSION(b))$$

$$AJOUTVECTEUR(a,v):=VECTOR(IF(IF(k\le DIMENSION(a),ELEMENT(a,k),v),k,1,DIMENSION(a)+1)$$

$$GS_1(a):=ITERATE(AJOUTVECTEUR(m,ELEMENT(a,DIMENSION(m)+1)-\sum_{i=1}^{DIMENSION(m)}\frac{ELEMENT(a,DIMENSION(m)+1).ELEMENT(m,i)}{(ELEMENT(m,i).ELEMENT(m,i))}ELEMENT(m,i),m,[ELEMENT(a,1)],DIMENSION(a)-1)$$

$$GS_2(a):=NORMALISATIONMATRICE(GS_1(a))$$

Exemples d'utilisation



NOTE : La dernière ligne du fichier graphique ci-dessus est la vérification de

l'orthogonalité de la matrice g_s_2(A).

Programmation récursive

$$GSRAUX(a,m,k):=IF(k=0,m,GSRAUX(a,AJOUTVECTEUR(m,ELEMENT(a,DIMENSION(a),DIMENSION(a)-k+1)-\sum_{i=1}^{DIMENSION(m)}\frac{ELEMENT(a,DIMENSION(m)+1).ELEMENT(m,i)}{(ELEMENT(m,i).ELEMENT(m,i))}ELEMENT(m,i)),k-1))$$

$$GSR(a):=GSRAUX(a,[\,],DIMENSION(a))$$

$$GSR\left(\begin{bmatrix} 3 & -4 & -5 & 5 & -4 & 2 \\ -1 & -3 & 1 & 3 & 0 & -3 \\ 3 & 0 & 2 & 4 & -5 & 4 \\ 1 & -1 & 0 & 5 & -5 & 4 \\ -1 & -3 & -3 & 2 & 0 & -4 \\ 2 & 0 & -2 & 2 & -4 & 1 \end{bmatrix}\right)$$

$$\begin{bmatrix} 3 & -4 & -5 & 5 & -4 & 2 \\ -\frac{154}{95} & -\frac{233}{95} & \frac{33}{19} & \frac{44}{19} & \frac{52}{95} & -\frac{311}{95} \\ \frac{1462}{1293} & \frac{1693}{1293} & \frac{6379}{1293} & \frac{3791}{1293} & -\frac{3715}{1293} & \frac{2737}{1293} \\ -\frac{110343}{57821} & \frac{29829}{115642} & -\frac{19351}{57821} & \frac{28976}{57821} & -\frac{18136}{57821} & \frac{86797}{115642} \\ -\frac{42552}{546015} & \frac{153194}{182005} & -\frac{168606}{546015} & \frac{21934}{546015} & \frac{159316}{182005} & -\frac{153278}{182005} \\ -\frac{5029}{143524} & -\frac{31779}{287048} & \frac{8239}{287048} & -\frac{35417}{287048} & -\frac{3638}{35881} & \frac{2461}{287048} \end{bmatrix}$$