Chapitre 7

Les Tableaux

> Si on ment à un compilateur, il prendra sa revanche.
>
> Henry Spencer

Bonne nouvelle ! Je vous avais annoncé qu'il y a avait en tout et pour tout quatre structures logiques dans la programmation. Eh bien, ça y est, on les a toutes passées en revue. Mauvaise nouvelle, il vous reste tout de même quelques petites choses à apprendre...

7.1 Utilité des tableaux

Imaginons que dans un programme, nous ayons besoin simultanément de 12 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer douze variables, appelées par exemple Notea, Noteb, Notec, etc... Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc...

Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, et après une succession de douze instructions input distinctes, cela donnera obligatoirement une atrocité du genre :

```
Moy=(N1+N2+N3+N4+N5+N6+N7+N8+N9+N10+N11+N12)/12
```

Ouf ! C'est tout de même bigrement laborieux. Et pour un peu que nous soyons dans un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter, alors là c'est le suicide direct. Cerise sur le gâteau, si, en plus, on est dans une situation on l'on ne peut pas savoir d'avance combien il y aura de valeurs à traiter, là on est carrément cuits.

C'est pourquoi la programmation nous permet de rassembler toutes ces variables en une seule, au sein de laquelle chaque valeur sera désignée par un numéro. En bon français, cela donnerait donc quelque chose du genre « la note numéro 1 », « la note numéro 2 », « la note numéro 8 ». C'est largement plus pratique, vous vous en doutez.

Un ensemble de valeurs portant le même nom de variable et repérées par un nombre, s'appelle un tableau, ou encore une variable indicée.

Le nombre qui, au sein d'un tableau, sert à repérer chaque valeur s'appelle – ô surprise – l'indice. Chaque fois que l'on doit désigner un élément du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément, entre parenthèses.

7.2 Notation et utilisation algorithmique

Dans notre exemple, nous créerons donc un tableau appelé Note. Chaque note individuelle (chaque élément du tableau Note) sera donc désignée Note(1), Note(2), etc. Attention, pour certains langages, l'indice commence à zéro et pas à 1.

Un tableau doit être déclaré comme tel, en précisant le nombre et le type de valeurs qu'il contiendra (la déclaration des tableaux est susceptible de varier d'un langage à l'autre. Certains langages réclament le nombre d'éléments, d'autre le plus grand indice... C'est donc une affaire de conventions).

En nous calquant sur les choix du MATLAB, nous déciderons ici arbitrairement et une bonne fois pour toutes que les "cases" sont numérotées à partir de un. Lors de la déclaration d'un tableau, on précise le nombre de cases du tableau.

```
% Tableau Note(11) en Entier
```

On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, bien sûr, mais aussi tableaux de caractères, tableaux de booléens, tableaux de tout ce qui existe dans un langage donné comme type de variables. Par contre, hormis dans quelques rares langages, on ne peut pas faire un mixage de types différents de valeurs au sein d'un même tableau.

L'énorme avantage des tableaux, c'est qu'on va pouvoir les traiter en faisant des boucles. Par exemple, pour effectuer le calcul de moyenne, cela donnera par exemple :

```
% Tableau Note(11) en Numérique
% Variables Moy, Som en Numérique
% Début
for i= 1:12
  Note(i)=input(['Entrez la note n$^\circ$',num2str(i)]);
end
Som=0
for i= 1:12
  Som=Som + Note(i)
end
Moy=Som/12
% Fin
```

NB : On a fait deux boucles successives pour plus de lisibilité, mais on aurait tout aussi bien pu n'en écrire qu'une seule dans laquelle on aurait tout fait d'un seul coup.

Remarque générale : l'indice qui sert à désigner les éléments d'un tableau peut être exprimé directement comme un nombre en clair, mais il peut être aussi une variable, ou une expression calculée. Dans un tableau, la valeur d'un indice doit toujours :

- être égale au moins à 1.
- être un nombre entier Quel que soit le langage, l'élément Truc(3,1416) n'existe jamais.
- être inférieure ou égale au nombre d'éléments du tableau. Si le tableau Bidule a été déclaré comme ayant 25 éléments, la présence dans une ligne, sous une forme ou sous une autre, de Bidule(32) déclenchera automatiquement une erreur.

7.3 Le gag de la journée

Il consiste à confondre, dans sa tête et / ou dans un algorithme, l'indice d'un élément d'un tableau avec le contenu de cet élément. La troisième maison de la rue n'a pas forcément trois habitants, et la vingtième vingt habitants. En notation algorithmique, il n'y a aucun rapport entre i et truc(i). Holà, Tavernier, prépare la cervoise !

7.4 Tableaux dynamiques

Il arrive fréquemment que l'on ne connaisse pas à l'avance le nombre d'éléments que devra comporter un tableau. Bien sûr, une solution consisterait à déclarer un tableau gigantesque (10 000 éléments, pourquoi pas, au diable les varices) pour être sûr que « ça rentre ». Mais d'une part, on n'en sera jamais parfaitement sûr, d'autre part, en raison de l'immensité de la place mémoire réservée - et la plupart du temps non utilisée, c'est un gâchis préjudiciable à la rapidité, voire à la viabilité, de notre algorithme.

Aussi, pour parer à ce genre de situation, a-t-on la possibilité de déclarer le tableau sans préciser au départ son nombre d'éléments.

Notons que, en MATLAB, les tableaux sont, par défaut dynamiques, et le simple fait d'écrire un élément hors du tableau change la taille du tableau. Pour tous les problèmes de ce cours, ce n'est pas préjudiciable, car les tableaux seront petits. Toutefois, lorsqu'on a de gros tableaux, cela ralenti considérablement le temps de calcul.

Par contre, il est interdit d'appeler un élément non existant. Si, par exemple, la matrice A est définie par

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix},$$

La commande

```
A(3,2)=1
```

a du sens, et donnera

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 0 & 1 \end{pmatrix},$$

La commande

```
A(3,2)=A(3,2)+1
```

7.5 Tableaux Multidimensionnels

Ceci n'est pas un dérèglement de votre téléviseur. Nous contrôlons tout, nous savons tout, et les phénomènes paranormaux que vous constatez sont dus au fait que vous êtes passés dans... la quatrième dimension (musique angoissante : « tintintin... »). Oui, enfin bon, avant d'attaquer la quatrième, on va déjà se coltiner la deuxième.

7.5.1 Pourquoi plusieurs dimensions ?

Une seule ne suffisait-elle pas déjà amplement à notre bonheur, me demanderez-vous ? Certes, répondrai-je, mais vous allez voir qu'avec deux (et davantage encore) c'est carrément le nirvana.

Prenons le cas de la modélisation d'un jeu de dames, et du déplacement des pions sur le damier. Je rappelle qu'un pion qui est sur une case blanche peut se déplacer (pour simplifier) sur les quatre cases blanches adjacentes. Avec les outils que nous avons abordés jusque là, le plus simple serait évidemment de modéliser le damier sous la forme d'un tableau. Chaque case est un emplacement du tableau, qui contient par exemple 0 si elle est vide, et 1 s'il y a un pion. On attribue comme indices aux cases les numéros 1 à 10 pour la première ligne, 11 à 20 pour la deuxième ligne, et ainsi de suite jusqu'à 100. Un pion placé dans la case numéro i, autrement dit la valeur 1 de Cases(i), peut bouger vers les cases contiguës en diagonale. Cela va nous obliger à de petites acrobaties intellectuelles : la case située juste au-dessus de la case numéro i ayant comme indice i-10, les cases valables sont celles d'indice i-9 et i-11. De même, la case située juste en dessous ayant comme indice i+10, les cases valables sont celles d'indice i+9 et i+11. Bien sûr, on peut fabriquer tout un programme comme cela, mais le moins qu'on puisse dire est que cela ne facilite pas la clarté de l'algorithme. Il serait évidemment plus simple de modéliser un damier par... un damier !

7.5.2 Tableaux à deux dimensions

L'informatique nous offre la possibilité de déclarer des tableaux dans lesquels les valeurs ne sont pas repérées par une seule, mais par deux coordonnées. Un tel tableau se déclare ainsi :

```
% Tableau Cases(10, 10) en Numérique
```

Cela veut dire : réserve moi un espace de mémoire pour 10 x 10 entiers, et quand j'aurai besoin de l'une de ces valeurs, je les repèrerai par deux indices (comme à la bataille navale, ou Excel, la seule différence étant que pour les coordonnées, on n'utilise pas de lettres, juste des chiffres). Pour notre problème de dames, les choses vont sérieusement s'éclaircir. La case qui contient le pion est dorénavant Cases(i, j). Et les quatre cases disponibles sont Cases(i-1, j-1), Cases(i-1, j+1), Cases(i+1, j-1) et Cases(i+1, j+1).

REMARQUE ESSENTIELLE : Il n'y a aucune différence qualitative entre un tableau à deux dimensions (i, j) et un tableau à une dimension (i * j). De même que le jeu de dames qu'on vient d'évoquer, tout problème qui peut être modélisé d'une manière peut aussi être modélisé de l'autre. Simplement, l'une ou l'autre de ces techniques correspond plus spontanément à tel ou tel problème, et facilite donc (ou complique, si on a choisi la mauvaise option) l'écriture et la lisibilité de l'algorithme.

Une autre remarque : une question classique à propos des tableaux à deux dimensions est de savoir si le premier indice représente les lignes ou le deuxième les colonnes, ou l'inverse. Je ne répondrai pas à cette question non parce que j'ai décidé de bouder, mais parce qu'elle n'a aucun sens. « Lignes »et « Colonnes »sont des concepts graphiques, visuels, qui s'appliquent à des objets du monde réel ; les indices des tableaux ne sont que des coordonnées logiques, pointant sur des adresses de mémoire vive. Si cela ne vous convainc pas, pensez à un jeu de bataille navale classique : les lettres doivent-elles désigner les lignes et les chiffres les colonnes ? Aucune importance ! Chaque joueur peut même choisir une convention différente, aucune importance ! L'essentiel est qu'une fois une convention choisie, un joueur conserve la même tout au long de la partie, bien entendu.

7.5.3 Tableaux à n dimensions

Si vous avez compris le principe des tableaux à deux dimensions, sur le fond, il n'y a aucun problème à passer au maniement de tableaux à trois, quatre, ou pourquoi pas neuf dimensions. C'est exactement la même chose. Si je déclare un tableau Titi(2, 4, 3, 3), il s'agit d'un espace mémoire contenant 3 x 5 x 4 x 4 = 240 valeurs. Chaque valeur y est repérée par quatre coordonnées. Le principal obstacle au maniement systématique de ces tableaux à plus de trois dimensions est que le programmeur, quand il conçoit son algorithme, aime bien faire des petits gribouillis, des dessins immondes, imaginer les boucles dans sa tête, etc. Or, autant il est facile d'imaginer concrètement un tableau à une dimension, autant cela reste faisable pour deux dimensions, autant cela devient l'apanage d'une minorité privilégiée pour les tableaux à trois dimensions (je n'en fais malheureusement pas partie) et hors de portée de tout mortel au-delà. C'est comme ça, l'esprit humain a du mal à se représenter les choses dans l'espace, et crie grâce dès qu'il saute dans l'hyperespace (oui, c'est comme ça que ça s'appelle au delà de trois dimensions).

Donc, pour des raisons uniquement pratiques, les tableaux à plus de trois dimensions sont rarement utilisés par des programmeurs non matheux (car les matheux, de par leur formation, ont une fâcheuse propension à manier des espaces à n dimensions comme qui rigole, mais ce sont bien les seuls, et laissons-les dans leur coin, c'est pas des gens comme nous).

7.6 Exercices corrigés

1. Ecrire un tableau des chiffres, une fois en numérique et une autre fois en caractère.

```
% Variable i, en Entier
% tableau cnum en numérique
% tableau cnum en caractère
% Début
for i=0:9
   cnum(i)=i
   cchar(i)=num2str(i)
end
%Fin
```

2. Corriger l'algorithme de décomposition en facteurs premiers pour qu'il remplisse un tableau (dynamiquement) avec les facteurs.

```
% Variable i, ind, n en Entier
% tableau fact en numérique
% Début
ind=1
for i=2:sqrt(n)
   while (n/i==fix(n/i))
       a(ind)=i
       n=n/i
       ind=ind+1
   end
end
if (n~=1)
    a(ind)=n
end
%Fin
```

3. Que produit l'algorithme suivant ?

```
% Tableau Nb(5) en Entier
% Variable i en Entier
% Début
for i=1:5
  Nb(i)=i*i
end
for i=1:5
  display(Nb(i))
end
%Fin
```

Peut-on simplifier cet algorithme avec le même résultat ?
Dans la première boucle, on met dans Nb le carré des nombres entre 1 et 5. Dans la seconde, les carrés sont affichés. On peut donc simplifier en groupant les deux boucles :

```
% Tableau Nb(5) en Entier
% Variable i en Entier
% Début
for i=1:5
  Nb(i)=i*i
  display(Nb(i))
end
%Fin
```

ou même

```
% Variable i en Entier
% Début
for i=1:5
  display(i*i)
end
%Fin
```

mais là, c'est triché parce qu'on n'utilise même plus les tableaux.

4. Que produit l'algorithme suivant ?

```
% Tableau N(6) en Entier
% Variables i, k en Entier
% Début
N(1)=1
for k=2:6
```

```
  N(k)=N(k-1)+2
  display(N(i))
end
%Fin
```

La première valeur du tableau est donnée, c'est 1. Les valeurs suivantes sont calculées itérativement : N(2)=N(1)+2=1+2=3. N(3)=N(2)+2=5. N(4)=7, N(5)=9 et N(6)=11. Ces valeurs sont affichées.

5. Que produit l'algorithme suivant ?

```
% Tableau Suite(7) en Entier
% Variable i en Entier
% Début
Suite(1)=1
Suite(2)=1
for i=3:7
  Suite(i)=Suite(i-1) + Suite(i-2)
  display(Suite(i))
end
% Fin
```

6. Ecrivez un algorithme qui calcule la médiane des valeurs d'un tableau de 7 données. La médiane est la valeur du tableau qui a autant de valeurs plus grandes et que de valeurs plus petites qu'elle parmi les valeurs du tableau. Par exemple, la médiane de 1 4 2 6 8 1 5 est 4, puisqu'il y a trois valeurs plus petites (1, 1, 2) et trois valeurs plus grandes (5, 6, 8).

```
% Tableau donn, pg, pp en entier
% Tableau i,j en numérique
donn=input('Entrer un tableau de 7 données');
pg=zeros(1,7);
pp=zeros(1,7);
for i=1:7
   for j=1:7
      if (donn(i)>donn(j))
        pg(i)=pg(i)+1
      end
   end
   if (pg(i)==3)
     display(['La médiane est ',num2str(donn(i))])
   end
end
```

7. Quel résultat produira cet algorithme ?

```
% Tableau X(2, 3) en Entier
% Variables i, j, val en Entier
% Début
Val=1
for i=1:2
  for j=1:3
    X(i,j)=Val;
    Val=Val+1;
  end
end
for i=1:2
  for j=1:3
    display(X(i, j))
  end
end
```

```
% Fin
```

Au début, Val=1, on met cette valeur dans X(1,1), puis on incrémente Val qui vaut à présent 2. On rentre cette valeur dans X(1,2), puis on rentre 3 dans X(1,3), puis 4 dans X(2,1). Le tableau final va donc être

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

L'ordinateur égrainera ensuite les nombres entre 1 et 6.

8. Plus rigolo : écrivons un algoritme qui donne le même résultat, mais en faisant une boucle sur Val plutôt que sur i etj.

   Réfléchissons deux minutes : les 3 premières valeurs de Val correspondront à i=1, les trois suivantes à i=2. Par conséquent, on a i=fix(Val/3)+1, et j=Val-3*i. L'algorithme s'écrira donc :

```
% Tableau X(2, 3) en Entier
% Variables i, j, val en Entier
% Début
for Val=1:6
  i=fix(Val/3)+1
  j=Val-3*i
  X(i,j)=Val;
end
...
% Fin
```

7.7 Exercices non corrigés

1. Ecrivez un algorithme permettant à l'utilisateur de saisir un nombre quelconque de valeurs, qui devront être stockées dans un tableau. L'utilisateur doit donc commencer par entrer le nombre de valeurs qu'il compte saisir. Il effectuera ensuite cette saisie. Enfin, une fois la saisie terminée, le programme affichera le nombre de valeurs négatives et le nombre de valeurs positives.
2. Ecrivez un algorithme constituant un tableau, à partir de deux tableaux de même longueur préalablement saisis. Le nouveau tableau sera la somme des éléments des deux tableaux de départ.
3. Toujours à partir de deux tableaux précédemment saisis, écrivez un algorithme qui calcule le schtroumpf des deux tableaux. Pour calculer le schtroumpf, il faut multiplier chaque élément du tableau 1 par chaque élément du tableau 2, et additionner le tout.
4. Ecrivez un algorithme permettant, toujours sur le même principe, à l'utilisateur de saisir un nombre déterminé de valeurs. Le programme, une fois la saisie terminée, renvoie la plus grande valeur en précisant quelle position elle occupe dans le tableau. On prendra soin d'effectuer la saisie dans un premier temps, et la recherche de la plus grande valeur du tableau dans un second temps.
5. Ecrire un algorithme qui demande un nombre et crée un tableau avec sa table de multiplication par 20.
6. Quel résultat produira cet algorithme ?

```
% Tableau X(2,3) en Entier
% Variables i, j, val en Entier
% Début
Val=1
for i=1:2
  for j=1:3
    X(i,j)=Val
    Val=Val + 1
  end
end
```

```
for j=1:3
  for i=1:2
    display(X(i, j))
  end
end
% Fin
```

7. Quel résultat produira cet algorithme ?

```
% Tableau X(2,3) en Entier
% Variables i, j, val en Entier
% Début
Val=1
for i=1:2
  for j=1:3
    X(i,j)=Val^2+2*i
    Val=Val + 1
  end
end
for i=1:3
  for j=1:2
    display(X(i, j))
  end
end
% Fin
```

8. Quel résultat produira cet algorithme ?

```
% Tableau T(4, 2) en Entier
% Variables k, m, en Entier
% Début
for k=1:4
  for m=1:2
    T(k, m)=k + m
  end
end
for k=1:4
  for m=1:2
    display(T(k, m))
  end
end
% Fin
\end{enumerate}
```

9. Soit un tableau T à deux dimensions (12, 8) préalablement rempli de valeurs numériques. Écrire un algorithme qui recherche la plus grande valeur au sein de ce tableau, et en donne les numéros de ligne et de colonne.

10. Écrire un algorithme de jeu de dames très simplifié. L'ordinateur demande à l'utilisateur dans quelle case se trouve son pion (quelle ligne, quelle colonne). On met en place un contrôle de saisie afin de vérifier la validité des valeurs entrées. Ensuite, on demande à l'utilisateur quel mouvement il veut effectuer : 0 (en haut à gauche), 1 (en haut à droite), 2 (en bas à gauche), 3 (en bas à droite). Si le mouvement est impossible (i.e. on sort du damier), on le signale à l'utilisateur et on s'arrête là . Sinon, on déplace le pion et on affiche le damier résultant, en affichant un « O »pour une case vide et un « X »pour la case où se trouve le pion.

Chapitre 8

Techniques Rusees

> Informatique : alliance d'une science inexacte et d'une activité humaine faillible
>
> Luc Fayard

Une fois n'est pas coutume, ce chapitre n'a pas pour but de présenter un nouveau type de données, un nouveau jeu d'instructions, ou que sais-je encore.

Son propos est de détailler quelques techniques de programmation qui possèdent deux grands points communs :

- leur connaissance est parfaitement indispensable
- elles sont un rien finaudes

Et que valent quelques kilos d'aspirine, comparés à l'ineffable bonheur procuré par la compréhension suprême des arcanes de l'algorithmique ? Hein ?

8.1 Tri d'un tableau : le tri par insertion

Première de ces ruses de sioux, et par ailleurs tarte à la crème absolue du programmeur, donc : le tri de tableau.

Combien de fois au cours d'une carrière (brillante) de développeur a-t-on besoin de ranger des valeurs dans un ordre donné ? C'est inimaginable. Aussi, plutôt qu'avoir à réinventer à chaque fois la roue, le fusil à tirer dans les coins, le fil à couper le roquefort et la poudre à maquiller, vaut-il mieux avoir assimilé une ou deux techniques solidement éprouvées, même si elles paraissent un peu ardues au départ.

Il existe plusieurs stratégies possibles pour trier les éléments d'un tableau ; nous en verrons deux : le tri par insertion, et le tri à bulles. Champagne !

Commençons par le tri par insertion.

Admettons que le but de la manIJuvre soit de trier un tableau de 12 éléments dans l'ordre croissant. La technique du tri par sélection est la suivante : on met en bonne position l'élément numéro 1, c'est-à-dire le plus petit. Puis en met en bonne position l'élément suivant. Et ainsi de suite jusqu'au dernier. Par exemple, si l'on part de :

```
45 122 12 3 21 78 64 53 89 28 84 46
```

On commence par rechercher, parmi les 12 valeurs, quel est le plus petit élément , et où il se trouve. On l'identifie en quatrième position (c'est le nombre 3), et on l'échange alors avec le premier élément (le nombre 45). Le tableau devient ainsi :

```
3 122 12 45 21 78 64 53 89 28 84 46
```

On recommence à chercher le plus petit élément, mais cette fois, seulement à partir du deuxième (puisque le premier est maintenant correct, on n'y touche plus). On le trouve en troisième position (c'est le nombre 12). On échange donc le deuxième avec le troisième :

```
3 12 122 45 21 78 64 53 89 28 84 46
```

On recommence à chercher le plus petit élément à partir du troisième (puisque les deux premiers sont maintenant bien placés), et on le place correctement, en l'échangeant, ce qui donnera in fine :

```
3 12 21 45 122 78 64 53 89 28 84 46
```

Et cetera, et cetera, jusqu'à l'avant dernier. En bon français, nous pourrions décrire le processus de la manière suivante :

Boucle principale : prenons comme point de départ le premier élément, puis le second, etc, jusqu'à l'avant dernier.

Boucle secondaire : à partir de ce point de départ mouvant, recherchons jusqu'à la fin du tableau quel et le plus petit élément.

Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ. Cela s'écrit (en omettant les déclarations de variables et tout ce genre de trucs) :

```
% boucle principale : le point de départ se décale à chaque tour
for i=1:11
% on considère provisoirement que t(i) est le plus petit élément
  posmini=i
% on examine tous les éléments suivants
  for j=i+1:11
    if (t(j)<t(posmini))
       posmini=j
    end
  end
% A cet endroit, on sait maintenant où est le plus petit élément. Il ne reste plus qu'à effectuer la
  temp=t(posmini)
  t(posmini)=t(i)
  t(i)=temp
% On a placé correctement l'élément numéro i, on passe à présent au suivant.
end
```

Notons ici l'emploi des commentaires pour expliquer, à chaque étape, ce que l'on fait... Ca a l'air bête, mais ca rend le code lisible pour un autre être humain, ce qui n'est pas si mal. Dans le même ordre d'idée, il n'aura pas échappé à votre sagacité que j'ai décalé progressivement les instructions, de façon telle que les `end` se trouvent en face de l'instruction qu'ils cloturent. Ce n'est pas rigoureusement indispensable, mais c'est drôlement pratique.

Par conséquent, l'oubli de convivialité dans les codes sera chatié, par le correcteur, de la façon qui s'impose, avec la plus grande sévérité.

8.2 Un exemple de flag : la recherche dans un tableau

Nous allons maintenant nous intéresser au maniement habile d'une variable booléenne : la technique dite du « flag ».

Le flag, en anglais, est un petit drapeau, qui va rester baissé aussi longtemps que l'événement attendu ne se produit pas. Et, aussitôt que cet événement a lieu, le petit drapeau se lève (la variable booléenne change de valeur). Ainsi, la valeur finale de la variable booléenne permet au programmeur de savoir si l'événement a eu lieu ou non.

Tout ceci peut vous sembler un peu fumeux, mais cela devrait s'éclairer à l'aide d'un exemple extrêmement fréquent : tester la présence d'une valeur dans un tableau. On en profitera au passage pour

corriger une erreur particulièrement fréquente chez le programmeur débutant. (Si... je vous assure, il arrive qu'un programmeur débutant fasse des erreurs... C'est assez rare, heureusement).

Soit un tableau comportant, disons, 20 valeurs. L'on doit écrire un algorithme saisissant un nombre au clavier, et qui informe l'utilisateur de la présence ou de l'absence de la valeur saisie dans le tableau.

La première étape, évidente, consiste à écrire les instructions de lecture / écriture, et la boucle - car il y en a manifestement une - de parcours du tableau :

```
% Tableau Tab(20) en Numérique
% Variable N en Numérique
% Début
N=input('Entrez la valeur à rechercher')
for i=1:20
???
end
% Fin
```

Il nous reste à combler les points d'interrogation de la boucle for. Évidemment, il va falloir comparer N à chaque élément du tableau : si les deux valeurs sont égales, alors bingo, N fait partie du tableau. Cela va se traduire, bien entendu, par un if. Et voilà le programmeur raisonnant hâtivement qui se vautre en écrivant :

```
% Tableau Tab(20) en Numérique
% Variable N en Numérique
% Début
N=input('Entrez la valeur à rechercher')
for i=1:20
  if (N==Tab(i))
    display('N fait partie du tableau')
  else
    display('N ne fait pas partie du tableau')
  end
end
% Fin
```

Et patatras, cet algorithme est une véritable catastrophe.

Il suffit d'ailleurs de le faire tourner mentalement pour s'en rendre compte. De deux choses l'une : ou bien la valeur N figure dans le tableau, ou bien elle n'y figure pas. Mais dans tous les cas, l'algorithme ne doit produire qu'une seule réponse, quel que soit le nombre d'éléments que compte le tableau. Or, l'algorithme ci-dessus envoie à l'écran autant de messages qu'il y a de valeurs dans le tableau, en l'occurrence pas moins de 20 !

Il y a donc une erreur manifeste de conception : l'écriture du message ne peut se trouver à l'intérieur de la boucle : elle doit figurer à l'extérieur. On sait si la valeur était dans le tableau ou non uniquement lorsque le balayage du tableau est entièrement accompli. Nous réécrivons donc cet algorithme en plaçant le test après la boucle. Faute de mieux, on se contentera de faire dépendre pour le moment la réponse d'une variable booléenne que nous appellerons Trouvé.

```
% Tableau Tab(20) en Numérique
% Variable N en Numérique
% Variable Trouvé en boléen
% Début
N=input('Entrez la valeur à rechercher')
for i=1:20
   ???
end
if Trouvé
  display('N fait partie du tableau')
```

```
else
  display('N ne fait pas partie du tableau')
end
% Fin
```

Il ne nous reste plus qu'à gérer la variable Trouvé. Ceci se fait en deux étapes. un test figurant dans la boucle, indiquant lorsque la variable Trouvé doit devenir vraie (à savoir, lorsque la valeur N est rencontrée dans le tableau). Et attention : le test est asymétrique. Il ne comporte pas de else. On reviendra là-dessus dans un instant.

Last, but not least, l'affectation par défaut de la variable Trouvé, dont la valeur de départ doit être évidemment false.

```
% Tableau Tab(20) en Numérique
% Variable N en Numérique
% Variable Trouvé en boléen
% Début
N=input('Entrez la valeur à rechercher')
Trouvé=false
for i=1:20
  if (N==Tab(i))
    Trouvé=true
  end
end
if Trouvé
  display('N fait partie du tableau')
else
  display('N ne fait pas partie du tableau')
end
% Fin
```

Méditons un peu sur cette affaire.

La difficulté est de comprendre que dans une recherche, le problème ne se formule pas de la même manière selon qu'on le prend par un bout ou par un autre. On peut résumer l'affaire ainsi : il suffit que N soit égal à une seule valeur de Tab pour qu'elle fasse partie du tableau. En revanche, il faut qu'elle soit différente de toutes les valeurs de Tab pour qu'elle n'en fasse pas partie. Voilà la raison qui nous oblige à passer par une variable booléenne , un « drapeau »qui peut se lever, mais jamais se rabaisser. Et cette technique de flag (que nous pourrions élégamment surnommer « gestion asymétrique de variable booléenne ») doit être mise en IJuvre chaque fois que l'on se trouve devant pareille situation.

Autrement dit, connaître ce type de raisonnement est indispensable, et savoir le reproduire à bon escient ne l'est pas moins.

8.3 Tri de tableau + flag = tri à bulles

Et maintenant, nous en arrivons à la formule magique : tri de tableau + flag = tri à bulles.

L'idée de départ du tri à bulles consiste à se dire qu'un tableau trié en ordre croissant, c'est un tableau dans lequel tout élément est plus petit que celui qui le suit. Cette constatation percutante semble digne de M. de Lapalisse, un ancien voisin à moi. Mais elle est plus profonde – et plus utile – qu'elle n'en a l'air.

En effet, prenons chaque élément d'un tableau, et comparons-le avec l'élément qui le suit. Si l'ordre n'est pas bon, on permute ces deux éléments. Et on recommence jusqu'à ce que l'on n'ait plus aucune permutation à effectuer. Les éléments les plus grands « remontent »ainsi peu à peu vers les dernières places, ce qui explique la charmante dénomination de « tri à bulle ». Comme quoi l'algorithmique n'exclut pas un minimum syndical de sens poétique.

En quoi le tri à bulles implique-t-il l'utilisation d'un flag ? Eh bien, par ce qu'on ne sait jamais par avance combien de remontées de bulles on doit effectuer. En fait, tout ce qu'on peut dire, c'est qu'on

devra effectuer le tri jusqu'à ce qu'il n'y ait plus d'éléments qui soient mal classés. Ceci est typiquement un cas de question « asymétrique » : il suffit que deux éléments soient mal classés pour qu'un tableau ne soit pas trié. En revanche, il faut que tous les éléments soient bien rangés pour que le tableau soit trié.

Nous baptiserons le flag Yapermute, car cette variable booléenne va nous indiquer si nous venons ou non de procéder à une permutation au cours du dernier balayage du tableau (dans le cas contraire, c'est signe que le tableau est trié, et donc qu'on peut arrêter la machine à bulles). La boucle principale sera alors :

```
% Variable Yapermute en Booléen
% Début
...
while(Yapermute)
...
end
% Fin
```

Que va-t-on faire à l'intérieur de la boucle ? Prendre les éléments du tableau, du premier jusqu'à l'avant-dernier, et procéder à un échange si nécessaire. C'est parti :

```
% Variable Yapermute en Booléen
% Début
...
while(Yapermute)
   for i=0:10
      if (t(i)>t(i+1))
        temp=t(i)
        t(i)=t(i+1)
        t(i+1)=temp
      end
   end
end
% Fin
```

Mais il ne faut pas oublier un détail capital : la gestion de notre flag. L'idée, c'est que cette variable va nous signaler le fait qu'il y a eu au moins une permutation effectuée. Il faut donc :

- lui attribuer la valeur true dès qu'une seule permutation a été faite (il suffit qu'il y en ait eu une seule pour qu'on doive tout recommencer encore une fois).
- la remettre à false à chaque tour de la boucle principale (quand on recommence un nouveau tour général de bulles, il n'y a pas encore eu d'éléments échangés),
- ne pas oublier de lancer la boucle principale, et pour cela de donner la valeur true au flag au tout départ de l'algorithme.

La solution complète donne donc :

```
% Variable Yapermute en Booléen
% Début
...
Yapermut=Vrai
while (Yapermut)
  Yapermut=false
  for i=1:11
      if (t(i)>t(i+1))
        temp=t(i)
        t(i)=t(i+1)
        t(i+1)=temp
        Yapermute=true
      end
```

```
  end
end
% Fin
```

Au risque de me répéter, la compréhension et la maîtrise du principe du flag font partie de l'arsenal du programmeur bien armé.

8.4 La recherche dichotomique

Je ne sais pas si on progresse vraiment en algorithmique, mais en tout cas, qu'est-ce qu'on apprend comme vocabulaire !

Blague dans le coin, nous allons terminer ce chapitre migraineux par une technique célèbre de recherche, qui révèle toute son utilité lorsque le nombre d'éléments est très élevé. Par exemple, imaginons que nous ayons un programme qui doive vérifier si un mot existe dans le dictionnaire. Nous pouvons supposer que le dictionnaire a été préalablement entré dans un tableau (à raison d'un mot par emplacement). Ceci peut nous mener à, disons à la louche, 40 000 mots.

Une première manière de vérifier si un mot se trouve dans le dictionnaire consiste à examiner successivement tous les mots du dictionnaire, du premier au dernier, et à les comparer avec le mot à vérifier. Ca marche, mais cela risque d'être long : si le mot ne se trouve pas dans le dictionnaire, le programme ne le saura qu'après 40 000 tours de boucle ! Et même si le mot figure dans le dictionnaire, la réponse exigera tout de même en moyenne 20 000 tours de boucle. C'est beaucoup, même pour un ordinateur.

Or, il y a une autre manière de chercher, bien plus intelligente pourrait-on dire, et qui met à profit le fait que dans un dictionnaire, les mots sont triés par ordre alphabétique. D'ailleurs, un être humain qui cherche un mot dans le dictionnaire ne lit jamais tous les mots, du premier au dernier : il utilise lui aussi le fait que les mots sont triés.

Pour une machine, quelle est la manière la plus rationnelle de chercher dans un dictionnaire ? C'est de comparer le mot à vérifier avec le mot qui se trouve pile poil au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans le première moitié du dico. Sinon, on sait maintenant qu'on devra le chercher dans la deuxième moitié . A partir de là, on prend la moitié de dictionnaire qui nous reste, et on recommence : on compare le mot à chercher avec celui qui se trouve au milieu du morceau de dictionnaire restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite. A force de couper notre dictionnaire en deux, puis encore en deux, etc. on va finir par se retrouver avec des morceaux qui ne contiennent plus qu'un seul mot. Et si on n'est pas tombé sur le bon mot à un moment ou à un autre, c'est que le mot à vérifier ne fait pas partie du dictionnaire.

Regardons ce que cela donne en terme de nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire.

- Au départ, on cherche le mot parmi 40 000.
- Après le test n°1, on ne le cherche plus que parmi 20 000.
- Après le test n°2, on ne le cherche plus que parmi 10 000.
- Après le test n°3, on ne le cherche plus que parmi 5 000.
- etc.
- Après le test n°15, on ne le cherche plus que parmi 2.
- Après le test n°16, on ne le cherche plus que parmi 1.

Et là, on sait que le mot n'existe pas.

Moralité : on a obtenu notre réponse en 16 opérations contre 40 000 précédemment ! Il n'y a pas photo sur l'écart de performances entre la technique barbare et la technique futée. Attention, toutefois, même si c'est évident, je le répète avec force : la recherche dichotomique ne peut s'effectuer que sur des éléments préalablement triés.

Eh bien maintenant que je vous ai expliqué comment faire, vous n'avez plus qu'à traduire ! Au risque de me répéter, la compréhension et la maîtrise du principe du flag font partie du bagage du programmeur bien outillé.

8.5 Exercices corrigés

1. Ecrivez un algorithme qui inverse l'ordre des éléments d'un tableau dont on suppose qu'il a été préalablement saisi.

   Ici, il y a différente manière de procéder. La plus simple est de partir d'un tableau provisoir de meme longueur.

```
% tableau tab, provis en numérique
% variable i en numérique
%
% Début
N=length(tab);
for i=1:N
   provis(i)=tab(N-i+1);
end
for i=1:N
   tab(i)=provis(i)
end
% fin
```

2. Ecrivez un algorithme qui permette à l'utilisateur de supprimer une valeur d'un tableau préalablement saisi. L'utilisateur donnera l'indice de la valeur qu'il souhaite supprimer. Attention, il ne s'agit pas de remettre une valeur à zéro, mais bel et bien de la supprimer du tableau lui-même ! Si le tableau de départ était :

   ```
   12 8 4 45 64 9 2
   ```

   Et que l'utilisateur souhaite supprimer la valeur d'indice 4, le nouveau tableau sera :

   ```
   12 8 4 45 9 2
   ```

   Le plus simple, c'est de créer un tableau avec toutes les valeurs sauf celle-là. Il faut juste faire attention aux bords.

```
%
% tableau tab, provis en numérique
% variable i,j en numérique
% Début
i=input('Quelle valeur voulez-vous supprimer?')
N=length(tab)
if (i==1)
  for j=2:N
    provis(j-1)=tab(j)
  end
elseif (i==N)
  for j=1:N-1
    provis(j)=tab(j)
  end
elseif (i>N)
  display('Seulement ',num2str(N), 'valeurs')
else
  for j=1:i-1
    provis(j)=tab(j)
  end
  for j=i:N-1
    provis(j)=tab(j+1)
  end
end
```

3. Ecrivez un algorithme qui permette de saisir un nombre quelconque de valeurs, et qui les range au fur et à mesure dans un tableau. Le programme, une fois la saisie terminée, doit dire si les éléments du tableau sont ordonnés ou non.

   Notons pour commencer qu'il suffit d'une seule paire de nombres non ordonnée pour que les nombres ne soient pas consécutifs.

```
% tableau tab en numérique
% variable flag en boléen
% variable x en numérique
%
% Début
%
N=input('Combien de valeurs voulez-vous entrer?')
for i=1:N
   tab(i)=input(['Entrer la valeur ',num2str(i)])
end
flag=true
for i=1:N-1
   if(tab(i)>tab(i+1))
     flag=false
   end
end
if flag
  display('Les nombres sont ordonnés');
else
  display('Les nombres ne sont pas ordonnés');
end
```

4. Ecrivez un algorithme qui place un élément dans un tableau de nombres ordonnés, de façon à conserver l'ordre. On suppose que l'élément à une valeur comprise dans l'intervalle des valeurs des nombres du tableau.

   Bon... Allons-y dans la bonne humeur...

   Soit un tableau tab de longueur N. L'élément x doit se placer en tab(j), avec j tel que x>tab(j-1)& x<tab(j+1), c'est-à-dire entre l'élément j et l'élément j+1, avant son introduction. Il ne reste donc qu'à trouver j.

   On procède comme pour la recherche d'un mot dans le dictionnaire. On part en j=N/2 (supposons N pair), et compare x aux valeurs j et j+1. Trois cas peuvent se présenter :

   x>tab(j+1) , alors, on prend la moitié de la partie au-dessus (j=j+(N-j)/2), et on reteste.

   x<tab(j) , alors, on prend la moitié de la partie au-dessus (j=j-(N-j)/2), et on reteste.

   x>=tab(j)& x<=tab(j+1) , alors, on a fini, on met x en tab(j+1) après avoir décalé tous les autres.

   Si on doit retester, on refait pareil, mais on ne fait plus que la moitié du déplacement :

   x>tab(j+1) , alors, on prend la moitié de la partie au-dessus (j=j+(N-j)/4), et on reteste.

   x<tab(j) , alors, on prend la moitié de la partie au-dessus (j=j+(N-j)/4), et on reteste.

   x>=tab(j)& x<=tab(j+1) , alors, on a fini, on met x en tab(j+1) après avoir décalé tous les autres.

   et ainsi de suite jusqu'à ce que cela marche. Ecrivons donc cela...

```
% Tableau tab en numérique
% variable x, N, dj, j, ntour en numérique
% variable flag en boléen
%
% Début
%
% On définit le tableau
```

```
tab=[1 2 3 4 5 6 7 8 9];
% L'élément à ranger.
x=3.5;
% La longueur du tableau
N=length(tab);
% On initialise le flag.
flag=false;
% Et le pas de recherche.
dj=fix(N/2);
% On part du milieu
j=dj;
% Le numéro du tour, pour la variation du pas de recherche
ntour=1;
%   Boucle while
while(flag==false)
% On change le pas de recherche, mais on le garde entier et non nul
  dj=fix(dj/2/ntour);
  if (dj<1)
      dj=1;
  end
% On compare x aux valeurs avant et apres j dans le tablea
  if (x>tab(j+1))
    j=j+dj;
  elseif (x<tab(j))
    j=j-dj;
  else
% S'il est entre les 2 (en autorisant une égalité), on le met là...
% Donc on décale ceux apres
    for k=N+1:-1:j+2
      tab(k)=tab(k-1);
    end
    tab(j+1)=x;
% Et on change le flag pour arreter de boucler
    flag=true;
  end
% on incrémente ntour
  ntour=ntour+1;
end
% Fin
```

8.6 Exercices non corrigés

1. Ecrivez un algorithme qui trie un tableau dans l'ordre décroissant. Vous écrirez bien entendu deux versions de cet algorithme, l'une employant le tri par insertion, l'autre le tri à bulles.
2. Ecrivez un algorithme qui inverse l'ordre des éléments d'un tableau dont on suppose qu'il a été préalablement saisi (« les premiers seront les derniers... »)
3. Ecrire un algorithme qui recherche un nombre donné dans un tableau de nombre, et qui sort le nombre de fois qu'il apparaît et la liste des indices correspondants.
4. Ecire un algoritme qui prend un mot et dit combien de voyelles et de consonnes il contient.

Chapitre 9

Les Fichiers

> On ne peut pas davantage créer des fichiers numériques non copiables que créer de l'eau non humide
>
> Bruce Schneier

> Comme la Hongrie, le monde informatique a une langue qui lui est propre. Mais il y a une différence. Si vous restez assez longtemps avec des Hongrois, vous finirez bien par comprendre de quoi ils parlent.
>
> Dave Barry

Jusqu'à présent, les informations utilisées dans nos programmes ne pouvaient provenir que de deux sources : soit elles étaient inclues dans l'algorithme lui-même, par le programmeur, soit elles étaient entrées en cours de route par l'utilisateur. Mais évidemment, cela ne suffit pas à combler les besoins réels des informaticiens.

Imaginons que l'on veuille écrire un programme gérant un carnet d'adresses. D'une exécution du programme à l'autre, l'utilisateur doit pouvoir retrouver son carnet à jour, avec les modifications qu'il y a apportées la dernière fois qu'il a exécuté le programme. Les données du carnet d'adresse ne peuvent donc être inclues dans l'algorithme, et encore moins être entrées au clavier à chaque nouvelle exécution !

Les fichiers sont là pour combler ce manque. Ils servent à stocker des informations de manière "permanente", entre deux exécutions d'un programme. Car si les variables, qui sont je le rappelle des adresses de mémoire vive, disparaissent à chaque fin d'exécution, les fichiers, eux sont stockés sur des périphériques à mémoire de masse (disquette, disque dur, CD Rom...).

9.0.1 Organisation des fichiers

Vous connaissez tous le coup des papous : « chez les papous, il y a les papous papas et les papous pas papas. Chez les papous papas, il y a les papous papas à poux et les papous papas pas à poux, etc. »Eh bien les fichiers, c'est un peu pareil : il y a des catégories, et dans les catégories, des sortes, et dans les sortes des espèces. Essayons donc de débroussailler un peu tout cela...

Un premier grand critère, qui différencie les deux grandes catégories de fichiers, est le suivant : le fichier est-il ou non organisé sous forme de lignes successives ? Si oui, cela signifie vraisemblablement que ce fichier contient le même genre d'information à chaque ligne. Ces lignes sont alors appelées des enregistrements.

Afin d'illuminer ces propos obscurs, prenons le cas classique, celui d'un carnet d'adresses. Le fichier est destiné à mémoriser les coordonnées (ce sont toujours les plus mal chaussées, bien sûr) d'un certain nombre de personnes. Pour chacune, il faudra noter le nom, le prénom, le numéro de téléphone et l'email. Dans ce cas, il peut paraître plus simple de stocker une personne par ligne du fichier (par enregistrement). Dit autrement, quand on prendra une ligne, on sera sûr qu'elle contient les informations concernant une personne, et uniquement cela. Un fichier ainsi codé sous forme d'enregistrements est appelé un fichier texte.

En fait, entre chaque enregistrement, sont stockés les octets correspondants aux caractères CR (code Ascii 13) et LF (code Ascii 10), signifiant un retour au début de la ligne suivante. Le plus souvent, le langage de programmation, dès lors qu'il s'agit d'un fichier texte, gèrera lui-même la lecture et l'écriture de ces deux caractères à chaque fin de ligne : c'est autant de moins dont le programmeur aura à s'occuper. Le programmeur, lui, n'aura qu'à dire à la machine de lire une ligne, ou d'en écrire une.

Ce type de fichier est couramment utilisé dès lors que l'on doit stocker des informations pouvant être assimilées à une base de données. Le second type de fichier, vous l'aurez deviné, se définit a contrario : il rassemble les fichiers qui ne possèdent pas de structure de lignes (d'enregistrement). Les octets, quels qu'il soient, sont écrits à la queue leu leu. Ces fichiers sont appelés des fichiers binaires. Naturellement, leur structure différente implique un traitement différent par le programmeur. Tous les fichiers qui ne codent pas une base de données sont obligatoirement des fichiers binaires : cela concerne par exemple un fichier son, une image, un programme exécutable, etc. . Toutefois, on en dira quelques mots un peu plus loin, il est toujours possible d'opter pour une structure binaire même dans le cas où le fichier représente une base de données.

Autre différence majeure entre fichiers texte et fichiers binaires : dans un fichier texte, toutes les données sont écrites sous forme de... texte (étonnant, non ?). Cela veut dire que les nombres y sont représentés sous forme de suite de chiffres (des chaînes de caractères). Ces nombres doivent donc être convertis en chaînes lors de l'écriture dans le fichier. Inversement, lors de la lecture du fichier, on devra convertir ces chaînes en nombre si l'on veut pouvoir les utiliser dans des calculs. En revanche, dans les fichiers binaires, les données sont écrites à l'image exact de leur codage en mémoire vive, ce qui épargne toutes ces opérations de conversion. Ceci a comme autre implication qu'un fichier texte est directement lisible, alors qu'un fichier binaire ne l'est pas(sauf bien sîr en écrivant soi-même un programme approprié). Si l'on ouvre un fichier texte via un éditeur de textes, comme le bloc-notes de Windows, on y reconnaîtra toutes les informations (ce sont des caractères, stockés comme tels). La même chose avec un fichier binaire ne nous produit à l'écran qu'un galimatias de scribouillis incompréhensibles.

9.1 Structure des enregistrements

Savoir que les fichiers peuvent être structurés en enregistrements, c'est bien. Mais savoir comment sont à leur tour structurés ces enregistrements, c'est mieux. Or, là aussi, il y a deux grandes possibilités. Ces deux grandes variantes pour structurer les données au sein d'un fichier texte sont la délimitation et les champs de largeur fixe. Reprenons le cas du carnet d'adresses, avec dedans le nom, le prénom, le téléphone et l'email. Les données, sur le fichier texte, peuvent être organisées ainsi :

```
Structure nř1
'Fonfec';'Sophie';0142156487;'fonfec@yahoo.fr'
'Zétofrais';'Mélanie';0456912347;'zétofrais@free.fr'
'Herbien';'Jean-Philippe';0289765194;'vantard@free.fr'
'Hergébel';'Octave';0149875231;'rg@aol.fr'
```

ou ainsi :

```
Structure nř2
Fonfec          Sophie          0142156487fonfec@yahoo.fr
Zétofrais       Mélanie         0456912347zétofrais@free.fr
Herbien         Jean-Philippe   0289765194vantard@free.fr
Hergébel        Octave          0149875231rg@aol.fr
```

La structure n°1 est dite délimitée; Elle utilise un caractère spécial, appelé caractère de délimitation, qui permet de repérer quand finit un champ et quand commence le suivant. Il va de soi que ce caractère de délimitation doit être strictement interdit à l'intérieur de chaque champ, faute de quoi la structure devient proprement illisible.

La structure n°2, elle, est dite à champs de largeur fixe. Il n'y a pas de caractère de délimitation, mais on sait que les x premiers caractères de chaque ligne stockent le nom, les y suivants le prénom, etc. Cela impose bien entendu de ne pas saisir un renseignement plus long que le champ prévu pour l'accueillir.

L'avantage de la structure n°1 est son faible encombrement en place mémoire; il n'y a aucun espace perdu, et un fichier texte codé de cette manière occupe le minimum de place possible. Mais elle possède en revanche un inconvénient majeur, qui est la lenteur de la lecture. En effet, chaque fois que l'on récupère une ligne dans le fichier, il faut alors parcourir un par un tous les caractères pour repérer chaque occurrence du caractère de séparation avant de pouvoir découper cette ligne en différents champs.

La structure n°2, à l'inverse, gaspille de la place mémoire, puisque le fichier est un vrai gruyère plein de trous. Mais d'un autre côté, la récupération des différents champs est très rapide. Lorsqu'on récupère une ligne, il suffit de la découper en différentes chaînes de longueur prédéfinie, et le tour est joué.

A l'époque où la place mémoire coûtait cher, la structure délimitée était souvent privilégiée. Mais depuis bien des années, la quasi-totalité des logiciels - et des programmeurs - optent pour la structure en champs de largeur fixe. Aussi, sauf mention contraire, nous ne travaillerons qu'avec des fichiers bâtis sur cette structure.

Remarque importante : lorsqu'on choisit de coder une base de données sous forme de champs de largeur fixe, on peut alors très bien opter pour un fichier binaire. Les enregistrements y seront certes à la queue leu leu, sans que rien ne nous signale la jointure entre chaque enregistrement. Mais si on sait combien d'octets mesure invariablement chaque champ, on sait du coup combien d'octets mesure chaque enregistrement. Et on peut donc très facilement récupérer les informations : si je sais que dans mon carnet d'adresse, chaque individu occupe mettons 75 octets, alors dans mon fichier binaire, je déduis que l'individu n°1 occupe les octets 1 à 75, l'individu n°2 les octets 76 à 150, l'individu n°3 les octets 151 à 225, etc.

9.2 Types d'accès

On vient de voir que l'organisation des données au sein des enregistrements du fichier pouvait s'effecteur selon deux grands choix stratégiques. Mais il existe une autre ligne de partage des fichiers : le type d'accès, autrement dit la manière dont la machine va pouvoir aller rechercher les informations contenues dans le fichier. On distingue :

L'accès séquentiel : on ne peut accéder qu'à la donnée suivant celle qu'on vient de lire. On ne peut donc accéder à une information qu'en ayant au préalable examiné celle qui la précède. Dans le cas d'un fichier texte, cela signifie qu'on lit le fichier ligne par ligne (enregistrement par enregistrement).

L'accès direct (ou aléatoire) : on peut accéder directement à l'enregistrement de son choix, en précisant le numéro de cet enregistrement. Mais cela veut souvent dire une gestion fastidieuse des déplacements dans le fichier.

L'accès indexé : pour simplifier, il combine la rapidité de l'accès direct et la simplicité de l'accès séquentiel (en restant toutefois plus compliqué). Il est particulièrement adapté au traitement des gros fichiers, comme les bases de données importantes.

A la différence de la précédente, cette typologie ne caractérise pas la structure elle-même du fichier. En fait, tout fichier peut être utilisé avec l'un ou l'autre des trois types d'accès. Le choix du type d'accès n'est pas un choix qui concerne le fichier lui-même, mais uniquement la manière dont il va être traité par la machine. C'est donc dans le programme, et seulement dans le programme, que l'on choisit le type d'accès souhaité.

Dans le cadre de ce cours, on se limitera volontairement au type de base : le fichier texte en accès séquentiel. Pour des informations plus complètes sur la gestion des fichiers binaires et des autres types d'accès, il vous faudra... chercher ailleurs.

9.3 Instructions (fichiers texte en accès séquentiel)

Si l'on veut travailler sur un fichier, la première chose à faire est de l'ouvrir. Cela se fait en attribuant au fichier un numéro de canal. On ne peut ouvrir qu'un seul fichier par canal, mais quel que soit le langage, on dispose toujours de plusieurs canaux, donc pas de soucis.

L'important est que lorsqu'on ouvre un fichier, on stipule ce qu'on va en faire : lire, écrire ou ajouter.

- Si on ouvre un fichier pour lecture, on pourra uniquement récupérer les informations qu'il contient, sans les modifier en aucune manière.
- Si on ouvre un fichier pour écriture, on pourra mettre dedans toutes les informations que l'on veut. Mais les informations précédentes, si elles existent, seront intégralement écrasées Et on ne pourra pas accéder aux informations qui existaient précédemment.
- Si on ouvre un fichier pour ajout, on ne peut ni lire, ni modifier les informations existantes. Mais on pourra, comme vous commencez à vous en douter, ajouter de nouvelles lignes (je rappelle qu'au terme de lignes, on préférera celui d'enregistrements).

Au premier abord, ces limitations peuvent sembler infernales. Au deuxième rabord, elles le sont effectivement. Il n'y a même pas d'instructions qui permettent de supprimer un enregistrement d'un fichier !

Toutefois, avec un peu d'habitude, on se rend compte que malgré tout, même si ce n'est pas toujours marrant, on peut quand même faire tout ce qu'on veut avec ces fichiers séquentiels.

Pour ouvrir un fichier texte, on écrira par exemple :

```
fid=fopen('exemple.txt','r')
```

Ici, 'exemple.txt' est le nom du fichier sur le disque dur, fid est le numéro de canal, il a été attribué automatiquement par l'ordinateur, et ce fichier a donc été ouvert en lecture, comme l'indique le 'r'. Vous l'aviez sans doute pressenti. Allons plus loin :

```
% Variables Truc, Nom, Prénom, Tel, Mail en Caractères
% variable fid en numérique
% Début
fid=fopen('exemple.txt','r')
Truc=fgets(fid)
Nom=Truc(1:20)
Prénom=Truc(21:15)
Tel=Truc(36:10)
Mail=Truc(46:20)
```

L'instruction fgets récupère donc dans la variable spécifiée l'enregistrement suivant dans le fichier... "suivant", oui, mais par rapport à quoi ? Par rapport au dernier enregistrement lu. C'est en cela que le fichier est dit séquentiel. En l'occurrence, on récupère donc la première ligne, donc, le premier enregistrement du fichier, dans la variable Truc. Ensuite, le fichier étant organisé sous forme de champs de largeur fixe, il suffit de tronçonner cette variable Truc en autant de morceaux qu'il y a de champs dans l'enregistrement, et d'envoyer ces tronçons dans différentes variables. Et le tour est joué.

La suite du raisonnement s'impose avec une logique impitoyable : lire un fichier séquentiel de bout en bout suppose de programmer une boucle. Comme on sait rarement à l'avance combien d'enregistrements comporte le fichier, MATLAB renvoie gentiment la valeur -1 lorsqu'il n'y a plus d'argument à lire. L'algorithme, ultra classique, en pareil cas est donc :

```
% Variable Truc en Caractère
% variable fid, ff en numérique
% Début
fid=fopen('exemple.txt','r')
ff=-1
while (ff==-1)
  Truc=fgets(fid)
end
fclose(fid)
%Fin
```

Et neuf fois sur dix également, si l'on veut stocker au fur et à mesure en mémoire vive les informations lues dans le fichier, on a recours à un ou plusieurs tableaux. Et comme on ne sait pas d'avance combien il y aurait d'enregistrements dans le fichier, on ne sait pas davantage combien il doit y avoir d'emplacements dans les tableaux. Qu'importe, les programmeurs avertis que vous êtes connaissent la combine des tableaux dynamiques. En rassemblant l'ensemble des connaissances acquises, nous pouvons donc écrire le prototype du code qui effectue la lecture intégrale d'un fichier séquentiel, tout en recopiant l'ensemble des informations en mémoire vive :

```
% Tableaux Nom, Prénom, Tel, Mail en Caractère
% variable i en numérique
% Début
fid=fopen('exemple.txt','r')
ff=-1
i=1
while (ff==-1)
  Truc=fgets(fid)
  Nom(i,:)=Truc(1:20)
  Prénom(i,:)=Truc(21:15)
  Tel(i,:)=Truc(36:10)
  Mail(i,:)=Truc(46:20)
end
fclose(fid)
%Fin
```

Ici, on a fait le choix de recopier le fichier dans quatre tableaux distincts. On aurait pu également tout recopier dans un seul tableau : chaque case du tableau aurait alors été occupée par une ligne complète (un enregistrement) du fichier. Cette solution nous aurait fait gagner du temps au départ, mais elle alourdit ensuite le code, puisque chaque fois que l'on a besoin d'une information au sein d'une case du tableau, il faudra aller procéder à une extraction. Ce qu'on gagne par un bout, on le perd donc par l'autre. Mais surtout, comme on va le voir bientôt, il y a autre possibilité, bien meilleure, qui cumule les avantages sans avoir aucun des inconvénients.

Néanmoins, ne nous impatientons pas, chaque chose en son temps, et revenons pour le moment à la solution que nous avons employée ci-dessus.

Pour une opération d'écriture, ou d'ajout, il faut d'abord impérativement, sous peine de semer la panique dans la structure du fichier, constituer une chaîne équivalente à la nouvelle ligne du fichier. Cette chaîne doit donc être « calibrée »de la bonne manière, avec les différents champs qui « tombent »aux emplacements corrects. Le moyen le plus simple pour s'épargner de longs traitements est de procéder avec des chaînes correctement dimensionnées dès au départ :

```
% Variable truc, nom, prenom, tel, mail en caractère
%
truc=banks(65);
nom='Tim'
prenom='Vincent'
tel='444719'
mail='vincent_tim@1euro'
truc(1:length(nom))=nom
truc(21:21+length(prenom)-1)=prenom
truc(36:36+length(tel)-1)=tel
truc(46:46+length(mail)-1)=mail
fid=fopen('Exemple.txt','a')
pif=fprintf(fid,'%s',truc)
fclose(fid)
```

Et pour finir, une fois qu'on en a terminé avec un fichier, il ne faut pas oublier de fermer ce fichier. On libère ainsi le canal qu'il occupait (et accessoirement, on pourra utiliser ce canal dans la suite du programme pour un autre fichier... ou pour le même).

9.4 Stratégies de traitement

Il existe globalement deux manières de traiter les fichiers textes :
l'une consiste à s'en tenir au fichier proprement dit, c'est-à-dire à modifier directement (ou presque) les informations sur le disque dur. C'est parfois un peu acrobatique, lorsqu'on veut supprimer un élément d'un fichier : on programme alors une boucle avec un test, qui recopie dans un deuxième fichier tous les éléments du premier fichier sauf un ; et il faut ensuite recopier intégralement le deuxième fichier à la place du premier fichier... Ouf.
l'autre stratégie consiste, comme on l'a vu, à passer par un ou plusieurs tableaux. En fait, le principe fondamental de cette approche est de commencer, avant toute autre chose, par recopier l'intégralité du fichier de départ en mémoire vive. Ensuite, on ne manipule que cette mémoire vive (concrètement, un ou plusieurs tableaux). Et lorsque le traitement est terminé, on recopie à nouveau dans l'autre sens, depuis la mémoire vive vers le fichier d'origine.
Les avantages de la seconde technique sont nombreux, et 99 fois sur 100, c'est ainsi qu'il faudra procéder :

- la rapidité : les accès en mémoire vive sont des milliers de fois plus rapides (nanosecondes) que les accès aux mémoires de masse (millisecondes au mieux pour un disque dur). En basculant le fichier du départ dans un tableau, on minimise le nombre ultérieur d'accès disque, tous les traitements étant ensuite effectués en mémoire.
- la facilité de programmation : bien qu'il faille écrire les instructions de recopie du fichier dans le tableau, pour peu qu'on doive tripoter les informations dans tous les sens, c'est largement plus facile de faire cela avec un tableau qu'avec des fichiers.

Pourquoi, alors, demanderez-vous haletants, ne fait-on pas cela à tous les coups ? Y a-t-il des cas où il vaut mieux en rester aux fichiers et ne pas passer par des tableaux ? La recopie d'un très gros fichier en mémoire vive exige des ressources qui peuvent atteindre des dimensions considérables. Donc, dans le cas d'immenses fichiers (très rares, cependant), cette recopie en mémoire peut s'avérer problématique. Toutefois, lorsque le fichier contient des données de type non homogènes (chaînes, numériques, etc.) cela risque d'être coton pour le stocker dans un tableau unique : il va falloir déclarer plusieurs tableaux, dont le maniement au final peut être aussi lourd que celui des fichiers de départ. A moins... d'utiliser une ruse : créer des types de variables personnalisés, composés d'un « collage »de plusieurs types existants (10 caractères, puis un numérique, puis 15 caractères, etc.). Ce type de variable s'appelle un type structuré. Cette technique, bien qu'elle ne soit pas vraiment difficile, exige tout de même une certaine aisance... Voilà pourquoi on va maintenant en dire quelques mots.

9.5 Données structurées

Nostalgiques du Lego, cette partie va vous plaire. Comment construire des trucs pas possibles et des machins pas croyables avec juste quelques éléments de base ? Vous n'allez pas tarder à le savoir...

Jusqu'à présent, voilà comment se présentaient nos possibilités en matière de mémoire vive : nous pouvions réserver un emplacement pour une information d'un certain type. Un tel emplacement s'appelle une variable (quand vous en avez assez de me voir radoter, vous le dites). Nous pouvions aussi réserver une série d'emplacement numérotés pour une série d'informations de même type. Un tel emplacement s'appelle un tableau (même remarque).

Eh bien toujours plus haut, toujours plus fort, voici maintenant que nous pouvons réserver une série d'emplacements pour des données de type différents. Un tel emplacement s'appelle une variable structurée. Son utilité, lorsqu'on traite des fichiers texte (et même, des fichiers en général), saute aux yeux : car on va pouvoir calquer chacune des lignes du fichier en mémoire vive, et considérer que chaque enregistrement sera recopié dans une variable et une seule, qui lui sera adaptée. Ainsi, le problème du "découpage" de chaque enregistrement en différentes variables (le nom, le prénom, le numéro de téléphone, etc.) sera résolu d'avance, puisqu'on aura une structure, un gabarit, en quelque sorte, tout prêt d'avance pour accueillir et prédécouper nos enregistrements. Attention toutefois ; lorsque nous utilisions des variables de type prédéfini, comme des entiers, des booléens, etc. nous n'avions qu'une seule opération à effectuer : déclarer la variable en utilisant un des types existants. A présent que nous voulons créer un nouveau type de variable (par assemblage de types existants), il va falloir faire deux choses : d'abord, créer le type. Ensuite seulement, déclarer la (les) variable(s) d'après ce type.

Reprenons une fois de plus l'exemple du carnet d'adresses. Je sais, c'est un peu comme mes blagues, ça lasse (là, pour ceux qui s'endorment, je signale qu'il y a un jeu de mots), mais c'est encore le meilleur moyen d'avoir un point de comparaison. Nous allons donc, avant même la déclaration des variables, créer un type, une structure, calquée sur celle de nos enregistrements, et donc prête à les accueillir. Ici MATLAB se distingue fortement d'autres langages comme le C, par exemple. Nous allons nous limiter au point de vue MATLAB, à chaque jour suffit sa peine.

```
bottin=struct('nom',{},'prenom',{},'tel',{},'mail',{})
```

Après cela, on peut remplir cette structure facilement :

```
clear all
bottin=struct('nom',{},'prenom',{},'tel',{},'mail',{});
bottin(1,1).nom='Canari'
bottin(1,2).nom='Yes'
bottin(1,3).nom='Bambel'
bottin(1,4).nom='Atrou'
bottin(1,1).prenom='John'
bottin(1,2).prenom='Jacques'
bottin(1,3).prenom='Henri'
bottin(1,4).prenom='Marcel'
bottin(1,1).tel='01 25484578'
bottin(1,2).tel='01 38454646'
bottin(1,3).tel='03 78798798'
bottin(1,4).tel='00 32 25484578'
bottin(1,1).mail='John@pifpaf.fr'
bottin(1,2).mail='Jacques@youpeee.fr'
bottin(1,3).mail='Henri@plouf.com'
bottin(1,4).mail='Marcel@zimboumtralala.be'
```

Ainsi, écrire correctement une information dans le fichier est un jeu d'enfant, puisqu'on dispose d'une variable Individu au bon gabarit. Une fois remplis les différents champs de cette variable - ce qu'on vient de faire -, il n'y a plus qu'à envoyer celle-ci directement dans le fichier. Et zou ! Malheureusement, à ma connaissance, en MATLAB, les structures ne s'écrivent qu'en binaire.

```
save saver_willy bottin
```

De la même manière, dans l'autre sens, lorsque j'effectue une opération de lecture dans le fichier, ma vie en sera considérablement simplifiée : la structure étant faite pour cela, je peux charger le bottin, le chipoter, et c'est fini.

9.6 Exercices corrigés

1. Quel résultat cet algorithme produit-il ?

   ```
   % Variable Truc en Caractère
   % Début
   fid=('monfichier.txt','r')
   Truc='';
   while(Truc~=-1)
     Truc=fgets(fid)
     display(Truc)
   end
   fclose(fid)
   % Fin
   ```

   On ouvre le fichier monfichier.txt en lecture. Tant que l'on atteint pas la fin du fichier, la condition du while n'est pas remplie, et on va donc lire toute les lignes, puis les écrire une à une à l'écran.

2. Ecrire un algorithme qui lit un fichier, et recopie dans un autre toutes les lignes qui ne commencent pas par le symbôle %.

```
% Variable Truc en Caractère
% Début
fid=('monfichier.txt','r')
fid=('monfichier_2.txt','w')
Truc='';
while(Truc~=-1)
  Truc=fgets(fid)
  if (Truc(1)~='%')
     fprintf(fid,'%s', Truc)
  end
end
fclose(fid)
% Fin
```

9.7 Exercices non corrigés

1. On travaille avec le fichier du carnet dŠadresses en champs de largeur fixe. Ecrivez un algorithme qui permet à lŠutilisateur de saisir au clavier un nouvel individu qui sera ajouté à ce carnet dŠadresses
2. Même question, mais cette fois le carnet est supposé être trié par ordre alphabétique. LŠindividu doit donc être inséré au bon endroit dans le fichier.
3. Ecrire un algorithme qui supprime dans notre carnet d'adresses tous les individus dont le mail est invalide (pour employer un critère simple, on considèrera que sont invalides les mails ne comportant aucune arobase, ou plus d'une arobase).

Chapitre 10

Les fonctions

Il y a deux méthodes pour écrire des programmes sans erreurs. Mais il n'y a que la troisième qui marche

Anonyme

L'informatique semble encore chercher la recette miracle qui permettra aux gens d'écrire des programmes corrects sans avoir à réfléchir. Au lieu de cela, nous devons apprendre aux gens comment réfléchir

Anonyme

10.1 Les fonctions prédéfinies

Certains traitements ne peuvent être effectués par un algorithme, aussi savant soit-il. D'autres ne peuvent l'être qu'au prix de souffrances indicibles.

C'est par exemple le cas du calcul du sinus d'un angle : pour en obtenir une valeur approchée, il faudrait appliquer une formule d'une complexité à vous glacer le sang. Aussi, que se passe-t-il sur les petites calculatrices que vous connaissez tous ? On vous fournit quelques touches spéciales, dites touches de fonctions, qui vous permettent par exemple de connaître immédiatement ce résultat. Sur votre calculatrice, si vous voulez connaître le sinus de 35°, vous taperez 35, puis la touche SIN, et vous aurez le résultat. Tout langage de programmation propose ainsi un certain nombre de fonctions ; certaines sont indispensables, car elles permettent d'effectuer des traitements qui seraient sans elles impossibles. D'autres servent à soulager le programmeur, en lui épargnant de longs - et pénibles - algorithmes.

10.1.1 Structure générale des fonctions

Reprenons l'exemple du sinus. Les langages informatiques, qui se doivent tout de même de savoir faire la même chose qu'une calculatrice à 2€, proposent généralement une fonction SIN. Si nous voulons stocker le sinus de 35 dans la variable A, nous écrirons :

```
A=sin(35)
```

Une fonction est donc constituée de trois parties :

- le nom proprement dit de la fonction. Ce nom ne s'invente pas ! Il doit impérativement correspondre à une fonction proposée par le langage. Dans notre exemple, ce nom est SIN.
- deux parenthèses, une ouvrante, une fermante. Ces parenthèses sont toujours obligatoires, même lorsqu'on n'écrit rien à l'intérieur.
- une liste de valeurs, indispensables à la bonne exécution de la fonction. Ces valeurs s'appellent des arguments, ou des paramètres.

Certaines fonctions exigent un seul argument, d'autres deux, etc. et d'autres encore aucun. A noter que même dans le cas de ces fonctions n'exigeant aucun argument, les parenthèses restent obligatoires. Le nombre d'arguments nécessaire pour une fonction donnée ne s'invente pas : il est fixé par le langage. Par exemple, la fonction sinus a besoin d'un argument (ce n'est pas surprenant, cet argument est la valeur de l'angle). Si vous essayez de l'exécuter en lui donnant deux arguments, ou aucun, cela déclenchera une erreur à l'exécution. Notez également que les arguments doivent être d'un certain type, et qu'il faut respecter ces types. Et d'entrée, nous trouvons : ŝubsectionLE GAG DE LA JOURNEE Il consiste à affecter une fonction, quelle qu'elle soit. Toute écriture plaçant une fonction à gauche d'une instruction d'affectation est aberrante, pour deux raisons symétriques. d'une part, parce que nous le savons depuis le premier chapitre de ce cours extraordinaire, on ne peut affecter qu'une variable, à l'exclusion de tout autre chose. ensuite, parce qu'une fonction a pour rôle de produire, de renvoyer, de valoir (tout cela est synonyme), un résultat. Pas d'en recevoir un, donc. L'affectation d'une fonction sera donc considérée comme l'une des pires fautes algorithmiques, et punie comme telle. Tavernier...

10.1.2 Les fonctions de texte

Une catégorie privilégiée de fonctions est celle qui nous permet de manipuler des chaînes de caractères. Nous avons déjà vu qu'on pouvait facilement « coller »deux chaînes l'une à l'autre avec l'opérateur de concaténation. Mais ce que nous ne pouvions pas faire, et qui va être maintenant possible, c'est pratiquer des extractions de chaînes (moins douloureuses, il faut le noter, que les extractions dentaires). Tous les langages, je dis bien tous, proposent peu ou prou les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

length(chaîne) : renvoie le nombre de caractères d'une chaîne, comme elle le faisait pour la longueur d'un tableau.

chaine(n1 :n2) : renvoie un extrait de la chaîne, commençant au caractère n1 et faisant n2 caractères de long, on y reviendra.

findstr(chaîne1,chaîne2) : renvoie un nombre correspondant à la position de chaîne2 dans chaîne1. Si chaîne2 n'est pas comprise dans chaîne1, la fonction renvoie un tableau vide. Si le string apparait plusieurs fois, le tableau comprend les différents indices.

Exemples :

length('Bonjour, ça va ?')	vaut	16
length('')	vaut	0
s='Zorro is back'		
s(4 :7)	vaut	'ro is b'
s(12 :12)	vaut	'c'
findstr('Un pur bonheur', 'pur')	vaut	4
findstr('Un pur bonheur', 'pif')	vaut	[]

10.1.3 Trois fonctions numériques classiques

Partie Entière Une fonction extrêmement répandue est celle qui permet de récupérer la partie entière d'un nombre :
Après : A=fix(3.228), A vaut 3. Cette fonction est notamment indispensable pour effectuer le célébrissime test de parité (voir exercice dans pas longtemps). On l'a déjà utilisée subrebtissement.

Modulo Cette fonction permet de récupérer le reste de la division d'un nombre par un deuxième nombre. Par exemple :

- A=mod(10,3), A vaut 1 car $10 = 3*3 + 1$
- B=mod(12,2), B vaut 0 car $12 = 6*2$
- C=mod(44,8), C vaut 4 car $44 = 5*8 + 4$

Cette fonction peut paraître un peu bizarre, et réservée aux seuls matheux. Mais vous aurez là aussi l'occasion de voir dans les exercices à venir que ce n'est pas le cas.

Génération de nombres aléatoires Une autre fonction classique , car très utile, est celle qui génère un nombre choisi au hasard. Tous les programmes de jeu, ou presque, ont besoin de ce type d'outils, qu'il s'agisse de simuler un lancer de dés ou le déplacement chaotique du vaisseau spatial de l'enfer de la mort piloté par l'infâme Zorglub, qui veut faire main basse sur l'Univers (heureusement vous êtes là pour l'en empêcher, ouf).

Mais il n'y a pas que les jeux qui ont besoin de générer des nombres aléatoires. La modélisation (physique, géographique, économique, etc.) a parfois recours à des modèles dits stochastiques (chouette, encore un nouveau mot savant !). Ce sont des modèles dans lesquels les variables se déduisent les unes des autres par des relations déterministes (autrement dit des calculs), mais où l'on simule la part d'incertitude par une « fourchette »de hasard.

Par exemple, un modèle démographique supposera qu'une femme a en moyenne x enfants au cours de sa vie, mettons 1.5. Mais il supposera aussi que sur une population donnée, ce chiffre peut fluctuer entre 1.35 et 1.65 (si on laisse une part d'incertitude de 10%). Chaque année, c'est-à-dire chaque série de calcul des valeurs du modèle, on aura ainsi besoin de faire choisir à la machine un nombre au hasard compris entre 1.35 et 1.65. Dans tous les langages, cette fonction existe et produit le résultat suivant :

```
Après :   Toto =rand(), On a : 0 =< Toto < 1
```

En fait, on se rend compte avec un tout petit peu de pratique que cette fonction rand peut nous servir pour générer n'importe quel nombre compris dans n'importe quelle fourchette. Je sais bien que mes lecteurs ne sont guère matheux, mais là, on reste franchement en deçà du niveau de feu le BEPC :

si rand génère un nombre compris entre 0 et 1, rand multiplié par Z produit un nombre entre 0 et Z. Donc, il faut estimer la « largeur »de la fourchette voulue et multiplier Alea par cette « largeur »désirée. ensuite, si la fourchette ne commence pas à zéro, il va suffire d'ajouter ou de retrancher quelque chose pour « caler »la fourchette au bon endroit. Par exemple, si je veux générer un nombre entre 1.35 et 1.65 ; la « fourchette »mesure 0,30 de large. Donc : $0 =<$ rand $*0{,}30 < 0{,}30$ Il suffit dès lors d'ajouter 1,35 pour obtenir la fourchette voulue. Si j'écris que :

```
Toto =rand()*0.3+1.35
```

Toto aura bien une valeur comprise entre 1,35 et 1,65. Et le tour est joué !

10.1.4 Les fonctions de conversion

Dernière grande catégorie de fonctions, là aussi disponibles dans tous les langages, car leur rôle est parfois incontournable, les fonctions dites de conversion.

On en a déjà vu une, num2str.

Rappelez-vous ce que nous avons vu dans les premières pages de ce cours : il existe différents types de variables, qui déterminent notamment le type de codage qui sera utilisé. Prenons le chiffre 3. Si je le stocke dans une variable de type alphanumérique, il sera codé en tant que caractère, sur un octet. Si en revanche je le stocke dans une variable de type entier, il sera codé sur deux octets. Et la configuration des bits sera complètement différente dans les deux cas. Une conclusion évidente, et sur laquelle on a déjà eu l'occasion d'insister, c'est qu'on ne peut pas faire n'importe quoi avec n'importe quoi, et qu'on ne peut pas par exemple multiplier '3' et '5', si 3 et 5 sont stockés dans des variables de type caractère. Jusque là, pas de scoop me direz-vous, à juste titre vous répondrai-je, mais attendez donc la suite.

Pourquoi ne pas en tirer les conséquences, et stocker convenablement les nombres dans des variables numériques, les caractères dans des variables alphanumériques, comme nous l'avons toujours fait ? Parce qu'il y a des situations où on n'a pas le choix ! Nous allons voir dès le chapitre suivant un mode de stockage (les fichiers textes) où toutes les informations, quelles qu'elles soient, sont obligatoirement stockées sous forme de caractères. Dès lors, si l'on veut pouvoir récupérer des nombres et faire des opérations dessus, il

va bien falloir être capable de convertir ces chaînes en numériques. Aussi, tous les langages proposent-ils une palette de fonctions destinées à opérer de telles conversions. La fonction réciproque de num2str est, forcément str2num.

10.2 Fonctions personnalisées

10.2.1 De quoi s'agit-il ?

Une application, surtout si elle est longue, a toutes les chances de devoir procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits de son déroulement. Par exemple, la saisie d'une réponse par oui ou par non (et le contrôle qu'elle implique), peuvent être répétés dix fois à des moments différents de la même application.

La manière la plus évidente, mais aussi la moins habile, de programmer ce genre de choses, c'est bien entendu de répéter le code correspondant autant de fois que nécessaire. Apparemment, on ne se casse pas la tête : quand il faut que la machine interroge l'utilisateur, on recopie presque mot à mot les lignes de codes voulues, et roule Raoul. Mais en procédant de cette manière, la pire qui soit, on se prépare des lendemains qui déchantent...

D'abord, parce que si la structure d'un programme écrit de cette manière peut paraître simple, elle est en réalité inutilement lourdingue. Elle contient des répétitions, et pour peu que le programme soit joufflu, il peut devenir parfaitement illisible. Or, c'est un critère essentiel pour un programme informatique, qu'il soit facilement modifiable donc lisible, y compris - et surtout - par ceux qui ne l'ont pas écrit ! Dès que l'on programme non pour soi-même, mais dans le cadre d'une organisation (entreprise ou autre), cette nécessité se fait sentir de manière aiguë.

En plus, à un autre niveau, une telle structure pose des problèmes considérables de maintenance : car en cas de modification du code, il va falloir traquer toutes les apparitions de ce code pour faire convenablement la modification ! Et si l'on en oublie une, patatras, on a laissé un bug.

Il faut donc opter pour une autre stratégie, qui consiste à séparer ce traitement du corps du programme et à appeler ces instructions (qui ne figurent donc plus qu'en un seul exemplaire) à chaque fois qu'on en a besoin. Ainsi, la lisibilité est assurée; le programme devient modulaire, et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité de l'application.

Le corps du programme s'appelle alors la procédure principale, et ces groupes d'instructions auxquels on a recours s'appellent des fonctions et des sous-procédures (nous verrons un peu plus loin la différence entre ces deux termes). Reprenons un exemple de question à laquelle l'utilisateur doit répondre par oui ou par non.

Mauvaise Structure :

```
on=input('Etes-vous marié ?')
while(on ~='Oui'&on~='Non')
  on=input('Tapez Oui ou Non')
end
on2=input('Avez-vous des enfants ?')
while(on2 ~='Oui'&on2~='Non')
  on=input('Tapez Oui ou Non')
end
```

On le voit bien, il y a là une répétition quasi identique du traitement à accomplir. A chaque fois, on demande une réponse par Oui ou Non, avec contrôle de saisie. La seule chose qui change, c'est le nom de la variable dans laquelle on range la réponse. Alors, il doit bien y avoir un truc.

La solution consiste à isoler les instructions demandant une réponse par Oui ou Non, et à appeler ces instructions à chaque fois que nécessaire. Ainsi, on évite les répétitions inutiles, et on a découpé notre problème en petits morceaux autonomes. Nous allons donc créer une fonction dont le rôle sera de renvoyer la réponse (oui ou non) de l'utilisateur. Ce mot de "fonction", en l'occurrence, ne doit pas nous surprendre : nous avons étudié précédemment des fonctions fournies avec le langage, et nous avons vu

que le but d'une fonction était de renvoyer une valeur. Eh bien, c'est exactement la même chose ici, sauf que c'est nous qui allons créer notre propre fonction, que nous appellerons repouinon :

```
function on=repouinon()
% variable on,
on=input('')
while(on ~='Oui'&on~='Non')
  on=input('Tapez Oui ou Non')
end
% Fin
```

Certain langage (pas MATLAB) utilise un mot clé additionnel : Renvoyer, qui indique quelle valeur doit prendre la fonction lorsqu'elle est utilisée par le programme. Cette valeur renvoyée par la fonction (ici, la valeur de la variable on) est en quelque sorte contenue dans le nom de la fonction lui-même, exactement comme c'était le cas dans les fonctions prédéfinies.

Une fonction s'écrit toujours en-dehors de la procédure principale. Selon les langages, cela peut prendre différentes formes. Mais ce qu'il faut comprendre, c'est que ces quelques lignes de codes sont en quelque sorte des satellites, qui existent en dehors du traitement lui-même. Simplement, elles sont à sa disposition, et il pourra y faire appel chaque fois que nécessaire. Si l'on reprend notre exemple, une fois notre fonction RepOuiNon écrite, le programme principal comprendra les lignes :

```
display('Etes-vous marié ?')
on=repouinon()
display('Avez-vous des enfants ?')
on2=repouinon()
```

Et le tour est joué ! On a ainsi évité les répétitions inutiles, et si d'aventure, il y avait un bug dans notre contrôle de saisie, il suffirait de faire une seule correction dans la fonction repouinon pour que ce bug soit éliminé de toute l'application. C'est-y pas beau, la vie ?

Toutefois, les plus sagaces d'entre vous auront remarqué, tant dans le titre de la fonction , que dans chacun des appels, la présence de parenthèses. Celles-ci, dès qu'on crée on qu'on appelle une fonction, sont obligatoires. Et si vous avez bien compris tout ce qui précède, vous devez avoir une petite idée de ce qu'on va pouvoir mettre dedans...

10.2.2 Passage d'arguments

Reprenons l'exemple qui précède et analysons-le. Nous écrivons un message à l'écran, puis appelons la fonction repouinon pour poser une question ; puis, un peu plus loin, on écrit un autre message à l'écran, et on appelle de nouveau la fonction pour poser la même question, etc. C'est une démarche acceptable, mais qui peut encore être améliorée : puisque avant chaque question, on doit écrire un message, autant que cette écriture du message figure directement dans la fonction appelée. Cela implique deux choses : lorsqu'on appelle la fonction, on doit lui préciser quel message elle doit afficher avant de lire la réponse la fonction doit être « prévenue »qu'elle recevra un message, et être capable de le récupérer pour l'afficher. En langage algorithmique, on dira que le message devient un argument de la fonction. Cela n'est certes pas une découverte pour vous : nous avons longuement utilisé les arguments à propos des fonctions prédéfinies. Eh bien, quitte à construire nos propres fonctions, nous pouvons donc construire nos propres arguments. Voilà comment l'affaire se présente... La fonction sera dorénavant déclarée comme suit :

```
function on=repouinon(question)
% variable on, question en caractère
on=input(question)
while(on ~='Oui'&on~='Non')
  on=input('Tapez Oui ou Non')
end
% Fin
```

Et puis///

```
on=repouinon('Etes-vous marié ?')
on2=repouinon('Avez-vous des enfants ?')
```

Et voilà le travail.

Une remarque importante : là, on n'a passé qu'un seul argument en entrée. Mais bien entendu, on peut en passer autant qu'on veut, et créer des fonctions avec deux, trois, quatre, etc. arguments ; Simplement, il faut éviter d'être gourmands, et il suffit de passer ce dont on en a besoin, ni plus, ni moins ! Dans le cas que l'on vient de voir, le passage d'un argument à la fonction était élégant, mais pas indispensable. La preuve, cela marchait déjà très bien lors de la première version. Nous allons voir maintenant une situation où il faut absolument passer deux arguments à une fonction si l'on veut qu'elle puisse remplir sa tâche. Imaginons qu'à plusieurs reprises au cours du programme, on ait à calculer la moyenne des éléments de différents tableaux. Plutôt que répéter le code à chaque fois, on va donc créer une fonction Moy, chargée spécialement de calculer cette moyenne. Voyons voir un peu de quoi cette fonction a besoin : du tableau, bien sûr. Comment calculer la moyenne de ses éléments sans cela ? mais il faut également le nombre d'éléments du tableau, ou, au choix, l'indice maximal du tableau. Enfin, quelque chose qui permette à la fonction de savoir combien de tours de boucle elle devra faire. Voilà donc une situation où la fonction a absolument besoin de deux arguments. Écrivons-la, juste histoire de vérifier qu'on est bien d'accord :

```
function y=moy(tab, n)
% variable y, n en numérique
% tableau tab en numérique
%
Som=0;
for i=1:n
  Som=Som + T(i);
end
y=Som / n
% Fin Fonction
```

Quant aux différents appels dans la procédure principale, si j'ai un tableau Riri de 43 éléments, un tableau Fifi de 5 éléments et un tableau Loulou de k éléments, et que je range respectivement les moyennes dans les variables M1, M2 et M3, cela donnera :

```
M1=moy(Riri,43)
M2=moy(Fifi,5)
M3=moy(Fifi,k)
```

En fait, tout cela, c'est simple comme bonjour... Le plus important, c'est d'acquérir le réflexe de constituer systématiquement les fonctions adéquates quand on doit traiter un problème donné.

Cette partie de la réflexion s'appelle d'ailleurs l'analyse fonctionnelle d'un problème, et c'est toujours par là qu'il faut commencer : en gros, dans un premier temps, on découpe le traitement en modules, et dans un deuxième temps, on écrit chaque module. Cependant, avant d'en venir là, il nous faut découvrir un dernier outil, qui prend le relais là où les fonctions deviennent incapables de nous aider.

10.2.3 Variables publiques et privées

L'existence de fonctions pose le problème de la « durée de vie »des variables, ce qu'on appelle leur portée. Pour adopter une classification simple (mais qui devient parfois plus complexe dans certains langages), une variable peut être déclarée :

- Comme privée, ou locale (c'est neuf fois sur dix l'option par défaut). Cela Cela signifie que la variable disparaît (et sa valeur avec) dès que prend fin la procédure ou elle a été créée.
- Comme publique, ou globale. Ce qui signifie qu'une telle variable est conservée intacte pour toute l'application, au-delà des ouvertures et fermetures de procédures. La variable conserve sa valeur et peut être traitée par différentes procédures du programme.

La manière dont ces déclarations doivent être faites est évidemment fonction de chaque langage de programmation. En matlab, les variables sont privées pour les fonctions, ce qui veut dire que les fonctions ne savent rien de ce qui est fait dans le programme principal et réciproquement. Comment choisir de déclarer une variable en Public ou en Privé ? C'est très simple : les variables globales consomment énormément de ressources en mémoire. En conséquence, le principe qui doit présider au choix entre variables publiques et privées doit être celui de l'économie de moyens : on ne déclare comme publiques que les variables qui doivent absolument l'être. Et chaque fois que possible, lorsqu'on crée une sous-procédure, on utilise le passage de paramètres plutôt que des variables publiques. Donc, pour ce qui suit, on supposera que toutes les variables sont privées, na !

10.2.4 Algorithmes fonctionnels

Pour clore ce chapitre, quelques mots à propos de la structure générale d'une application. Celle-ci va couramment être formée d'une procédure principale, et de fonctions et de sous-procédures (qui vont au besoin elles-mêmes en appeler d'autres, etc.). L'exemple typique est celui d'un menu, ou d'un sommaire, qui « branche »sur différents traitements, donc différentes sous-procédures. L'algorithme fonctionnel de l'application est le découpage et/ou la représentation graphique de cette structure générale, ayant comme objectif de faire comprendre d'un seul coup d'IJil quelle procédure fait quoi, et quelle procédure appelle quelle autre. L'algorithme fonctionnel est donc en quelque sorte la construction du squelette de l'application. Il se situe à un niveau plus général, plus abstrait, que l'algorithme normal, qui lui, détaille pas à pas les traitements effectués au sein de chaque procédure. Dans la construction - et la compréhension - d'une application, les deux documents sont indispensables, et constituent deux étapes successives de l'élaboration d'un projet. La troisième - et dernière - étape, consiste à écrire, pour chaque procédure et fonction, l'algorithme détaillé.

Exemple de réalisation d'un algorithme fonctionnel : Le Jeu du Pendu Vous connaissez tous ce jeu : l'utilisateur doit deviner un mot choisi au hasard par l'ordinateur, en un minimum d'essais. Pour cela, il propose des lettres de l'alphabet. Si la lettre figure dans le mot à trouver, elle s'affiche. Si elle n'y figure pas, le nombre des mauvaises réponses augmente de 1. Au bout de dix mauvaises réponses, la partie est perdue.

Ce petit jeu va nous permettre de mettre en relief les trois étapes de la réalisation d'un algorithme un peu complexe ; bien entendu, on pourrait toujours ignorer ces trois étapes, et se lancer comme un dératé directement dans la gueule du loup, à savoir l'écriture de l'algorithme définitif. Mais, sauf à être particulièrement doué, mieux vaut respecter le canevas qui suit, car les difficultés se résolvent mieux quand on les saucissonne...

Etape 1 : le dictionnaire des données : Le but de cette étape est d'identifier les informations qui seront nécessaires au traitement du problème, et de choisir le type de codage qui sera le plus satisfaisant pour traiter ces informations. C'est un moment essentiel de la réflexion, qu'il ne faut surtout pas prendre à la légère... Or, neuf programmeurs débutants sur dix bâclent cette réflexion, quand ils ne la zappent pas purement et simplement. La punition ne se fait généralement pas attendre longtemps ; l'algorithme étant bâti sur de mauvaises fondations, le programmeur se rend compte tout en l'écrivant que le choix de codage des informations, par exemple, mène à des impasses. La précipitation est donc punie par le fait qu'on est obligé de tout reprendre depuis le début, et qu'on a au total perdu bien davantage de temps qu'on en a cru en gagner...

Donc, avant même d'écrire quoi que ce soit, les questions qu'il faut se poser sont les suivantes : de quelles informations le programme va-t-il avoir besoin pour venir à bout de sa tâche ? pour chacune de ces informations, quel est le meilleur codage ? Autrement dit, celui qui sans gaspiller de la place mémoire, permettra d'écrire l'algorithme le plus simple ? Encore une fois, il ne faut pas hésiter à passer du temps sur ces questions, car certaines erreurs, ou certains oublis, se payent cher par la suite. Et inversement, le temps investi à ce niveau est largement rattrapé au moment du développement proprement dit.

Pour le jeu du pendu, voici la liste des informations dont on va avoir besoin :

- une liste de mots (si l'on veut éviter que le programme ne propose toujours le même mot à trouver, ce qui risquerait de devenir assez rapidement lassant...)

- le mot à deviner
- la lettre proposée par le joueur à chaque tour
- le nombre actuel de mauvaises réponses
- et enfin, last but not least, l'ensemble des lettres déjà trouvées par le joueur. Cette information est capitale; le programme en aura besoin au moins pour deux choses : d'une part, pour savoir si le mot entier a été trouvé. D'autre part, pour afficher à chaque tour l'état actuel du mot (je rappelle qu'à chaque tour, les lettres trouvées sont affichées en clair par la machine, les lettres restant à deviner étant remplacées par des tirets).
- à cela, on pourrait ajouter une liste comprenant l'ensemble des lettres déjà proposées par le joueur, qu'elles soient correctes ou non; ceci permettra d'interdire au joueur de proposer à nouveau une lettre précédemment jouée.

Cette liste d'informations n'est peut-être pas exhaustive; nous aurons vraisemblablement besoin au cours de l'algorithme de quelques variables supplémentaires (des compteurs de boucles, des variables temporaires, etc.). Mais les informations essentielles sont bel et bien là. Se pose maintenant le problème de choisir le mode de codage le plus futé. Si, pour certaines informations, la question va être vite réglée, pour d'autres, il va falloir faire des choix (et si possible, des choix intelligents!).
C'est parti, mon kiki :
Pour la liste des mots à trouver, il s'agit d'un ensemble d'informations de type alphanumérique. Ces informations pourraient faire partie du corps de la procédure principale, et être ainsi stockées en mémoire vive, sous la forme d'un tableau de chaînes. Mais ce n'est certainement pas le plus judicieux. Toute cette place occupée risque de peser lourd inutilement, car il n'y a aucun intérêt à stocker l'ensemble des mots en mémoire vive. Et si l'on souhaite enrichir la liste des mots à trouver, on sera obligé de réécrire des lignes de programme... Conclusion, la liste des mots sera bien plus à sa place dans un fichier texte, dans lequel le programme ira piocher un seul mot, celui qu'il faudra trouver. Nous constituerons donc un fichier texte, appelé dico.txt, dans lequel figurera un mot par ligne (par enregistrement).
Le mot à trouver, lui, ne pose aucun problème : il s'agit d'une information simple de type chaîne, qui pourra être stocké dans une variable appelée mot, de type caractère.
De même, la lettre proposée par le joueur est une information simple de type chaîne, qui sera stockée dans une variable appelée lettre, de type caractère.
Le nombre actuel de mauvaises réponses est une information qui pourra être stockée dans une variable numérique de type entier simple appelée MovRep.
L'ensemble des lettres trouvées par le joueur est typiquement une information qui peut faire l'objet de plusieurs choix de codage; rappelons qu'au moment de l'affichage, nous aurons besoin de savoir pour chaque lettre du mot à deviner si elle a été trouvée ou non. Une première possibilité, immédiate, serait de disposer d'une chaîne de caractères comprenant l'ensemble des lettres précédemment trouvées. Cette solution est loin d'être mauvaise, et on pourrait tout à fait l'adopter. Mais ici, on fera une autre choix, ne serait-ce que pour varier les plaisirs : on va se doter d'un tableau de booléens, comptant autant d'emplacements qu'il y a de lettres dans le mot à deviner. Chaque emplacement du tableau correspondra à une lettre du mot à trouver, et indiquera par sa valeur si la lettre a été découverte ou non (faux, la lettre n'a pas été devinée, vrai, elle l'a été). La correspondance entre les éléments du tableau et le mot à deviner étant immédiate, la programmation de nos boucles en sera facilitée. Nous baptiserons notre tableau de booléens du joli nom de « verif ».
Enfin, l'ensemble des lettres proposées sera stockée sans soucis dans une chaîne de caractères nommée Propos. Nous avons maintenant suffisamment gambergé pour dresser le tableau final de cette étape, à savoir le dictionnaire des données proprement dit :

```
Nom Type Description
Dico.txt Fichier texte Liste des mots à deviner
Mot Caractère Mot à deviner
Lettre Caractère Lettre proposée
MovRep Entier Nombre de mauvaises réponses
Verif() Tableau de Booléens Lettres précédemment devinées, en correspondance avec Mot
Propos Caractère Liste des lettres proposées
```

Etape 2 : l'algorithme fonctionnel On peut à présent passer à la réalisation de l'algorithme fonctionnel, c'est-à-dire au découpage de notre problème en blocs logiques. Le but de la manIJuvre est multiple :

- faciliter la réalisation de l'algorithme définitif en le tronçonnant en plus petits morceaux.
- gagner du temps et de la légèreté en isolant au mieux les sous-procédures et fonctions qui méritent de l'être. Eviter ainsi éventuellement des répétitions multiples de code au cours du programme, répétitions qui ne diffèrent les unes des autres qu'à quelques variantes près.
- permettre une division du travail entre programmeurs, chacun se voyant assigner la programmation de sous-procédures ou de fonctions spécifiques (cet aspect est essentiel dès qu'on quitte le bricolage personnel pour entrer dans le monde de la programmation professionnelle, donc collective).

Dans notre cas précis, un premier bloc se détache : il s'agit de ce qu'on pourrait appeler les préparatifs du jeu (choix du mot à deviner). Puisque le but est de renvoyer une valeur et une seule (le mot choisi par la machine), nous pouvons confier cette tâche à une fonction spécialisée ChoixDuMot (à noter que ce découpage est un choix de lisibilité, et pas une nécessité absolue ; on pourrait tout aussi bien faire cela dans la procédure principale).

Cette procédure principale, justement, va ensuite avoir nécessairement la forme d'une boucle Tantque : en effet , tant que la partie n'est pas finie, on recommence la série des traitements qui représentent un tour de jeu. Mais comment, justement, savoir si la partie est finie ? Elle peut se terminer soit parce que le nombre de mauvaises réponses a atteint 10, soit parce que toutes les lettres du mot ont été trouvées. Le mieux sera donc de confier l'examen de tout cela à une fonction spécialisée, PartieFinie, qui renverra une valeur numérique (0 pour signifier que la partie est en cours, 1 en cas de victoire, 2 en cas de défaite).

Passons maintenant au tour de jeu. La première chose à faire, c'est d'afficher à l'écran l'état actuel du mot à deviner : un mélange de lettres en clair (celles qui ont été trouvées) et de tirets (correspondant aux lettres non encore trouvées). Tout ceci pourra être pris en charge par une sous-procédure spécialisée, appelée AffichageMot. Quant à l'initialisation des différentes variables, elle pourra être placée, de manière classique, dans la procédure principale elle-même.

Ensuite, on doit procéder à la saisie de la lettre proposée, en veillant à effectuer les contrôles de saisie adéquats. Là encore, une fonction spécialisée, SaisieLettre, sera toute indiquée.

Une fois la proposition faite, il convient de vérifier si elle correspond ou non à une lettre à deviner, et à en tirer les conséquences. Ceci sera fait par une sous-procédure appelée VérifLettre.

Enfin, une fois la partie terminée, on doit afficher les conclusions à l'écran ; on déclare à cet effet une dernière procédure, FinDePartie.

Nous pouvons, dans un algorithme fonctionnel complet, dresser un tableau des différentes procédures et fonctions, exactement comme nous l'avons fait juste avant pour les données (on s'épargnera cette peine dans le cas présent, ce que nous avons écrit ci-dessus suffisant amplement. Mais dans le cas d'une grosse application, un tel travail serait nécessaire et nous épargnerait bien des soucis).

On peut aussi schématiser le fonctionnement de notre application sous forme de blocs, chacun des blocs représentant une fonction ou une sous-procédure : A ce stade, l'analyse dite fonctionnelle est terminée. Les fondations (solides, espérons-le) sont posées pour finaliser l'application.

Etape 3 : Algorithmes détaillés Normalement, il ne nous reste plus qu'à traiter chaque procédure isolément. On commencera par les sous-procédures et fonctions, pour terminer par la rédaction de la procédure principale.

Toutes ces joyeuses choses seront l'objet d'un chouette exercice sur l'ordinateur... Parce que, bon...

10.3 Exercices corrigés

1. Écrivez une fonction qui renvoie la somme de cinq nombres fournis en argument. Proposez un exemple d'application.

```
function s=somme(x1,x2,x3,x4,x5)
% variables s,x1,x2,x3,x4,x5 en numérique
```

```
% Début
s=x1+x2+x3+x4+x5
% fin
```

Par exemple on aura

```
so=somme(1,2,3,4,5);
```

2. Ecrire une fonction qui mélange les lettres d'un mot.

```
function m=anagram(mot)
% tableau m,mot en caractere
% tableau position en numérique
% variable r,L en numérique
% variable flag en boléen
% tableau f en boléen
L=length(mot)
f=true(1,L)
for i=1:L
   flag=false
   while(flag==false)
     r=fix(rand()*L)+1
     if (f(r))
       m(r)=mot(i)
       f(r)=false
       flag=true
     end
   end
end
```

10.4 Exercices non corrigés

1. Ecrire une fonction qui trie les lettres d'un mot par ordre alphabétique.
2. Ecrire un programme qui saisit des données, les trie dans un tableau et regarde si les nombres sont consécutifs ou pas, en utilisant une fonction de saisie des données et une fonction de tri.
3. Écrivez une fonction qui renvoie le nombre de voyelles contenues dans une chaîne de caractères passée en argument. Au passage, notez qu'une fonction a tout à fait le droit d'appeler une autre fonction.

Chapitre 11

La programmation vectorielle

La révolution informatique fait gagner un temps fou aux hommes, mais ils le passent avec leur ordinateur !

Khalil Assala

Les hommes viennent de Mars, les femmes de Vénus. Les ordinateurs viennent de l'enfer.

Anonyme

Les ordinateurs sont inutiles. Ils ne savent que donner des réponses.

Pablo Picasso

11.1 Les manipulations élémentaires

De nos jours, la mémoire n'est, le plus souvent, plus un obstacle en programmation. Par conséquent, on peut manipuler plusieurs choses en même temps. Cela a permis de mettre en oeuvre une méthode sympathique en diable que, faute de mieux, je nomme ici programmation vectorielle.

Supposons que l'on ait un tableau Trucmuch qui contient, pour une raison qui lui est propre, les nombres entre 1 et 5. Et on veut le multiplier par 2. Avant, quand c'était pas bien, on faisait ça...

```
%
% tableau Trucmuch en numérique
% variable i en numérique^
for i=1:5
   Trucmuch(i)=i
end
for i=1:5
  Trucmuch(i)=Trucmuch(i)*2
end
```

bèèèè, c'est pas beau, c'est lent (Ouuuuuuhhhhhh du public)

Heureusement est arrivé :

```
%
% tableau Trucmuch en numérique
% variable i en numérique
Trucmuch=1:5
Trucmuch=Trucmuch*2
```

C'est joli (Ooooooohhhhh admiratifs du public), c'est facile et surtout très lisible. En plus, ce qui ne gâte rien, on voulait multiplier le tableau par deux, et bien c'est exactement ce qui est écrit.

Supposons que l'on veuille copier ce tableau dans un autre... Avant, on faisait ça :

```
%
% tableau Trucmuch, retrucmuch en numérique
% variable i en numérique
for i=1:5
   Trucmuch(i)=i
end
for i=1:5
  refoitrucmuch(i)=Trucmuch(i)
end
```

Alors que maintenant,

```
%
% tableau Trucmuch, retrucmuch en numérique
% variable i en numérique
Trucmuch=1:5
retrucmuch=Trucmuch
```

C'est-à-dire que l'on applique l'opérateur à un vecteur, et on ne s'inquiète de rien, l'ordinateur gère.

En réalité, on a déjà appliqué cela une fois, sans le dire (encore Monsieur Jourdin).

En effet, l'instruction for de MATLAB permet de prendre un tableau de valeurs pour i. Or, lorsque j'écrivais :

```
for i=1:15
   ...
end
```

L'ordinateur comprenait ceci :

1. Créer un tableau provisoire qui contient les entiers entre 1 et 15.
2. Faire prendre à i successivement les différentes valeurs dans le tableau.
3. Pour chaque valeur de i, exécuter les instructions de la boucle.

11.2 Les tests sur les tableaux

Même pas peur, si MATLAB est malin, on peut faire des tests sur les tableaux...

Essayons donc ceci :

```
% tableau x en numérique
% tableau b en boléen
x=1:100
b=(x>50)
%
```

Qu'est donc b ? Un tableau boléen de la même longueur que x qui contient, pour les indices tels que x<=50 la valeur false et pour les autres la valeur true.

C'est top-chouette, on continue :

```
% tableau x,i en numérique
x=1:100
i=find(x>50)
```

Le tableau i contient la liste des indices pour lesquels c'est vrai... Donc, si on combine tout cela, que nous donne :

```
% tableau x,i en numérique
% variable j en numérique
x=1:100
i=find(x/2==fix(x/2))
x(i)=x(i)+1
```

La première ligne crée un vecteur avec les nombres de 1 à 100. La deuxième crée un vecteur des indices qui des x pairs. A ceux-là, la troisième ligne ajoute 1.

Le résultat est donc une série de tous les nombres impairs entre 1 et 101, répété deux fois, sauf 1 et 101 qui ne sont pas répétés. C'est magique, quand même.

Revoyons, par exemple, le tri à bulle en version vectorisée :

```
% On crée un vecteur d'une permutation aléatoire des nombres entre 1 et 101, ce que l'on va trier.
rx=randperm(101);
% On a besoin de 2 flag...
flag=1;
flag2=1;
% Il sera utile de savoir si j'ai un nombre pair de données, ou un nombre impair.
imp=length(x)/2~=fix(length(x)/2);
% imp vaut 1 si le nombre de données est impair.
%
% Tant que les flags ne sont pas nuls tous les 2...
while (flag+flag2>0)
%
% Je dois tester si les nombres qui se suivent sont bien ordonnés. En version vecteur, je le fais en
% les 2 côtés: 1>2, 3>4, ..., 99>100, et 2>3,4>5,...
%
% On commence par 1>2...
    impair=rx(1:2:end-imp);
    pair=rx(2:2:end);
% Ceux qui sont mals rangés, dans ce groupe, on met leurs indices dans i
    ii=find(impair>pair);
% Et on les inverse avec ceux qui suivent, vectoriellement...
    prov=rx(2*ii);
    rx(2*ii)=rx(2*ii-1);
    rx(2*ii-1)=prov;
% On doit continuer tant qu'il y a des inversions. Ce qui a lieu tant que ii n'est pas de longueur n
    flag=length(ii);
% Maintenant, on s'occupe des autres relations d'ordre: 2>3 etc
    impair=rx(3:2:end);
    pair=rx(2:2:end-(1-imp));
    ii=find(pair>impair);
    prov=rx(2*ii+1);
    rx(2*ii+1)=rx(2*ii);
    rx(2*ii)=prov;
```

```
% Et on met le deuxième flag...
    flag2=length(ii);
end
```

Joli, non ? En plus, l'algorithme va entre 3 et 5 fois plus vite, une fois vectorisé.rx=randperm(101) ;

Chapitre 12

Exercice sur ordinateur

L'ordinateur obéit à vos ordres,
pas à vos intentions.

Anonyme

L'homme est sujet à l'erreur.
Mais s'il veut vraiment
commettre la gaffe absolue, alors
là, il lui faut un ordinateur.

Dan Rather

12.1 Additionner des fractions

Le problème semble bête comme chou, mais en pratique, ce qu'on s'amuse!!!

Bon, je résume. On suppose deux fractions, données en termes de deux numérateurs et deux dénominateurs. On veut calculer leur somme et la simplifier.

12.1.1 L'initialisation

On va créer une fonction. Elle aura quatre paramètres d'entrées (num1, num2, den1, den2). Elle sort deux nombres, num et den.

12.1.2 Le calcul simple

On sait que

$$\begin{aligned} \frac{\text{num}}{\text{den}} &= \frac{\text{num}_1\text{den}_2 + \text{num}_2\text{den}_1}{\text{den}_1\text{den}_2} \\ \text{num} &= \text{num}_1\text{den}_2 + \text{num}_2\text{den}_1 \\ \text{den} &= \text{den}_1\text{den}_2 \end{aligned}$$

Calculons déjà cela. Maintenant, il y a lieu de simplifier, et c'est là où l'on rit.

12.1.3 La simplification

Pour ce faire, il faut calculer le PGCD (plus grand commun diviseur). Commençons par nous doter d'une fonction qui décompose le nombre en facteurs premiers (voir fin du chapitre 6, on ne va pas

réinventer la roue à chaque fois...). Par cette fonction, on crée les tableaux NUM et DEN des facteurs premiers décomposant les numérateurs et dénominateurs.

Cela fait, il faut simplifier. Là, je vous laisse réfléchir à la méthode. Si vous le faites pédestrement, vous n'êtes pas arrivés. En revanche, il existe une solution simple et assez élégante.

12.1.4 Les cas particuliers

On est presqu'au bout. Il faut encore traiter les cas particuliers :

1. si le numérateur vaut zéro,
2. si le dénominateur vaut zéro,
3. si les deux sont nuls,
4. si l'un ou les deux sont négatifs.

12.2 Le jeu de la vie

Cet algoritme simule l'évolution d'une colonie bactérienne. Le problème est simple. On prend une grille, genre 100 sur 100. On l'initialise à zéro : toutes les bactéries sont mortes. On tire au sort un certain nombre de points de la grille, et on les mets à 1. Après celà, on observe l'évolution avec les lois suivantes : si le nombre de bactéries vivantes voisines d'une bactérie est compris entre K1 et K2, la bactérie sera vivante au tour suivant. Sinon, elle sera morte.

12.2.1 L'initialisation

On définit un tableau de 100 sur 100, le nombre N de points que l'on va "allumer", le nombre nt de pas de temps, et les constantes K1 et K2 (ça marche bien avec K2=2). On tire au sort (fonction rand) les N points à allumer, et on change leur valeur pour 1.

12.2.2 L'évolution

On doit faire une boucle pour calculer un pas de temps après l'autre. Comme il y a des instructions graphiques, celle-là, je vous la donne :

```
% ouvre un fichier game_of_live.avi, qui contiendra le film.
aviobj=avifile('game_of_live.avi','fps',25,'Compression','Indeo5');
% Ici, c'est un peu tordu, je veux que les points de valeurs 1 soient noirs et les points de valeurs
colormap gray
map=colormap;
mm=flipud(map);
colormap(mm)
% ouf, c'est fini
for i=1:nt
% La magie est dans la fonction 'tour' qui calcule la valeur de la grille au pas suivant.
    grid=tour(grid,K1,K2);
% ici, on dessine le résultat
    pcolor(grid); shading flat
% et ici, on le met dans le film
    frame=getframe(gcf);
    aviobj=addframe(aviobj,frame);
    clf
% voila, la boucle est bouclée
end
% Et on ferme le fichier du film
aviobj=close(aviobj)
```

Bon, là, on a déjà bien bossé... mais le plus dur reste à faire...

12.2.3 La fonction tour

Que doit faire cette fonction ?
(1) Calculer le nombre de voisins,
(2) En fonction de ce nombre, calculer pour chaque case de la grille ce qu'il en advient au tour suivant.

Pour calculer le nombre de voisin, il y a un problème au bord. Pour ne pas m'embéter, j'ai pris une géométrie torique, c'est-à-dire que la ligne au-dessus de la première ligne est la dernière ligne, que la ligne après la dernière ligne est la première ligne (ce qui nous fait un cylindre), et que la colonne qui suit la dernière colonne est la première et inversément (ce qui achève le tore).

Le choix le plus judicieux, ici, est de créer un halo, c'est-à-dire de rajouter une colonne devant et une colonne derrière, une ligne au-dessus et une ligne en dessous, où l'on recopie la ligne/colonne adéquate. Comme cela, tant que l'on ne traite que les case de la grille, elles ont toutes tous leurs voisins, donc pas besoin de faire de teste.

Bon, il ne reste plus qu'à faire la somme de la valeur des cases voisines, et de voir si cette somme est entre K1 et K2. Pour ces points-là, on met un, pour les autres, on met zéro. Et c'est fini. Ici, la programmation vectorielle est infinement plus rapide et élégante que la programmation par boucle. Donc...

12.3 Compteur des points au tennis

Bon... En voila un exercie bête et méchant... Le problème est simple, on rentrera en temps réel qui a gagné chaque échange, et l'ordinateur nous donne les points (y compris les jeux et les sets), nous dit s'il y a tie-break et tout. L'unité élémentaire, c'est le jeu, nous commencerons là.

12.3.1 Les règles

La plupart du temps, il est nécessaire de remporter deux sets afin de gagner la partie. La seule exception est celle des matches du tableau masculin lors de rencontres dans des tournois du Grand Chelem, ou en Coupe Davis. Pour gagner une manche, il faut être le premier à marquer six jeux avec au moins deux jeux d'écart, dans le cas contraire la manche se poursuit. Les scores possibles pour remporter un set sont ainsi : 6/0, 6/1, 6/2, 6/3, 6/4 et 7/5 (si les deux joueurs n'ont pu se départager au bout de dix jeux). Si les deux joueurs n'ont pas été en mesure de se départager au cours des douze premiers jeux (donc à égalité à 6/6), ils disputent un jeu décisif, qui vaut un jeu, et permet donc de remporter la manche 7/6. En revanche, dans les tournois du grand chelem, exception faite de l'US Open, chez les hommes comme chez les femmes, il n'y a pas de tie-break dans la manche décisive (la cinquième chez les hommes, la troisième chez les femmes), et le match n'est remporté que lorsque l'on parvient à avoir deux jeux d'avance sur l'adversaire ; par exemple 8/6, 9/7, 10/8, etc.

L'invention du jeu décisif date de 1970, soit deux ans après le début de l'ère open. La finalité de ce jeu était d'empêcher des matches interminables, car il arrivait à l'époque que des sets soient gagnés sur le score de 29/27 par exemple. Le principe du jeu décisif est assez simple. Les joueurs servent à tour de rôle. Celui qui débute ne sert qu'une fois de droite à gauche, puis son adversaire sert deux fois de suite, de gauche à droite, puis de droite à gauche, et ainsi de suite. Le gagnant de la manche est le premier joueur à atteindre sept points avec au moins deux points d'écart (Ex : 7/2, 7/5, 9/7...) La manche est alors gagnée sur le score de 7-6.

Une manche se remporte donc en marquant un certain nombre de jeux. Afin de remporter un jeu, il est nécessaire de marquer au moins quatre points, soit sur son service lorsque l'on sert, soit sur le service adverse lorsque l'on reçoit. Il est donc possible, soit pour le serveur, soit pour le receveur de remporter un jeu, même si théoriquement, le serveur est avantagé par rapport au receveur. Si les deux adversaires marquent trois points, on a une situation d'égalité, expliquée ci-après. Lors d'un jeu, voici la manière dont les points sont décomptés : zéro, quinze pour un point marqué, trente pour deux points marqués, quarante pour trois points marqués. Lorsque les deux joueurs ont marqué trois points, (donc à 40/40), il y a égalité. Celui qui marque le point suivant obtient un avantage. Pour marquer le jeu, un joueur qui

a l'avantage doit marquer un autre point. Si c'est le joueur qui n'a pas l'avantage qui marque le point suivant, on revient à égalité, et ainsi de suite jusqu'à ce que l'un des deux joueurs remporte le jeu.

Au niveau de l'arbitrage, on donne toujours le score du serveur en premier. Par exemple, si le serveur marque trois points contre deux à son adversaire, le score est 40/30. Dans le cas contraire, le score est 30/40. Il en est de même au niveau des avantages, lorsqu'il y a égalité dans un jeu. Lorsque c'est le serveur qui a l'avantage, l'arbitre annoncera avantage service, et si c'est le receveur qui a l'avantage, on aura avantage dehors.

Je propose de faire cet algorithme par boucles while imbriquées, l'une pour le gain de la manche, l'autre pour le gain du match.

12.3.2 L'initialisation

Il faut demander à l'utilisateur en combien de sets gagnant est la partie, et si on applique la règle des tie-breaks tout le temps, jamais ou tout le temps sauf en cas de manche décisive.

12.3.3 Le jeu

D'abbord, il faut compter avec la façon bizarre de compter les points (15-30-40) et puis avec les avantages et égalité. On ne s'occupera pas du service, et on aura un joueur 1 et un joueur 2, en donnant toujours les points du joueur 1 d'abord.

12.3.4 La manche

Il faut compter les jeux, et voir s'il y a lieu de faire une manche décisive. Dans ce cas, la gérer.

12.3.5 Le match

Bon, ben là, il suffit de voir si l'une des deux parties à remporter le bon nombre de manches.