Sur la programmation rationnelle des algorithmes numériques

A. BOSSAVIT* et B. MEYER**

I. INTRODUCTION

Aux premiers temps de l'expansion du téléphone aux Etats-Unis, la compagnie ATT s'avisa qu'elle ne pouvait continuer dans cette voie fructueuse sans transformer toutes les Américaines en standardistes ; et il fallut bien inventer la commutation automatique.

On peut redouter le développement d'une situation analogue parmi les spécialistes du calcul scientifique, dont beaucoup tendent – bien malgré eux – à devenir des programmeurs à plein temps, experts, mais amers.

Or le calcul scientifique n'est ni le seul ni le premier domaine d'application de l'informatique où soient apparus des problèmes de *maîtrise de la complexité* qui, mal surmontés, conduisent à une situation où les difficultés propres à la programmation prennent le pas sur l'analyse des problèmes qu'on cherchait à résoudre en premier lieu.

L'une des causes de cette situation est la coupure trop brutale qui intervient entre les différents niveaux de la résolution d'un problème numérique sur ordinateur. Cette coupure apparaît bien dans la présentation traditionnelle des algorithmes numériques (voir par exemple [3] ou [4]), présentation qui comprend en général deux parties. La première est un exposé mathématique développant l'"idée" qui préside à la méthode employée ; elle a les caractéristiques générales de rigueur et de clarté que l'on attend d'une discussion mathématique. La seconde partie – présente seulement dans les articles et les ouvrages qui veulent présenter un aspect "pratique" – fournit un mode d'implantation de la méthode sur ordinateur ; elle consiste généralement en un programme, écrit le plus souvent en FORTRAN ou ALGOL, et accompagné de peu d'explications.

L'observation comparée de ces deux étapes ne peut que faire ressortir le caractère intuitif et peu scientifique de la seconde d'entre elles dans les présentations traditionnelles. Un programme FORTRAN cité à l'appui d'une méthode est souvent long, difficile à comprendre de prime abord, truffé de particularités liées par exemple à l'ordinateur sur lequel il a été testé par son auteur ; il contient de nombreux choix de représentation implicites (concernant par exemple le mode de rangement en mémoire des éléments d'une matrice, l'affectation d'un même tableau à deux vecteurs qui n'ont pas à être utilisés en même temps, etc.) ; il s'appuie sur toute une série d'a priori censés être "évidents", et par conséquent non documentés (test de nullité des pivots, produits scalaires en précision étendue, choix d'un traitement par lignes plutôt que par colonnes ou inversement, etc.). Toutes ces décisions de mise en œuvre, cachées et plus ou moins rationnelles, font qu'un bonne dose de foi est nécessaire au lecteur s'il veut se convaincre qu'un programme concret est bien la représentation (l'"implantation") d'une méthode mathématique exposée par ailleurs.

---

* Chef de Division au Département Traitement de l'Information et Etudes Mathématiques.
** Ingénieur chercheur au Département Méthodes et Moyens de l'Informatique.

E.D.F. – BULLETIN DE LA DIRECTION DES ETUDES ET RECHERCHES
SERIE C – MATHEMATIQUES, INFORMATIQUE n° 1, 1979, pp. 61-74.

La thèse du présent article est qu'on peut combler, au moins partiellement, ce fossé grâce aux progrès considérables effectués ces dix dernières années en matière de programmation. Nous essayerons de montrer, en nous appuyant sur un exemple de problème numérique, comment on peut obtenir des programmes par une suite de transformations systématiques, et être à même d'accorder au produit final, à sa validité et à sa "solidité", une confiance non pas absolue — là comme ailleurs, l'erreur est humaine — mais beaucoup plus élevée que dans les méthodes traditionnelles.

Nous tenterons de montrer, sur l'exemple bien connu de la factorisation de Choleski, l'intérêt de l'axiomatique des *structures de contrôle,* des méthode de *démonstration de validité* des programmes, et de *l'approche descendante* en programmation des algorithmes numériques. Il conviendrait de compléter ces techniques par l'utilisation des *types abstraits,* non abordés ici.

Les notations employées sont tirées de la référence [2] ; les principales sont définies au paragraphe II. Les autres ne devraient pas poser de difficulté à tout lecteur connaissant un langage de programmation.

II. NOTATIONS – PRINCIPES DE DEMONSTRATION DE VALIDITE

Nous utiliserons une notation algorithmique [2] destinée à permettre une expression claire des méthodes de résolution des problèmes, et facilement traduisible dans l'un des langages de programmation usuels.

II.1. Programmes et instructions

Un "programme", ou "sous-programme", a la forme générale suivante :

```
programme p   (données x , y : ENTIERS,
                        z : REEL ;
               résultats h , l : REELS ;
               données modifiées m, n, p : ENTIERS,
                                 r : LOGIQUE)
              |
              |  . . . . . corps de programme . . . . .
```

Parmi les arguments, on distingue les "données" (qui ne peuvent être modifiées par p), les "résultats" (dont la valeur est calculée par p), et les "données modifiées" (dont la valeur initiale est transmise à p qui peut la modifier).

Le "corps de programme" est une suite de déclarations et d'instructions séparées par des points virgules.

Parmi les instructions, nous aurons :

1) des affectations, de la forme

variable ← *expression*

2) des instructions conditionnelles, de la forme

```
si condition alors
              | instruction 1
sinon
              | instruction 2
```

ou simplement

si condition **alors**
| *instruction 1*

où *instruction 1* et *instruction 2* sont des instructions quelconques.

3) des *"blocs"*, de la forme

| *instruction 1 ; instruction 2 ; . . . ;*
| *instruction n*

où *instruction 1 , . . . , instruction n* sont des instructions quelconques. L'exécution d'un tel bloc est l'exécution des instructions qui le composent, dans l'ordre.

4) des boucles, de la forme

tant que *condition* ***répéter***
| *instruction*

dont l'effet est nul si *condition* est vraie, et, sinon, consiste à exécuter une fois *instruction* et à recommencer. Une seconde forme de boucle est :

pour *i* ***variant de*** *m* ***à*** *n* ***répéter***
| *instruction*

équivalant à

| $i \leftarrow m$;
| ***tant que*** $i \leqslant n$ ***répéter***
| | *instruction;*
| | $i \leftarrow i + 1$

Nous emploierons aussi la notation

pour *i* ***variant de*** *m* ***à*** *n* ***tant que*** *condition* ***répéter***
| *instruction 1*

équivalant à

| $i \leftarrow m$;
| ***tant que*** $i \leqslant n$ ***et*** *condition* ***répéter***
| | *instruction 1 ;*
| | $i \leftarrow i + 1$

5) des appels de sous-programmes, de la forme

$p(a_1, a_2, a_3, \ldots)$

où $a_1, a_2, a_3, \ldots$ sont des objets du programme appelant, de même type que les arguments correspondant définis dans le sous-programme p.

II.2. Eléments d'axiomatique des programmes

La présentation précédente est succincte et incomplète. Plus que les notations elles-mêmes, cependant, deux idées sont importantes ici :

1) Le fait que les "structures" définies permettent, par combinaison répétée, de créer des programmes aussi complexes qu'on le désire ; ainsi, dans

tant que c *répéter*
| a

a pourra être un bloc, de la forme

| a_1 ; a_2 ;
| a_3

et a_1, à son tour, peut être une instruction conditionnelle :

si c *alors*
| b_1
sinon
| b_2

etc. Le programme résultant, aussi long soit-il, possède une structure simple obtenue par répétion d'un petit nombre de mécanismes de combinaison fondamentaux, et commodément décrite par un arbre.

2) Les instructions de base choisies ont la caractéristique importante de pouvoir être caractérisées par des *axiomes,* permettant ensuite de *démontrer* des propriétés sur les programmes qu'elles servent à construire. Cette idée est l'une des plus importantes qui aient été mises en lumière par les progrès de la méthodologie de la programmation : elle conduit à considérer le texte d'un programme comme un *objet formel,* manipulable dans un certain système mathématique.

L'axiome associé à une instruction I s'écrit sous la forme :

$$\{P\}\ I\ \{Q\}$$

où P et Q, écrits sous la forme de commentaires (pour lesquels notre notation utilise des accolades " { " et " } "), sont des assertions caractérisant — en général sous forme de prédicats du premier ordre — des propriétés vérifiées par les objets du programme. Un tel axiome signifie que si l'assertion P, dite *précondition,* est vérifiée, et que I est exécutée, alors Q, dite *postcondition,* sera vérifiée après exécution.

A titre d'exemple, pour toute assertion P, l'affectation $v \leftarrow e$ vérifie

$$\{P[e \rightarrow v]\} \quad v \leftarrow e\ \{P\}$$

où $P[e \rightarrow v]$ désigne la propriété obtenue par substitution de e pour chaque occurence de v dans P.

L'axiomatique complète des structures de contrôle a été développée par Hoare [1]. Nous donnerons ici l'une des règles les plus importantes, celle qui caractérise la boucle ***tant que*** vue plus haut. Cette règle comporte en fait deux parties :

a) ***tant que*** c ***répéter*** A $\{$***non*** $c\}$

En d'autres termes, à la sortie d'une boucle *tant que,* la condition de boucle est fausse (il n'y a pas ici de précondition).

b) $\{P$ ***et*** $c\}\ A\ \{P\} \Rightarrow \{P\}$ ***tant que*** c ***répéter*** $A\ \{P\}$

Cette dernière règle indique que si une propriété quelconque P est *invariante* pour une exécution de l'instruction A — du moins dans le cas où la condition de boucle c est vérifiée — alors elle est invariante pour un nombre quelconque d'exécutions de A, donc pour la boucle ***tant que*** c ***répéter*** A.

La notion *d'invariant de boucle* joue un rôle important dans les démonstrations de validité. A titre d'exemple simple – nous en verrons de plus compliqués au paragraphe suivant –, soit le programme contenant les déclarations

variables i : *ENTIER*,
$somme$: *REEL* ;
tableau $t[1 : 100]$: *REEL*

et la boucle

$i \leftarrow 1$; $somme \leftarrow 0$;
tant que $i \leqslant 100$ ***répéter***
$somme \leftarrow somme + t[i]$;
$i \leftarrow i + 1$

On vérifiera aisément que la propriété P suivante :

$$(i \leqslant 101) \textbf{ et } \left(somme = \sum_{j=1}^{i-1} t[j]\right)$$

est bien invariante, dans l'environnement $i \leqslant 100$, pour le corps de boucle

$somme \leftarrow somme + t[i]$;
$i \leftarrow i + 1$

Comme par ailleurs, les actions d'initialisation

$$i \leftarrow 1 \ ; \ somme \leftarrow 0$$

assurent bien la validité initiale de cet invariant de boucle P, on peut affirmer qu'il reste vrai à la sortie de la boucle. Par ailleurs, la propriété *a)* nous assure qu'à cette sortie de boucle la condition de boucle $i \leqslant 100$ est fausse, donc $i > 100$. On a donc à la sortie de boucle

a) $i > 100$

et *b)* $i \leqslant 101$ ***et*** $somme = \sum_{j=1}^{i-1} t[j]$

c'est-à-dire :

$$i = 101 \textbf{ et } somme = \sum_{j=1}^{100} t[j]$$

ce qui montre que le programme calcule bien la somme des 100 éléments de t (on aurait pu exprimer le même calcul par une boucle ***pour***, à laquelle se généralise facilement la notion d'invariant).

La présentation des propriétés de la boucle ***tant que*** doit s'accompagner d'une restriction : elles ne sont valables que si la boucle se termine. La finitude d'une boucle ***tant que*** est assurée par l'existence d'un *variant*, soit v, fonction des variables du programme, et tel que

a) $v \geqslant 0$ à l'entrée de la boucle ;

b) $\{v \geqslant 0\}$ est un invariant de la boucle ;

c) chaque exécution du corps de boucle A fait décroître v strictement.

Il est important de noter que les méthodes de démonstration de programmes esquissées ici ont pour intérêt essentiel, non pas de permettre la démonstration *a posteriori* de programmes déjà écrits – exercice d'une utilité limitée –, mais d'influer sur l'écriture des pro-

grammes et de favoriser ainsi la fiabilité des produits obtenus. Il s'agit de donner un rôle de tout premier plan aux *assertions* successives, qui sont la justification même des éléments de programme venant de glisser entre elles. Ces assertions visent à expliciter à chaque instant du déroulement d'un programme les propriétés qui doient être vérifiées par les objets de ce programme. Idéalement, la suite des assertions, construite "à l'envers" (en partant de la conclusion) serait écrite avant tout élément de programme ; en pratique, on développera concurremment le programme et les assertions, qui forment la base de sa démonstration.

Cette démarche exige plus de rigueur, et un développement plus méthodique, que les approches traditionnelles. D'une part, en effet, elle oblige à expliciter, sous forme d'assertions mises noir sur blanc, nombre de connaissances et d'hypothèses sur le problème, qui étaient sans nul doute présentes à l'esprit du programmeur, mais sous forme implicite, dans l'approche usuelle. Il est clair ainsi que l'écriture

$$Y = 1 / (1 - X)$$

suppose l'assertion $\{X \neq 1\}$ (ou peut être $\{|X - 1| > \epsilon\}$), qu'on aura tout intérêt à exprimer clairement.

L'approche proposée conduit par ailleurs à mettre l'accent sur la notion de *spécification,* en insistant sur la nécessité de définir expressément et aussi complètement que possible les problèmes à résoudre avant de commencer à programmer. L'hypothèse est évidemment que c'est la première phase qui est véritablement difficile, la seconde étant de nature technique (cf. [2]). On considèrera la phase de spécification *statique* comme une activité de programmation à part entière, et le reste du processus comme une suite de transformations systématiques transformant peu à peu la spécification en un programme exécutable.

La méthode proposée tend à diminuer autant que possible la part de l'à peu près et de l'intuition dans le processus de programmation, pour en faire une suite de choix clairs et conscients.

III. UN EXEMPLE DE DEVELOPPEMENT DE PROGRAMME

Nous appliquerons les principes précédents, en leur associant l'idée de programmation descendante, à un exemple bien connu, celui de la résolution d'équations linéaires par factorisation de Choleski, en commençant par le cas des matrices triangulaires auquel cette méthode permet de se ramener.

Un mot de précaution s'impose : nous ne prétendons pas que qui ce soit ait inventé cet algorithme de la façon qui va être décrite ; s'agissant d'une méthode classique, ce serait évidemment absurde. De la même manière, nul n'affirmera que les grands théorèmes mathématiques ont été découverts à l'origine grâce à des démonstrations progressives, rigoureuses et systématiques semblables à celles qu'on trouve dans les manuels. Notre but est méthodologique et pédagogique : la démarche adoptée devrait permettre de retrouver les algorithmes en question de façon aussi sûre que possible, de bien les comprendre, de bien les faire comprendre, et peut être de se retrouver mieux armé pour l'étude et la recherche de nouveaux algorithmes.

III.1. En quoi la résolution des systèmes triangulaires est-elle "simple" ?

Comparons les deux programmes suivants :

```
programme résol (données a : n-MATRICE, b : n-VECTEUR ;
                 résultats x : n-VECTEUR, singulière : LOGIQUE)
    si A est régulière alors
        | singulière ← faux ;
        | affecter à x la solution de Ax = b
    sinon
        | singulière ← vrai
```

puis *résoltri* qui ne diffère du précédent que par l'adjonction du commentaire initial :

$$\{a[i,j] = 0 \quad si \quad j > i\}$$

Chacun conviendra que *résoltri* est plus "simple" que *résol.* En fait, la notion de "complexité" ou "simplicité" sous-jacente n'est peut être pas aussi évidente qu'il y paraîtrait à première vue. Essayons de la ramener à la complexité des assertions à satisfaire et des structures de contrôle d'un programme de résolution.

Dans le cas de *résoltri,* nous voulons que soit vraie la propriété

$$(Q) \qquad 0 < i \leqslant n \Rightarrow \sum_{j \leqslant i} a_{ij} x_j = b_i$$

à la sortie du programme. L'idée naturelle est de réaliser "progressivement" (Q). Cela signifie par exemple que nous introduisons une variable k dans le programme, destinée à progresser de 0 à n, la propriété

$$(P) \qquad (0 < i \leqslant k \Rightarrow \sum_{j \leqslant i} a_{ij} x_j = b_i) \ \textbf{et} \ (0 \leqslant k \leqslant n)$$

devant rester vraie. Or elle est vraie pour $k = 0$. Donc nous cherchons une action A qui, enchaînée avec l'incrémentation de k d'une unité, fera de P un *invariant* pour la boucle suivante :

```
tant que k < n répétér
    | k ← k + 1 ;
    | A
```

Alors *résoltri* sera

```
programme résoltri (a, b, x, singulière)
    |          k ← 0 ;
    |          {P}
    | tant que k < n répéter
    |          | k ← k + 1 ;
    |          | A
    | {P et non (k < n)}
```

L'assertion finale signifie que $k \geqslant n$ et que P a lieu, donc que $k = n$ et que Q est bien vérifiée.

Une action A satisfaisante est assez facile à trouver :

$$(A) \qquad x_k \leftarrow \left(b_k - \sum_{j<k} a_{kj} x_j\right) / a_{kk}$$

A un détail près, il est clair que $\{P \ \textbf{et} \ C\} \ A \ \{P\}$, et le programme serait correct n'était ce détail : a_{kk} peut être nul ; l'action A ne peut alors être exécutée. On peut la remplacer par l'instruction conditionnelle suivante, notée A' :

(A')

```
si a_kk = 0 alors
    | singulière ← vrai
sinon
    | A
```

et remplacer la condition C, $k < n$, par C' :

(C') $\quad k < n$ ***et*** *non-singulière*

On montre alors (par les propriétés de l'instruction conditionnelle, que nous n'avons pas introduites) :

$$\{P \textit{ et } C'\}\ A'\ \{P\}$$

et notre programme s'écrit maintenant

```
programme résoltri (a, b, x, singulière)
    | k ← 0 ;
    | singulière ← faux ;
    | {P}
    | tant que k < n et non singulière répéter
    |     | k ← k + 1 ;
    |     | si a_kk = 0 alors
    |     |     | A
    |     | sinon
    |     |     | singulière ← vrai
    | {P et non C'}
```

Le programme est donc correct. Comme nous l'avons annoncé à la section II, la démonstration de validité n'intervient pas *après* mais s'intègre au processus de construction du programme.

Il ne reste plus qu'à détailler l'action A s'il y a lieu. En FORTRAN, par exemple, la sommation qui figure dans A n'est pas une opération élémentaire et se fait par une boucle.

III.2. La factorisation de Choleski

Soit M une matrice symétrique définie positive. Nous voulons un programme résolvant $Mx = b$. D'après ce qui précède, si l'on sait trouver une matrice S triangulaire inférieure telle que

(P) $\quad SS^t = M$

le programme consistera en l'enchaînement de trois actions :

```
| Factorisation-Choleski de M ;
| Résolution de Sy = b ;
| Résolution de S^t x = y
```

et l'on dispose d'un programme pour les deux dernières, puisque S et S^t sont triangulaires. C'est évidemment pour cela qu'on pose a priori P. Notre espoir qu'une telle factorisation de M existe est fondé puisque P équivaut à :

$$(Q) \qquad 1 \leqslant j \leqslant i \leqslant n \Rightarrow \sum_{k=1}^{j} s_{jk}\, s_{ik} = m_{ij}$$

soit $n(n+1)/2$ équations pour $n(n+1)/2$ inconnues. Nous considérons donc que Q est la propriété souhaitée à la sortie du programme.

Une technique souvent fructueuse pour construire un programme consiste à partir de la conclusion — ici Q — et à en chercher une version affaiblie qui servira d'invariant de boucle. Ici l'assertion suivante est un bon candidat :

$$(Q') \quad \begin{cases} 0 \leqslant l \leqslant n \\ \textit{et} \quad 1 \leqslant j \leqslant i \leqslant l \Rightarrow \sum_{k=1}^{j} s_{jk}\, s_{ik} = m_{ij} \end{cases}$$

L'intérêt de Q' (a priori) vient de ce que :

a) pour $l = 0$, Q' est vraie trivialement

b) pour $l = n$, Q' est équivalente à notre conclusion Q.

Le problème est donc ramené à celui de trouver une action A telle que Q' soit invariante pour la boucle ci-dessous (1) :

tant que $l < n$ ***répéter***
$\quad l \leftarrow l + 1$;
$\quad A$

Le dessin de la figure 1 aide à trouver l'action A :

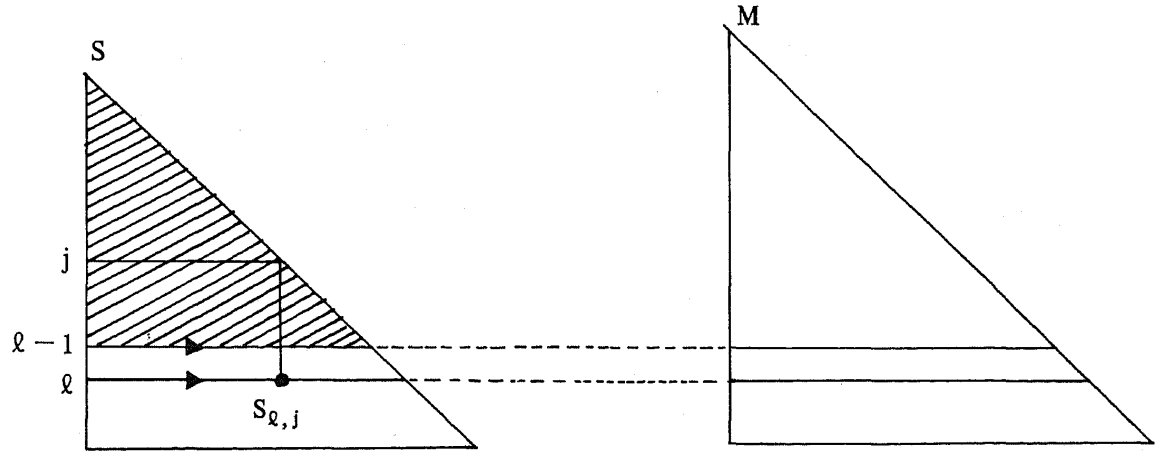

Figure 1

Lorsqu'on passe de $l - 1$ à l, les égalités suivantes doivent être réalisées *en plus* de celles qui l'étaient déjà au stade $l - 1$:

$$(R) \quad \begin{cases} \sum_{k=1}^{j} s_{jk}\, s_{lk} = m_{lj} \,, \quad \textit{pour} \quad 1 \leqslant j \leqslant l \,. \end{cases}$$

On peut raisonner sur R comme sur Q, c'est-à-dire introduire l'invariant de boucle souhaité :

$$(R') \quad \begin{cases} 0 \leqslant i \leqslant l \\ \textit{et} \quad 1 \leqslant j \leqslant i \Rightarrow \sum_{k=1}^{j} s_{jk}\, s_{lk} = m_{lj} \end{cases}$$

qui redonne R pour $i = l$. On cherche alors l'action B qui, associée à l'incrémentation de i, laisse R' invariant. C'est évidemment :

---

(1) Les deux boucles tant que qui suivent pourraient aussi s'exprimer par des boucles *pour* avec progres-

$$(B)\quad \left|\begin{array}{ll} s_{li} \leftarrow \left(m_{li} - \sum_{k<i} s_{ik}\, s_{lk}\right) / s_{ii} & \text{pour} \quad 1 \leqslant i \leqslant l \\ s_{ll} \leftarrow \sqrt{m_{ll} - \sum_{k<l} s_{lk}^2} & \text{pour} \quad i = l \end{array}\right.$$

Ceci nous donne l'action A qui servira à former le corps de la boucle principale :

$\{Action\ A\}$

$i \leftarrow 0$ $\{R'$ *est vrai*$\}$;
tant que $i < l - 1$ ***répéter*** $\{Action\ B\}$
$i \leftarrow i + 1$;
$s_{li} \leftarrow \left(m_{li} - \sum_{k<i} s_{ik}\, s_{lk}\right) / s_{ii}$;
$s_{ll} \leftarrow \sqrt{m_{ll} - \sum_{k<l} s_{lk}^2}$
$\{s_{ll}$ *n'est pas nul*$\}$ $\{R$ *est vrai*$\}$

On trouvera page 72 une version FORTRAN de cet algorithme. C'est une traduction littérale, respectant les règles de [2] ; la seule modification est l'adjonction d'une variable logique, *DEFPOS*, servant à vérifier au cours des calculs que la matrice initiale est bien définie positive (c'est le principe de la "programmation défensive" et de la "méfiance mutuelle" entre sous-programmes : lorsque c'est matériellement possible, on évite de croire sur parole une assertion d'entrée).

Le compte d'opérations est le suivant (en remontant des boucles les plus internes vers les plus externes) :

	Multiplications	Divisions	Extractions de racines
Action B	$i - 1$	1	
Boucle sur l'action B (le calcul précédent pour i variant de 1 à $l - 1$)	$\dfrac{(l-1)(l-2)}{2}$	l	
Action A (la boucle précédente plus le calcul de s_{ll})	$\dfrac{(l-1)(l-2)}{2} + l - 1 = \dfrac{l(l-1)}{2}$	l	1
Ensemble du programme (l'action A pour l variant de 1 à n)	$\sum_{l=1}^{n} \dfrac{l(l-1)}{2} \simeq \dfrac{n^3}{6}$	$\dfrac{n(n+1)}{2}$	n

On retrouve bien le compte d'opérations traditionnel.

Remarque :

Nous n'avons pas suivi la présentation traditionnelle de l'algorithme. Sans doute par imitation de l'élimination de Gauss, on travaille en général "par colonnes" : supposant connu le trapèze de la figure 2, on montre qu'il est "facile" de calculer la colonne l.

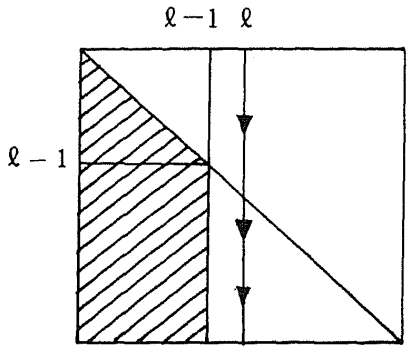

Figure 2

Pour retrouver cette variante, revenons à notre conclusion Q. Il entrait un certain arbitraire dans le choix de l'invariant de boucle Q'. Il serait aussi légitime de choisir l'invariant

$$(Q_1') \quad \left| \begin{array}{l} 0 \leqslant l \leqslant n \\ \textbf{\textit{et}} \quad j \leqslant i,\ 1 \leqslant j \leqslant l \Rightarrow \sum_{k=1}^{j} s_{jk}\, s_{ik} = m_{ij} \end{array} \right.$$

"libérant" en quelque sorte l'indice i. On trouve aisément le corps de boucle correspondant :

$$(A_1) \quad \left| \begin{array}{l} s_{ll} \leftarrow \sqrt{m_{ll} - \sum_{k<l} s_{lk}^2}\ ; \\ \textbf{\textit{pour tous les}}\ i\ \textbf{\textit{tels que}}\ l < i \leqslant n\ \textbf{\textit{répéter}} \\ \quad \left| \begin{array}{l} s_{il} \leftarrow \left(m_{il} - \sum_{k<l} s_{ik}\, s_{lk}\right) / s_{ll} \end{array} \right. \end{array} \right.$$

On écrirait facilement le programme correspondant [4].

Ces variantes doivent être étudiées de près dès qu'on se pose des problèmes de représentation physique (grande matrice rangée par lignes), en particulier en liaison avec les problèmes de pagination en mémoire virtuelle. La question des structures de données amène rapidement à s'intéresser à un autre aspect des recherches récentes en programmation : la théorie des *types abstraits,* qui permet de distinguer nettement les propriétés externes des objets manipulés (des matrices dans le cas qui nous occupe) des problèmes de représentation physique. Cette étude fera l'objet d'un article ultérieur.

IV. CONCLUSION

Qu'avons-nous obtenu ? Certainement pas un algorithme révolutionnaire. Nous avons "pris du recul" par rapport au problème de la factorisation de Choleski, et essayé de ne pas mettre la charrue avant les bœufs : il est malsain de commencer à coder un programme à l'aveuglette, en suivant son intuition ou l'habitude, sans avoir posé clairement le problème à résoudre. De l'exposé précis de ce problème, nous avons vu qu'il était possible de déduire un invariant de boucle, puis la boucle elle-même et le programme. L'invariant n'est pas déterminé de façon univoque ; différents choix, tous justifiables, mènent à différents algorithmes ; la démarche adoptée permet de comprendre en profondeur, nous semble-t-il, les points communs de ces algorithmes et leurs différences réelles.

```
      SUBROUTINE CHOFAC ( N , NMAX , M , S , DEFPOS )
      INTEGER N , NMAX
      REAL M (NMAX,N) , S (NMAX,N)
      LOGICAL DEFPOS
C  FACTORISATION DE CHOLESKI DE LA MATRICE M, M = S*TRANSP(S).
C  SI ELLE EST POSSIBLE, DEFPOS AURA LA VALEUR .TRUE., SINON, .FALSE.
C  VARIANTE PAR LIGNES, MATRICES PLEINES.
C  NMAX = DIMENSION DES COLONNES DE M ET S DANS L'APPELANT.
      DEFPOS = .TRUE.
      L = 0
C     /TANT QUE  L < N  ET  DEFPOS  REPETER/
1     IF ((L.GE.N) .OR. (.NOT. DEFPOS)) GOTO 9
            L = L+1
            I = 0
C           /TANT QUE  I < L  REPETER/
2           IF (I.GE.L) GOTO 8
                  I = I+1
C                 -PRODUIT SCALAIRE DES LIGNES I ET L :-
                        SOM = M(L,I)
                        K = 0
C                       /TANT QUE  K < I  REPETER/
3                       IF (K.EQ.I) GOTO 4
                              K = K+1
                              SOM = SOM - S(I,K)*S(L,K)
                              GOTO 3
C                       /FIN TANT QUE/
4                       CONTINUE
C                 /SI  I # L  DIVISER/
                  IF (I.EQ.L) GOTO 5
                        S(L,I) = SOM / S(I,I)
                        GOTO 7
C                 /SINON SI  SOM <= 0  MATRICE NON DEFINIE POSITIVE/
5                 IF (SOM.GT.0.) GOTO 6
                        DEFPOS = .FALSE.
                        GOTO 7
C                 /SINON TERME DIAGONAL = RACINE CARREE/
6                       S(L,L) = SQRT (SOM)
C                 /FIN SI/
7                 GOTO 2
C           /FIN TANT QUE/
8           GOTO 1
C     /FIN TANT QUE/
9     RETURN
      END
```

Enfin, la démarche même de construction de l'algorithme assure un degré de confiance en sa validité, certes pas absolu, mais sans aucun doute supérieur à tout ce que l'on peut attendre des méthodes traditionnelles.

Une réflexion méthodologique de cet ordre, et son application à l'enseignement dans l'Université et les Grandes Ecoles, nous paraissent nécessaires si l'on veut éviter aux spécialistes du calcul scientifique d'être submergés à court terme par les difficultés matérielles de la programmation, jusqu'à devenir des "opérateurs de la programmation" comme il existe des "opérateurs du téléphone".

BIBLIOGRAPHIE

[1] HOARE C.A.R. – An Axiomatic Basis for Computer Programming. *Communications of the ACM, 12,* 10, pp. 576-583 (octobre 1969).

[2] MEYER Bertrand et BAUDOIN Claude. – *Méthodes de Programmation.* Paris : Eyrolles 1978.

[3] WILKINSON J.H. – *The Algebraic Eigenvelue Problem.* Oxford : Clarendon Press 1965.

[4] WILKINSON J.H. et REINSCH C. – *Linear Algebra (Handbook for Numerical Computation, vol. II).* Berlin : Springer-Verlag 1971.