

J.-P. BENZÉCRI

F. BENZÉCRI

**Sources de programmes d'analyse de données en
langage PASCAL : (I) : analyse des correspondances
(IA) : déclarations et procédures externes**

Les cahiers de l'analyse des données, tome 22, n° 1 (1997),
p. 13-38

http://www.numdam.org/item?id=CAD_1997__22_1_13_0

© Les cahiers de l'analyse des données, Dunod, 1997, tous droits réservés.

L'accès aux archives de la revue « Les cahiers de l'analyse des données » implique l'accord avec les conditions générales d'utilisation (<http://www.numdam.org/conditions>). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM

Article numérisé dans le cadre du programme
Numérisation de documents anciens mathématiques

<http://www.numdam.org/>

**SOURCES DE PROGRAMMES
D'ANALYSE DE DONNÉES EN LANGAGE PASCAL:
(I) : ANALYSE DES CORRESPONDANCES
(IA) : DÉCLARATIONS ET PROCÉDURES EXTERNES**

[SOURCES PASCAL (IA)]

J.-P. & F. BENZÉCRI

0 Introduction à la publication des sources d'un logiciel

Depuis quelque dix ans, paraissent, dans *CAD*, les résultats d'analyses effectuées sur micro-ordinateur Macintosh, avec le logiciel MacSAIF. Si l'on met à part les premiers essais, les versions successives des programmes sont compatibles entre elles; en ce sens qu'on a conservé les formats des fichiers par lesquels communiquent entre eux les programmes d'analyse factorielle, de classification, d'affichage de graphiques, de codage et de création de tableaux, d'aides à l'interprétation, d'analyse discriminante...

Pour élaborer, après un codage approprié, le tableau décrivant quelques centaines d'individus, de diverses classes, par un ensemble hétérogène de variables, il faut, même en se bornant à ce que nous considérons comme indispensable, utiliser une dizaine de programmes différents; dont la plupart ont pour sources, outre un programme principal, diverses unités qui peuvent être communes à plusieurs programmes.

Cette complexité s'explique, en partie, parce que les micro-ordinateurs n'acceptaient, il y a dix ans, que des segments de code exécutable occupant chacun en mémoire centrale moins de 32k; mais, dans une certaine mesure, la structure ramifiée du programme a aussi l'avantage d'être conforme à celle de la méthode numérique et statistique; et de permettre d'introduire d'utiles améliorations de détail sans bouleverser l'ensemble.

Par rapport aux programmes exécutés, il y a 25 ou 30 ans, en déposant un paquet de cartes dans un grand centre de calcul, dont pourtant la puissance réelle était moindre que celle de calculateurs qu'on transporte aujourd'hui dans un cartable, les logiciels conversationnels ont ouvert à l'analyse des données des perspectives insoupçonnées. Or sont toujours utilisés, en 1997, des logiciels de statistique, héritiers de lointains prédécesseurs dont ils ont peu étendu les fonctions, même si la vitesse d'exécution et la capacité en taille en sont incomparablement plus grandes.

Ayant déjà montré, sur de nombreux exemples et diverses notices, comment opère un logiciel tel que MacSAIF, nous voudrions aussi enseigner les principes d'écriture de programmes coordonnés. Les algorithmes des méthodes multidimensionnelles que nous utilisons sont certes bien connus, au moins des lecteurs de *CAD*. Il suffit de dire qu'en dehors de calculs algébriques indéfiniment répétés mais très simples, nous utilisons: le tri par fusion (cf. §4); la diagonalisation de matrices carrées symétriques par SYMQR (cf. §5); et la CAH par l'algorithme d'agrégation avec recherche en chaîne des paires de plus proches voisins réciproques (IIB§2); ce dernier algorithme étant le seul que nous ayons contribué à fixer. Toute l'originalité d'un logiciel conversationnel réside donc dans l'enchaînement.

Cependant, il est difficile de décrire cet enchaînement par un discours général: on nous comprendra mieux en suivant le commentaire de sources écrites dans un langage compilable et ayant effectivement servi à créer des programmes exécutables. Mais, si la structure subsiste, les sources, mêmes écrites en un langage de haut niveau, doivent souvent être reprises.

L'ordinateur Macintosh a été initialement conçu en parfaite harmonie avec le langage Pascal; présentement, l'on recommande, et imposerait presque, le langage C. Beaucoup, voyant que le Macintosh, et le système MacOS, ont des rivaux plus répandus, souhaiteraient qu'on publiât des sources acceptées par ces rivaux. Mais, considérant que, d'une part, il faut publier et commenter des sources éprouvées par un long usage, et que, d'autre part, un langage de haut niveau et un système opératoire ne conviennent à un logiciel conversationnel que s'ils possèdent les fonctions de ceux que nous avons utilisés, nous ne sortirons pas, ici, du Pascal et du Macintosh.

Reste que, dans ces limites, nous avons utilisé, pour le seul Pascal: l'interpréteur MacPascal (sur Macintosh+); les compilateurs TML et TML2, Think Pascal et Code Warrior. Seul ce dernier produit des programmes utilisant toutes les ressources des processeurs PowerPC qui équipent tous les Macintosh produits actuellement. La vitesse d'exécution obtenue dépasse de beaucoup ce qu'on obtient autrement avec les mêmes algorithmes: par exemple, la CAH de plusieurs milliers d'individus ne demande que quelques minutes. Mais le nouveau système MacOS a, notamment pour la recherche de fichiers, des instructions différentes de celles répertoriées dans les éditions successives de "Inside Macintosh". On donnera donc ici les sources acceptées par le compilateur TML2. Certes, pour créer des programmes exécutables, ce compilateur ne fonctionne que sur des Macintosh munis de processeurs 680xx; mais les programmes, une fois créés, sont acceptés, à notre connaissance, par tous le Macintosh, depuis le Mac+, jusqu'au Performa. La seule précaution étant qu'avec le système 7, on doit, ainsi que l'offre la rubrique "lire les informations", configurer le programme en sorte qu'il s'exécute sur une zone de mémoire centrale suffisant aux exemples qu'on désire traiter.

Dans une première partie, (I), nous nous bornerons à l'analyse des correspondances; à l'exclusion de l'examen des données, sur histogrammes et tris croisés, et du codage. Toutefois, sans créer des graphiques plans, les sources du seul programme 'qori' comprennent six unités distinctes:

- qori.p: source principale;
- udiaq.p: diagonalisation par SYMQR;
- utrir.p: tri par fusion d'un ensemble de nombres réels;
- uens.p: choix, au sein d'un ensemble, d'un sous-ensemble tel que celui des lignes ou colonnes supplémentaires;
- ulire.p: lecture d'un tableau rectangulaire, sous tous les formats de fichier, texte ou binaire, acceptés par 'qori';
- uver.p: définitions des types de données, notamment des séquences et pointeurs; et procédures de vérification, d'affichage et de lecture; toutes choses communes à l'ensemble du logiciel.

Dans le présent article, (IA), on considère les cinq unités appelées par le programme 'qori'. Le programme principal, 'qori.p' fait l'objet d'un deuxième article, (IB). Dans (IC), sont publiés les programmes graphiques; dans ID, l'adjonction de tableaux en supplément. Viendront ensuite d'autres parties du logiciel MacSAIF: analyse discriminante (IIA), Classification (IIB) et aides à l'interprétation (IIC, D), examen et codage de tableaux de données, constructions de tableaux (III); sans exclure les programmes de tris et de concordance qui permettent de construire rapidement, à partir de textes, des tableaux destinés à l'analyse de correspondance.

C'est par l'explication de 'uver.p' que nous commençons la notice. Certes, le texte des procédures n'est qu'une suite d'instructions propres au Macintosh (et même, comme on l'a dit, périmées pour le système MacOS des PowerPC). Mais, avec les constantes et types, le lecteur connaîtra les notions que nous avons choisies; et c'est par des procédures qu'on accède à ces notions.

Il n'est pas dans notre habitude d'écrire, dans les sources mêmes, aucun de ces commentaires entre accolades {} qu'acceptent les compilateurs. Nous en dispensons un style de notation créé en ajoutant aux contraintes du langage d'autres contraintes qui en facilitent l'interprétation; principalement en ce que les variables individuelles et les tableaux ou pointeurs sont conçus non seulement comme rentrant dans l'un des types usuels d'entier ou de réel, etc.; mais comme renvoyant à des ensembles: I, J, ... Le programme lui-même édifie sur ces ensembles des tableaux et des procédures qui sont comme les objets et les morphismes d'une catégorie; ce terme étant compris comme le font les mathématiciens quand ils décrivent les structures d'une théorie. (Voir, notamment, les "*Cahiers de topologie et Géométrie différentielle catégoriques*", publiés dans la tradition de notre Maître Ch. EHRESMANN).

Les commentaires ajoutés ici aux sources, seront donc écrits comme le texte de l'article; et le lecteur comprendra que la suite des définitions et instructions se poursuit, sans interruption, en enjambant ces commentaires.

1 Notions de base du logiciel: l'unité 'uver.p'

```
UNIT uver;
INTERFACE uses mentypes, quickdraw, osintf, toolintf, sane;
const jmax =300;imax=1000;iamax=2700;ibmax=5400;ipmax=8000;xtn=12;
aamax=93;abmax=95;ddir=193;titab=256;
```

1.1 Les constantes

Dans la version présentée ici, le nombre des variables, ou colonnes du tableau de base, est fixé à $jmax=300$. Le maximum, $imax$, du nombre des lignes, n'est pas absolu: il ne concerne que les tableaux écrits en forme de texte (et non suivant le format binaire que Pascal donne aux entiers ou aux réels). Pour chaque colonne d'un tel tableau, (cf. *infra* 'ulire.p'), le programme qui le lit réserve 1000 positions; à moins qu'en tête ne soit spécifiée une autre dimension, inférieure ou supérieure à 1000, mais, en tout cas, $\leq ipmax=8000$.

Puisqu'un réel, sous le format simple de 'single', occupe 4 octets, une colonne formée de $cari$ (i.e. $cardI$) nombres occupe $4*cari$ octets; donc, si $cari=ipmax=8000$, la colonne occupe un peu moins de 32k octets (8000 est choisi pour être un nombre rond); c'est parce que l'instruction du TML2 pour demander un segment de mémoire de longueur donnée ne fonctionnant sans accident qu'au-dessous de 32k, qu'on a fixé ainsi: $ipmax$.

Pour des segments de mémoire plus longs, la souplesse est moindre: il faut définir une structure de longueur déterminée, et en demander la création. Ainsi, un sigle (cf. *infra* §§1.2 et 1.3.4) étant codé sur 6 octets, la colonne des $cari$ sigles n'est inférieure à 32k, que si $cari \leq ibmax$; au delà, on demande un segment mémoire de longueur $6*ipmax$, convenant pour les plus grands ensembles I que nous traitions.

Sous un autre format dit 'extended', le réel, est, pour plus de précision, codé sur xtn octets; selon que les instructions font ou non appel à un coprocesseur, xtn vaut 12 ou 10. Sans entrer dans ces distinctions on pose ici: $xtn=12$ (bien que, pour plus de généralité, les programmes, considérés ici n'appellent pas de coprocesseur arithmétique). Une colonne de $iaamax=2700$ réels écrits dans ce format occupe moins de 32k.

Les constantes { $aamax$, $abmax$, $ddir$, $titab$ } apparaissent dans la description de l'en-tête d'un fichier contenant un tableau codé sous forme binaire. Les valeurs sont héritées d'anciennes versions et conservées pour la compatibilité des nouveaux programmes avec les fichiers antérieurement créés. La contrainte majeure est ici, que, comme on le précisera dans la suite, l'étendue du fichier des facteurs afférents à un ensemble est limitée à $aamax=93$ facteurs; soit, au total, avec les poids et inerties, $abmax =95$ colonnes. Même si ce nombre surprend, il ne gêne pas; car les facteurs de rang élevé ne servent que pour des CAH; et, même là, 93 axes suffisent.

```

type
  zigle=packed array[0..5]of byte;pzgl=^zigle;plng=^longint;
  zgi=array[1..ipmax]of zigle;pzgi=^zgi;
  pww=^single;pint=^integer;
  tk=array[0..jmax]of ptr;ptk=^tk;
  rtj=array[1..jmax]of extended;prtj=^rtj;
  sjj=array[1..jmax]of prtj;psjj=^sjj;
  itj=array[1..jmax]of integer;pitj=^itj;
  tini=array[1..ipmax]of integer;ptni=^tini;
  tdri=array[1..ipmax]of extended;ptdri=^tdri;
  ti=array[1..ipmax]of single;pti=^ti;kji=array[0..jmax]of pti;
  ricla=record sis:zigle;Fac:array[1..abmax] of single end;
  tric=array[1..ddir]of integer;ptrc=^tric;

```

1.2 Les types de données

Pour écrire le logiciel MacSAIF, on a adjoint, aux types usuels, {entier, réel, caractère, chaîne...}, d'autres types, définis dans 'uver.p'. Il s'agit, principalement, de suites de nombres (array) ou de pointeurs, permettant de réserver en mémoire l'espace où seront logées de telles suites, et donnant accès à celles-ci.

Quand, accoutumés depuis longtemps à écrire des algorithmes en ALGOL, en laissant à des programmeurs le soin de les faire exécuter en passant par le langage FORTRAN, nous avons entrepris de travailler seuls, en PASCAL, sur un micro-ordinateur, l'accès à la mémoire du Macintosh, nous a d'abord paru très difficile. Déclarer un tableau tel quel, en tête d'un programme, n'est possible qu'avec des dimensions fixées, e.g., 50×10, et sous réserve que l'espace total demandé par l'ensemble des déclarations n'excède pas 32k.

Pourtant, un autre espace de mémoire existe; restreint à 32k dans les premiers Macintosh, mais étendu à environ un méga dès l'apparition du Mac+, et constamment extensible depuis. À cet espace, on accède par des pointeurs. L'habitude une fois prise, on apprécie cet adressage indirect. Ainsi, prenons un tableau de 100 colonnes; dont 25 éparpillées du rang 1 au rang 100 (et non groupées de 76 à 100), doivent être mises en supplément. Il suffit de renuméroter les pointeurs des colonnes, sans recopier aucunement celles-ci, pour calculer comme si les colonnes supplémentaires étaient les 25 dernières.

Plus remarquable encore, est le fait qu'un seul et même espace de mémoire centrale, avec son contenu physique parfaitement décrit, du point de vue du calcul, comme un ensemble d'octets, peut être écrit et lu sous différentes interprétations, soit comme entier, soit comme réel, soit comme comprenant du texte avec des nombres.

Pour plus de précision, considérons le type, 'ricla', qui est au bas de la liste. Cette structure comprend un 'zigle' (qui n'est autre qu'un sigle usuel, sur la représentation duquel on reviendra); puis une suite de abmax nombre réels. En fait, 'ricla' est une forme adaptée du nom 'licla' utilisé dans une version

ancienne pour signifier 'ligne pour la classification'; i.e., ligne contenant les informations requises sur chacun des éléments d'un ensemble (I, J, Is, Js, principaux ou supplémentaires; etc.). Initialement, appréciant l'accord entre la notion de type du PASCAL, et les structures considérées dans la description des théories mathématiques, nous pensions recourir systématiquement à des types complexes et variés, et créer, notamment des fichiers de 'licla', pour conserver sur disque les résultats d'une analyse factorielle et les transmettre aux programmes de CAH, de tracé de graphiques plans, etc.

En fait, les types du PASCAL, ne comportant pas de paramètres variables, ne peuvent servir à créer, pour toute analyse, des fichiers appropriés écrits sans perdre d'espace.

Supposons qu'une analyse ait produit 74 facteurs non triviaux; dont on ne veut conserver que les 50 premiers pour des applications ultérieures. La ligne afférente à un élément devrait donc comprendre 214 octets, 6 pour le sigle, et 52×4 pour les facteurs et les colonnes PDS, INR. En fait, dans l'accès à la mémoire centrale et au disque, il vaut mieux prendre pour unité le mot de 2 octets, soit un entier au format minimum (car l'accès à l'octet isolé est en butte aux caprices des compilateurs; et doit être vérifié quand on y a recours). Nous dirons donc: 107 mots (ou entiers) utiles pour chaque élément. Or un 'ricla' tel qu'on l'a défini, occupe $6+(4 \times 95) = 386$ octets = 193 mots (notons, au passage, l'origine de la constante ddir, déclarée sans commentaire au §1.1). Dans le cas considéré, en écrivant les résultats de l'analyse sur un fichier de 'ricla', on perdrait près de la moitié de l'espace occupé sur le disque.

On écrit donc un fichier d'entiers (file of integer); dont les segments de 107, afférents à un élément, sont lus dans un espace de la mémoire centrale qui est à la fois considéré comme rentrant dans le type 'tric' (array of integer) et dans le type 'ricla', selon le pointeur par lequel on y accède.

Revenons maintenant au type 'zigle', mis en haut de la liste, et figurant comme un des termes de la structure 'ricla'. Le sigle d'un élément ayant, selon nos conventions, quatre caractères, au plus, il semble naturel de déclarer un type 'sigle' = string(4), chaîne de 4 caractères. De façon précise, en PASCAL, un chaîne de p caractères ($p < 255$) occupe (p+1) octets; dont le premier indique le nombre des caractères qui doivent effectivement être lus; ces caractères étant donnés ensuite par leurs numéros dans le code ASCII étendu. Avec la version initiale du compilateur TML, le type string(4) est considéré comme occupant 3 mots (soit: 6 octets, bien que 5 suffisent). En TML2, string(4) est assimilé à 'string' (sans restriction de longueur); et occupe donc 256 octets=128 mots. Pour gérer au plus juste la mémoire, il a donc fallu déclarer le 'zigle'; dont on est sûr qu'il occupe trois mots; et qu'on peut lire comme un sigle usuel, e.g. au moyen d'un 'stringptr' (cf. *infra*, §1.3.4, les procédures: 'sigler', 'zigler').

```

procedure benzecri;
function verif (sptr0:stringptr):integer;
function dialof (sptr0:stringptr):char;
procedure verichaine (sptr0,sptr1:stringptr;var rep:char);
procedure detruire (nnn:str255);
procedure drwnum (ntg:integer);
procedure drw4 (ntg:integer);
procedure drwfl (rfl:extended);
function sigler (zig0:zigle):str255;
function zigler (sig0:str255):zigle;
procedure lir (var rep:char);
procedure lirr (var rep:char);

```

1.3 Fonctions des procédures et implementation

```

implementation
procedure benzecri;
var cadre:rect;
wpl>windowptr;
begin
textbook (thePort);
with cadre do begin left:=0;top:=38;right:=520;bottom:=342 end;
initgraf (@theport);initfonts;initwindows;
wpl:=newwindow (nil,cadre,'MacSAIF',true,0,pointer (-1),true,111);
setport (wpl);setorigin (0,0);pensize (1,1);penmode (patXor);
textfont (4);textsize (9);
moveto (5,25);
end;

```

1.3.1 Initialisation

En TML2 existe une procedure 'textbook(thePort)' qui permet, en bref, d'écrire, de lire et de dessiner dans une fenêtre de l'écran du Macintosh; le programme s'exécutant en dialogue avec l'opérateur qui répond du clavier aux questions affichées.

Cette procédure s'appelle "textbook" (i.e. *livre d'enseignement*) parce qu'elle est destinée aux étudiants qui, sans connaître les ressources propres au Macintosh, s'initient au PASCAL en exécutant les algorithmes classiques des manuels d'analyse numérique. Un développeur professionnel n'a cure de cette commodité; mais un statisticien qui, dans la Pratique de l'Analyse des Données, se tient à mi chemin entre l'abstraction mathématique et l'anatomie des machines, ne fait pas fi de "textbook"!

La procedure 'benzecri' (ainsi nommée parce qu'une version commence en affichant le nom de l'auteur) repose essentiellement sur "textbook"; mais, de plus, afin de maîtriser le tracé des graphiques (plans en analyse factorielle, histogrammes, cartes,...), elle fixe la taille de la fenêtre, indépendamment de celle de l'écran du micro-ordinateur particulier qu'on utilise.

Toujours placée en tête de programme, elle se termine en envoyant le point d'insertion du texte en une place déterminée de la fenêtre (en haut et à gauche); pour un premier dialogue qui fixe les données à traiter et arrête les modalités du traitement.

1.3.2 Existence des fichiers

```
function verif;
var nomf:str255;fin:file of integer;
vrnl,i1,i2,i3,rfnl:integer;finfl:finfo;
begin i2:=0;
if (setvol(sptr0,99)=noerr) then i2:=1;
if (i2=1) then begin
  i1:=getvol(@nomf,vrnl);
  nomf:=copy(sptr0^,length(nomf)+1,length(sptr0^)-length(nomf));
  i1:=fsopen(nomf,vrnl,rfnl);
  if (i1=noerr) then begin i3:=fsclose(rfnl);i2:=2 end end;
if (i2=2) then begin
  reset(fin,sptr0^);if eof(fin) then i2:=1;close(fin);
  if (i2=1) then i3:=fsdelete(nomf,vrnl) end;
verif:=i2;
end;
```

Recevant, par un pointeur, le nom d'un fichier, tel que Disq:dos:fich, la fonction 'verif' répond en prenant la valeur zéro, s'il n'y a même pas de disque du nom donné (ici: 'Disq'); la valeur 1, si le disque existe, mais non le fichier; la valeur 2, si le fichier existe sur le disque. Une difficulté est que, pour autant que nous sachions, toute procédure qui s'enquiert de l'existence d'un fichier sur un disque, crée le fichier s'il n'existe pas; en sorte qu'on n'a de l'absence du fichier qu'une preuve indirecte, qui est qu'on se trouve à la fin de celui-ci (end of file = eof), qui est vide; on doit donc, en ce cas, détruire (par: fsdelete) ce fichier vide, inopportunément créé.

```
function dialof;
begin dialof:='O';
if (verif(sptr0)=0) then begin dialof:='N';
  writeln('ERREUR le disque manque') end
else if (verif(sptr0)=1) then begin dialof:='N';
  writeln('ERREUR le fichier manque') end;
end;
```

La fonction 'dialof' affiche à l'écran le résultat de la recherche de fichier effectuée par 'verif'; et elle convertit la réponse en un caractère 'O'= Oui, ou 'N'=Non.

```
procedure verichaine;
var str2:string;
begin
if (rep='O') then writeln(' BIEN');
if not(rep='N') then begin
  str2:=concat(sptr0^,sptr1^);
  write('verification pour ',sptr1^,'':8-length(sptr1^));
  rep:=dialof(stringptr(@str2)) end;
end;
```

La procédure 'verichaine' sert lorsqu'à propos d'un fichier de base unique, l'exécution d'un programme requiert l'existence de plusieurs fichiers contenant des données et les résultats de calculs déjà faits. Ainsi dans le programme 'discr' d'analyse discriminante, la procédure 'principer' appelle 'verichaine', pour vérifier l'existence d'une série de 4 fichiers (cf. ID§1).

```

procedure detruire;
var vrn1,ii:integer;nnf,nnc:string;sptr0:stringptr;
begin
  sptr0:=stringptr(@nnc);nnc:=nnn;
  ii:=setvol(sptr0,99);ii:=getvol(@nnf,vrn1);
  nnf:=copy(sptr0^,length(nnf)+1,length(sptr0^)-length(nnf));
  ii:=fsdelete(nnf,vrn1);
end;

```

Eventuellement, on doit détruire un fichier qu'on a créé.

1.3.3 Écriture de nombres à l'écran

La procédure "textbook", exécutée en tête du programme, permet à celui-ci d'écrire, sur l'écran, des lignes successives de texte, par les ordres usuels du PASCAL: 'write' et 'writeln'. De même, par l'ordre 'readln', on lit une suite de caractères, se terminant par un aller à la ligne (CR), et affichée à l'écran, à partir du clavier.

Mais, en TML2, pour plus de rapidité, les ordres 'writeln' ne sont exécutés que par blocs de lignes; et non dès leur apparition dans le cours du calcul. Si l'on utilise 'writeln' pour créer un graphique où des écritures de texte se mêlent à des tracés de lignes et de polygones propres au Macintosh, le résultat est incohérent. (En fait, le seul moyen de s'assurer qu'une suite d'ordres 'write' est exécutée jusqu'au bout, est de la faire suivre d'un ordre 'readln' qui implique la mise à jour de l'écran).

C'est pourquoi, on utilise l'ordre 'drawstring'; d'autant plus que, sur un graphique complexe, l'insertion des segments de texte, ne peut se faire au fil des aller à la ligne, mais doit être commandée par des ordres explicites 'moveto', propres au Macintosh.

Or tandis qu'avec 'write' ou 'writeln', le programme commande l'écriture, non seulement de chaînes de caractères explicitement spécifiées, mais aussi de nombres entiers ou réels, convertis en texte par des procédures appelées par TML2, l'ordre 'drawstring' demande une chaîne en laquelle il revient au programmeur de convertir les nombres qu'il désire afficher.

```

procedure drwnum;var catnum:string;ctg:integer;begin
  catnum:='';ctg:=ntg;
  while not(ctg=0) do begin
    catnum:=concat(chr(48+(ctg mod 10)),catnum);
    ctg:=ctg div 10 end;
  if (ntg=0) then catnum:='0';
  drawstring(catnum);
end;

```

Telle est la raison d'être des procédures {drwnum, drw4, drwf1}. Il est facile de voir comment 'drwnum' convertit un entier positif en un chaîne.

Le cas de 'drw4' est particulier: la procédure ne sert que pour des nombres positifs inférieurs à 10000; et, afin de respecter l'alignement des

```

procedure drw4;var catnum:string;ctg:integer;begin
  catnum:='';ctg:=ntg;
  while not (ctg=0) do begin
    catnum:=concat(chr(48+(ctg mod 10)),catnum);
    ctg:=ctg div 10 end;
  while (length(catnum)<4) do catnum:=concat(' ',catnum);
  drawstring(catnum);
end;

```

colonnes dans un tableau, la chaîne est complétée, s'il y a lieu, par des blancs, jusqu'à comprendre 4 caractères.

```

procedure drwfl;var catfl:string;sgn:char;xp,ml:integer;
begin
  sgn:=' ';if (rfl<0) then sgn:='-';catfl:='';
  if (rfl=0) then catfl:=' 0  ';
  if ((9.9e+9<rfl) or (rfl<-9.9e+9)) then catfl:='inf';
  if (catfl='') then begin
    xp:=0;
    if (rfl<0) then rfl:=rfl;
    if (rfl<1) then catfl:='- ' else catfl:='+';
    while (rfl<1) do begin xp:=xp+1;rfl:=rfl*10 end;
    while not (rfl<10) do begin xp:=xp+1;rfl:=rfl/10 end;
    catfl:=concat('e',catfl,chr(48+xp));
    ml:=trunc(rfl*100);
    catfl:=concat(chr(48+(ml mod 10)),catfl);ml:=ml div 10;
    catfl:=concat(chr(48+(ml mod 10)),catfl);ml:=ml div 10;
    catfl:=concat(chr(48+(ml mod 10)),'.',catfl);
  end;
  catfl:=concat(sgn,catfl);
  drawstring(catfl);
end;

```

La procedure 'drwfl' écrit un nombre réel, rfl, de signe quelconque, sous le format flottant, comme un chaîne de longueur 8. (Font seuls exception, les nombres dont la valeur absolue excède 9.9e+9, et qui sont notés 'inf', infini). En tête, le caractère sgn donne le signe. Le nombre donné est mis sous la forme du produit d'une puissance de 10, dont l'exposant comprend un seul chiffre précédé d'un signe, par un nombre réel dont le module est ≥ 1 et < 10 . Ce module, multiplié par 100, est écrit en 3 chiffres, dont le premier est suivi d'un point.

1.3.4 Les sigles, comme chaînes ou suites d'octets

```

function sigler;var str0:str255;il:integer;
begin
  for il:=0 to 5 do str0[il]:=chr(zig0[il]);
  sigler:=str0;
end;
function zigler;var zig0:zigue;il:integer;
begin
  for il:=0 to 5 do zig0[il]:=ord(sig0[il]);
  zigler:=zig0;
end;

```

Ainsi qu'on l'a dit au §1.2, un sigle déclaré comme string(4), occuperait, dans la mémoire gérée par TML2, la même place qu'une chaîne de 255 caractères. C'est pourquoi est défini le type 'zigue', suite de [0..5] octets. En

fait, dans la mémoire, le zigle et la chaîne ne sont qu'une même suite d'octets; dont le premier donne la longueur du sigle, et les suivants, les caractères par leurs numéros du code ASCII. Entre zigle et sigle, la seule différence est dans le format de lecture: d'où les deux procédures de conversion, 'sigler' et 'zigler', inverses l'une de l'autre.

1.3.5 Lecture d'une réponse à l'écran

```

procédure lir;
var P:point;
zz,cc,ww:integer;
vnt:eventrecord;boule:boolean;
begin
zz:=1;ww:=0;
while not (zz=0) do begin
  boule:=getnextevent (keydownmask, vnt) ;
  cc:=bitand(vnt.message, charcodemask);
  if (32<cc) and (ww=0) then begin rep:=chr(cc);
    ww:=1;drawchar(rep);getpen(P);moveto(2,P.v+11) end;
  if (cc=13) and (1=ww) then zz:=0; end;
end;
procédure lirr;
var P:point;
zz,cc,ww:integer;
vnt:eventrecord;boule:boolean;
begin
zz:=1;ww:=0;
while not (zz=0) do begin:
  boule:=getnextevent (keydownmask, vnt) ;
  cc:=bitand(vnt.message, charcodemask);
  if (32<cc) and (ww=0) then begin rep:=chr(cc);
    ww:=1;drawchar(rep);end;
  if (cc=13) and (1=ww) then zz:=0; end;
end;
end.

```

Une commande propre au Macintosh, permet, à partir du clavier, de commander la création d'un fichier graphique qui représente l'ensemble de l'écran, en traduisant les points blancs et noirs par une suite ordonnée de chiffres 0 et 1 (bitmap). Ainsi, peuvent être saisis un graphique usuel d'analyse factorielle, un histogramme, un tableau de tri croisé, une carte..., affichés à l'écran par le programme.

Mais, en TML2, cette commande ne peut s'exécuter quand le programme a envoyé un ordre 'readln' de lecture à l'écran. Or, afin que se poursuive, ou s'achève l'exécution du programme, tout graphique ou tableau, affiché à l'écran, doit nécessairement être effacé par un ordre reçu du clavier. Il faut donc disposer d'ordres de lecture, autres que 'readln', écrits avec des instructions propres au Macintosh. Les procédures 'lir' et 'lirr' lisent et affichent une réponse formée d'un seul caractère; 'lir', à la différence de 'lirr', met à la ligne le point d'insertion (par un 'moveto').

On notera que le listage de 'uver.p' se termine par trois 'end'; qui ferment, successivement, la boucle: while not(zz=0)..., la procédure 'lirr', et l'unité 'uver' elle-même.

```

UNIT ulire;
INTERFACE uses memtypes, quickdraw, osintf, toolintf, sane, uver;
procedure litab (var pj:ptr; pk, pn, pt:ptr; var rc:char;
                var ic, jc:integer; var pi:ptr);

```

2 Lecture d'un tableau: l'unité 'ulire.p'

L'unité 'ulire.p' ne déclare pas de constante ou de type qui lui soit propre; car elle renvoie à 'uver.p' qui contient toutes les déclarations qui lui sont nécessaires. Et son interface ne mentionne qu'une seule procédure 'litab'; dont toutefois l'implémentation comprend deux procédures qui lui sont propres, 'lecsig' et 'lecnombre'.

2.1 Arguments de la procédure 'litab'

Les arguments de la procédure 'litab' comprennent plusieurs pointeurs (ptr), un caractère (char), deux entiers (integer), dont on doit d'abord expliquer les rôles.

Quand 'litab' est appelée par un programme qui requiert un tableau de données, l'opérateur doit, dans l'exécution de 'litab', choisir ce tableau par un dialogue à l'écran. Si le choix parvient à se fixer sur tableau qui existe effectivement sur un disque, la réponse rc est: 'O' (oui); sinon la réponse est: 'N' (non).

En cas de succès, les entiers: ic et jc, donnent les cardinaux des ensembles, I et J, des lignes et des colonnes; les pointeurs: pn et pt, pointent vers des chaînes de caractères qui sont, respectivement, le nom du fichier contenant le tableau et le titre de celui-ci.

Plus exactement, quant au nom du tableau, il faut prendre garde que si, dans celui-ci, les données numériques ne sont pas des entiers écrits tels quels, comme texte, le nom comprend un suffixe: 'yy', pour des réels écrits, comme texte, en virgule fixe ou flottante; 'z' ou 'w', pour des entiers ou réels écrits en binaire. Ce suffixe n'est pas transmis avec le pointeur pn.

Quand on appelle 'litab' les pointeurs pj et pi ne pointent vers aucune information pourvue de sens; mais, en cas de succès, ils pointent, respectivement, vers la suite des sigles des lignes et des colonnes.

Le pointeur pk, quant à lui, pointe initialement vers une suite de jmax pointeurs: en cas de succès, ces pointeurs (ou, plutôt, les premiers d'entre eux, en nombre égal à l'entier jc, nombre des colonnes, augmenté de 1) donnent accès aux colonnes du tableau lu par 'litab' (et à une colonne initiale, n°0).

Dans certains cas (,mais non en analyse des correspondances), la procédure 'litab' est appelée pour mettre en mémoire centrale un tableau auquel le programme doit adjoindre des lignes: le nombre maximum de celles-ci est alors mis dans 'ic', afin que 'litab' réserve la mémoire appropriée. Sinon, lors de l'appel, afin d'épargner la mémoire centrale, ic doit être nul.

2.2 Variables déclarées dans la procédure 'litab'

```
implementation
procedure litab;
var fin:file of integer;ft:text;paux:ptr;
    sig:zicle;carac,rpf:char;nomf:string;ffl,ff2:longint;
    erl,c,j,i,il,iz1,iz2,cartj,carti,memi,sgn:integer;rrt:real;
    sgli:pzgl;ttri,sptit,sptr0:stringptr;ffli:plng;wwi:pw;
    tampi:array[1..titab]of integer;
    sigli,siglj:pzgi;k1ji:kji;ptlk:ptk;
```

De l'écriture du tableau à lire, dépend le type du fichier servant à y accéder: 'text', pour un texte; mais 'file of integer' dans le cas d'un codage binaire; car, ainsi qu'on l'a dit au §1.2, les données de tout type peuvent être transférées comme des suites d'entiers (ou mots de deux octets), dont la valeur, de nombre ou de texte, est saisie par des pointeurs appropriés.

Outre les deux fichiers 'fin' et 'ft', sont déclarées des variables dont la plupart rentrent dans les types usuels du PASCAL (et de tout langage de même classe): 'char' (caractère); 'string' (chaîne de caractères); 'integer' (entier écrit sur 4 octets, compris de -32768 à +32767); 'longint' (entier écrit sur 8 octets, compris entre -2147483648 et +2147483647);...

Les nombres réels offrent quelque difficulté: nous utilisons, d'une part, un format court, 'single' (sur 4 octets); et, d'autre part, un format long 'extended'; ce dernier format, qui occupe selon les cas, 10 ou 12 octets (cf. §1.1, la constante xtn), doit être distingué du 'double' (8 octets) généralement utilisé en dehors des Macintosh. Le terme de 'real' est diversement compris selon les compilateurs: pour TML2, c'est un 'extended'; et les procédures offertes par TML2 pour lire ou écrire un réel, n'opèrent correctement que pour un 'extended'; en particulier, si w1 est déclaré comme 'single', la procédure write(ft,w1) n'écrira pas correctement: mais il suffit de demander: write(ft,0+w1); car l'opération d'addition, toujours effectuée par le processeur en format 'extended', donne à 0+w1 le format approprié. Le lecteur se souviendra de cette remarque pour excuser, dans la suite, la bizarrerie de certaines instructions.

Les divers types de pointeur propres à MacSAIF sont décrits au §1.2. Il vaut la peine de commenter de ce point de vue la dernière ligne du tableau des variables de 'litab'. Les pointeurs sigli et siglj pointent chacun vers une suite de 'zicle': on verra, dans la suite, comment ils correspondent aux pointeurs, non typés, pi et pj, qui figurent parmi les arguments de 'litab'. De même, ptlk et k1ji sont liés au pointeur, non typé, pk, autre argument de 'litab'. Par ptlk, la suite de pointeurs, non typés, vers laquelle pointe pk est explicitement livrée à l'usage de la procédure 'litab'; avec k1ji, ces pointeurs sont typés, i.e. considérés comme pointant, chacun, vers une suite de 'single'. Finalement, de façon précise, k1ji[j]^i désignera le i-ème 'single' de la suite vers laquelle pointe le j-ème pointeur k1ji[j] (propre à la j-ème colonne). On notera que les arguments entiers i et j, se trouvent écrits dans le même ordre (j,i) que dans le nom k1ji: j=colonne; i=ligne.

2.3 Procédures de lecture sur un fichier de texte

```

procEDURE lecsig;begin sig[0]:=0;
  while (ord(carac) in [0..32]) and not eof(ft) do read(ft, carac);
  while not ((ord(carac) in [0..32]) or (sig[0]=4)) do begin
    sig[0]:=sig[0]+1;sig[sig[0]]:=ord(carac);
    if eof(ft) then carac:= ' ' else read(ft, carac) end;
  while not ((ord(carac) in [0..32]) or eof(ft)) do read(ft, carac);
  if eof(ft) then carac:= ' ';end;

```

On a dans TML2 des procédures de lecture de chaîne se terminant par un aller à la ligne (CR); désirant lire des sigles qui peuvent se trouver en toute position, nous avons dû écrire une procédure particulière, 'lecsig'. On notera que la lecture du sigle se termine soit, devant un caractère spécial (dont le code en ASCII est entre 0 et 32; i.e. un blanc, un aller à la ligne); soit parce que 4 caractères ont déjà été lus (i.e. sig[0]=4). Mais, dans ce dernier cas, la lecture des caractères se poursuit sur le fichier de texte ft, jusqu'à rencontrer un signal d'arrêt. La raison en est qu'en tête des informations (écrites sur une ou plusieurs lignes) afférentes à un individu i, le texte des données comporte souvent, non un simple sigle de 4 lettres au plus, mais un mot plus long, dont la lecture préalable peut faciliter la lecture des nombres qui suivent.

Car la lecture des nombres est délicate. Certes, il y a, à cet effet, dans TML2 une procédure 'read'; que nous utilisons, faute de mieux, pour lire les nombres réels écrits, sous tout format, dans un texte; mais l'ordre read(ft,rrt) ne s'exécute correctement que s'il ne rencontre devant lui, avant le nombre proprement dit, que des blancs ou des aller-à-la-ligne. Notre procédure 'lecsig' assure cette condition même si le nom de l'individu est un mot (connexe) de longueur supérieure à 4.

```

procEDURE lecnombre;begin ffl:=0;sgn:=1;
  while not ((ord(carac) in [48..57]) or eof(ft)) do begin
    read(ft, carac);
    if not (ord(carac) in [48..57]) then sgn:=1;
    if (carac='-') then sgn:=-1;end;
  while (ord(carac) in [48..57]) do begin
    ffl:=(10*ffl)+(ord(carac)-48);
    if eof(ft) then carac:= ' ' else read(ft,carac) end;
  ffl:=sgn*ffl; end;

```

Quand le texte des données ne comporte que des entiers (même affectés d'un signe: ce qui n'est pas directement admis en analyse des correspondances; mais est acceptable avant codage par 'zrang': cf. IIIA), ces entiers sont lus par la procédure 'lecnombre': celle-ci, absorbe, sans incident, tous les caractères non numériques qu'elle rencontre; et n'affecte le signe négatif à un nombre que si la séquence des chiffres est immédiatement précédée par le caractère '-'. Ainsi, dans un tableau de données entières, on peut, dans les informations afférentes à un individu i, insérer tout commentaire ne comportant pas de chiffre. La présence de chiffres étant toujours permise à l'intérieur d'un sigle (ou même du nom, connexe, de longueur quelconque, qui comprend celui-ci).

2.4 Choix du tableau à lire

```

begin
  sptr0:=stringptr(pn);
  rc:='N';erl:=0;
  while not ((rc='O') or (erl=6)) do begin rc:='O';
    write('le fichier des données est ');
    readln(nomf);sptr0^:=nomf;
    iz1:=verif(stringptr(@nomf));iz2:=0;
    if (iz1=1) then begin nomf:=concat(sptr0^,'yy');
      iz2:=verif(stringptr(@nomf));
      if (iz2=2) then begin iz1:=2;iz2:=3 end end;
    if (iz1=1) then begin nomf:=concat(sptr0^,'.z');
      iz2:=verif(stringptr(@nomf)) end;
    if (iz2=1) then nomf:=concat(sptr0^,'.w');
    rc:=dialof(stringptr(@nomf));
    if (rc='N') then erl:=erl+1 else begin writeln(nomf);
      write('ce nom est-il confirmé oui(O) ou non(N) ');
      readln(rc) end end;

```

La procédure 'litab' proprement dite commence par un dialogue au cours duquel est éventuellement fixé le nom du fichier contenant le tableau.

Supposons qu'un nom, tel que 'Disq:Dosx:tab', a été proposé par l'utilisateur: ainsi qu'on l'a dit au §2.1, le jeu des suffixes permet de comprendre sous ce nom, divers formats.

Le programme, appelant la fonction 'verif', décrite au §1.3.2, cherche d'abord un fichier de ce nom (texte où les données sont des entiers); et, s'il ne le trouve pas (sous réserve qu'au moins, le disque, 'Disq', existe: iz1=1), il cherche un fichier avec le suffixe 'yy' (texte où les données sont écrites en format numérique quelconque); puis, en cas de second échec, on essaye le suffixe '.z' (fichier binaire où les données sont des entiers); et, finalement, le suffixe '.w' (fichier binaire où les données sont des réels).

Si, un fichier a été trouvé, son nom, avec le suffixe éventuel, est soumis à l'approbation de l'utilisateur: rc='O', oui; ou 'N', non. Sinon, le nombre, erl, des propositions erronées est augmenté d'une unité. La boucle 'while not' est parcourue, sous la réserve que erl<6, tant qu'un fichier n'a pas été trouvé et accepté par l'utilisateur.

Au sortir de la boucle 'while not', la procédure ne peut se poursuivre que si, un fichier ayant été accepté: rc='O'. Les entiers iz1 et iz2 donnent alors le type de ce fichier: on a respectivement:

```

pour Disq:Dosx:tab, texte avec des entiers: iz1=2 ; iz2=0;
pour Disq:Dosx:tabyy, texte avec des nombres quelconques: iz1=2 ; iz2=3;
pour Disq:Dosx:tab.z, fichier binaire d'entiers: iz1=1 ; iz2=2;
pour Disq:Dosx:tab.w, fichier binaire de réels: iz1=1 ; iz2=1;

```

C'est d'après ces résultats qu'on procède ensuite à la lecture.

2.5 Lecture du tableau des données

```
if (rc='0') then begin
  ptlk:=ptk(pk);sptit:=stringptr(pt);
  ffli:=plng(@tampi);wwi:=pww(@tampi);
  sgli:=pzgl(@tampi);ttri:=stringptr(@tampi);
  pj:=newptr(6*jmax);siglj:=pzgi(pj);end;
```

Pour nous, la lecture d'un tableau s'effectue en mettant en mémoire centrale les données énumérées ci-après:

Le titre: chaîne de caractères vers laquelle pointe le pointeur: sptit, créé en donnant à pt le type 'stringptr'.

Les sigles des lignes et colonnes: avant toute lecture, on réserve pour les sigles des colonnes, j, l'espace maximum, correspondant à jmax colonnes; le pointeur pj, y afférent, recevant le type 'pzgi', est nommé siglj. Quant aux sigles des lignes, leur nombre (ou seulement une borne supérieure de ce nombre) ne peut être fixé qu'ensuite, d'après le contenu du fichier des données.

Les données numériques elles-mêmes: on sait qu'à chaque colonne correspond un pointeur: *a priori*, on a prévu un tableau de jmax pointeurs, vers lequel pointe ptlk, qui n'est autre que l'argument pk (cf. *supra* §2.1), muni du type approprié, ptk. Mais les pointeurs ptlk^[j] ne pointent, eux-mêmes, présentement vers aucun espace déterminé: pour leur assigner un tel espace, il faut, ici encore, fixer (ou majorer) le nombre ic des lignes.

La lecture, ainsi préparée, du fichier des données se poursuit suivant deux voies différentes selon qu'on lit un fichier binaire (izl=1) ou un fichier d'entiers (izl≠1). Dans le premier cas, toutes les informations sont lues sur 'fin' ('file of integer'), dans le tableau 'tampi' qui sert de tampon; et elles sont comprises, en pointant sur l'adresse initiale @tampi, des pointeurs ayant le type approprié: ttri, pour le titre; sgli, pour les sigles; ffli, pour un entier long (cas du suffixe '.z'); et wwi, pour un réel simple ('single'; cas du suffixe '.w').

2.5.1 Lecture d'un tableau en format binaire

Une fois ouvert le fichier des données, au format d'entiers, le programme lit d'abord dans, 'tampi', un bloc d'entiers dont la longueur est 'titab' (=256).

Le bloc commence par les octets d'une chaîne de caractères, laquelle n'est autre que le titre. Ce titre, sitôt lu, est affiché à l'écran, par l'instruction: writeln (sptit^).

Dans le cas particulier où le fichier à lire est un fichier de facteurs (cf. *supra*, §1.2, explication du type 'ricla'), cas qui n'est pas celui où le tableau lu doit être analysé, mais se rencontre pour une Classification, un tracé de graphique plan, vient ensuite, dans ce même bloc, la suite des valeurs propres.

Enfin le dernier entier n'est autre que le nombre des individus.

```

if (iz1=1) and (rc='0') then begin
  reset(fin,nomf);
  for c:=1 to titab do read(fin,tampi[c]);
  sptit^:=ttri^;writeln(sptit^);
  memi:=tampi[titab]+ic;read(fin,jc);
  writeln(memi:6,' x',jc:4);write(jc:4);
  if (memi<ibmax) then begin
    paux:=newptr(6*memi);pi:=paux;sigli:=pzgi(paux);end
  else begin new(sigli);pi:=ptr(sigli);end;
  for j:=0 to jc do begin
    paux:=newptr(4*memi);ptlk^[j]:=paux;k1ji[j]:=pti(paux) end;
  for j:=1 to jc do begin
    for c:=1 to 3 do read(fin,tampi[c]);siglj^[j]:=sgli^;
    if (j<14) then write(sigler(siglj^[j]):6) end;
  i:=0;writeln;
  while not (eof(fin) or (i=memi)) do begin i:=i+1;
    for c:=1 to 3 do read(fin,tampi[c]);sigli^[i]:=sgli^;
    write(sigler(sigli^[i]):4);
    for j:=1 to jc do begin read(fin,tampi[1]);read(fin,tampi[2]);
      if (iz2=1) then k1ji[j]^ [i]:=wwi^ else k1ji[j]^ [i]:=ffli^;
      if (j<14) then write(round(0+k1ji[j]^ [i]):6) end;
    writeln end;
  ic:=i;close(fin) end;

```

Ce nombre, `tampi[titab]`, n'est toutefois pas accepté tel quel pour calculer la valeur, `memi`, du cardinal à affecter aux colonnes: car, ainsi qu'on l'a dit au §2.1, la procédure 'litab' peut être appelée avec une valeur de `ic` positive qui désigne un nombre supplémentaire de lignes qui doit être prévu, en mémoire centrale, au-delà des lignes effectivement lues. Toutefois, en analyse des correspondances, la procédure 'litab' est appelée avec `ic=0`. Le nombre, `jc`, des colonnes du tableau des données est lu, ensuite, sur le fichier 'fin'. Et l'on affiche à l'écran les dimensions prévues, en mémoire centrale, pour le tableau.

Pour réserver la mémoire pour la colonne des sigles et pour les colonnes numériques des variables, on utilise un pointeur auxiliaire 'paux'; auquel est ensuite attribué le type approprié. Pour les sigles; l'espace requis est de $6 \times \text{memi}$ octets; par l'instruction 'newptr', TML2 ne peut réserver plus de 32k; ce qui correspond à `memi=ibmax=5400`; au-delà, on doit créer par un ordre 'new' la structure 'sigli' qui convient jusqu'à la valeur `memi=ipmax=8000`. (Pour la définition des constantes, nous renvoyons au §1.1). Les `jc` colonnes numériques sont créées avec un segment de mémoire $4 \times \text{memi} < 32k$. Plus précisément, est créée une colonne 0, qui ne contient pas de données lues sur 'fin', mais sert, ultérieurement, au programme 'qori' qui a appelé 'litab'.

La procédure lit ensuite les sigles des colonnes, chacun comme trois entiers, en les interprétant par le pointeur 'sgli'; et, compte tenu de la largeur de la fenêtre réservée par la procédure d'initialisation 'benzecri', les affiche jusqu'à concurrence de 13. Puis on lit les lignes successives; en s'arrêtant quand on rencontre soit la fin du fichier des données, soit le maximum, 'memi', fixé par la place réservée en mémoire. On notera que, comme indiqué au début du §2.5, les nombres sont lus comme deux entiers, et interprétés par le pointeur 'wwi' ou 'ffli', selon que le suffixe est '.w' ou '.z'. Comme pour les sigles, l'affichage (à l'écran) s'arrête à la 13-ème colonne.

2.5.2 Lecture d'un tableau en format de texte

```

if (rc='O') and not (iz1=1) then begin
  reset (ft,nomf);
  readln(ft,sptit^);writeln(sptit^);carac=' ';
  lecnombre;memi:=imax;
  if (1000<ffl) then begin ff2:=ffl div 1000;memi:=ff2 end;
  memi:=memi+ic;
  ff2:=ffl mod 1000;jc:=ff2;
  writeln(memi:6, ' x',jc:4);
  if (memi<ibmax) then begin
    paux:=newptr(6*memi);pi:=paux;sigli:=pzgi (paux);end
    else begin new (sigli);pi:=ptr (sigli);end;
  for j:=0 to jc do begin
    paux:=newptr(4*memi);klji[j]:=pti (paux);ptlk^[j]:=paux end;
  write(jc:4);
  for j := 1 to jc do begin lecsig;siglj^[j]:= sig;
    if (j<14) then write(sigler(siglj^[j]):6) end;i:=0;writeln;
  while not (eof(ft) or (i=memi)) do begin lecsig;
    if not (sig[0]=0) then begin i:=i+1;
      sigli^[i] := sig;write(sigler(sigli^[i]):4);
      for j := 1 to jc do begin
        if (iz2=0) then begin lecnombre;klji[j]^ [i]:=ffl end else
          begin read(ft,rrt);klji[j]^ [i]:=rrt end;
        if (j<14) then write(round(0+klji[j]^ [i]):6) end;
        writeln end end;
    ic:=i;close (ft);end;
  if (rc='O') and (ic*jc=0) then rc:='N';
end;end.

```

Une fois ouvert le fichier des données, au format de texte, le programme, par l'instruction `readln(ft, sptit^)`, lit d'abord la ligne de titre, dans l'espace pointé par `sptit`. La procédure `'lecnombre'` sert alors pour fixer le nombre des colonnes et des lignes. Dans les versions initiales de `'qori'`, la place en mémoire était toujours réservée pour un nombre de lignes maximum, `imax=1000`. Dans la suite, on a voulu, d'une part, épargner la mémoire, notamment dans le cas d'un tableau comptant de nombreuses colonnes; et, d'autre part, traiter, éventuellement plus de 1000 individus.

Ainsi qu'on l'a expliqué au §2.3, `'lecnombre'` glisse sur tous les commentaires ou caractères divers autres des chiffres (venant après la ligne de titre), jusqu'à lire, dans `ffl` un entier, au format long. Ordinairement, cet entier n'est autre que le nombre des colonnes: `carj=jc≤jmax=300`. Si l'on désire spécifier aussi le nombre des lignes, `cari`, le nombre, `ffl`, inscrit en tête du tableau des données combine `cari` et `carj` suivant la formule:

$$\text{ffl} = (1000 \times \text{cari}) + \text{carj} \quad ;$$

(e.g.: `ffl=7500024`, si `cari=7500` et `carj=24`). On reconnaît qu'il s'agit d'une telle combinaison si $(1000 < \text{ffl})$; auquel cas: `cari = ffl div 1000`; sinon, `cari` est pris égal à 1000. Dans tous les cas, `carj = ffl mod 1000`.

L'allocation de mémoire se fait exactement comme dans le cas d'un tableau en format binaire: en particulier, on tient compte du nombre, `ic`, de lignes supplémentaires, demandé dans l'appel de la procédure `'litab'`.

On lit alors le tableau proprement dit. D'abord, les sigles des colonnes, par la procédure 'lecsig', expliquée au §2.3. Ensuite les lignes, l'une après l'autre. Il importe de noter que la procédure 'litab' est indifférente aux aller à la ligne que comporte le fichier de texte mis en lecture: seul compte, et de façon essentielle, l'aller à la ligne qui termine le titre. Ensuite, la disposition des informations et les séparateurs entre celles-ci n'ont aucun rôle. L'expression de 'ligne du tableau' doit donc être comprise ici relativement à la structure mathématique $I \times J$; et non au texte.

Si les nombres sont écrits comme des entiers (éventuellement pourvus d'un signe), on les lit par la procédure 'lecnombre'. Sinon, et, dans ce cas, le nom du fichier de texte comporte le suffixe 'yy', on exécute l'instruction: read(ft,rrt), qui exige des conditions expliquées, ci-dessus, au §2.3.

```

RGO données d'activité gastrique (2739 cas)
examens de Ph-métrie effectués en MCMLXXXVI
2739010 pathologie  âge nrf kay nrg dur rg>5 rg>> %ph4 %ph5
  1 5 sophage 108 2 28 4 11 0 74 271 380
 38 2 pulmonaire 540 44 30 2 11 0 77 252 438
125 6 Malaises 29 80 125 8 27 3 371 1278 2032
131 1 ORL pure 106 17 24 4 22 0 86 277 328
 68 3 ORL + Pulm 24 66 16 1 4 0 36 167 242

```

Le tableau ci-dessus offre un exemple des libertés laissées dans le cas où les nombres sont exclusivement des entiers. La ligne de titre se compose de caractères quelconques, y compris des chiffres. La ligne de commentaire qui la suit ne donne la date qu'en lettres (chiffres romains). D'après le nombre 2739010, on sait que le nombre des variables est 10, et on attend, au maximum, 2739 individus. Les sigles des colonnes ont tous 3 ou 4 caractères, excepté le premier 'pathologie': ce mot sera lu par lecsig comme 'path', et les sigles qui suivent seront bien lus. Dans les lignes afférentes aux individus, le sigle est, en fait, un numéro de dossier. La variable 'path' vient ensuite, comme un numéro, qu'on a pu faire suivre d'un libellé en texte (sans chiffre) qui est lu par 'lecnombre' avant la valeur de l'âge (donnée ici en mois).

```

RGO données d'activité gastrique (2739 cas)
examens de Ph-métrie effectués en MCMLXXXVI
2739010 pathologie âge nrf kay nrg dur rg>5 rg>> %ph4 %ph5
  1 sophage 5 108 2 28 4 11 0 74 2.71 3.80
 38pulmonaire 2 540 44 30 2 11 0 77 2.52 4.38
125Malaises 6 29 80 125 8 27 3 371 12.78 20.32
131ORLpure 1 106 17 24 4 22 0 86 2.77 3.28
 68ORL+Pulm 3 24 66 16 1 4 0 36 1.67 2.42

```

Dans la variante ci-dessus, les nombres des deux dernières colonnes sont écrits avec un point décimal; le nom du fichier doit avoir le suffixe 'yy'. On ne peut introduire de commentaires parmi les nombres, mais le sigle peut être prolongé par le nom de la pathologie; avec pour seul inconvénient que 'lecsig' lira pour sigles {1Eso, 38pu, 125M, 121O, 68OR}. On notera que dans 'ORL pure' et 'ORL + Pulm', les blancs ont été supprimés; faute de quoi, 'lecsig' s'arrêterait après 'ORL' et le jeu de read(ft,rrt) serait perturbé par des lettres.

3 Définition d'un sous-ensemble: l'unité 'uens.p'

Dans l'analyse d'un tableau de correspondance donné, $I \times J$, les éléments supplémentaires peuvent avoir deux origines différentes. D'une part, le programme 'qori' permet de mettre en supplément des sous-ensembles, I_s et J_s , des ensembles, I et J , des lignes et des colonnes. D'autre part, par le programme 'qorlsup' (cf. ID), on adjoint au tableau $I \times J$ des tableaux externes, $I_a \times J$ (lignes supplémentaires) ou $I \times J_b$ (colonnes supplémentaires). Dans le premier cas, les sous-ensembles I_s et J_s peuvent être spécifiés, soit par un fichier de texte (,que lit la procédure 'suplirer', propre à 'qori': cf. IB§3.2.1), soit par un dialogue à l'écran, conduit suivant la procédure 'ensembler', contenue dans 'uens.p' (,procédure qui sert également au programme 'soustab' pour définir un sous-tableau d'un tableau de base, donné en format '.z' ou '.w').

La procédure 'ensembler' a 5 arguments:

L'entier 'carte' n'est autre que le nombre des éléments de l'ensemble de base à traiter, E.

Les pointeurs 'plmt' et 'padj' indiquent, respectivement, le nom de E (ses 'éléments') et la qualité ('adjectif') du sous-ensemble, E_s , à définir. Pour 'qori', 'plmt' pointe vers 'colonne' ou vers 'ligne' ; 'padj' pointe vers 'suppl'. (Pour 'soustab', 'padj' pointe vers: 'à garder').

Le pointeur 'psigl' donne accès à la liste des sigles des éléments de l'ensemble E: aussi psigl reçoit-il le type: pzgi.

Enfin le pointeur psu, désigne un tableau de nombres entiers. Initialement, avant appel de 'ensembler', ce tableau a été mis à zéro; après exécution de 'ensembler', sont mis à 1 les nombres dont le rang correspond à un élément de E_s .

Pour désigner les éléments de E_s , l'utilisateur peut procéder soit par éléments isolés, soit par blocs. On ne sort de la grande boucle 'while not...' que si l'utilisateur a confirmé son choix par la réponse `rup='O'`. la boucle commence par afficher la suite des sigles des éléments de l'ensemble E, chacun précédé de son rang, i. Il faut prendre garde que le nombre, nu, des chiffres de ce nombre doit être 4, si `care ≥ 1000`. Le nombre, li, des éléments inscrits sur une ligne dépend de nu. De plus les éléments ne peuvent s'afficher, dans la fenêtre, que sur 23 lignes, i.e. par paquets, dont le cardinal est: `pa=23×li`; l'utilisateur commandant, éventuellement, la succession des paquets en entrant un caractère (e.g.: '*').

Selon que les éléments sont désignés isolément ou par blocs, on entre dans l'une ou l'autre des deux instructions composées introduites, respectivement, par 'if (repg='U')' et 'if (repg='B')'. Dans les deux cas, la procédure demande à l'utilisateur de confirmer le nombre, 'cares', des éléments

```

UNIT uens;
INTERFACE uses memtypes,quickdraw,osintf,toolintf,sane,uver;
procedure ensembler(carte:integer;plmt,padj,psigt,psu:ptr);
implementation
procedure ensembler;
var cares,i,is,ip,ih,nu,li,pa:integer;rup,regp,rvp:char;chif:string;
    lmt1,adj1,pchf:stringptr;siglt:pzgi;sul:ptni;
begin
lmt1:=stringptr(plmt);adj1:=stringptr(padj);pchf:=stringptr(@chif);
siglt:=pzgi(psigt);sul:=ptni(psu);
if (carte<1000) then nu:=3 else nu:=4;
li:=81 div (6+nu);pa:=23*li;
rup:='N';
while not (rup='O') do begin rup:='N';
write('les ',lmt1^,'s ',adj1^,' seront-elles designées');
write('une par une(U) ou par blocs(B) ');readln(regp);
if not (regp='B') then regp:='U';
writeln ('ci-dessous sont rappelés les num et sigles des ',lmt1^,'s');
for i:=1 to carte do begin
write(i:nu,sigler(siglt^[i]):5,'');
if (i mod li=0) then begin writeln;
if ((i mod pa=0) and (i<carte)) then begin
write('pour afficher la suite de l'ensemble entrer *');
readln(rvp) end end end ;
if not (carte mod li=0) then writeln;
if (regp='U') then begin
while not (rup='O') do begin
write('le nombre de ',lmt1^,'s ',adj1^,' sera ');
readln(cares);if (carte<cares) then cares:=carte;
writeln(cares:8);
write('ce nombre est-il confirmé O ou N ');
readln(rup) end;
writeln ('l'utilisateur donnera ci-dessous les num des ',lmt1^,'s ',adj1^);
writeln ('NB l'ordre dans lequel ces num sont donnés est indifférent');
for i:=1 to cares do begin
write('le numéro initial de la ',lmt1^,' ',adj1^,i:3,' est ');
readln(is);if (is<1) then is:=1;if (carte<is) then is:=carte;
sul^[is]:=1 end end;
if (regp='B') then begin
while not (rup='O') do begin
write('le nombre des blocs de ',lmt1^,'s ',adj1^,' sera ');
readln(cares);if (carte<cares) then cares:=carte;
writeln(cares:8);
write('ce nombre est-il confirmé O ou N ');
readln(rup) end;
for i:=1 to cares do begin
write('le numéro de la lère ',lmt1^,' du bloc ',adj1^,i:3,' est ');
readln(ip);
write('le num de la dernière ',lmt1^,' du bloc ',adj1^,i:3,' est ');
readln(ih);
if (ip<1) then ip:=1;if (carte<ih) then ih:=carte;
for is:=ip to ih do sul^[is]:=1 end end;
write('le choix des ',lmt1^,'s ',adj1^,' est il confirmé O ou N ');
readln(rup);if not (rup='O') then for i:=1 to carte do sul^[i]:=0 end end;
end.

```

ou des blocs; car l'usage nous a appris qu'on peut, par inadvertance, s'engager dans une boucle de désignation tournant des dizaines, voire des centaines, de fois.

4 Tri par fusion d'un ensemble de nombres réels: l'unité 'utrir.p'

```

UNIT utrir;
INTERFACE uses memtypes, quickdraw, osintf, toolintf, sane, uver5;
procedure trire (parp, ppra, pVr, ptVr:ptr; cr:integer);
implementation procedure trire;
var arpl, pral:ptni; Vr1, tVr1:ptdri; ina, la, lb, n, na, nb, nz, x:integer;
begin
arpl:=ptni (parp); pral:=ptni (ppra); Vr1:=ptdri (pVr); tVr1:=ptdri (ptVr);
for n:=1 to cr do arpl^[n]:=n; la:=1; ina:=1;
while (ina+la<=cr) do begin
x:=cr-(ina+la-1); if (x<la) then lb:=x else lb:=la;
na:=ina; nb:=ina+la;
for n:=ina to (ina+la+lb)-1 do begin
if (na<ina+la) and (nb<ina+la+lb) then
if (Vr1^[na]<=Vr1^[nb]) then begin nz:=na; na:=na+1 end
else begin nz:=nb; nb:=nb+1 end
else if (ina+la=na) then begin nz:=nb; nb:=nb+1 end
else begin nz:=na; na:=na+1 end;
tVr1^[n]:=Vr1^[nz]; pral^[n]:=arpl^[nz] end;
for n:=ina to ina+la+lb-1 do begin
Vr1^[n]:=tVr1^[n]; arpl^[n]:=pral^[n] end;
ina:=ina+la+lb;
if (cr<ina+la) then begin la:=2*la; ina:=1 end; end;
for n:= 1 to cr do pral^[arpl^[n]]:=n;
end; end.

```

La procédure 'trire' a pour arguments 4 pointeurs et un entier. Celui-ci n'est autre que la longueur 'cr' de la suite de nombres réels, qui doivent être rangés dans l'ordre croissant. Les pointeurs donnent accès, respectivement, à deux suites d'entiers (chacun écrit sur 2 octets) et à deux suites de réels (au format 'extended').

L'essentiel de la procédure 'trire' consiste en une boucle 'while', régie par les deux entiers 'ina' et 'la' dont la valeur initiale est 1.

La fusion proprement dite, opère sur deux segments, ou blocs, Va et Vb, de la suite Vr1[^]: le premier débute à l'adresse 'ina' (nom qui rappelle: initiale de a) et comprend 'la' nombres; le second débute à l'adresse 'inb' et comprend 'lb' nombres. En principe, lb est égal à la; mais si une suite de longueur la devait dépasser le dernier terme, cr, la longueur lb, du bloc Vb, est réduite à la valeur: cr-(ina+la-1). Mais il faut d'abord que la place existe pour le bloc Va et que celui-ci ne se termine pas à la dernière adresse, cr: tel est l'objet de la condition (ina+la<=cr) placée en tête de la boucle 'while'.

Ceci posé, dans la fusion, les deux blocs consécutifs de Vr1[^], Va et Vb, qu'on peut noter [ina..ina+la-1] et [ina+la..ina+la+lb-1], sont recopiés dans le bloc correspondant [ina..ina+la+lb-1], tVab, du tableau tVr1[^]. En bref, les blocs Va et Vb étant supposés déjà rangés de sorte qu'ils contiennent des nombres non décroissants, les nombres sont recopiés en avançant dans Va et Vb. À un instant donné, sont à comparer, les nombres de Va et Vb ayant, dans Vr1[^], pour rangs respectif: na et nb: est recopié au rang n de tVr1[^], dans tVab, qui est en cours d'écriture, le plus petit des deux nombres (le choix se portant,

en cas d'égalité, sur celui du bloc Va). Quand Va et Vb ont ainsi été fondus dans tVab, ce bloc de, tVr1^, maintenant convenablement trié, est recopié dans Vr1^, à la place des blocs consécutifs Va et Vb.

Puis on tente de placer, sur Vr1^, à partir de l'adresse ina+la+lb, deux nouveaux blocs, Va et Vb, soumis, de même, à la fusion.

Si cela n'est pas possible, s'exécute l'instruction composée:

```
if (cr<ina+la) then begin la:=2*la;ina:=1 end;
```

qui double la longueur, la, des blocs à considérer, et ramène à l'adresse ina=1 de Vr1^, pour définir de nouveaux blocs.

Reste à expliquer le rôle des tableaux d'entiers arp1^ et pral^. Initialement, arp1^ contient la suite des entiers de 1 à cr. Ces entiers sont considérés comme les adresses initiales des nombres réels donnés dans Vr1^. Au cours des itérations de la boucle générale 'while', on effectue, entre les deux tableaux d'entiers arp1^ et pral^, les mêmes transferts qu'entre les deux tableaux de réels, Vr1^ et tVr1^; en sorte que, après exécution de la procédure 'trire', il y a, d'une part, dans Vr1^, (ainsi que dans tVr1^), l'ensemble des réels donnés initialement, mais rangés en une suite non décroissante; et, d'autre part, dans arp1^, un rappel des adresses qu'occupaient initialement ces nombres: e.g., si le réel qui est finalement dans Vr1^[n] était, initialement, dans Vr1^[un], on a arp1^[n]=un. De plus, l'instruction:

```
for n:= 1 to cr do pral^[arp1^[n]]:=n;
```

met, dans pral^[un], l'adresse finale, n, du réel qui était, initialement, dans Vr1^[un].

Dans le programme 'qori', la procédure 'trire' est appelée par la procédure 'diagu' de diagonalisation d'une matrice carrée symétrique (cf. ci-après, §5) afin de ranger les vecteurs propres (facteurs) dans l'ordre décroissant des valeurs propres.

Dans 'zrang' (cf. IIIA§2.5), 'trirs' (variante de 'trire') sert à l'affichage d'une variable (colonne j) du tableau des données, suivant des histogrammes, dont chaque créneau porte les sigles des individus (lignes i) qui y sont compris; et éventuellement, c'est par 'trirs' qu'une variable est découpée en classes.

En Classification Ascendante Hiérarchique, l'algorithme d'agrégation en chaîne des paires de plus proches voisins réciproques ne crée pas nécessairement les nœuds dans l'ordre croissant de leurs niveaux (il crée seulement, nécessairement, chaque nœud après tous ses descendants). On doit donc renuméroter les nœuds dans l'ordre des niveaux croissants. Les niveaux étant conservés en format entier, on utilise une procédure 'trier' qui ne diffère de 'trire' qu'en ce qu'elle traite des entiers (cf. IIB§2.1.1).

5 Diagonalisation d'une matrice carrée symétrique: l'unité 'udiaq.p'

```

UNIT udiac;
INTERFACE uses memtypes, quickdraw, osintf, toolintf, sane, uver, utrir;
procedure diagu (pha:0, fjj:0, dj:0, zz:0, pza:0, rj:0, lama:0:ptr; carj:integer);
implementation procedure diagu;
var
  fjj:sjj:dj,rj,lama:prtj;zz:pitj;zjj:psjj;
  a,ap,j,jj,jp,js,shft,it:integer;crit:boolean;
  k1,k2,kk,max,som,soml,px,qx,xx,yy,co,si,norm,res:extended;
procedure sinkos;begin
  if ((px=0) and (qx=0)) then begin co:=1;si:=0;norm:=0 end
  else begin if (abs(px)<abs(qx)) then
    norm:=abs(qx)*sqrt(1+sqr(px/qx)) else
    norm:=abs(px)*sqrt(1+sqr(qx/px));
    co:=px/norm;si:=qx/norm end;end;
begin
  zjj:=psjj(fjj:0);for j:=1 to jmax do fjj[j]:=zjj^[j];
  dj:=prtj(dj:0);rj:=prtj(rj:0);lama:=prtj(lama:0);zz:=pitj(zz:0);
  for j:=1 to carj do begin dj^[j]:=fjj[j]^j;fjj[j]^j:=1;rj^[j]:=0;
  if (j<carj-1) then begin max:=0;
  write(j:3);if (j mod 27 =0) then writeln;
  for jp:=j+1 to carj do
    if (max<abs(fjj[jp]^j)) then max:=abs(fjj[jp]^j);
  if (0<max) then begin som:=0;
  for jp:=j+1 to carj do begin fjj[jp]^j:=fjj[jp]^j/max;
  som:=somsqr(fjj[jp]^j) end;
  som:=sqrt(som);if (fjj[j+1]^j<0) then som:=-som;
  rj^[j]:=-som*max;fjj[j+1]^j:=fjj[j+1]^j+som;
  fjj[j]^j:=som*fjj[j+1]^j;soml:=0;
  for jp:=j+1 to carj do begin som:=0;
  for js:=j+1 to jp do som:=som+(fjj[jp]^js)*fjj[js]^j);
  if (jp<carj) then for js:=jp+1 to carj do
    som:=som+(fjj[js]^j)*fjj[js]^jp);
  rj^[jp]:=som/fjj[j]^j;soml:=soml+(fjj[jp]^j)*rj^[jp] end;
  som:=soml/(2*fjj[j]^j);
  for jp:=j+1 to carj do begin rj^[jp]:=rj^[jp]-(som*fjj[jp]^j);
  for js:=j+1 to jp do fjj[jp]^js
    :=fjj[jp]^js-((fjj[jp]^j)*rj^[js])+(fjj[js]^j)*rj^[jp])
  end end end;writeln;
  rj^[carj-1]:=fjj[carj]^carj-1;fjj[carj]^carj-1:=0;fjj[carj-1]^carj:=0;
  if (2<carj) then for j:=carj-1 downto 2 do begin
  write(j:3);if ((carj-j) mod 27 =0) then writeln;
  for jp:=j to carj do begin som:=0;
  for js:=j to carj do
    som:=som+(fjj[js]^j-1)*fjj[js]^jp);
  fjj[j-1]^jp:=som/fjj[j-1]^j-1 end;
  for jp:=j to carj do for js:=j to carj do
    fjj[jp]^js:=fjj[jp]^js-(fjj[jp]^j-1)*fjj[j-1]^js);
  for jp:=j to carj do begin
    fjj[jp]^j-1:=0;fjj[j-1]^jp:=0 end;
  fjj[j-1]^j-1:=1 end;

```

Nous ne tenterons pas d'expliquer l'algorithme SYMQR, mis en œuvre par la procédure 'diagu'; mais considérerons seulement les accès et le déroulement de 'diagu'. Plus précisément, 'diagu' n'étant, présentement, appelée que dans 'qori', nous spécifions, d'après cet appel, la signification des variables de 'diagu': i.e., un entier et 7 pointeurs.

```

shft:=1;k1:=5;
for j:=carj downto 2 do begin it:=0;writeln;write(j:3);
  while ((rj^[j-1]<>0) and (it<50)) do begin jp:=j;crit:=false;
    while ((crit=false) and (1<jp)) do begin jp:=jp-1;
      som:=abs(dj^[jp]);if (abs(dj^[jp+1])<som) then som:=abs(dj^[jp+1]);
      som:=(1e-12)*som;if abs(rj^[jp])<som then
        begin crit:=true;rj^[jp]:=0;jp:=jp+1 end end;
    if (jp<j) then begin it:=it+1;write(it:2);
      som:=(dj^[j-1]-dj^[j])/2;soml:=sqrt(sqr(som)+sqr(rj^[j-1]));
      if (som<0) then soml:=-soml;
      k2:=dj^[j]-(sqr(rj^[j-1])/(som+soml));
      if (shft=1) then if (abs(k2-k1)<abs(k2)/2) then shft:=0
        else begin k1:=k2;kk:=5 end ;
      if (shft<>1) then kk:=k2; px:=dj^[jp]-kk;qx:=rj^[jp];sinkos;
      for js:=jp to j-1 do begin
        for jj:=1 to carj do begin xx:=fjj[jj]^[js]; yy:=fjj[jj]^[js+1];
          fjj[jj]^[js]:=(co*xx)+(si*yy);fjj[jj]^[js+1]:=(co*yy)-(si*xx) end;
        dj^[js]:=(co*dj^[js]*co)+((2*co*rj^[js])+(si*dj^[js+1]))*si;
        dj^[js+1]:=(co*dj^[js+1])-(si*rj^[js]);rj^[js]:=-si*kk;
        if (js<>j-1) then if (abs(co)<abs(si)) then begin
          px:=(co*rj^[js])+(si*dj^[js+1]);qx:=si*rj^[js+1];
          rj^[js+1]:=co*rj^[js+1];dj^[js+1]:=((co*px)/si)+kk;
          sinkos; rj^[js]:=norm end
        else begin
          res:=si/co; dj^[js+1]:=(co*dj^[js+1])-(si*rj^[js]);
          px:=dj^[js+1]-kk; rj^[js+1]:=co*rj^[js+1]; qx:=rj^[js+1];
          sinkos;rj^[js]:=res*norm end end;
        xx:=rj^[j-1];yy:=dj^[j];
        dj^[j]:=(co*yy)-(si*xx);rj^[j-1]:=(co*xx)+(si*yy) end end end;
  trire(zz0,pza0,dj0,lama0,carj);
  for a:=1 to carj do lama^[a]:=dj^[1+carj-a];
  for j:=2 to carj do for jp:=1 to j do begin
    res:=fjj[j]^[jp];fjj[j]^[jp]:=fjj[jp]^[j];fjj[jp]^[j]:=res end;
  zjj:=psjj(Phaj0);
  for a:=1 to carj do begin ap:=zz^[1+carj-a];zjj^[a]:=fjj[ap] end;
end;end.

```

5.1 Arguments de la procédure 'diagu'

L'entier carj n'est autre que le nombre des lignes et colonnes de la matrice carrée à diagonaliser: carj délimite les parties utiles des espaces auxquels donnent accès les pointeurs.

Par fjj0, 'diagu' reçoit la matrice à diagonaliser, explicitée, suivant ses colonnes, par le tableau de pointeurs: fjj^. Ce tableau: fjj^, sert à calculer les vecteurs propres (normalisés: pour que vaille 1 la somme des carrés des composantes). Le contenu final des zones auxquelles fjj0 donne accès, n'est pris en compte par 'qori' que par l'intermédiaire du pointeur Pha0 (cf. *infra*).

Les pointeurs {zz0, pza0, dj0, rj0} offrent à 'diagu' des espaces de travail, déjà réservés par 'qori', qui appelle 'diagu', mais dont le contenu initial et final est indifférent, pour 'qori' et 'diagu'.

Les pointeurs {Phaj0, lama0} n'apportent pas de données à 'diagu'; mais servent à renvoyer à 'qori' les résultats de la diagonalisation: Phaj0, donne accès à un tableau de pointeurs; lesquels, en fin de calcul, pointent vers les vecteurs propres; de même, lama0 pointe vers le tableau des valeurs propres.

5.2 Déroulement de la procédure 'diagu'

Si l'on met à part les initialisations, la procédure 'diagu' comprend trois parties, d'inégale importance.

A) La tridiagonalisation de la matrice donnée, effectuée par les deux boucles successives:

```
for j:=1 to carj do...
```

```
  if (2<carj) then for j:=carj-1 downto 2 do...
```

au cours de chacune desquelles s'affiche à l'écran, sur deux blocs de lignes consécutifs, la suite (respectivement croissante et décroissante) des valeurs de l'indice j.

B) La diagonalisation, ou calcul des vecteurs propres normalisés (i.e., avec des composantes dont la somme des carrés est 1), effectuée par la boucle:

```
for j:=carj downto 2 do begin it:=0;writeln;write(j:3);...
```

boucle au cours de laquelle s'affiche à l'écran, sur une ligne commençant par le nombre j, au fur et à mesure du calcul, le numéro de l'itération en cours pour le vecteur propre dont le rang (initial) est j: soit, e.g.,

```
26
25 1 2 3 4
```

si après calcul d'un vecteur j=26, obtenu immédiatement, du seul fait des calculs ayant fourni les vecteurs de rang j>26, est en cours, pour j=25, une 4-ème itération. [On notera que, par la boucle: `downto`, les vecteurs propres sont créés dans l'ordre décroissant des rangs j.]

C) Le tri, par 'tire', des vecteurs propres et valeurs propres; celles-ci formant, finalement, une suite décroissante:

```
{lama^[1], lama^[2], ..., lama^[carj]} .
```

Ainsi qu'on l'a dit au §5.1, c'est aux vecteurs propres et valeurs propres, ainsi rangés, que la procédure 'qori' accède par les pointeurs {Phaj0, lama0}.

Référence bibliographique (voir, aussi, ID, *in fine*)

K. Ben SALEM : "Associativité de la fusion et parallélisme dans les algorithmes de tri"; [ASS. FUS. TRI]; in *CAD*, Vol.XV, n°2; pp. 133-138; (1990).

N.B.: Dans le présent exposé, les articles ultérieurs de la série [SOURCES PASCAL] sont cités par leurs numéros: IB, IC, ..., IIA...