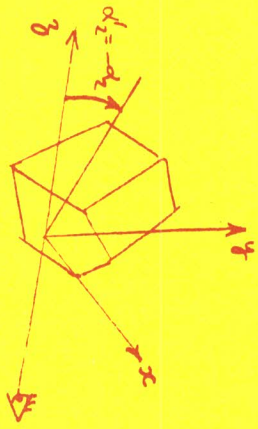
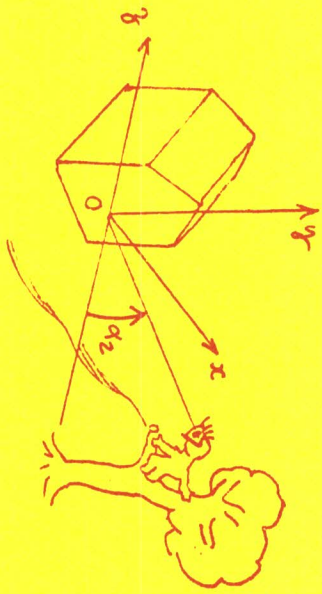
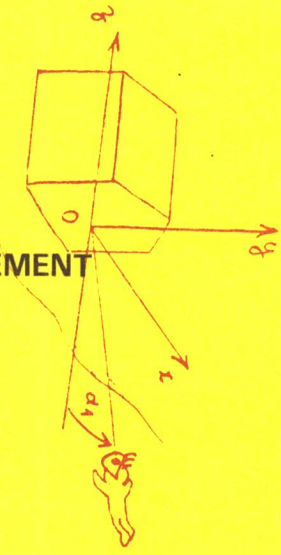






UNIVERSITÉ
DE NANCY I

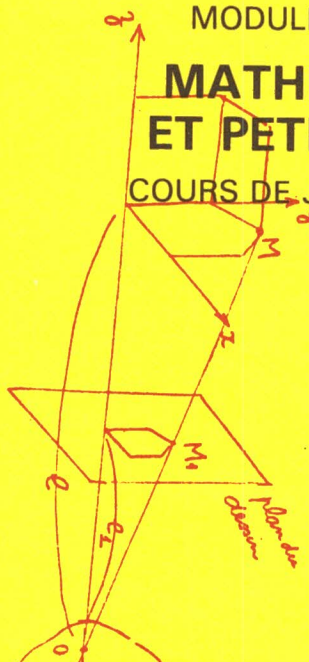
CENTRE
DE TÉLÉ-ENSEIGNEMENT
UNIVERSITAIRE
DE NANCY II

mathématiques et programmation

MODULE **MP 2**

MATHÉMATIQUES ET PETITS LOGICIELS (II)

COURS DE J.-P. FERRIER



DIPLÔME D'ÉTUDES UNIVERSITAIRES GÉNÉRALES

SCIENCES DES STRUCTURES ET DE LA MATIÈRE
MATHÉMATIQUES PHYSIQUE INFORMATIQUE

SCIENCES DE L'ÉDUCATION

la maquette de la couverture a été réalisée par le L.E.P. Cyfflé - NANCY

B.P. 239 - 54506 VANDOEUVRE-les-NANCY CEDEX
Dépôt légal : 1er trimestre 1988
n° de la publication : 2-85406-106-3
Le Responsable de la collection : Philippe LOMBARD

Réf. N 511

Introduction

Les ambitions du module MP1 en matière d'informatique sont très limitées; on n'a pas recherché autre chose qu'un premier contact, une occasion de faire des mathématiques ou des sciences physiques.

Le module MP2 reste, lui aussi, modeste. Cependant, en plus d'une nouvelle occasion de faire encore un peu de mathématiques, on devrait y trouver au moins les éléments nécessaires pour écrire de manière propre un petit programme en BASIC.

L'orientation générale donnée à la composante informatique de ce module n'a pas été définie sans peine. Le lecteur trouvera peut-être dans le présent fascicule un peu plus de rigueur que dans le précédent; elle est inspirée par des documents pédagogiques de J.-P Bertrandias, professeur à l'Université de Grenoble 1.

Si répandre l'informatique dans tous les milieux est une idée à la mode, il n'est pas sûr que l'on ait trouvé la bonne manière de le faire. Le présent traité ne le prétend pas non plus; d'ailleurs ce n'est pas exactement son rôle, l'informatique n'étant pas sa seule motivation. Faute de pouvoir définir la bonne solution, il est au moins possible d'inventorier quelques écueils, ce que nous allons commencer par faire.

L'informatique bidouillante

C'est l'informatique qui se pratiquerait dans les clubs et que les informaticiens dénoncent depuis longtemps; ils la rendent responsable des erreurs qui fleurissent dans les programmes, avec parfois des conséquences graves. La généralisation du langage BASIC, qui n'est pas un langage structuré, la favorise. Le fascicule de l'unité MP1 s'approche dangereusement de cet écueil dans la mesure où l'on ne fournit pas de méthodes, ce qui va être en partie réparé dans celui-ci. Le choix est délibéré et le risque calculé: procéder par essais et erreurs est la base de l'apprentissage. On y reviendra plus loin.

L'informatique bourbachisante

Le groupe qui a pris le nom collectif de Nicolas Bourbaki a apporté, dès la fin des années 30, ouverture et renom aux mathématiques françaises. Malheureusement ceux qui ont voulu transposer ses écrits dans l'enseignement de tous les niveaux, maternelle comprise, ont produit la catastrophe des "mathématiques modernes". En informatique, il y a une quinzaine d'années, on a développé la programmation systématique ou structurée pour réagir contre le bidouillage. Il est normal que les cours universitaires s'en inspirent. En revanche ce serait également commettre une erreur de transposer ce formalisme dans les autres niveaux de l'enseignement. De ce côté, comme pour les

mathématiques, ce sont les cercles pédagogiques qu'il faut redouter.

Vous ne verrez pas l'ombre de cet écueil dans le présent fascicule.

L'informatique débile

Elle se rencontre par exemple à la télévision à l'occasion d'émissions grand public d'initiation. On y apprend comment écrire "bonjour" à l'écran en frappant PRINT "bonjour" au clavier, ou bien à colorer l'écran en bleu ... Elle n'est pas absente des revues ou des ouvrages d'initiation où elle se mêle au bidouillage. Chaque fois que l'on insiste sur les commandes graphiques de tel langage, que l'on s'attarde sur les particularités de tel matériel ... on touche l'écueil. La tentation est forte car la pratique de cette informatique est accessible à tous. Le fascicule de l'unité MP1 y a un peu cédé au chapitre 2; il est parfois difficile de faire autrement. L'utilisation d'une austère calculatrice avec un écran de petite taille est un remède assez efficace.

Un test : si vous savez par exemple à quoi correspond l'instruction BOXF, vous n'y avez pas échappé.

L'informatique presse-bouton

C'est l'utilisation de logiciels évolués dont on apprend à se servir sans avoir la moindre idée de la manière dont ils fonctionnent. Bien sûr cette pratique est utile à certains, comme au journaliste qui écrit de nombreux rapports à l'aide d'un traitement de textes, ou à l'enseignant qui se sert d'un bon didacticiel dans sa classe. Mais ce n'est pas davantage faire de l'informatique que regarder la télévision n'est faire de l'électronique. La comparaison amène d'ailleurs cette remarque : utiliser un poste de télévision s'apprend beaucoup plus vite ; les logiciels de traitements de texte ont des progrès à faire à cet égard et apprendre aujourd'hui leur maniement à quelqu'un qui n'en a pas l'usage immédiat est du temps perdu.

Cet écueil est pernicieux et a pour lui la mode. Il est mis en avant par toutes les initiatives récentes de l'Education Nationale : informatique pour tous, nanoréseau ...

Chercheur ou Bricoleur

Heureusement on peut voir les choses de manière moins négative, le bidouillage et l'abstraction sont aussi complémentaires, à condition de ne se laisser enfermer ni dans l'un ni dans l'autre. La complaisance à l'égard du premier écueil vient de la crainte de donner aux gens les moyens de bien programmer sans leur donner le goût de le faire, au point qu'ils n'écrivent jamais d'eux-mêmes un seul vrai programme. Ce reproche s'applique aussi bien à l'enseignement des mathématiques.

A cet égard il est intéressant de comparer la démarche du mathématicien professionnel et celle de l'informaticien amateur. La découverte d'un théorème par un mathématicien comprend souvent trois phases :

une phase de tâtonnements : de calculs dans des cas particuliers, de tentatives de réduction du problème à un autre ou de généralisation simplificatrice ... qui s'achève sur un sentiment d'échec mais aura permis d'apporter sur le chantier le matériel qui sera plus tard utilisé pour la solution,

une phase de décantation, d'une semaine ou d'une année, pendant laquelle le chercheur pense officiellement à autre chose, les idées se mettant inconsciemment dans l'ordre,

l'éclair qui montre que les éléments étaient là, prêts à assembler, la démonstration s'écrivant toute seule.

Poincaré a admirablement décrit cela quelque part, ceux qui connaissent reliront.

L'informaticien amateur qui veut écrire un programme présentant quelque difficulté passe souvent aussi par au moins trois étapes :

une étape de bricolage : d'écriture de sous-programmes, de décomposition du programme en modules ... qui aboutit à un programme qui tourne mais mal, donnant déjà une idée de ce qui sera possible ou non compte-tenu des limitations matérielles,

une étape de remise en ordre, consciente, pour mieux spécifier le problème et l'analyser de manière plus systématique,

une étape finale d'écriture, apportant une impression de facilité et d'harmonie.

Ici on peut lire Knuth qui consacre quelquepart dans le tome 1 deux pages à expliquer comment il travaille, et qui reste un amateur, mais au sens noble du terme.

Ces lignes sont pour montrer que d'une certaine manière la pratique de l'informatique à un niveau accessible à tous se rapproche de la recherche scientifique de haut niveau. En tout cas elle tranche sur une forme d'enseignement ne laissant aucune initiative et débouchant sur des exercices dont l'énoncé contient la solution.

Ce qui restera

La science évolue vite et il est permis de se demander si ce que l'on apprend aujourd'hui servira encore dans vingt ans. Comme on l'a dit cela écarte l'informatique presse-boutons. Cela ne plaide pas non plus en faveur de la manipulation de fichiers dans des langages comme BASIC ou Pascal tant qu'un effort d'uniformisation n'aura pas abouti.

En revanche on peut dire sans grande chance de se tromper qu'il y aura toujours des choix, des itérations, que l'on emploiera toujours le formalisme des mathématiques là où ces dernières servent. Les techniques algorithmiques qui n'ont pas beaucoup évolué depuis le début évolueront peu dans l'avenir.

Paradoxalement les perspectives d'évolution plaident en faveur d'un vieux BASIC minimum, ne nécessitant qu'un investissement léger et permettant de faire déjà beaucoup de choses par soi-même dans des situations très diverses, intermédiaire entre les langages de bas niveau et les langages évolués.

Avec INPUT, PRINT, IF ... THEN, GOTO, GOSUB, RETURN, FOR ... TO ... STEP ... NEXT ..., des variables numériques et un tableau on peut déjà travailler.

On peut se passager de ELSE qui ne se programme bien qu'avec un parenthésage comme Begin ... end, et des itérations conditionnelles comme WHILE ... WEND qui rendent désagréable la présence de numéros de ligne.

Informatique et enseignement des sciences

Il y a au moins deux manières d'associer l'informatique à l'enseignement des mathématiques.

D'une part on peut limiter l'informatique à un rôle d'illustration, de prolongement naturel à l'occasion d'une application numérique : on écrira un programme pour résoudre des équations linéaires, des équations différentielles ... La part de l'informatique est légère. Cette attitude se trouve dans les exemples inspirés du module AN1.

D'autre part on peut imaginer d'utiliser l'informatique pour renouveler complètement l'enseignement des mathématiques. Partant d'un problème concret où les mathématiques ne sont pas apparentes, pour écrire un programme on va découvrir de vrais problèmes mathématiques, ce qui donnera une occasion d'en faire. L'avantage est d'avoir fixé un objectif qui peut être motivant : ce peut être la réalisation d'un jeu réaliste et complexe. C'est aussi celui d'une sanction indiscutable : une solution inexacte conduit à un programme faux avec des conséquences évidentes. Ce peut être une voie à expérimenter là où les méthodes traditionnelles ont échoué. Cependant l'investissement est important, les détails pratiques ne pouvant être escamotés. Certains thèmes des modules MP1

et MP2 sont un peu orientés dans ce sens.

On pourrait penser que les liens entre les mathématiques et la nouvelle science qu'est l'informatique vont détourner les premières de leurs partenaires traditionnels que sont par exemple les sciences physiques. Il n'en est rien : au contraire l'usage de l'informatique renforce ces liens, au même titre que ceux avec les sciences biologiques, économiques ... Prenons l'exemple d'une équation différentielle. Quel besoin aurait-on de calculer une valeur approchée de telle solution pour telle valeur de la variable, dans un exemple d'école ? Quel sens affectif attacherait-on à une valeur de 10^6 ou 10^{-6} ? Le traitement numérique n'est gratifiant que si le problème lui-même a des origines concrètes et donc des conclusions interprétables.

L'art pour l'art

Dans un ouvrage collectif qui fait autorité dans l'enseignement primaire, on peut apprendre que les mathématiques n'ont ni rôle social, ni rôle professionnel, leur seul but étant la maîtrise d'un langage. Certains, qui voient dans l'informatique un allié au service exclusif de l'acquisition de notions abstraites, partagent peut-être ce point de vue. D'autres pensent être plus réalistes mais semblent ignorer que l'écriture de programmes pour réaliser ce que l'on peut faire à la main sans effort avec du papier quadrillé n'est pas si éloigné des exercices stériles sur les relations d'équivalence ou les patatoïdes.

Tout cela donne une vision déformée de la réalité scientifique. Il n'est pas mauvais d'insister sur ceci : l'objet des mathématiques, comme de l'informatique et de la science en général, est d'accroître la productivité.

En informatique on doit chercher le gain de temps, l'adaptation à des situations multiples, la suppression de tâches répétitives. Les exemples de ce module sont imparfaits, mais on a eu le souci de se conformer à ces critères.

En mathématiques le respect du précepte aurait pu éviter des déboires. Pour le calcul de limites simples, comme celles qui servent à déterminer des asymptotes, on a préconisé il y a quelques années les filtres ; maintenant certains font refaire le même calcul numérique approché à chaque fois qu'il est dans l'un ou l'autre cas. Le gain de productivité ? A un niveau plus élémentaire, en quoi la numération en base 3, 4 ou 5 améliore-t-elle la productivité du français moyen?

1

Variables

1. Introduction : utilisation de variables dans l'écriture d'un programme.

Choisissons un problème de l'école élémentaire à titre d'illustration : sachant qu'un produit se vend 59 F avec un taux de TVA de 18%, combien se vendra-t-il lorsque le taux de TVA sera porté à 33% ?

Mettons-en parallèle la manière dont ce petit problème était résolu il y a encore quelques années et l'écriture d'un programme.

Rappel des données

le taux de TVA faible est 18

le taux de TVA fort est 33

le prix avec taux faible est 59

Nombre de centièmes du prix HT dans un prix avec taux faible :

100 + 18 = 118

Id dans un prix avec taux fort :

100 + 33 = 133

Centième du prix HT :

59 : 118 = 0,50 F

Prix avec taux fort :

0,50 × 133 = 66,50 F

Introduction des données

```
10  T = 18 :
    U = 33
20  P = 65
```

Calcul des multiplicateurs associés au taux faible

```
30  M = 100+T :
```

et au taux fort

```
    N = 100+U
```

Calcul du centième du prix HT

```
40  X = P/M
```

et détermination du résultat

```
50  Q = X*N
```


Ces quelques lignes amènent un premier commentaire. Dans la colonne de droite, lorsqu'on affecte à une variable une valeur numérique ou la valeur calculée d'une expression, on commence par écrire le nom (l'identificateur) de la variable. L'usage qui voulait que l'on écrive le résultat cherché avant de poser l'opération procède de la même logique ; c'est une bonne raison pour le réintroduire. De même, dans l'écriture d'un programme, on décrira les variables avant leur utilisation dans des commentaires ou un lexique.

On a rappelé les données au début de la colonne de gauche, ce qui n'était pas indispensable. On aurait pu également à droite se dispenser des lignes 10 et 20 et remplacer les variables par les valeurs numériques particulières du problème. Il est cependant judicieux d'utiliser des variables, même en mode direct, sans programme ; cela facilite l'écriture de formules, permet d'utiliser une donnée sans la réécrire. Surtout, l'écriture d'un programme pour un problème aussi simple ne se justifie que si on l'utilise plusieurs fois avec des données différentes.

Il faudra alors qu'à chaque lancement du programme, l'utilisateur communique à la machine les valeurs particulières des données. Rappelons que cela se fait par l'instruction d'entrée INPUT en BASIC ; on réécrira donc la ligne 20 sous la forme

```
20  INPUT P
```

si l'on veut faire varier le prix avec taux faible, avec

éventuellement un commentaire

```
20  INPUT "PRIX1="; P
```

De même, pour ne pas avoir à interroger les variables après l'exécution du programme on ajoutera à la fin une instruction de sortie

```
60  PRINT  Q
```

ou

```
60  PRINT "PRIX2="; Q
```

La solution indiquée relève d'une « analyse guidée par les données ». On a en effet cherché de proche en proche ce qui pouvait être déterminé à partir des données du problème : nombre de centièmes du prix HT, centième du prix HT... jusqu'à obtenir le résultat. L'avantage de cette méthode est qu'elle fournit l'ordre des opérations qui sera aussi celui du programme.

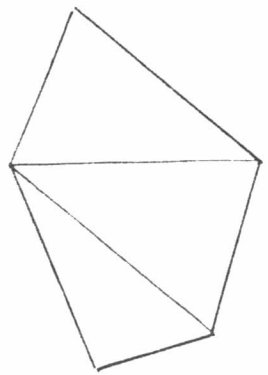
Une autre attaque est celle de l'« analyse guidée par les résultats ». Elle consiste à partir ici du prix Q avec taux fort que l'on cherche, pour constater qu'il s'obtient par la formule

```
Q = X*N
```

où X est le prix HT en centimes et N le multiplicateur pour le taux fort. Or on sait précisément calculer à la fois X et N à partir des données P et M. L'écriture du programme nécessite ici de remonter les étapes. Cela peut sembler lourd mais c'est pourtant ainsi qu'il faut partir ; quand on veut aller quelque part mieux vaut regarder le but. La première méthode supposait cette première analyse était inconsciente.

2. Composer les expressions

Passons du cours moyen à la classe de seconde pour envisager un nouveau problème. Chargé d'évaluer la surface d'un certain nombre de terrains aux formes polygonales mais irrégulières, vous envisagez de les découper en triangles dont vous mesurerez les côtés à l'aide d'un double décamètre sur le terrain ou d'un double décimètre sur le plan cadastral, il s'agit d'obtenir la surface par calcul.



C'est un exemple typique de problème menant à des calculs répétitifs et atroces à effectuer à la main, pour lequel l'appoint de moyens informatiques légers est utile.

Une première solution consiste à chercher dans un manuel la formule adéquate, de choisir une forme dans laquelle ne figurent au second membre que les seules données a, b, c qui sont les côtés du triangle, par exemple

$$S = \frac{1}{2}\sqrt{b^2c^2 - \left(\frac{b^2+c^2-a^2}{2}\right)^2}$$

et de reproduire directement cette formule dans une expression en BASIC avec plus ou moins de parenthèses suivant la machine comme

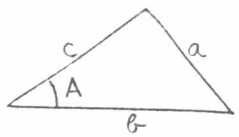
```
S=(√((B↑2)*(C↑2)-((B↑2+C↑2-A↑2)/2)↑2))/2
```

ce qui, précisément à cause des parenthèses, représente un exercice périlleux. Il est déjà plus simple de remplacer les

suivés par les produits, il faut donner

$$S = (\sqrt{(B*B+C*C-(B*B+C*C-A*A)*(B*B+C*C-A*A)/4)})/2 .$$

Une autre manière de travailler est de lire les quelques lignes qui précèdent la formule dans le manuel. On trouvera ainsi



$$S = \frac{1}{2} bc \sin A .$$

Il suffit donc de déterminer $\sin A$. Or

$$\sin A = \sqrt{1-\cos^2 A} ,$$

alors qu'à son tour $\cos A$ peut être tiré de

$$a^2 = b^2 + c^2 - 2bc \cos A$$

soit

$$\cos A = \frac{b^2+c^2-a^2}{2bc} .$$

En remontant les calculs on aboutit aux lignes de programme

```
D = (B*B+C*C-A*A)/2/B/C
E = √(1-D*D)
S = B*C*E/2
```

On a supprimé tout problème de parenthésage et en même temps gagné en efficacité. On peut faire mieux en utilisant $bc \cos A$ comme variable intermédiaire d'où

```
F = (B*B+C*C-A*A)/2
S = (√(B*B*C*C-F*F))/2 .
```

Quelles conclusions tirer ?

Que l'analyse guidée par les résultats a rendu ici une service dans un problème pourtant résolu.

Que l'utilisation de variables intermédiaires plutôt que

celle de formules entièrement développées est avantageuse à la fois pour la lisibilité en mathématiques et l'efficacité en programmation

Que l'informatique n'est pas le biais pour parler du parenthésage. Les langages de programmation sont encore lourdauds par comparaison à la notation mathématique usuelle ; c'est normal ils sont moins intelligents que l'homme qui les a créés.

Que ce type d'activités fait voir davantage les fonctions comme des formules opératoires que des graphes abstraits, ce qui finalement rejoint le point de vue des anciens comme Cauchy. Attention cependant : quelques difficultés seront signalés plus loin.

3. Le hiatus

Entre le cours moyen où l'on peut se contenter de travailler comme autrefois et le lycée où le cours de mathématiques donne des occasions de programmer, c'est-à-dire au collège, y-a-t-il une place pour des activités analogues? Très probablement mais bien des erreurs ont été faites.

Dire qu'une touche de calculatrice comme [COS] est une fonction est absurde. La calculatrice possède des touches de fonctions, mais les fonctions sont $\sqrt{x}$, $\cos x$, $\sin x$ et non les touches. Quel est incidemment le domaine de définition d'une touche ?

A genre de considérations contribue à faire des mathématiques le royaume de l'inexpliqué ; au contraire l'écriture d'un petit programme calculant maladroitement la fonction cosx est instructif et libérateur

4. Les variables dans la mémoire

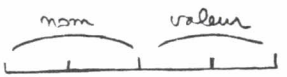
Nous évoquons très brièvement l'utilisation de la mémoire pour représenter les variables. Il a été dit dans le fascicule MP1 qu'à chaque variable correspondait une case en mémoire, symbolisée par le petit dessin

Les dimensions de la case varient suivant le type de variable concerné et la quantité d'information nécessitée. Son organisation dépend aussi de la taille des mots sur lesquels travaille le microprocesseur. Un TO7 par exemple utilise des mots de 8 bit, ou octets, c'est à dire contenant 8 informations élémentaires de 1 bit, lequel prend seulement 2 valeurs symbolisés par 0 et 1.

De 00000000
à 11111111,

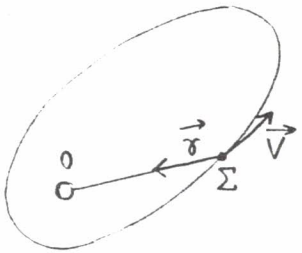
il y a $2^8 = 256$ valeurs possibles pour un tel mot. Si l'on veut représenter un entier de 0 à 65536 ou de −32768 à 32767 il suffit de 2 octets. C'est du moins vrai pour une machine dans laquelle une variable comme A se trouve toujours à la même place, avec pour conséquence une organisation figée de la mémoire. Les calculatrices SHARP ont une organisation de ce genre, et par suite ne reconnaissent qu'un seul type de variable numérique, perdant du même coup beaucoup de place pour les entiers.

Si l'on veut utiliser par exemple des noms de variables de deux lettres, pour un petit nombre de variables, réellement utilisées, une organisation figé hypothèquerait une mémoire considérable. Il plus efficace, reprenant le cas d'entiers sur deux octets, de placer devant les deux octets contenant la valeur deux autres contenant les lettres du nom, avec par exemple le codage ASCII.



S'il n'y a que des variables entiers placés à partir d'une adresse connue on pourra retrouver la valeur d'une variable dont on connait le nom sans risque de confondre les octets d'un nom et ceux d'une valeur (qui sont dans les deux cas des suites de 0 et de 1). Si l'on veut plusieurs types et des noms de grande longueur une organisation plus élaboré est nécessaire.

Exercice : le satellite



On veut écrire un programme aidant à représenter la trajectoire d'un satellite Σ ; il devra en fonction de la position et de la vitesse au temps t, déterminer la position et la vitesse au temps $t+\Delta t$ où Δt est petit par rapport au temps de révolution.

On place l'origine sur la planète attractrice et on choisit des axes dans le plan de la trajectoire. Le satellite se trouve en (a,b) avec la vitesse (m,n) en t et va se trouver en (c,d) avec la vitesse (p,q) en $t+\Delta t$. La vitesse ayant peu varié va représenter à peu près le taux d'accroissement vectoriel

$$\frac{(c,d)-(a,b)}{\Delta t},$$

ce qui donne

$$\frac{c-a}{\Delta t} = m \quad , \quad \frac{d-b}{\Delta t} = n$$

et donc

$$c = a + m\Delta t \quad , \quad d = b + n\Delta t .$$

Il faut maintenant corriger la vitesse pour un usage ultérieur. Son taux d'accroissement

$$\frac{(p,q)-(m,n)}{\Delta t}$$

représentant à peu près l'accélération $\vec{\gamma}$ au temps t, elle même proportionnelle à la force d'attraction, c'est à dire, d'après la loi de Newton, portée par le rayon $\overrightarrow{O\Sigma}$,

dirigée vers O et en longueur proportionnelle à l'inverse du carré de la distance de Σ à O. Si ℓ est cette longueur, donnée par

$$\ell = \sqrt{a^2+b^2},$$

l'accélération sera simplement proportionnelle à $\frac{\overrightarrow{\Sigma O}}{\ell^3}$, vecteur de composantes $\left(\frac{-a}{\ell^3}, \frac{-b}{\ell^3}\right)$. On obtient ainsi

$$p = m - k\frac{a}{\ell^3}$$

$$q = n - k\frac{b}{\ell^3}$$

Ecrire maintenant le programme, en remarquant qu'il ne sert à rien de développer les formules

4

Variables (suite) : passage de paramètres

1. Un exemple

Supposons que la résolution d'un problème demande d'effectuer à plusieurs reprises et dans des conditions différentes la transformation linéaire f qui au point du plan de coordonnées x, y associe le point du plan de coordonnées u, v suivant les formules :

$$\begin{cases} u = x - y \\ v = x + y \end{cases},$$

que l'on résume dans le symbolisme

$$(u, v) = f((x, y)).$$

Il est immédiat de réaliser un petit sous-programme en langage BASIC traduisant les relations précédentes. Par exemple :

```
100   U=X-Y
110   V=X+Y
120   RETURN
```

et alors GOSUB produira dans certains cas le résultat cherché.

Imaginons maintenant que l'on veuille calculer le point, toujours noté (u, v) obtenu à partir de (x, y) par l'application de f deux fois, c'est à dire

$$(u, v) = f(f((x, y)))$$

Peut-on se contenter de GOSUB 100 : GOSUB 100 ? Faites l'essai dans le petit programme

```
10   INPUT X,Y
20   GOSUB 100:GOSUB 100
30   PRINT U,V
40   END
```

Vous constaterez que le nombre d'appels GOSUB 100 de la ligne 20 est indifférent. Dans un langage comme BASIC il y a donc moins de souplesse dans l'utilisation des variables qu'en mathématiques.

2. Paramètres d'entrée

Regardons ce que fait le sous-programme. Il prend les valeurs dans les cases mémoires des variables X, Y et met les résultats dans celles des variables U, V. Si nous voulons que les transformations s'enchaînent il faut prendre comme valeurs de départ de la seconde les résultats de la première. Cela demandera des affectations, la ligne 20 étant transformée en

```
GOSUB 100:
X=U:
Y=V:
GOSUB 100
```

Maintenant le résultat est bon. Malheureusement si on veut se resservir du point (x, y) on constatera avec désolation que les valeurs des variables X, Y ont changé.

Pour éviter cet inconvénient il ne faut pas confondre sous les mêmes noms X, Y les variables qui sont les paramètres d'entrée du sous-programme et les variables du programme principal ; on réservera donc aux paramètres d'entrée des noms particuliers, par exemple S, T, réécrivant le sous-programme sous la forme

```
100   U=S-T
110   V=S+T
120   RETURN
```

L'appel du sous-programme sera précédé d'une affectation aux paramètre d'entrée suivant le modèle

```
S =
T =
```

Le premier appel donnera ainsi lieu à

```
S=X :
T=Y :
GOSUB 100
```

et le second

```
S=U :
T=V :
GOSUB 100
```

3. Paramètres de sortie

Jusqu'ici nous avons toujours appelé u, v les éléments du résultat, c'est à dire les paramètres de sortie. Il est bien évident qu'une telle limitation est insupportable. Il serait agréable d'indiquer au sous-programme quelles sont les variables souhaitées pour le résultat, comme on le fait en mathématiques en écrivant

simplement

$$(p, q) = f(x, y)$$

Pour ce faire le sous-programme n'a pas besoin comme pour les paramètres d'entrée de transmettre des valeurs mais des noms, ceux des variables dans lesquelles on veut le résultat. Cela n'est pas possible en BASIC normal. Si l'on veut affecter le résultat à d'autres variables, il faudra le faire par des affectations au retour de l'appel. On trouvera par conséquent assez souvent le schéma suivant

```
paramètre d'entrée = ...
GOSUB sous-programme
      = expression utilisant les
        résultats
```

4. Conclusion provisoire

Lorsque l'étude d'un problème a mis en évidence des sous-actions donnant lieu à des modules, il est important de bien noter pour chacun des modules destinés à servir en plusieurs endroits du programme principal

- les paramètres d'entrée : quelles informations le module doit-il recevoir ?
- les paramètres de sortie : quelles informations le module doit-il fournir ?

Ce épargne beaucoup d'ennuis ultérieurs.

Une fois ce travail fait, et les précautions prises pour qu'il n'y ait pas d'interférence entre les variables du module et celles du programme principal, il n'y a plus à s'occuper de ce qui se passe à l'intérieur du module lorsqu'on l'utilise ; il suffit de connaître l'action qu'il est chargé d'exécuter.

5. Un mot sur les langages plus évolués

Certains BASIC dits évolués, et d'autres langages comme LSE ou Pascal ... disposent de procédures avec paramètres.

Il n'est plus nécessaire d'avoir recours à des affectations préalables du type de celles que nous avons évoquées. Si, en Pascal, dans la transformation linéaire que nous appellerons lin nous déclarons les paramètres d'entrée par

```
procedure lin (s,t : real)
```

en tête de la procédure, laquelle consistera en

```
begin
   u := s-t ;
   v := s+t
end;
```

la première application de f donnera simplement lieu à

```
lin(x,y)
```

et la double application à

```
lin(x,y) ; lin(u,v)
```

Quand le programme principal rencontre lin(x,y) les valeurs de x,y sont recopiées automatiquement dans les variables u,v

et la procédure lin est appelée.

Contrairement au BASIC les noms s,t ici pris comme paramètres d'entrée peuvent être réutilisés dans le programme principal sans risque d'interférence ; ces variables sont dites locales.

On peut également déclarer des paramètres de sortie sous une forme telle que

```
procedure lin ( s,t : real ; var u,v : real).
```

Alors lin(x,y ; x₁,y₁) réalisera

$$(x_1, y_1) = f(x, y)$$

et lin(x,y ; x₁,y₁) ; lin(x₁,y₁ ; x₂,y₂) réalisera

$$(x_2, y_2) = f(f(x, y))$$

6. Exercice : le satellite (suite)

On veut maintenant utiliser le programme de la première partie du chapitre comme sous-programme pour faire afficher les positions successives ou les faire dessiner sur un écran avec une instruction de tracé de segment.

Utiliser ce qui a été vu dans ce chapitre à cette fin

2

Discussion, étude par cas

1. Introduction

Supposons que l'on veuille écrire un petit programme donnant la solution de l'équation $ax+b$ lorsqu'on lui fournit les coefficients a, b.

Au lieu de se jeter sur la formule $x=-b/a$, d'écrire une ligne telle que $X=-B/A$ et de constater que la machine répond dans certains cas par un message d'erreur, on peut réfléchir à la manière dont la solution est obtenue.

Après avoir écrit l'équation

$$ax=-b$$

on essaie de diviser par a les deux membres. On distingue donc

un 1er cas : $a \neq 0$,

dans lequel on aboutit évidemment au résultat indiqué.

Mais il faudra considérer également le cas restant, c'est à dire

un 2ème cas : $a=0$,

dans lequel l'équation se résume à

$$0=-b$$

Ce cas se divise alors lui-même en

un 1er sous-cas $b \neq 0$,

pour lequel l'équation n'a pas de solution,

et un 2ème sous-cas : $b=0$,

pour lequel toute valeur est solution

Au 1 nous n'avions envisagé que des actions en "séquence". Ici la programmation de notre solution va utiliser ce qu'en langage algorithmique on appelle des "choix", ce que l'on schématise en général sous la forme

Si (condition) alors (action X)

sinon (action Y),

et, qui dans le cas envisagé, peut donner lieu à

si $a \neq 0$ alors écrire $-b/a$

sinon si $b \neq 0$ écrire "impossible"

sinon écrire "équation indéterminée"

Remarque : l'exemple précédent montre que l'étude méthodique d'un problème peut fournir de manière automatique l'organisation algorithmique. En revanche il est difficile de se retrouver à partir d'éléments disparates de la solution. Si, à un informaticien qui ne connaîtrait rien de l'équation du premier degré, vous indiquez que la solution est $-b/a$, mais que si $a=0$ c'est impossible encore que si $a=b=0$ c'est finalement indéterminé, il devra démêler des informations qui se contredisent. Vous serez peut-être conduit à tout lui préciser, à savoir que la solution est

$-b/a$ si $a \neq 0$

impossible si $a = 0$ et $b \neq 0$

indéterminé si $a = 0$ et $b = 0$,

indications qu'il fera figurer dans un tableau à deux entrées tel que

	$a \neq 0$	$a = 0$
$b \neq 0$	$-b/a$	impossible
$b = 0$	$-b/a$	indéterminé

après quoi il constatera qu'il est plus commode de faire porter le premier choix sur a et retrouvera la formulation algorithmique que vous avez si naturellement obtenue.

S'il est un peu distrait, il pourra aussi bien aboutir à la formulation suivante, tout aussi correcte mais maladroite en l'absence de raisons spéciales

si $b = 0$ alors si $a = 0$ alors ...

sinon ...

sinon si $a = 0$ alors

sinon ...

2. Programmation des choix en BASIC minimum

Le BASIC minimum ne connaît pas d'instruction traduisant le "sinon". On peut programmer facilement un choix si l'action Y est vide et si l'action X peut tenir dans la fin de la ligne. Ainsi

si (condition) alors (action X)

sinon (rien)

se traduit alors par

IF (condition) THEN (instructions réalisant X séparées par des :)

Si l'action Y est non vide, nécessitant éventuellement plusieurs lignes, mais l'action X tenant toujours en une seule ligne, on traduira

si (condition) alors (action X)

sinon (action Y)

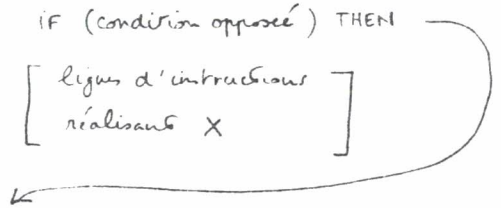
par

IF (condition) THEN (instructions réalisant X séparées par des :) : GOTO

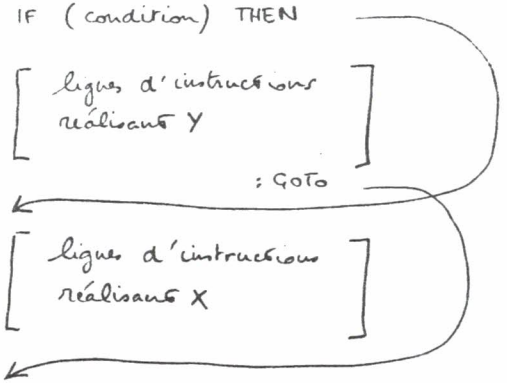
[lignes d'instructions réalisant l'action Y]

suite éventuelle

Dans le cas où l'action X nécessite plusieurs lignes, l'action Y étant vide, on se ramène au cas précédent en inversant la condition



Enfin le cas général donnera lieu à



Dans notre petit exemple nous écrirons par exemple

```
10  INPUT A,B
20  IF A<>0 THEN PRINT -B/A : GOTO 50
30  IF B<>0 THEN PRINT "IMPOSSIBLE": GOTO 50
40  PRINT "EQUATION INDETERMINEE"
50  suite ou fin
```

On a d'abord traité le cas n'utilisant qu'une ligne.

3. Exercice

Reprendre la discussion du 1. avec le système de deux équations

$$ax + by = e$$
$$cx + dy = f.$$

La méthode suggérée consiste à considérer d'abord le cas où l'un des quatre coefficients a, b, c, d n'est pas nul; supposons par exemple que $a \neq 0$; en multipliant la première équation par $-\frac{c}{a}$ et la retranchant à la seconde, on remplace le système en un système équivalent

$$ax + by = e$$
$$\left(-\frac{bc}{a} + d\right)y = -\frac{ec}{a} + f \quad \text{ou} \quad (ad - bc)y = af - ec.$$

Si $ad - bc \neq 0$ on tire y de la seconde équation et on en déduit x. Il y a une solution et une seule

Si $ad - bc = 0$ le système se réduit à

$$ax + by = e$$

soit $x = -b/a\, y + e/a$.

Géométriquement les solutions sont les points d'une droite. Les conclusions auraient été analogues pour b, c ou $d \neq 0$.

Si $a = b = c = d = 0$ alors le système se résume à

$$0 = e$$
$$0 = f,$$

la discussion étant facile.

4. Un mot à propos du langage Pascal

La formulation algorithmique indiquée en 1. se traduit directement en Pascal par

if (condition) **then** (instruction)
else (instruction),

la partie **else** étant facultative.

Par instruction on entend une instruction simple ou une séquence d'instructions encadrées par le parenthésage

begin
instructions séparées par des ;
end

Certains BASIC offrent également l'instruction ELSE. Cela simplifie l'écriture de choix lorsque les actions sont simples. En revanche l'avantage est faible lorsque la traduction des actions occupe plusieurs lignes. Ce n'est finalement pas la présence ou l'absence de ELSE qui importe, mais la structure en lignes du programme.

2

Etude par cas (suite)

1. Choix multiples

Dans les exemples choisis jusqu'ici la discussion, à chaque stade, s'est articulée autour d'une condition qui peut être réalisée ou non. Il n'y a avait donc que deux choix possibles. Dans d'autres situations il arrive que la discussion amène naturellement l'examen d'un nombre de cas supérieur à 2.

Ainsi, si, dans l'exemple du système d'équations, on veut détailler la discussion lorsque l'un au moins des coefficients a, b, c ou d n'est pas nul, on peut distinguer quatre cas.

1er cas : $a \neq 0$
2ème cas : $a = 0,\ b \neq 0$
3ème cas : $a = 0,\ b = 0,\ c \neq 0$
4ème cas : $a = 0,\ b = 0,\ c = 0,\ d \neq 0$.

On peut toujours ramener un choix multiples à une succession de choix simples, sous la forme

si (condition 1) alors ...
sinon si (condition 2) alors ...
sinon si ...

Dans l'exemple que nous venons de prendre, le choix multiple était par essence une succession de choix simples, à savoir

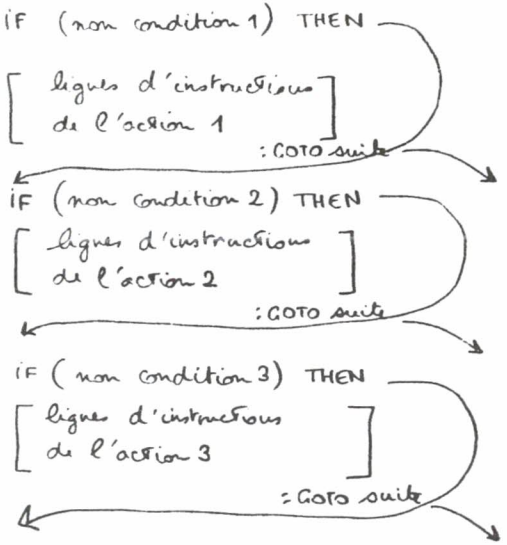
si $a \neq 0$ alors ...
sinon si $b \neq 0$ alors ...
sinon si $c \neq 0$ alors ...
sinon ...

Dans un choix multiple, il convient que les différents cas s'excluent mutuellement. De plus ils doivent couvrir toutes les éventualités, à moins que l'on ne souhaite conserver un cas restant pour l'action correspondante serait vide.

2. Programmation d'un choix multiple en BASIC minimum

Les limitations du langage BASIC dues à son organisation en lignes font qu'il est difficile de donner une formule universelle. Nous écartons par ailleurs ici l'utilisation de branchements calculés ou d'instructions du type ON... GOTO... ou ON... GOSUB..., dont l'intérêt est limité à des cas bien précis.

Voici quand même une manière assez propre de programmer un choix multiple, qui peut s'appliquer à toutes les situations; elle consiste à inverser les différentes conditions rencontrées de manière à provoquer un saut lorsque la condition n'est pas réalisée.



etc

GOTO est omis à la fin des choix

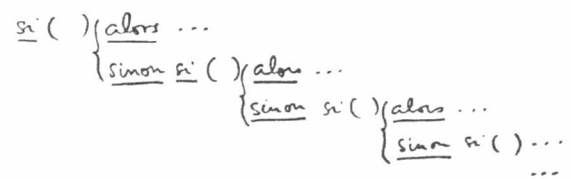
3. Priorités

Lorsqu'on transforme un choix multiple avec beaucoup de cas en une succession de choix simples, il est important de bien choisir l'ordre dans lequel les choix seront effectués. En effet, si l'on se trouve dans le 1er cas, le travail de la machine se résumera à évaluer la condition 1 et effectuer les instructions de l'action 1; dans le nème cas, avant de faire un travail analogue pour la condition n et l'action n, la machine aura dû évaluer successivement les conditions 1, .., n-1, pour constater qu'elles ne sont pas remplies. Si les conditions sont complexes le temps passé par la machine peut être beaucoup plus long.

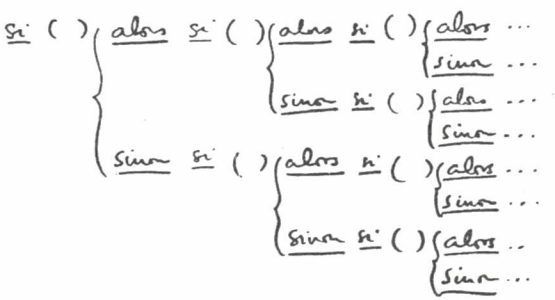
Voici quelques idées à retenir.

Lorsqu'un cas est nettement plus fréquent que les autres, le mettre si possible au début.

Lorsqu'aucune priorité n'est évidente, essayer de diminuer la "profondeur" en provoquant des regroupements. Autrement dit remplacer une cascade telle que



par un arbre tel que



Cela dit, on n'est pas toujours maître de l'ordre ; il arrive assez souvent que certaines conditions ne peuvent être évaluées que lorsque d'autres sont remplies. Une fois de plus une discussion méthodique apportera davantage que des recettes de programmation.

4. Un mot sur le langage Pascal

Les choix multiples s'y programment aisément grâce à l'instruction **case** qui s'emploie comme suit

```
case ( expression ) of
    (valeur 1) : (instruction 1) ;
    ...
    (valeur n) : (instruction n)
end ;
```

5. Exercices

- Ecrire un programme qui à partir d'une chaîne de trois caractères dont le premier est a, le troisième b et le second choisi parmi +, −, * et /, donne le résultat de l'opération suggérée entre les variables a et b.
- Reprendre la discussion sur le système du premier degré en détaillant la discussion comme indiqué au 1.

3

Itérations

1. Introduction

Supposons que l'on doive programmer le calcul d'une fonction transcendante comme la fonction exponentielle à partir de la série

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} .$$

Bien entendu on ne cherchera qu'une valeur approchée et on ne calculera donc qu'un nombre fini de termes. Si l'on peut se contenter d'un nombre fixé une fois pour toutes, 3 ou 5 par exemple, on peut à la rigueur travailler de manière séquentielle, en recopiant 3 ou 5 fois les mêmes instructions, ou en appelant 3 ou 5 fois de suite un sous-programme.

En revanche si le nombre de termes peut dépendre des exigences de l'utilisateur, ou bien de la valeur x considérée et a fortiori s'il n'est pas connu d'avance, il faut utiliser un outil algorithmique nouveau : l'itération, ou répétition ou boucle

2. Boucle à contrôle par compteur

Le cas le plus simple est celui où le nombre de fois que le traitement doit être effectué est connu à l'avance.

Par exemple, ayant limité x à l'intervalle [0,1], on peut démontrer que le reste de la série est inférieur à

$$\frac{e}{(n+1)!}$$

Si l'on cherche seulement 4 décimales exactes (il est plus correct de parler d'une erreur inférieure à 10^{-4}, ce qui n'empêche pas que la totalité des chiffres puisse être fausse), sachant que $8! > 40.000$ et que $e < 3$, il suffira de calculer 7 termes en plus du premier qui vaut 1. Si l'on cherche une précision de 10^{-6}, sachant que $10! > 3.000.000$ il suffira de calculer 9 termes en plus du premier.

Une première solution consiste à laisser à l'utilisateur le choix du nombre de termes à calculer, et le soin de vérifier par lui-même que ce nombre est adapté à ses besoins.

L'itération se réduit dans ce cas à ce qu'en notation algorithmique on peut écrire :

faire n fois l'action X .

La première chose à faire est de préciser en quoi consiste l'action X ; dans notre problème elle consistera en deux sous-actions à savoir

- calculer un terme t
- l'ajouter à la somme s

Concentrons-nous sur la première ; il s'agira de déterminer la valeur d'un entier i et celle de l'expression

$$t = \frac{x^i}{i!}$$

On remarquera que l'on peut déduire la valeur pour i de celle pour $i-1$ grâce à l'identité

$$\frac{x^i}{i!} = \frac{x}{i} \cdot \frac{x^{i-1}}{(i-1)!},$$

relation valable pour $i \geq 1$ avec les conventions $x^0 = 0! = 1$.

La sous-action de calcul d'un terme consistera ainsi à

- ajouter 1 à i
- multiplier t par $\frac{x}{i}$

Dans la seconde ligne la valeur de i est la nouvelle valeur, celle obtenue après addition de 1. Pour ne pas avoir à s'occuper dès le début de l'ordre dans lequel on effectue les modifications, on peut distinguer les anciennes valeurs des nouvelles en plaçant par exemple une barre au dessus ; on dira :

- donner à i la valeur $\overline{i} - 1$
- donner à t la valeur $\frac{x}{i} * \overline{t}$

Cette distinction disparaît au moment de l'écriture du programme, et c'est à ce moment qu'on se souciera de l'ordre.

L'action X, qui prend le nom de module itéré, ayant été détaillée, on doit se préoccuper de l'initialisation des variables.

Notre point de départ sera la somme réduite à son premier terme 1 correspondant à la valeur 0 de i.

L'initialisation se fera donc en donnant la valeur 1 à i, la valeur 1 à t et la valeur 1 à S.

3. Programmation en langage BASIC

Pour faire N fois l'action X, il suffit d'utiliser une variable auxiliaire, par exemple J, et d'utiliser l'instruction FOR ... TO ... NEXT ... sous la forme

```
[ module initial ]
FOR J=1 TO N
[ module itéré  ]
NEXT J
```

Dans notre exemple, cela donnera simplement

```
20  I=0 : T=1 : S=1
30  FOR J=1 TO N
40  I=I+1 : T=T*X/I : S=S+T
50  NEXT J
```

On laisse au lecteur d'ajouter en 10 et 50 des instructions d'entrée (de X et N) et de sortie (de S).

En réalité l'instruction FOR J=1 TO N ... NEXT J permet un peu plus ; elle assure bien que la boucle sera exécutée N fois, mais avec une variable J qui prendra les valeurs 1, 2 ... jusqu'à N. On peut utiliser la variable J dans le calcul d'une expression à l'intérieur de la boucle.

Dans notre exemple, comme dans tous les exemples de suite récurrences où l'ordre du terme intervient, on peut économiser une variable, puisque I et J prennent toujours les mêmes valeurs. On peut ainsi résumer le programme en

```
20  T=1 : S=1
30  FOR I = 1 TO N
40  T= T*X/I : S=S+T
50  NEXT I
```

En fait l'instruction

```
FOR I=1 TO N
...
NEXT I
```

traduit déjà l'itération

```
[I = 0]
faire N fois
[I = I+1
...        ]
```

Deux précautions cependant :

- Ne pas modifier la valeur de la variable compteur dans le module itéré, c'est autorisé mais dangereux pour la compréhension du programme.
- Ne pas chercher à utiliser la valeur de la variable compteur en sortie d'itération ; elle dépend de la version du BASIC.

L'instruction FOR... TO... NEXT autorise une borne de départ autre que 1, ainsi qu'un pas qui peut ne pas être 1 précisé alors par STEP... On peut utiliser un pas négatif aussi bien

```
FOR I = A TO B STEP P
...
NEXT I
```

traduit ainsi

```
[I = A-P]
faire (B-A) fois
[I = I+P
...      ]
```

3

Itérations (suite)

Nous abordons maintenant le cas d'itérations dont le nombre n'est pas connu à l'avance, en reprenant l'exemple du calcul de la fonction exponentielle.

1. Boucle à contrôle par condition de dernier élément.

Plaçons-nous, pour calculer e^x, dans le cas où x est négatif, ou bien intéressons-nous pour $x \geq 0$ à la fonction

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!}$$

Une étude mathématique montre que pour une telle série le reste est inférieur au dernier terme retenu de la somme partielle approchant la fonction. Pour obtenir une précision donnée, il convient alors d'ajouter des termes en s'arrêtant dès que l'on aura retenu un terme inférieur à la précision demandée.

En notation algorithmique, une telle itération s'écrit sous la forme

répéter l'action X jusqu'à la condition C

L'action X est ici la même que dans l'étude précédente d'itération avec contrôle par compteur. La nouveauté réside dans la présence d'une condition d'arrêt C.

Dans le cas qui nous occupe cette condition est simplement

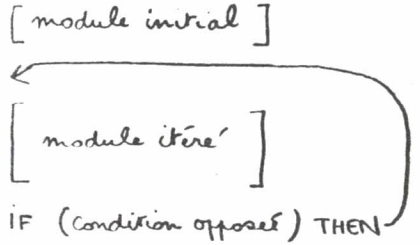
$$\text{Ts}(t) <= p$$

où p est la précision demandée.

Naturellement on ne doit pas omettre l'initialisation

2. Programmation en BASIC

On utilise un branchement en fin de boucle suivant le schéma



Dans l'exemple considéré cela donnerait par exemple

```
20  I=0:T=1:S=1
40  I=I+1:T=T*X/I:S=S+T
50  IF ABS(T)≥P THEN 40
```

On peut noter que la boucle

```
FOR I=1 TO N
...
NEXT I
```

peut-être vue comme une itération avec condition d'arrêt, à savoir

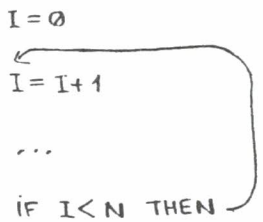
[I = 0]

répéter

[I = I+1
 ...]

jusqu'à I = N ,

ce qui se programme



3. Boucle à contrôle par condition de fin de liste.

Revenons au cas de la fonction e^x, avec des valeurs positives ou négatives. Admettons qu'une étude mathématique ait montré que le reste est inférieur à

$$\frac{x^{n+1}}{(n+1)!}\, e^x$$

Pour exploiter ce résultat il faut encore connaître une majoration de e^x par une valeur M. On pourrait en obtenir une en remplaçant x par un entier plus grand et effectuant une itération auxiliaire. Pour ne pas compliquer la discussion nous supposerons M connue sur l'intervalle de valeurs que l'on autorisera pour x.

Alors la quantité

$$\frac{x^{n+1}}{(n+1)!}\, M$$

majorant le reste n'est autre que le produit par M du premier terme du reste. Par conséquent on répètera l'action d'ajouter des termes à la somme tant que la condition

$$\text{abs}\left(\frac{x^{n+1}}{(n+1)!}\, M\right) \leq p$$

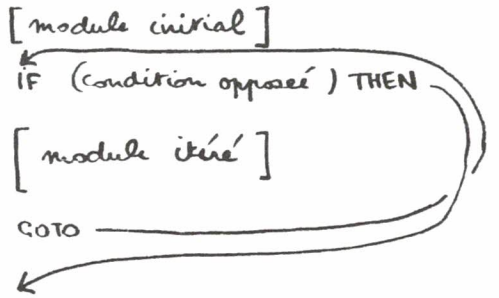
ne sera pas réalisée

En notation algorithmique, ce type d'itération s'écrit

tant que condition C faire l'action X

4. Programmation en BASIC

On utilise un branchement en début de boucle suivant le shéma

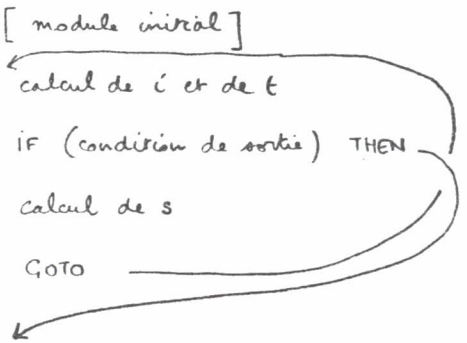


Dans notre exemple on aura ainsi

```
20  I=0 : T=1 : S=1
30  IF ABS(T*X*M/(I+1))≤ P THEN 60
40  I=I+1 : T=T*X/I : S=S+T
50  GOTO 30
60  ...
```

5. Sortie au milieu de la boucle

Dans le dernier exemple la condition d'arrêt fait intervenir $\frac{x^{n+1}}{(n+1)!}$ qui est le premier terme rejeté. En fait on pourrait fort bien ajouter ce terme à la somme, ce qui reviendrait au cas d'une boucle avec contrôle par condition de dernier élément ; imaginons donc, pour la beauté du discours, que nous soyons avares de tout calcul inutile et que nous nous interdisions d'ajouter ce terme. Notre souci d'économie nous suggère de calculer t avant de vérifier la condition d'arrêt, suivant le schéma



lequel donne

```
20  I=0 : T=1 : S=1
30  I=I+1 : T=T*X/I
40  IF ABS(T*M) ≤ P THEN 60
50  GOTO 30
60  ...
```


6. Un mot sur le langage Pascal

Une itération avec contrôle par compteur se programme à l'aide des instructions

for (variable) := (début) **to** (fin) **do** ...

alors qu'une itération avec contrôle par condition de dernier élément utilise

repeat

...

until (condition)

et une itération avec contrôle par condition de fin de liste

while (condition) **do** ...

On notera que, de la même façon que dans le modèle proposé pour la programmation, une itération avec repeat est exécuté au moins une fois, alors qu'une itération avec while peut ne pas l'être.

4

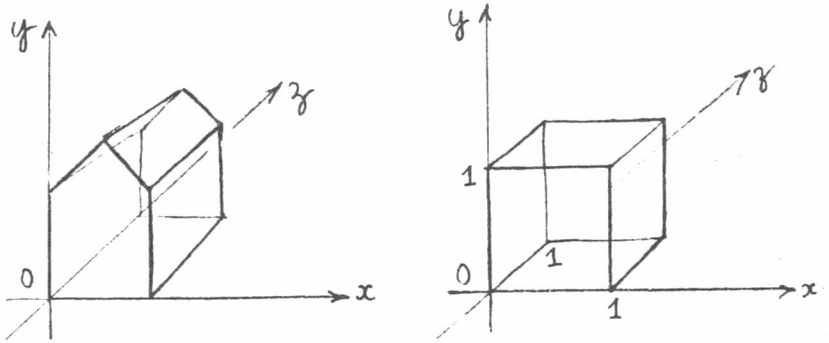
Dessin en perspective

Ce chapitre illustre le cours de géométrie projective en donnant l'occasion de manipuler les tableaux

- Position du problème ; organisation des données

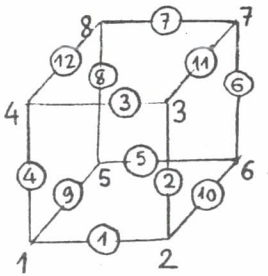
On se propose de faire préparer (ou réaliser si le matériel le permet) par l'ordinateur le dessin en perspective d'un objet à facettes, par exemple une petite habitation ou un cube, en style "fil de fer" figurant simplement les arêtes par des traits.

L'objet étant tridimensionnel, ses points sont repérés dans un système d'axes Oxyz adapté à l'objet comme suit



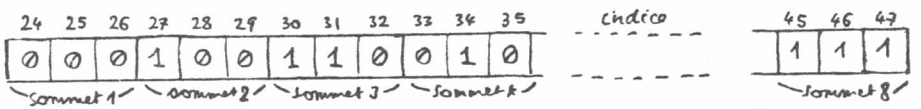
La donnée se réduira à un ensemble d'arêtes, chacune étant définie par l'ensemble des deux sommets qu'elle joint, chaque sommet étant à son tour défini par ses coordonnées dans le repère choisi.

Prenons l'exemple du cube dont les sommets ont été numérotés (de manière arbitraire) de 1 à 8 et les arêtes (de manière également arbitraire) de 1 à 12 :

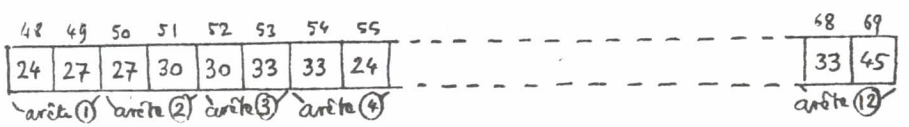


Commençons par envisager la gestion directe de la mémoire, ou du moins la situation s'en rapprochant dans laquelle on utilise un seul tableau A (comme celui des calculatrices SHARP) dont les indices 1 à 23 ont été réservés à d'autres usages.

On choisira ainsi de placer les coordonnées des sommets par groupe de 3 à partir de l'indice 24 :



Ensuite, pour définir les arêtes, on pourra placer les numéros des sommets joints (l'ordre est libre) ; cependant comme ce sont les coordonnées des sommets qui servent, on économisera des calculs en plaçant directement l'indice de la première coordonnée dans le tableau, c'est à dire 24 au lieu de 1, 27 au lieu de 2 ... (c'est encore 3 fois le numéro du sommet plus 21).



Envisageons maintenant le cas d'un BASIC standard pour lequel on travaillera autrement.

Les sommets seront définis par 3 tableaux X, Y, Z dimensionnés à 50 = 8 en perdant l'indice 0 pour plus de clarté : ainsi X(1) sera la première coordonnée du sommet numéro 1 ...

Les arêtes seront de leur côté définies par 2 tableaux S, T (de nombres entiers) dimensionnés à AR = 12 : ici S(J) sera le numéro du premier sommet de l'arête J ...

Cela donne

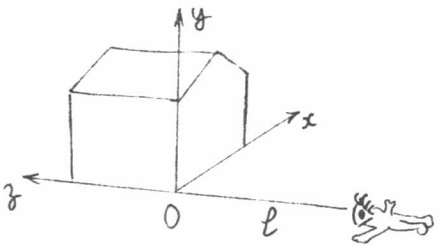
	1	2	3	4	5	6	7	8	indice du sommet
X	0	1	1	0	0	1	1	0	coordonnée x
Y	0	0	1	1	0	0	1	1	coordonnée y
Z	0	0	0	0	1	1	1	1	coordonnée z

	1	2	3	4	5	6	7	8	9	10	11	12	indice de l'arête
S	1	2	3	4	5	6	7	8	1	2	3	4	numéro du premier sommet
T	2	3	4	1	6	7	8	5	5	6	7	8	numéro du second sommet

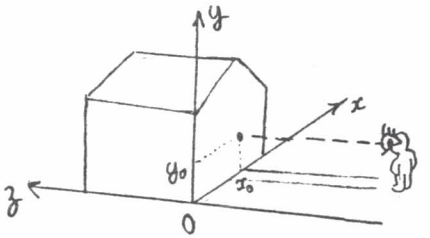
On pourrait encore, pour les sommets comme pour les arêtes, utiliser des tableaux à deux dimensions, un tableau dimensionné à 3 et 50 dans un cas et à 2 et AR dans l'autre : alors X(N, I) désignera par exemple la coordonnée numéro N du sommet numéro I ... Cette solution n'offre pas un grand intérêt parce qu'ici les trois coordonnées ne jouent pas le même rôle et que l'occasion d'une itération sur le numéro de coordonnée se présentera peu.

• Observation

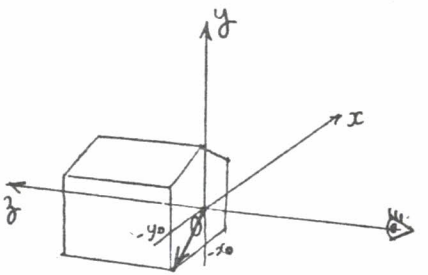
On s'intéresse à des objets suffisamment éloignés pour que la vision binoculaire ne joue pas.



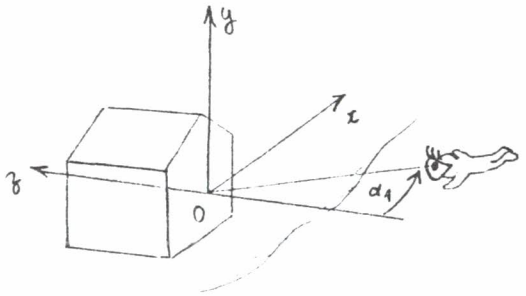
La manière la plus simple d'observer l'objet est de se placer face à lui, l'axe de la vision confondu avec l'axe Oz. Les conditions de l'observation sont alors entièrement déterminées par la distance ℓ de l'œil qui observe à l'origine O.



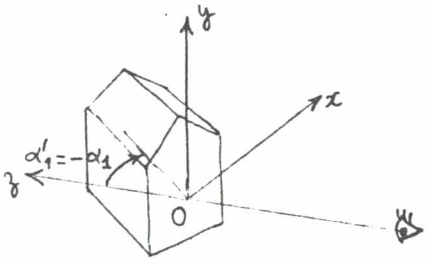
Un cas un peu plus général est celui où, la direction du regard étant la même, l'axe du regard passe par un point de coordonnées x_0, y_0, z_0 (on peut choisir $z_0 = 0$)



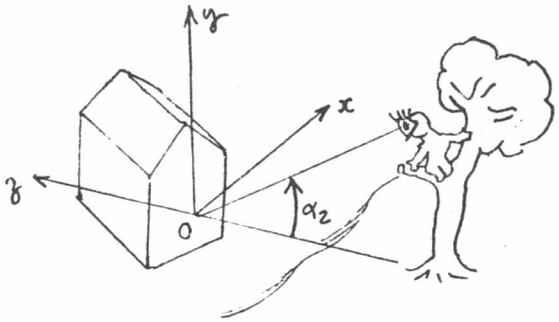
On se ramène au premier cas, c'est à dire à $x_0 = y_0 = z_0 = 0$ en translatant l'objet par rapport au système d'axes Oxyz du vecteur $(-x_0, -y_0, -z_0)$



Pour compliquer un peu, l'observateur peut, son axe de vision passant toujours par O, se déplacer dans le plan horizontal Oxz en tournant autour de l'axe Oy d'un angle α_1 dans un sens ou l'autre.

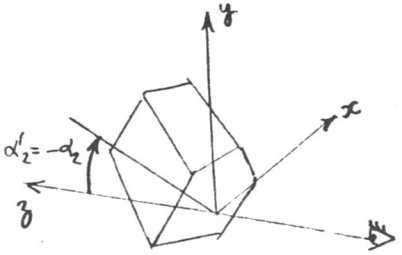


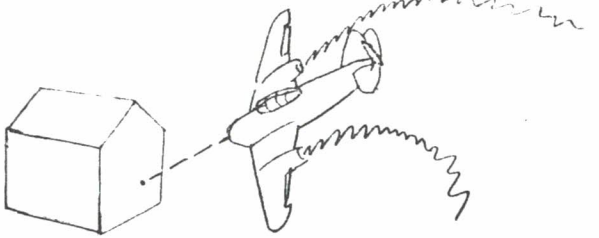
Pour se ramener au cas précédent il suffit maintenant de faire tourner l'objet autour du même axe de l'angle opposé $\alpha'_1 = -\alpha_1$.

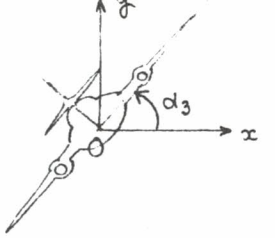


Toujours regardant O, l'observateur peut aussi s'élever (vue en plongée) ou s'abaisser (vue en contreplongée) en tournant autour de l'axe Ox d'un angle α_2.

On se ramène toujours à la situation du début par une rotation d'angle opposé $\alpha'_2 = -\alpha_2$ sur l'objet.



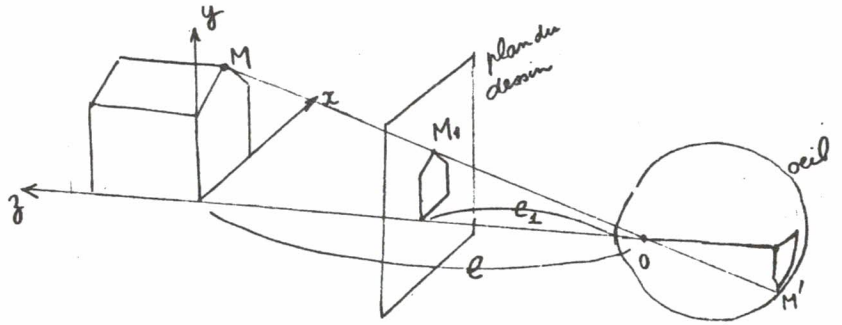




Enfin l'observateur peut lui-même s'incliner latéralement, ce qui arrive pour un avion piquant vers l'objet en amorçant un virage ; cela met en jeu une rotation d'angle α_3 par rapport à l'axe Oz. Cette situation est rare dans l'observation des éléments d'architecture.

- La perspective

Les réductions faites au paragraphe précédent nous ont ramené au cas où l'axe Oz coïncide avec l'axe de vision, l'œil étant situé à la distance ℓ de O.



Une dernière réduction va placer l'origine des axes au centre optique de l'œil (assimilé à une lentille simple ce qui ne trahit pas le raisonnement) ; cela revient à translater l'objet du vecteur $(0, 0, d)$. L'image M' au fond de la rétine est sur la droite joignant O à M.

Par définition un dessin réaliste donne l'illusion de l'original ; cela signifie qu'une fois le dessin placé correctement dans un plan perpendiculaire à l'axe de vision, le point M_1 du dessin qui représente M doit se trouver sur la droite OM de manière à avoir la même image M' au fond de l'œil.

La transformation qui fait passer du point M de coordonnées x, y, z au point M_1 de coordonnées x_1, y_1, z_1 avec $z_1 = l_1$, s'écrit simplement : l'alignement de M_1 et M avec O se traduit par la proportionalité des vecteurs $\overrightarrow{OM_1}$ et $\overrightarrow{OM}$ sous la forme

$$\overrightarrow{OM_1} = \lambda \overrightarrow{OM},$$

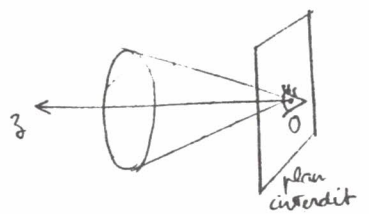
soit en composantes

$$x_1 = \lambda x$$
$$y_1 = \lambda y$$
$$l_1 = \lambda z,$$

d'où l'on tire $\lambda = \frac{l_1}{z}$ et les équations de la projection

$$\begin{cases} x_1 = l_1 \frac{x}{z} \\ y_1 = l_1 \frac{y}{z} \end{cases}$$

Bien entendu il faut que z soit non nul, autrement dit que M ne soit pas dans le plan Oxy. En pratique la vision est limitée à un cône autour de Oz qui empêche cette éventualité.



• Enchaînement des transformations

Trois rotations sont mises en jeu par la vue en perspective. Nous avons vu dans le fascicule MP1, au chapitre 9 pages 4-8 à 4-10 les équations d'une rotation d'angle α dans un plan. Effectuer une rotation par rapport à un axe dans l'espace consiste à laisser fixe la coordonnée suivant cet axe et à effectuer une rotation dans le plan des deux autres coordonnées. On peut toujours se ramener à une rotation par rapport au second axe en effectuant une permutation des axes.

Ainsi la première rotation se fait dans les axes

$Oxyz$ (autour de Oy)

la seconde dans les axes

$Ozxy$ (autour de Ox)

et la troisième dans les axes

$Oyzx$ (autour de Oz)

On passe des coordonnées x, y, z aux coordonnées z, x, y par un décalage vers la droite $\rightarrow\cdot\rightarrow$ et de la même façon de z, x, y à y, z, x et de y, z, x aux coordonnés x, y, z de départ.

Si on enchaîne une rotation d'axe Oy et d'angle α donnée par

$$x' = x\cos\alpha - z\sin\alpha$$
$$y' = y$$
$$z' = x\sin\alpha + z\cos\alpha$$

avec la permutation des axes donnée par

$$x'' = z',\ y'' = x',\ z'' = y'$$

On obtient les formules

$$\begin{cases} x'' = x\sin\alpha + z\cos\alpha \\ y'' = x\cos\alpha - z\sin\alpha \\ z'' = y \end{cases}$$

qu'il faudra appliquer trois fois avec trois angles successifs.

En fait, au lieu des angles d'Euler $\alpha_1, \alpha_2, \alpha_3$ nous travaillerons directement avec leurs opposés $\alpha'_1, \alpha'_2, \alpha'_3$ donnés dans un tableau A dimensionné à 3. Pour interpréter le signe de ces angles il faudrait régler des problèmes d'orientation. Par souci de simplicité nous passerons, laissant l'usage déterminer cette interprétation.

En résumé nous effectuerons sur les sommets

1) la translation de vecteur $(-x_0, -y_0, -z_0)$
2) l'enchaînement des rotations et permutations d'axes pour les angles $\alpha'_1, \alpha'_2, \alpha'_3$
3) la translation de vecteur $(0, 0, d)$
4) la projection avec $l_1 = 1$
5) une homothétie adéquate sur les valeurs trouvées.

Cette dernière opération s'impose car la valeur de l_1 n'intervient que par un coefficient d'échelle qui conditionne la dimension du dessin. Si on utilise un écran ou une feuille dont les dimensions sont limitées, l'idéal est que le dessin soit aussi grand que possible tout en tenant dans les limites permises.

• Le programme pour le cube

Ce qui suit est adapté aux petites calculatrices SHARP ; on suppose les valeurs correspondant aux sommets et arêtes déjà introduites dans le tableau A ; le programme détruit les coordonnées des sommets.

On récupère sur une même ligne les deux coordonnées d'un point de départ et celles d'un point d'arrivée pour l'image d'une arête. Il faudra placer et tracer dans un cadre carré $[-9, 9] \times [9, 9]$ de papier millimétré

```
introduction des données (L pour d,
D, E, F pour x0, y0, z0, et A(1).. A(3)
pour les angles)                          300 INPUT L : INPUT D, E, F :
                                              FOR I=1 TO 3 : INPUT A(I) : NEXT I
initialisation du rapport q               310 Q=0 : FOR W=24 TO 45 STEP 3
itération de
  translation (-x0, -y0, -z0)                   T=A(W)-D : U=A(W+1)-E : V=A(W+2)-
  itération de                            320 FOR N=1 TO 3
    rotation suivie de permutati                R=COS A(N) : S=SIN A(N)
    des axes                              330 Q=U : U=TR-VS : T=TS+VR : V=(
                                              NEXT N
  translation de (0,0,l)                  340 V=V+L : A(W)=T/V : A(W+1)=U/V
  projection
  q rendu plus grand que                  350 R=ABS(A(W)/9) : IF R>Q LET Q=
  x1/9 et y1/9                            360 R=ABS(A(W+1)/9) : IF R>Q LET Q=
                                          370 NEXT W
correction par homothétie                 400 FOR W=24 TO 45 STEP 3 :
de rapport 1/q                                A(W)=A(W)/Q : A(W+1)=A(W+1)/Q
                                              NEXT W
```

```
410  USING "##.#":
     FOR N= 48 TO 62 STEP 2
420  B = A(N): C = A(N+1): D = B+1.
     E = C+1
425  PRINT A(B); A(C); A(D); A(E)
430  NEXT N: END
```

affichage des coordonnées des points extrêmes de l'image d'une cible

- Améliorations possibles

Reprendre le programme précédent en BASIC standard avec un objet plus complexe et faire dessiner à l'écran le résultat, utilisant une instruction de tracé de segment.

- Pour ceux qui s'intéressent au langage Pascal

Un sommet, comme triplet de nombres réels peut se voir attribuer le type

```
array [0..2] of real
```

et de même une cible, comme couple d'entiers le type

```
array [0..1] of integer.
```

Cependant, comme nous l'avons vu, les trois coordonnées ne jouent pas le même rôle, même si elles sont toutes du même type (réel) un triplet de coordonnées pourra être représenté aussi bien par la structure record du langage Pascal, utilisant le type


```
record of
    x : real;
    y : real;
    z : real
end;
```

Pour définir la coordonnée x du sommet s on écrirait

```
with s do x. =
```

et pour utiliser cette coordonnée on écrirait simplement

```
s.x
```

(qui signifie champ x de l'enregistrement s)

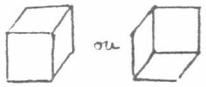
4

Dessin en perspective : suite

• L'élimination des parties cachées

On peut améliorer notablement le rendu d'un dessin en style "fil de fer" en éliminant les parties cachées, ce qui par exemple au lieu de

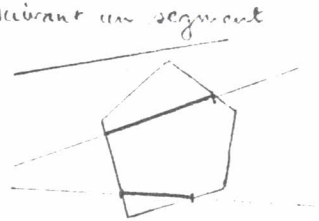
donnera



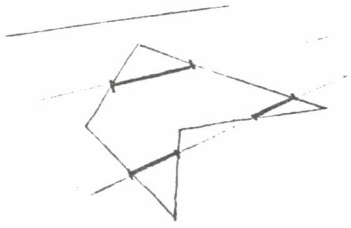
rendant plus facile l'interprétation du dessin dans un cas où la perspective est faible.

Rappelons que nous avons considéré un objet à facettes ; cet objet n'est pas seulement constitué d'arêtes mais aussi de faces opaques. Si ces dernières ne sont pas représentées, elles peuvent cacher des arêtes ou des portions d'arête.

Dans toute la suite les faces seront des polygônes convexes : un polygône est convexe si toute droite qui le rencontre le coupe suivant un segment



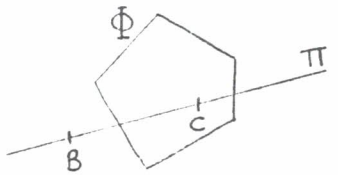
convexe

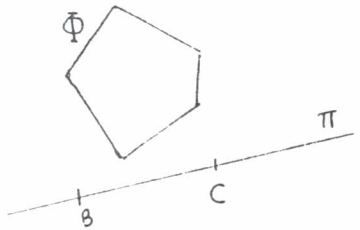


non convexe

En résumé nous ne supposons pas l'objet lui-même convexe. Dans le cas d'un objet convexe, comme le cube plein, le problème est simplifié ; une méthode consiste d'ailleurs à décomposer l'objet en objets convexes (notre choix), ce qui conduit à un algorithme peu performant en présence d'un grand nombre d'arêtes, évite les problèmes d'orientation et permet de traiter directement des exemples comme

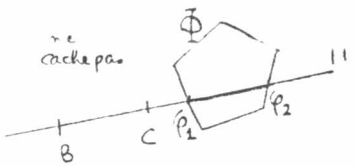
• Discussion



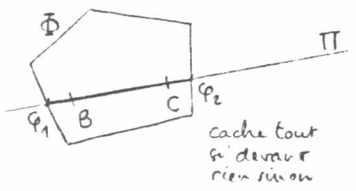


Considérons une arête BC et une face laquelle est un polygône convexe. Pour savoir si Φ cache une partie de BC il suffit de considérer les points du plan π se projetant sur la droite portant BC ; ce plan est vu en projection, c'est à dire sur le dessin, comme une droite.

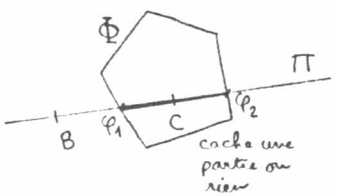
Si Φ ne rencontre pas π, évidemment la face en question ne peut rien cacher de l'arête BC.



Si Φ rencontre Π, c'est suivant un segment (φ_1, φ_2). Ce segment peut cacher ou non une partie de BC, la partie éventuellement cachée étant un segment dont les extrémités font partie des points $B, C, \varphi_1, \varphi_2$

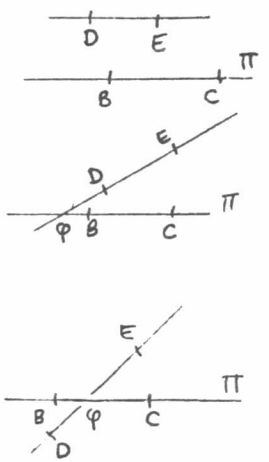


Il y a au moins un cas où l'on voit facilement que Φ ne cache rien ; c'est celui où l'un des points φ_1, φ_2 est entre B et C sur le dessin mais derrière le segment BC en réalité.



Cette discussion nous amène à chercher les intersections sur le dessin de la droite représentant Π avec les arêtes DE autres que BC.

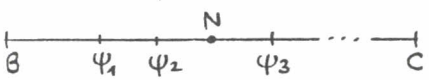
- On s'assure d'abord que la droite DE rencontre Π ; si elle est parallèle on rejette DE.
- On vérifie alors que l'intersection φ est entre D et E ; sinon on rejette
- Puis on élimine encore le cas où l'intersection φ est entre B et C mais où le point φ de l'arête DE est derrière celui de l'arête BC.



• Stratégie

Dans le cas où l'arête est acceptée on note sa référence, la position de l'intersection par rapport à BC ainsi que la profondeur du point de DE correspondant (nous verrons plus loin ce qu'il faut entendre par profondeur).

Lorsque le travail a été fait pour toutes les arêtes, on reprend les résultats (qui ont été enregistrés dans un tableau provisoire des intersections). On repère d'abord les points d'intersection φ situés entre B et C puisque c'est seulement en l'un de ces points que l'arête peut éventuellement changer de vue à caché ou l'inverse.



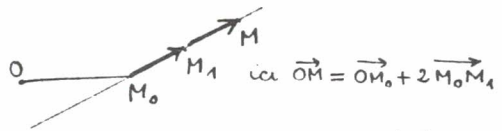
Chacun des segments $(B, \psi_1), (\psi_1, \psi_2), (\psi_2, \psi_3) \ldots$ est entièrement vu ou caché. Pour en décider il suffit de le savoir pour le milieu N.

A cette fin on reprend l'examen des faces pouvant le cacher, c'est à dire des segments (φ_1, φ_2) où φ_1, φ_2 font partie des points d'intersection φ enregistrés. Cependant il faut maintenant tous les considérer, et non seulement ceux entre B et C, en les groupant par deux correspondant à des arêtes bordant une même face et en comparant la profondeur du point N sur BC et celle du point qu'il représente sur la face, c'est à dire sur le segment (φ_1, φ_2).

• Position d'un point relativement à un segment

Soient M_0, M_1 deux points distincts du plan ou de l'espace rapporté à une origine O. On peut caractériser les points M de la droite M_0M_1 par la représentation vectorielle

$$\overrightarrow{OM} = \overrightarrow{OM_0} + \lambda \overrightarrow{M_0M_1}$$



ici $\overrightarrow{OM} = \overrightarrow{OM_0} + 2\overrightarrow{M_0M_1}$

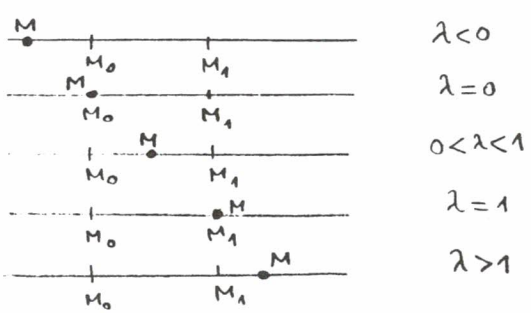
le second membre peut encore s'écrire $(1-\lambda)\overrightarrow{OM_0} + \lambda(\overrightarrow{OM_0} + \overrightarrow{M_0M_1})$

soit

$$(1-\lambda)\overrightarrow{OM_0} + \lambda\overrightarrow{OM_1}.$$

Cette représentation, de la forme $\alpha\overrightarrow{OM_0} + \beta\overrightarrow{OM_1}$ avec $\alpha+\beta=1$ est dite représentation barycentrique.

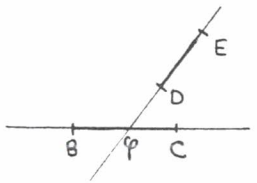
Lorsque λ est compris entre 0 et 1, le point M n'est autre que le centre de gravité des points M_0 et M_1 affectés des masses respectives $1-\lambda$ et λ. Et le fait que λ soit entre 0 et 1 caractérise les points de la droite qui sont situés entre M_0 et M_1 dans cette représentation.



La condition $0<\lambda<1$ (respectivement $0\leq\lambda\leq 1$) peut se tester par $\lambda^2<\lambda$ (respectivement $\lambda^2\leq\lambda$).

• Intersection de segments dans le plan

Soient (B,C) et (D,E) deux segments du plan. Cherchons



d'abord l'intersection des droites BC et DE, en identifiant un point de BC donné par

$$\overrightarrow{OB} + \lambda\overrightarrow{OC}$$

avec un point de DE donné par

$$\overrightarrow{OD} + \mu\overrightarrow{OE},$$

ce qui donne

$$\overrightarrow{OB} + \lambda\overrightarrow{BC} = \overrightarrow{OD} + \mu\overrightarrow{DE},$$

soit encore

$$\lambda\overrightarrow{BC} - \mu\overrightarrow{DE} = \overrightarrow{OD} - \overrightarrow{OB} = \overrightarrow{BD}.$$

En prenant des composantes pour $\overrightarrow{BC}\begin{vmatrix} p \\ q \end{vmatrix}$, $\overrightarrow{DE}\begin{vmatrix} r \\ s \end{vmatrix}$ et $\overrightarrow{BD}\begin{vmatrix} t \\ u \end{vmatrix}$, cela conduit au système d'équations linéaires

$$\begin{cases} p\lambda - r\mu = t \\ q\lambda - s\mu = u \end{cases}$$

Si on écarte le cas où le déterminant $-ps+qr$ est nul, c'est à dire le cas où les droites BC et DE sont parallèles, le système se résout (multipliant par exemple équation par s et la seconde par r et faisant la différence ...) par les formules

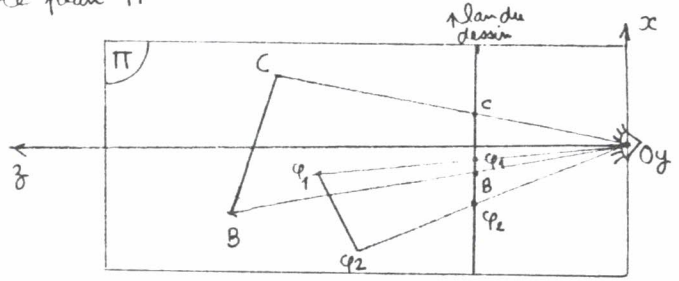
$$\lambda = \frac{ts-ur}{ps-qr}$$

$$\mu = \frac{qt-pu}{ps-qr}$$

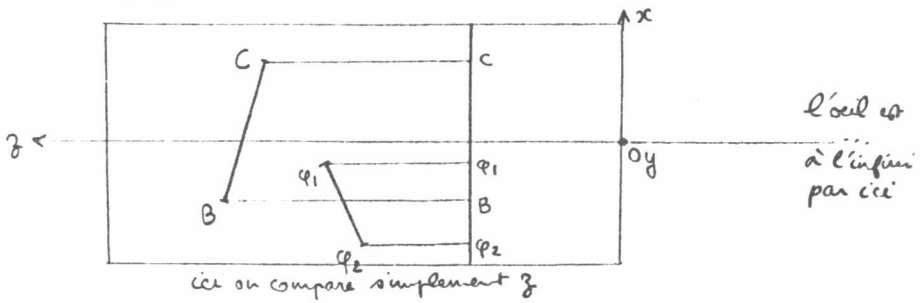
Pour savoir si φ est entre B et C d'une part ou entre D et E d'autre part il suffit de déterminer si λ ou μ est compris entre 0 et 1.

• Profondeur

Revenons trois paragraphes en arrière et plaçons-nous dans le plan Π



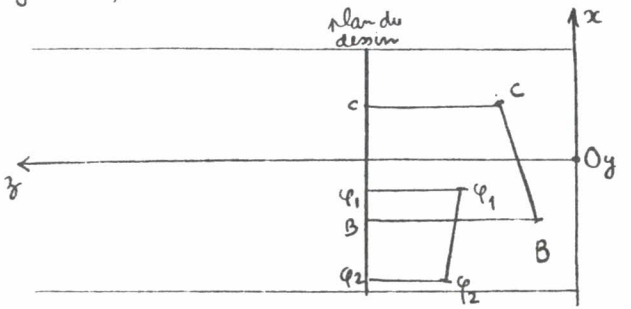
Le point φ_1 est devant le point correspondant du segment s'il est "plus proche" de l'œil. Malheureusement il est moins facile de comparer des distances à un point que de comparer simplement une coordonnée comme on le ferait dans le cas d'une projection parallèle.



C'est maintenant qu'il faut faire un peu de géométrie projective. Si nous complétons les formules de projection par

$$\begin{cases} x_1 = \dfrac{x}{z} \\ y_1 = \dfrac{y}{z} \\ z_1 = \dfrac{1}{z} \end{cases}$$

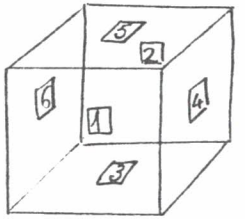
Le cours de Géométrie nous apprend que nous avons défini une transformation projective de l'espace à trois dimensions. En particulier les droites sont transformées en droites et les segments en segments. Pour obtenir le dessin il n'y a plus qu'à oublier la troisième coordonnée z, c'est à dire à projeter parallèlement à Oz. Cependant cette troisième coordonnée après transformation donne la profondeur ; plus exactement, la fonction $z \mapsto 1/z$ étant décroissante la profondeur est son opposé (laquelle est négative !)



La transformation $\frac{1}{3}$ échange plus loin et plus proche

• Ne pas perdre la face

Reprenons l'exemple du cube avec la numérotation (arbitraire) des faces



Le cube est ici fermé, mais on pourrait le transformer en boîte en enlevant une face.

Nous avons besoin de savoir si deux arêtes bordent la même face. Une première solution consiste à ajouter aux données pour chaque arête les numéros des deux faces qu'elle borde (ou par exemple 0 lorsqu'une face est absente). Il faudra en général effectuer 4 interrogations (sans compter le cas des faces absentes):

face 1 de l'arête U = face 1 de l'arête V ?
" 1 " " 2 "
" 2 " " 1 "
" 2 " " 2 "

Pour éviter ces interrogations multiples une deuxième solution consiste à se donner un tableau à deux dimensions F tel que F(J,K) = 1 ou 0 suivant que les arêtes J et K bordent la même face ou non. L'interrogation est facilitée (les faces absentes ne présentent plus de problème) mais la méthode est coûteuse en mémoire pour un tableau presque exclusivement constitué de zéros.

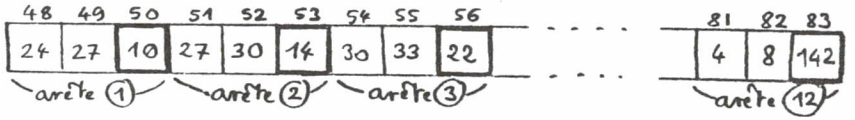
(Pour fanatiques seulement: si l'on dispose d'un BASIC effectuant les opérations logiques bit à bit sur les entiers, il est possible d'attribuer un bit à chaque face et d'ajouter aux données pour chaque arête un nombre dont les bits mis à 1 sont ceux des faces que l'arête borde. L'interrogation se résumera à comparer à zéro le résultat d'un AND bit à bit

6	5	4	3	2	1	n° de face
0	0	0	1	0	1	arête 1 (faces 1-3)
0	0	1	0	0	1	arête 2 (faces 1-4)
0	0	0	0	0	1	AND

)

Enfin pour ceux qui ne disposent que d'un BASIC simple et de peu de place en mémoire, voici une manière absurde mais amusante et relativement efficace de s'en tirer. On attribue à chaque face un nombre premier, utilisant par exemple: 2, 3, 5, 7, 11, 13 pour le cube. On ajoute aux données pour une arête le produit des nombres premiers associés aux faces qu'elle borde (avec 1 pour une face absente). L'interrogation consiste à déterminer si deux nombres ont un facteur commun, c'est à dire un PGCD différent de 1.

Cela consisterait ainsi à organiser la partie des données concernant les arêtes sous la forme



apparaissent $10 = 2\times5$ (faces 1.3), $14 = 2\times7$ (faces 1.4)...

• Le programme

Ecrit dans l'exemple du cube pour calculatrice SHARP, il suppose les données introduites et les transformations effectuées, y compris $z_1 = 1/z$, en ajoutant à la ligne 340 du programme de la semaine précédente l'instruction A(W+2) = 1/V.

Il écrit les coordonnées de deux points joints par un segment une

```
on introduit A pour tenir compte des
erreurs de calcul dans les tests;
itération sur les arêtes W de
  composantes p, q de BC
  initialisation du tableau provisoire
  des intersections
  itération sur les arêtes V de
    composantes r, s de DE et
    t, u de BD
    calcul du déterminant
    s'il n'est pas nul
      Calcul de μ
      si μ²≤ μ
        calcul de λ et de la
        différence entre la profondeur
        sur DE et celle sur BC
        si λ²≥λ et si la différence
        est ≥ 0 on complète le
        tableau des intersections
  initialisation et itération sur
  le sous-segment (μ,λ) de BC de
    recherche de l'intervalle (μ,λ) suivant
    par λ=1 et itération de
      si γ > μ et γ<λ on
      remplace λ par γ
    placement au milieu de (μ,λ);
    initialisation (Θ=0) et
    itération sur les
    intersections V de
      itération sur U jusqu'à V
      exclu de
        information sur les faces
        bordées pour les arêtes des
        intersections U et V
```

```
10  A=.001 : FOR W=48 TO 81 STEP 3
20  B=A(W): C=A(W+1): P=A(C)-A(B):
      Q=A(C+1)-A(B+1): K=81
30  FOR V=48 TO 81 STEP 3
40  D=A(V): E=A(V+1): R=A(E)-A(D):
      S=A(E+1)-A(D+1): T=A(D)-A(B):
      U=A(D+1)-A(B+1)
50  Θ=PS-QR : IF ABS Θ<A THEN 90
60  M=(QT-PU)/Θ : IF MM>M+A THEN 90
70  L=(TS-UR)/Θ : N=A(D+2)*(1-M)
      +A(E+2)*M-A(B+2)*(1-L)
      -A(C+2)*L
80  IF (LL>L-A)+(N>-A) LET K=K+3:
      A(K)=A(V+2): A(K+1)=L: A(K+2)=N
90  NEXT V: M=Θ
110 L=1 : FOR V=84 TO K STEP 3
120   G=A(V+1): IF (G>M+A)*(G<L-A)
        LET L=G
130 NEXT V
150 N=(L+M)/2 : Θ=0 : FOR V=84
      TO K STEP 3 : S=A(V+1)
160   FOR U=84 TO V-3 STEP 3:
        F=A(V) : E=A(U)
```

```
180     D=F-E*INT(F/E) : IF
          D>0 LET F=E: E=D: GOTO 180
190     IF E=1 THEN 220
200     R=A(U+1): IF (S-N)*(N-R)<A
          THEN 220
210     IF A(U+2)+A(V+2)/(S-N)*(N-R)
          >A LET Θ=1: U=V-3: V=K
220   NEXT U : NEXT V
240 IF Θ=1 THEN 270
250 D=A(B): E=A(B+1): S=E+MQ:
      T=D+LP: U=E+LQ
260 PRINT USING "#.##"; D+MP;
      S; T; U
270 IF L<1-A LET M=L : GOTO 110
280 NEXT W : END
```

recherche du PGCD par la méthode d'Euclide

si le résultat n'est pas 1 (les arêtes bordent la même face)

si le milieu est entre φ_1 et φ_2

si la différence des profondeurs est >0 alors Θ=1 (le sous-segment est caché)

si Θ n'est pas 1 (le sous-segment est vu)

on calcule les coordonnées x, y des extrémités

et on les affiche

tant que λ<1 — on remplace μ par λ

• Améliorations possibles

Reprendre le programme avec un objet plus complexe et un affichage graphique sur écran. Certaines arêtes peuvent servir de décor (dessin de porte, de fenêtre) et ne pas intervenir pour cacher d'autres éléments, tout en pouvant bien sûr elles-mêmes être cachées.

5

Intermède

Les programmes du présent chapitre concernent de simples récréations mathématiques. On y trouvera l'occasion d'illustrer une méthode algorithmique très courante de parcours d'arbre.

• Des suites particulières de nombres

Etant donné un entier positif n, on cherche les suites finies d'entiers entre 0 et n telles que chacun de ces entiers p apparaisse deux fois dans la suite en des emplacements différant de p ; le cas de 0 est particulier puisqu'il n'apparaît qu'une fois, ou en deux emplacements différant de 0, donc confondus.

Pour $n=0$ la seule suite possible est

0

et pour $n=1$ les seules possibles sont

0 1 1

et 1 1 0 .

Voici quelques autres exemples ; pour $n=2$

1 1 2 0 2 ,

pour $n=3$:

2 0 2 3 1 1 3 .

On peut aisément généraliser ces deux exemples pour en construire pour tout n, mais on veut s'aider de l'ordinateur pour les trouver toutes.

• Revenir sur ses pas

Prenons $n=4$. Les suites auront 9 termes et il s'agira d'occuper des emplacements que l'on peut numéroter

1 2 3 4 5 6 7 8 9

Pour occuper l'emplacement 1 nous avons le choix entre les chiffres de 0 à 4 ; retenons le premier, c'est à dire 0.

0 _ _ _ _ _ _ _ _
2

On ne peut plus remettre 0 et, pour occuper l'emplacement 2 on a le choix entre les chiffres de 1 à 4 ; retenons encore le premier qu'il faut alors placer aussi en 3.

0 1 1 _ _ _ _ _ _
4

Pour occuper l'emplacement 4 nous devons choisir a priori entre 2, 3 et 4 ; retenons 2 qu'il faut aussi placer en 6.

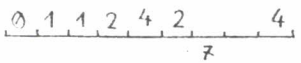
0 1 1 2 _ 2 _ _ _
5

L'emplacement 5 reste libre et nous disposons pour l'occuper de 3 et 5 ; choisissons 3 qu'il faut aussi placer en 8.

0 1 1 2 3 2 _ _ 3
7

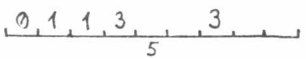
Pour l'emplacement 7 il ne reste maintenant plus que 4 et il est impossible de le placer de manière convenable ; nous sommes bloqués.

Revenons donc en arrière sur le choix pour l'emplacement précédent, c'est à dire l'emplacement 5 ; au lieu de 3 choisissons le suivant 4.

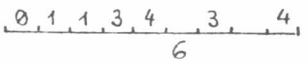


Pour l'emplacement 7 il ne reste que 3, et nous ne pourrons pas davantage aboutir. Nous avons même épuisé tous les candidats pour l'emplacement 5.

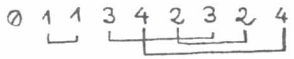
Revenons en arrière d'un cran de plus pour reconsidérer le choix fait pour l'emplacement 4. Au lieu de 2 choisissons maintenant 3.



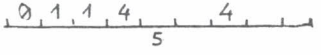
Pour l'emplacement 5 nous ne pourrons placer 2, donc passons à 4



Miracle, 2 vient maintenant se loger en 6 de manière acceptable

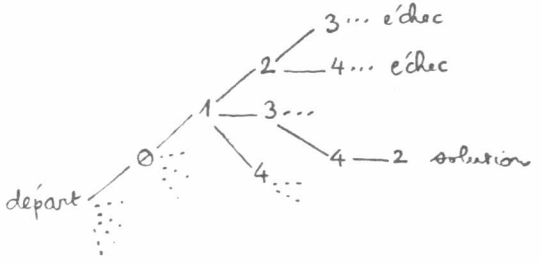


Voilà une solution. Si nous voulons les autres il faut remettre en cause le dernier choix non imposé, qui était celui pour l'emplacement 4 ; nous pouvons essayer 4.



etc...

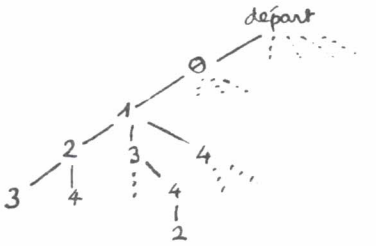
• Description à l'aide d'un arbre



On peut décrire le début de recherche que nous avons effectuée par le schéma ci-contre.

C'est ce que l'on appelle un arbre. Au départ on est à la racine de l'arbre. A chaque stade du parcours on rencontre un embranchement. La méthode consiste à explorer les branches l'une après l'autre, de manière systématique.

L'usage en informatique (pas en peinture) est de représenter les arbres avec la racine en haut, le parcours s'effectuant de gauche à droite, ce qui dans notre exemple aurait donné



• Analyse générale

A chaque stade nous sommes en présence de candidats qu'il faut essayer les uns après les autres. Le coeur du programme sera donc <u>une itération sur les candidats</u>.

Supposant l'initialisation faite par ailleurs (ne pas l'oublier), cela va conduire à écrire un noyau que nous appellerons **essais** comportant

- un test d'épuisement de la liste des candidats
  pour sortir de l'itération et effectuer un traitement ①
- un essai du candidat suivant
  pour, s'il est acceptable, effectuer un traitement ②
  et reprendre l'itération

Précisons les traitements ① et ②.

S'il n'y a plus de candidats, deux cas se présentent :

- on se trouve au stade du départ, alors le jeu est fini.
- dans le cas contraire on effectue un [retour] au stade précédent associé d'une [annulation] des placements qui avaient pu y être effectués ; on reprend les essais après le candidat dont le choix a été reconsidéré.

Sinon on essaie le candidat suivant avant de reprendre l'itération. S'il est acceptable on enregistre le choix avec [validation] des placements essayés ; deux cas se présentent alors :

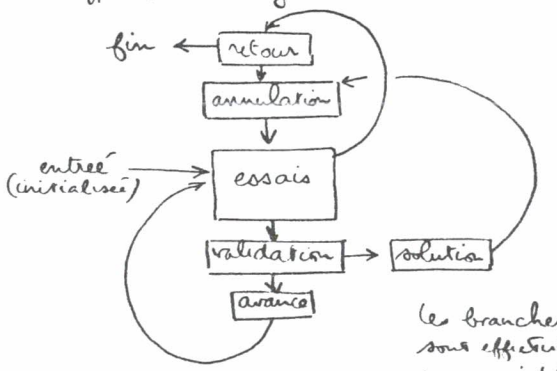
- on a une [solution] que l'on va par exemple afficher ; on procède à son annulation pour reprendre les essais afin d'en trouver d'autres.
- on n'en a pas encore une, alors on [avance] au stade suivant pour reprendre les essais initialisés au début de la liste.

Ce n'est pas une itération ordinaire puisqu'elle s'appelle elle-même. Une analyse plus satisfaisante reposerait sur la récursivité.

Dans cette démarche d'avance et de retour sur ses pas il faut garder en mémoire non seulement le dernier choix effectué mais toute la séquence de ceux qui sont encore valides à partir du départ.

Tantôt on ajoute de nouvelles informations, tantôt on en reprend et les supprime, en informatique cette structure d'information s'appelle une <u>pile</u> : ajouter est empiler et reprendre est dépiler

Il est pratique d'adopter pour la présentation de ce type d'algorithme dans un langage non récursif comme le BASIC une disposition type afin de s'y retrouver. En voici une



Les branchements principaux sont effectués vers le haut, par souci d'élégance.

- Organisation des données

Nous désignons par S un tableau d'entiers représentant la suite ; l'indice représentant l'emplacement varie de 1 à 2*N+1 : ainsi S(I) désigne le nombre occupant l'emplacement I.

La "pile" des choix sera représenté par un tableau P dont l'indice varie de 0 à n : ainsi P(K) désigne le nombre p choisi au stade K.

Pour éviter les recherches il est commode de conserver dans un tableau I le premier emplacement occupé par le nombre P, avec la convention I(P) = 0 lorsque I n'a pas été placé.

Si l'on doit gérer à la main les tableaux à l'aide d'un seul tableau A on choisira trois indices de départ et pour n ≤ 9 on remplacera par exemple S(I) par A(30+I), P(K) par A(50+K) et I(K) par A(60+K).

On marquera dans S par -1 un emplacement libre (on pourrait à la rigueur utiliser 0 car 0 peut être placé n'importe où).

• Le programme

```
10 INPUT N
20 DIM S(2*N+1): DIM P(N) : DIM I(N):
30 FOR I=1 TO 2*N+1 : S(I)=-1: NEXT I :
   I=1 : GOTO 70
```

entrée

si K=0 fin sinon retour avec dépilage de P et restitution de I

```
40 IF K=0 THEN END
50 K=K-1 : P=P(K) : I=I(P)
```

annulation

```
60 S(I)=-1 : S(I+P)=-1 : I(P)=0
```

essais si la liste est finie on retourne sinon
si P a été placé on saute
si I+P > 2*N+1 on peut abréger les essais et retourner
si la place I+P est prise on saute

```
70 P=P+1 : IF P>N THEN 40
80 IF I(P) THEN 70
90 IF I+P> 2*N+1 THEN 40
100 IF S(I+P)>=0 THEN 70
```

validation

```
110 S(I)=P: S(I+P)= P : I(P)=I :
    IF K=N THEN 150
```

avance avec empilage de P
recherche du premier emplacement vacant
initialisation des essais

```
120 P(K)= P : K= K+1
130 I=I+1 : IF S(I)>=0 THEN 130
140 P=-1 : GOTO 70
```

solution

```
150 FOR J=1 TO 2*N+1 : PRINT S(J) :
    NEXT J : GOTO 50
```

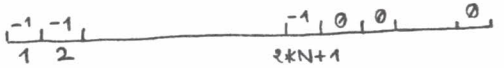

• Améliorations et variantes

Pour afficher toutes les valeurs de la suite en même temps sur une calculatrice Sharp, on peut déclarer DIM C$(0) en 10 et remplacer la ligne 150 par la suivante :

```
150 C$(0)="" : FOR J=1 TO 2*N+1 : C$(0)= C$(0)+ CHR$(48+S(J)) :
    NEXT I : PRINT C$(0) : GOTO 50
```

Pour n jusqu'à 7 compris, on construit une chaîne C$(0) (a priori limitée à 16 caractères) comprenant les chiffres S(J), lesquels interviennent par leur code ASCII 48+S(J).

On peut éviter le test I+P > 2*N+1 en 90 en plaçant des sentinelles à droite sous la forme de valeurs 0 simulant l'occupation par 0 en nombre au moins N



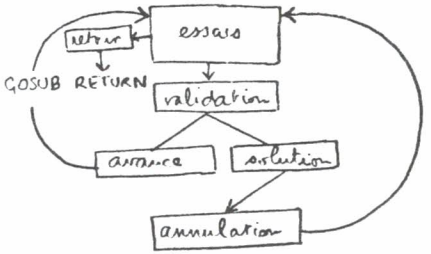
Pour cela il suffira de dimensionner S à 3*N+1 au début.

Si on renonce aux sentinelles, mais décide de ne pas placer 0 on peut supprimer la boucle de la ligne 30.

Une modification plus fondamentale consiste à ne pas chercher à occuper des places, mais à placer des nombres en essayant pour un nombre donné toutes les places possibles. Le programme sera plus simple, le tableau P devenant inutile puisque k lui-même sera placé au stade k, mais moins performant, laissant des emplacements libres disséminés ; l'écrire en exercice.

Pour l'une ou l'autre des méthodes reprendre la programmation en représentant l'itération principale comme un sous-programme qui s'appelle lui-même par GOSUB.

Voici une disposition type pour cela



- Pour ceux qui aiment le Langage Pascal.

Le langage Pascal est récursif, avec des variables locales. On peut écrire directement l'itération sans se soucier de gérer une pile de choix. Voici un exemple, pour n=4, d'application de la seconde méthode ; pour éviter le test i+p > 2*n+1 le tableau s est bordé à droite par des 0 simulation l'occupation.

```
Program Skolem;
var j : integer;
    s : array [1..13] of integer;
procedure print;
begin
  for j:=1 to 9 do write (s[k]:4);
  writeln
end;
```



```
procedure try (k: integer);
var i: integer;
begin
  for i:=1 to 9 do
  begin if s[i]=1 then if s[i+k]:=1 then
        begin s[i]:= k; s[i+k]:=k;
        if k<4 then try (k+1) else print;
        s[i]:=-1; s[i+k]:=-1 end
  end;
begin
  for j:=1 to 9 do s[j]:=-1;
  for j:=10 to 13 do s[j]:=0;
  try (0)
end.
```

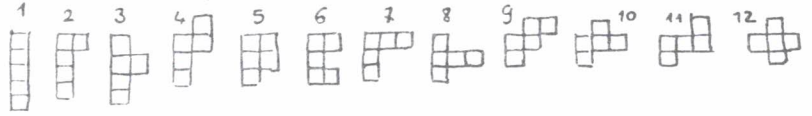
5

Intermède : suite

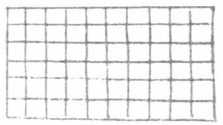
• Les pentaminos

Un vieil exemple de l'utilisation de l'ordinateur pour trouver toutes les solutions d'une récréation mathématique est celui des pentaminos. Il remplit encore les pages de publications grand public de micro-informatique.

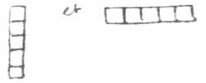
Décrivons-le rapidement. En assemblant 5 petits carrés identiques (5 d'où penta), on peut réaliser exactement 12 figures géométriques d'un seul tenant et non identifiables par déplacement ou retournement, à savoir



Le jeu dans sa version la plus courante consiste à placer ces 12 figures dans un rectangle 6×10 de 60 cases



Si on limite l'identification aux déplacements par translation chaque figure donne naissance à un certain nombre de variantes :

2 pour la figure 1 : 

8 pour la figure 2 : 

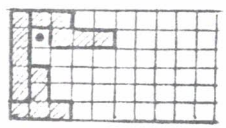
etc ...

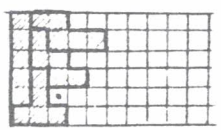
Il y a 73 variantes en tout

• Stratégie

Le principal général est toujours l'application d'une méthode d'aller et retour avec, pour un langage non récursif comme le BASIC, gestion d'une pile. Il faut cependant soigner les détails pour ne pas trop perdre en efficacité.

1) Plutôt que de considérer d'abord les figures et leur chercher successivement une place convenable, il est, comme c'est souvent le cas et comme on a pu le constater avec le jeu précédent, plus efficace de considérer d'abord les places libres et de chercher tant qu'il en reste une figure y logeant. Cela correspond à la démarche naturelle : la forme de l'espace libre découpé par le placement des premières figures guide le choix. On prendra comme principe de remplir la première case libre rencontrée lorsqu'on parcourt de haut en bas la première colonne, puis de la même façon la seconde.

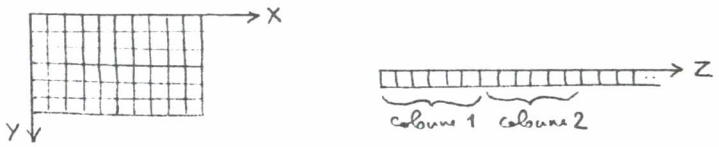




⊡ première case libre après 3 figures placées, après 4.

S'il peut arriver que le remplissage partiel enferme un groupe de cases ne permettant de loger aucune figure, cela sera décelé assez vite et l'usage d'une procédure spéciale pour l'empêcher compliquerait le programme au point d'en ralentir l'exécution plutôt que de l'accélérer.

2) Dans un problème où la situation à un moment donné est décrite par un tableau à deux dimensions (ici une pour les lignes, une pour les colonnes), on peut souvent simplifier le programme en se ramenant à un tableau à une seule dimension. Dans notre exemple on mettra à la suite les cases de la première colonne à partir de celle du haut, plus loin celles de la deuxième ...

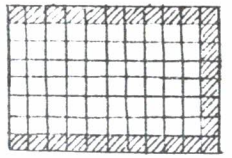


si on numérote à partir de 0
la formule est $Z = 6*X + Y$

De cette façon le parcours des cases se fait simplement de la gauche vers la droite sans qu'il soit besoin d'avancer tantôt vers le bas tantôt vers la droite.

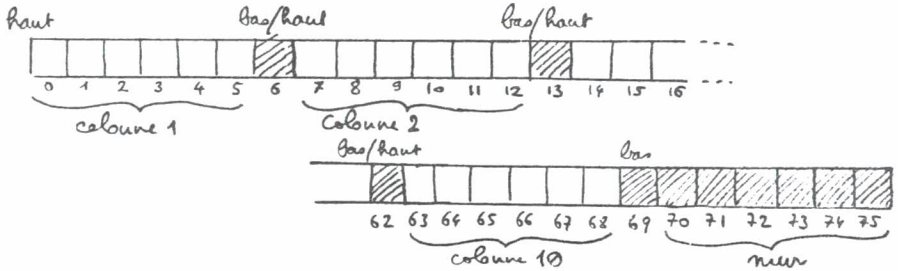
3) Une figure peut être logée si elle n'empiète pas sur celles déjà placées mais il faut encore qu'elle ne sorte pas du cadre. Plutôt que de vérifier à part cette dernière condition, on peut le faire en même temps que pour la première. Il suffit, à l'image de ce qui a été fait pour le problème précédent, d'ajouter des cases fictives

occupées autour du cadre ; c'est la méthode des sentinelles.



▨ sentinelle

Dans la description par un tableau à une seule dimension que nous avons donnée il suffit de placer une sentinelle unique entre les suites de cases correspondant à une colonne et un mur de sentinelles à la fin



L'addition de sentinelles impose de corriger la formule de traduction en $Z = 7*X + Y$

4) On peut tenir compte des symétries du cadre en décidant pour l'une des figures à 8 variantes (la 2 par exemple) de n'en conserver que 2 non déduites l'une de l'autre par une symétrie du cadre. On n'aura plus qu'une solution sur quatre, les trois autres s'en déduisant par symétrie. Cela en fait encore 2339 à trouver ! Noter que le temps de calcul est un peu diminué mais n'est pas divisé par 4.

• Organisation des données

Le cadre sera représenté par un tableau T à une dimension de $10 \times 7 + 6 = 76$ cases numérotées de 0 à 75. Les cases libres auront la valeur 0 alors qu'une case occupée aura comme valeur le numéro, de 1 à 12, de la figure qui l'occupe ou, si elle est occupée par une sentinelle, une autre valeur non nulle, par exemple -1.

On se donnera les 73 variantes sous la forme de 4 tableaux D1, D2, D3, D4 contenant les décalages entre le premier carré et chacun des 4 autres dans la représentation unidimensionnelle avec sentinelles indiquée. Rappelons que la formule de calcul du décalage sera $7 * D_X + D_Y$ où D_X est le décalage en X et D_Y le décalage en Y, ce dernier pouvant être négatif, mais le choix du premier carré étant fait de façon que le décalage total soit toujours positif. Les 4 décalages seront par exemple

1, 2, 3, 4 pour 

7, 14, 21, 28 pour 

4, 5, 6, 7 pour  (dans ce cas les valeurs sont 7_3, 7_2, 7_1 et 7_0)

L'itération porte bien sûr sur les variantes : lorsque la première case à occuper est fixée ces dernières correspondent aux différentes manières de placer une figure. Pour décider si une variante est acceptable il faut encore savoir si la figure a été déjà placée ou non ; il serait même maladroit d'essayer une variante d'une figure déjà placée. Les données doivent faire le lien entre variantes et figures.

Une manière de traiter le problème un peu indigne d'un langage évolué mais économisant les tableaux consiste à ajouter pour les tableaux D1 .. D4 un indice supplémentaire pour chaque figure avant ceux des variantes de ladite figure. On fixerait

D1 = 0 pour faire la distinction avec un déplacement,

D2 égal au numéro de 1 à 12 de la figure,

D3 égal au nombre de variantes + 1 ; c'est le nombre d'indices à ajouter pour se retrouver sur la figure suivante.

Pour éviter les recherches on ajoutera au tableau T un tableau U indiquera si la figure F a été placée ou non suivant qu'il contiendra F ou 0 à l'indice F.

De même la pile des choix, représentée par un tableau V, sera complétée par un tableau C indiquant la première case occupée au même stade et un tableau F précisant le numéro de figure de V.

• Le programme

on suppose les tableaux dimensionnés et D1, D2, D3, D4 remplis

entrée

```
10  DIM ...
    ... lecture de DATA
50  N=0:C=0:GOTO 190
```

si n=0 fin sinon retour avec dépilage

```
100 IF N=0 THEN END
110 N=N-1:V=V(N):C=C(N):F=F(N)
```

annulation

```
120 A=0:GOSUB 200:U
```

essais test de fin de liste

en cas de figure nouvelle, si elle est placée on saute

si on une case est déjà occupée on saute

```
130 V=V+1:IF V>84 THEN 100
140 IF D1(V)=0 THEN F=D2(V):IF U(F)
    THEN V=V+D3(V)
150 IF T(C+D1(V))+T(C+D2(V))+
    T(C+D3(V))+T(C+D4(V)) THEN 130
```

validation et si n=11 solution

```
160 A=F:GOSUB 200:IF N=11 THEN
    GOSUB 300:GOTO 120
```

avance avec empilage

case libre suivante

initialisation des essais

```
170 V(N)=V:C(N)=C:F(N)=F:N=N+1
180 C=C+1:IF T(C) THEN 180
190 V=-1:GOTO 130
```

Sous-programme de modification de T et U

```
200 T(C+D1(V))=A:T(C+D2(V))=A:
    T(C+D3(V))=A:T(C+D4(V))=A:
    U(F)=A
```

Sous-programme d'affichage d'une solution

```
300 FOR J=1 TO 12:FOR K=1 TO 6:
    PRINT T(I);:I=I+1:NEXT K:
    I=I+1:NEXT J:RETURN
```


• Variantes

Les calculatrices n'ont ni capacité mémoire ni vitesse suffisantes pour le jeu des pentaminos. On peut essayer celui des tetraminos

avec le cadre  .

Y-a-t-il des solutions ?

6

Echéancier

On reprend ici une partie du chapitre 5 du fascicule MP1 en choisissant une manière plus efficace de traiter le sujet. Ce sera une occasion de retrouver les arbres évoqués au chapitre précédent et d'évoquer timidement les pointeurs.

• Rappel du problème

Le possesseur d'un compte bancaire inscrit, dès qu'il en a connaissance, les prélèvements à venir sur son compte, chaque enregistrement comportant une date d'effet et un montant; par simple introduction de la date du jour, le programme doit lui indiquer les prélèvements qui auront dû être effectués et les retirer de l'échéancier.

La méthode utilisée se contentait de parcourir la liste des enregistrements en comparant la date à celle du jour, pour ne retenir que les éléments utiles, c'est à dire le plus souvent aucun ou un seul. Il fallait pour cela parcourir la liste complète; si elle comprend une vingtaine d'enregistrements, cela demandera avec un ordinateur de poche un délai d'une poignée de secondes, qui n'est pas intolérable mais qu'il serait agréable de réduire pour le confort de l'utilisateur.

• Discussion

En résumé nous disposons donc d'un ensemble d'enregistrements sur lequel nous voulons

- soit ajouter un ou plusieurs autres
- soit exploiter et retirer un ou plusieurs enregistrements caractérisés par une date inférieure à une date donnée.

Si nous pouvions déterminer rapidement l'élément de la liste pour lequel la date est la plus petite, et

- si la date est postérieure à celle du jour c'est terminé
- sinon on traite l'enregistrement et on recommence

Il serait donc avantageux de disposer d'une liste classée en fonction de la date. Malheureusement, si la liste est classée à un moment donné, les opérations prévues créent quelques difficultés.

- le retrait d'un enregistrement se fait au début de la liste; si on veut que le début de liste reste fixe, il faut décaler tout l'ensemble vers la gauche.
- l'addition d'un nouvel enregistrement se fait en un point quelconque de la liste; il faudra d'abord déterminer ce point par des comparaisons de date à partir du début, puis décaler toute la fin de liste pour faire une place au nouvel enregistrement.

On voit aussitôt que les manipulations sont plus complexes que celles que l'on a voulu éviter: ce n'est pas la solution.

• Disposition en arbre

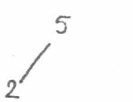
Oublions à partir de maintenant et pour toute la première partie de ce chapitre les détails du problème initial et contentons-nous de travailler des suites de nombres (les seules données), en cherchant une manière de les représenter pour faciliter les opérations :

- d'adjonction d'un élément
- de retrait du plus petit élément.

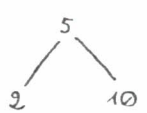
Prenons l'exemple dans lequel on se donne successivement

5, 2, 10, 3, 4, 0, 8, 1.

Plaçons d'abord 5 en ce qui sera la racine d'un arbre ;

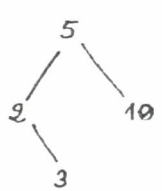


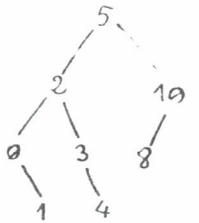
ensuite 2, qui est plus petit sera au bout d'une branche à gauche,



et 10, qui est plus grand, au bout d'une branche à droite.

Pour placer 3, nous le comparons à 5 ; comme il est plus petit nous glissons sur la branche gauche et le comparons à 2 : il est plus grand et sera donc au bout d'une nouvelle branche partant de 2 vers la droite





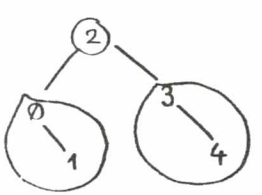
Pour ajouter 0, 8, 1 on fait de même, partant de la racine, glissant sur les branches, puis en créant une nouvelle

Il n'est pas difficile de trouver le plus petit élément : il suffit, partant de la racine, de glisser sur les branches de gauche tant que l'on peut, on tombe ici sur 0

Pour le supprimer il faut prendre une petite précaution lorsque, comme ici, il part de cet élément une branche vers la droite ; on doit alors la rattacher à gauche de son antécédent.

donne

Les arbres ainsi fabriqués ont une propriété qui les caractérisera dans une définition récursive ; plaçons-nous en un nœud quelconque et regardons ce qui en part vers la gauche d'une part et vers la droite de l'autre.



les éléments de la partie gauche sont inférieurs au nœud qui est lui-même inférieur aux éléments de la partie droite.

• Organisation des données

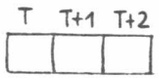
Pour chaque nœud de l'arbre nous devons réserver

- une place pour inscrire la valeur qui y est attachée,

mais nous devons aussi savoir passer de ce nœud à son éventuel fils gauche (celui qui est au bout de la branche gauche) ainsi qu'à son éventuel fils droit. Pour cela il faut également réserver

- une place pour noter où trouver les informations relatives au fils gauche
- une autre de manière analogue pour le fils droit.

Si nous disposons d'un grand tableau A pour gérer la mémoire, le plus simple est de réserver pour chaque nœud 3 indices consécutifs, le premier servant de référence pour désigner le nœud.

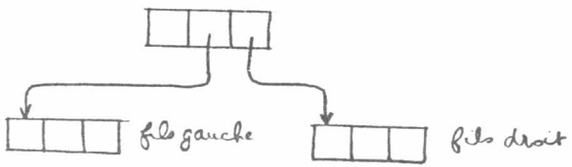


A(T) contiendra la valeur attachée au nœud T

A(T+1) contiendra l'indice du nœud fils gauche ou 0 s'il n'y en a pas.

A(T+2) contiendra celui du nœud fils droit ou 0 s'il n'y en a pas.

Pour bien marquer que A(T+1) par exemple ne contient pas directement une valeur mais désigne l'endroit où l'on peut la trouver on parle de "pointeur" et on utilise la représentation



• Constitution de l'arbre

Pour éviter des difficultés ultérieures, on suppose que la racine est déjà installée, avec une valeur fictive très grande qui ne sera jamais retirée. L'arbre aura ainsi la forme

Pour ajouter un nouvel élément, utilisant les indices U, U+1, U+2, supposant que la racine correspond elle-même à l'indice 27, il suffit de faire ce qui suit :

```
20  INPUT A(U) : V=27
40  IF A(U)<A(V) LET V=V+1 : GOTO 44
42  V=V+2
44  IF A(V) LET V=A(V) : GOTO 40
50  A(V)=U
```

partant de la racine on tère

la comparaison de A(U) et A(V) avec descente à gauche tant que l'on peut

on fait de U le fils gauche suivant le cas.

On peut remplacer les lignes 40 à 44 par la seule ligne

```
40  V=V+1+(A(U)>A(V)) : IF A(V)
      LET V=A(V) : GOTO 40
```

Pour essayer on ajoutera la ligne

```
10  T=27 : A(T)=E9 : U=30
```

et on modifiera

```
20  INPUT A(U) : V=27 : GOTO 40
30  END
50  A(V)=U : U=U+3 : GOTO 20
```

- Retrait du plus petit élément

Il faut simplement garder note de l'antécédent du noeud ; on utilise la variable S pour cela.

pointeur de la racine on itère

On descente à gauche tant que l'on peut

on affiche la valeur et on rattache le fils droit de T à son antécédent S comme fils gauche.

```
10  T=27
70  IF A(T+1) LET S=T :
        T= A(T+1): GOTO 70
90  PRINT A(T) : A(S+1)=A(T+2)
```


6

Echéancier : suite

- Retour à la gestion du compte

Dans notre problème initial nous avions non seulement à considérer des dates, mais aussi des montants. Cela nous conduit à modifier l'organisation des informations relatives à un noeud de l'arbre.

Nous placerons les pointeurs d'abord ; pour économiser de la place, ils seront tous les deux à l'indice T, la valeur de A(T) étant donnée par

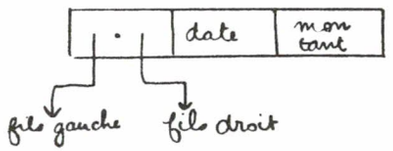
$$\text{pointeur gauche} + \frac{\text{pointeur droit}}{1000}$$

On obtiendra le pointeur gauche en prenant la partie entière, et le pointeur droit en prenant 1000 fois la partie fractionnaire.

A l'indice T+1 nous plaçons la donnée classée, qui est la date, elle même codée sous la forme

$$\text{n° du mois} + \frac{\text{jour du mois}}{100}$$

Enfin à l'indice T+2 sera placé le montant du prélèvement

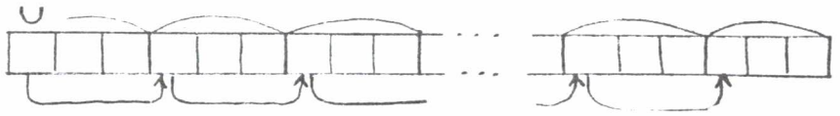


• Gestion des places libres

La solution consistant à utiliser un indice U qui augmente de 3 chaque fois que l'on a besoin d'ajouter un enregistrement ne permet pas de travailler longtemps. On pourrait bien sûr partir du début et chercher le premier emplacement libre comme cela a été proposé dans le fascicule MP1

Cependant on peut faire mieux en liant les places libres les unes aux autres en une liste, toutes, sauf celle qui sera la dernière, contenant l'indice de celle qui sera sa suivante.

En départ on aura la disposition



Chaque fois que l'on aura inscrit un nouvel enregistrement, utilisant en pratique A(U+1) et A(U+2), on actualisera U en lui donnant la valeur A(U) qui sera lui-même annulé

A l'inverse, si on retire de l'arbre l'enregistrement T, on rattache cette nouvelle place au début de la liste précédente par A(T)=U et on actualise U par U=T.

Compte tenu de ces modifications l'adjonction d'un enregistrement donnera lieu à la fraction de programme suivante


```
10  U = 27 : M = 3
15  FOR V = 27 TO 84 STEP 3 :
    A(V) = V + 3 : NEXT V
20  V = A(U) : A(U) = 0 : U = V :
25  INPUT A(U+1), A(U+2) : V = 27 : GOTO 35
30  END
35  P = A(V) : IF A(U+1) > A(V+1)
    THEN 50
40  IF P > 1 LET V = INT P : GOTO 35
45  A(V) = P + U : GOTO 20
50  IF P > INT P LET V = M*(P - INT P) :
    GOTO 35
55  A(V) = P + U/M : GOTO 20
```

nouvelle place libre partant de la racine itération de

tant qu'on le peut descente à gauche ...

... ou à droite

L'itération qui apparaît ici se présente de manière un peu différente des habitudes. Il y a deux possibilités pour la sortie qui donnent lieu à des traitements légèrement différents. On pourrait bien sûr les unifier.

• Retrait du plus petit élément : suite

La méthode que nous avons esquissée au début du chapitre consistait à partir de la racine. Cela est nécessaire au début mais ensuite, lorsqu'on a supprimé l'enregistrement T, il suffit de repartir de son antécédent, et cela même si l'on a ajouté de nouveaux enregistrements dans l'intervalle.

L'ennui est que si l'on doit supprimer l'antécédent en question il faut connaître son propre antécédent et ainsi de suite... Il faut donc, lorsqu'on descend suivant les branches de gauche, garder en mémoire tous les noeuds rencontrés. Cela peut se faire à l'aide d'une pile, pour laquelle nous réservons les premiers indices du tableau A.

Si nous enchaînons le retrait jusqu'à la date Q, cela donne lieu à

```
10  T=27 : A(T+1)=&9 : S=1 .
    M=&3
60  INPUT Q
70  IF A(T)>1 LET A(S)=T : S=S+1 :
    T=INT A(T) : GOTO 70
80  IF A(T+1)>Q THEN END
90  PRINT A(T+1), A(T+2) :
    R=M*A(T)-T :
    A(T)=U : U=T :
    S=S-1 : T=A(S) : A(T)=A(T)+R :
    GOTO 70
```

tant que l'on peut descendre à gauche avec empilage de la référence des noeuds

tant que la date ne dépasse pas Q on enchaîne avec les opérations de :

recherche du fils droit de T,

rattachement de T à la liste des places libres,
dépilage pour connaître l'antécédent et lui rattacher le fils droit, cet antécédent est la nouvelle valeur de T