Jean Berstel
Jean-Éric Pin
Michel Pocchiola

MATHÉMATIQUES ET INFORMATIQUE

Problèmes résolus

ALGÈBRE, COMBINATOIRE, ARITHMÉTIQUE

Ce livre a été publié par McGraw-Hill France en 1991

Version 15 janvier 2005

Avant-propos

Depuis quelques années, l'introduction d'une épreuve d'informatique aux concours d'entrée dans les grandes écoles scientifiques a créé un intérêt accru pour des exemples d'algorithmes et de programmes qui s'inspirent du programme des classes préparatoires. Ce livre s'inscrit dans ce cadre. Partant de sujets posés aux concours d'entrée à l'ENS d'Ulm (mais les thèmes abordés se retrouvent, dans une forme voisine, dans d'autres concours), nous proposons à la fois des solutions aux questions mathématiques et des programmes types pour les questions algorithmiques. Les réponses sont, la plupart du temps, accompagnées de développements qui permettent de replacer le problème dans son cadre mathématique et le programme dans un contexte informatique. Dans cette forme élargie, le livre intéresse également les étudiants des premier et deuxième cycles universitaires en mathématiques et en informatique; il est aussi un complément pour la préparation de l'option d'informatique de l'agrégation.

Les thèmes

L'apparition de l'informatique et de la programmation dans les classes préparatoires fait découvrir à un grand nombre d'étudiants leurs nombreuses applications possibles aux autres sciences. Cette utilisation ne se limite pas au seul calcul numérique car l'informatique permet aussi de formuler en termes algorithmiques et programmables des théories mathématiques, des phénomènes physiques, chimiques ou biologiques dont l'étudiant n'avait naguère qu'une connaissance théorique. Le présent livre permettra d'aborder cette découverte, en ce qui concerne l'emploi de la programmation pour l'étude de problèmes mathématiques.

Les thèmes abordés concernent principalement l'algèbre, la géométrie, l'arithmétique et la combinatoire. On traite des sujets traditionnels d'algèbre linaire, mais aussi des thèmes moins habituels comme les pseudo-inverses de matrices rectangulaires, les conditions de trigonalisation simultanée d'un ensemble de matrices, les matrices unimodulaires et les matrices irréductibles. Une section est consacrée aux polynômes : recherche des zéros par les suites de Sturm, polynômes symétriques et factorisation des polynômes à coefficients dans le corps à deux éléments. Nous avons choisi d'accorder une place importante aux problèmes combinatoires parce qu'ils fournissent, d'une part, d'excellents exemples de programmation et, d'autre part, soulèvent des questions mathématiques intéressantes et parfois ardues. Parmi les problèmes combinatoires classiques, nous traitons les nombres de Bernoulli et les partitions d'entiers. Un long chapitre est consacré à la combinatoire des mots, qui a d'étroits rapports avec la logique mathématique, la théorie combinatoire des groupes, l'arithmétique et l'informatique. La géométrie algorithmique est une discipline en plein essor. Nous abordons deux problèmes typiques de cette nouvelle théorie : la triangulation de Delaunay et le problème de la «galerie d'art». Tout particulièrement dans ce chapitre, les algorithmes décrits ne sont pas les meilleurs connus. La mise en œuvre des algorithmes optimaux dépasse très largement le cadre de ce livre. La dernière partie traite de problèmes arithmétiques. Il s'agit, d'une part, de la factorisation des entiers de Gauss et, d'autre part, de la manipulation de grands entiers ou de grands réels. L'arithmétique modulaire est une première technique mais elle ne permet pas la division. Le dernier chapitre est consacré à l'arithmétique positionnelle. On y détaille le calcul sur les grands entiers et les grands réels et on présente plusieurs algorithmes de calcul des décimales de π.

Les solutions

Les thèmes sont organisés sous forme de triptyques. La première partie présente un énoncé, proche de la forme où il a été donné au concours d'entrée à l'ENS (mais les thèmes abordés se retrouvent, comme nous l'avons déjà dit, dans une forme voisine dans d'autres concours). L'énoncé comporte à la fois des questions mathématiques et des questions de programmation. La deuxième partie du triptyque donne une solution aux questions mathématiques. Plutôt que de reprendre une à une les questions de l'énoncé, cette partie est l'occasion de présenter des résultats généraux sur le thème; les solutions aux questions y figurent soit explicitement, soit en filigrane. En général, les questions mathématiques préparent aux questions informatiques, dans la mesure où elles permettent d'élaborer ou de justifier le développement des algorithmes.

Les programmes

La troisième partie d'un triptyque contient les procédures ou fonctions qui répondent aux questions de programmation. Là aussi, on conçoit la réponse dans un cadre plus large : la plupart du temps, on s'appuie sur des bibliothèques de procédures dont l'intérêt peut dépasser le sujet et qui constituent le pendant des digressions mathématiques. L'idée qui nous conduit dans cette approche est que la solution aux questions informatiques est en général facile à concevoir si l'on dispose d'un «outillage» approprié. Si, dans quelques années, des systèmes de calcul formel remplaçaient Pascal comme support informatique, bon nombre des bibliothèques disparaîtraient, mais l'essentiel des programmes resterait. Chaque thème est complété par de brèves notes bibliographiques dont le but est d'aider le lecteur intéressé à approfondir le sujet.

Les dialectes de Pascal

Bien que le langage de programmation Pascal soit codifié par une norme internationale, de petites variations subsistent d'un compilateur à l'autre, d'une machine à l'autre. Les programmes présentés dans ce livre sont prévus pour fonctionner en Turbo Pascal sur des ordinateurs compatibles PC et en Think Pascal sur des ordinateurs Macintosh. Nous avons retenu la version 4 pour Turbo Pascal et la version 2 pour Think Pascal ; dans les deux cas, il s'agit d'un compromis entre ce qui est disponible sur le marché et ce qui est disponible dans les salles de travaux pratiques. Les différences entre les deux compilateurs sont faibles et sont reportées, dans la mesure où elles sont pertinentes pour nos programmes, dans l'annexe A.

Le langage de programmation

L'enseignement de la programmation dans les classes préparatoires constitue une introduction à l'utilisation des ordinateurs pour la résolution de problèmes scientifiques. Dans cette optique, l'apprentissage de l'informatique est limité à un petit nombre de concepts, et la pratique de la programmation s'appuie sur un sous-ensemble assez limité du langage Pascal. Ce sous-ensemble comprend les types de base booléen, entier, réel et, comme type composé, les tableaux. Sont en particulier exclus les pointeurs et les enregistrements, les procédures et fonctions récursives. Les déclarations de procédures et fonctions se limitent au niveau du programme principal (pas de déclaration à l'intérieur d'une procédure). Dans la mesure où il ne s'agit pas d'enseigner l'informatique, mais seulement de familiariser les élèves avec l'usage des ordinateurs, les considérations sur les structures de données ainsi que sur la complexité ou la validité des algorithmes mis en œuvre ne figurent pas au programme.

Nous nous efforçons, dans ce livre, de respecter ces contraintes. Dans certaines situations, nous donnons, en plus des programmes itératifs, des procédures ou fonctions récursives, pour montrer combien cette écriture peut être confortable. Parfois, nous déclarons aussi des procédures locales à d'autres procédures ou fonctions, dans le but d'en diminuer le nombre de paramètres. En revanche, nous avons systématiquement respecté les limitations sur les données structurées.

Les restrictions sur Pascal dans le programme des classes préparatoires sont par ailleurs fort intéressantes du point de vue pédagogique, puisqu'elles mettent à jour les contorsions auxquelles on doit se livrer pour représenter des données structurées. Par exemple, un polynôme à coefficients réels est naturellement représenté par un *couple* formé d'un entier qui est le degré du polynôme et d'un tableau de réels. La notion de couple (`record`) ne figurant pas au programme, le degré doit être intégré dans le tableau des réels. Ce genre de phénomènes a d'ailleurs contribué au développement de langages de programmation plus évolués. Le lecteur averti pourra sans peine introduire, à de nombreux endroits, des structures de données plus appropriées.

L'efficacité

Quant à la complexité des programmes, on peut faire plusieurs remarques. Les programmes que nous donnons ne sont pas les plus rapides possibles, loin s'en faut. Notre souci principal étant la clarté et la structuration, nos programmes comportent de nombreux appels de procédures ou de fonctions là où une affectation simple aurait fait gagner beaucoup de temps, au détriment de la clarté. De même, nous n'avons pas essayé de faire des économies de place et nous n'avons pas écrit de programmes qui, au prix d'une complication supplémentaire, permettent à plusieurs données de se partager une même place. Un programmeur astucieux peut, dans la plupart de nos programmes, diviser le temps d'exécution par trois, voire plus, et la place requise par deux. Si nous gaspillons aussi généreusement les deux ressources principales, le temps et la place, c'est qu'avec les matériels modernes, elles ne nous sont pas comptées, aussi longtemps que l'on réalise des exercices de programmation. L'expérience montre que le programme le plus inefficace a un temps d'exécution imperceptible, par rapport au temps pris par exemple par la frappe des coefficients d'une matrice d'ordre 4 et par l'impression des résultats. Très rares sont les cas (les programmes de géométrie algorithmique, ceux calculant les décimales de π ou encore celui qui calcule la base de l'algèbre de Lie en sont) où on attend vraiment le résultat et où on cherche donc à améliorer l'efficacité. On s'aperçoit alors que l'élaboration de programmes efficaces passe par la conception d'algorithmes efficaces qui, eux, ne sont faciles ni à trouver ni à prouver.

Mode d'emploi

Le livre est divisé en deux volumes. Le premier volume regroupe les sujets d'algèbre et le second volume ceux de combinatoire, géométrie et arithmétique. La numérotation des énoncés indique dans l'ordre le numéro de chapitre, le numéro de section et le numéro d'énoncé. Par exemple le théorème 5.1.3 est le troisième énoncé de la section 1 du chapitre 5. Nous conseillons au lecteur de chercher à résoudre le problème par ses propres moyens avant de consulter la solution. Afin d'éviter les duplications inutiles, la plupart des programmes figurant dans ce livre font appel à des bibliothèques qui sont fournies en annexe. Bien entendu, pour faire tourner les programmes figurant dans le livre, il suffit de recopier les parties de la bibliothèque réellement utilisées par le programme.

Remerciements

Nous avons bénéficié de l'aide, de conseils, de discussions et de remarques de beaucoup de personnes, à des titres divers. Nous tenons à remercier Jean-Paul Allouche, Marie-Pierre Béal, Danièle Beauquier, Bruno, Luc Boasson, Pierre Cartier, Maxime Crochemore, Marie-Thérèse Da Silva, Dominique Perrin, Christophe Reutenauer, Jean-Jacques Risler, Mariette Yvinec. Nous remercions Claude Puech et Jean-Marie Rifflet pour leur intérêt constant, et Yves Tremblay et Claire-Marie La Sade pour l'accueil qu'ils nous ont réservé et leur soutien logistique compétent.

Version 15 janvier 2005

Table des matières

Partie I

Algèbre linéaire

Chapitre 1

Calcul matriciel

Ce chapitre présente les algorithmes de base pour le calcul matriciel, en particulier les méthodes de Gauss et de Jordan pour la résolution de systèmes d'équations linéaires. Le calcul d'un déterminant et de l'inverse d'une matrice sont traités. La détermination du rang d'une matrice est exposée en détail; en effet, ce calcul est important puisqu'il équivaut à la détermination de la dimension de l'espace vectoriel engendré par un ensemble donné de vecteurs.

Le premier paragraphe décrit un certain nombre de procédures de base qui constituent, avec les procédures des sections suivantes, une bibliothèque utile de manipulation de matrices. Sur la façon d'employer une bibliothèque de procédures, voir l'annexe A.

1.1 Une bibliothèque

Nous commençons par la présentation d'une bibliothèque de manipulation de matrices. Quelques-unes de ces procédures sont faciles, d'autres seront présentées plus en détail dans la suite de ce chapitre.

Nous déclarons, sans surprise, le type des vecteurs et matrices par :

```
CONST
   OrdreMax = 12;                    L'ordre maximal des matrices et vecteurs.
TYPE
   vec = ARRAY[1..OrdreMax] OF real;
   mat = ARRAY[1..OrdreMax] OF vec;
   vecE = ARRAY[1..OrdreMax] OF integer;
VAR
   MatriceUnite: mat;                Matrice prédéfinie.
   MatriceNulle: mat;                Matrice prédéfinie.
```

En vue d'alléger les notations, on utilisera des procédures distinctes pour les matrices carrées et pour les matrices rectangulaires. En effet, considérons par exemple une procédure de multiplication de matrices. Si les matrices sont carrées, il suffit de passer en paramètre l'ordre commun des matrices, alors qu'il faut trois paramètres dans le cas de matrices rectangulaires.

Voici les en-têtes des procédures ou fonctions qui constituent la bibliothèque. Il y a trois familles de procédures : les procédures d'entrées-sorties, un groupe de procédures ancillaires, comme des copies de lignes et colonnes et des procédures réalisant des opérations simples (multiplication, transposition, etc.) ou plus élaborées, comme la résolution d'un système d'équations linéaires et le calcul de l'inverse.

Dans certaines procédures, un paramètre appelé `titre` permet d'afficher un texte avant la lecture ou l'impression; ceci facilite grandement l'interprétation des résultats. Le type `texte` doit être défini au préalable, sa définition dépend de la syntaxe du compilateur. En Turbo Pascal comme en Think Pascal, cela peut être `string[80]`. L'initialisation des deux matrices prédéfinies se fait dans la procédure :

`PROCEDURE InitMatrices;`
Définition de la matrice unité et de la matrice nulle.

Un certain nombre de procédures sont employées pour les entrées et sorties de matrices et de vecteurs :

`PROCEDURE EntrerMatrice (n: integer; VAR a: mat; titre: texte);`
Affichage du titre, puis lecture de la matrice a carrée d'ordre n.
`PROCEDURE EntrerMatriceRect (m, n: integer; VAR a: mat; titre: texte);`
Affichage du titre, puis lecture de la matrice a d'ordre (m, n).
`PROCEDURE EntrerVecteur (n: integer; VAR a: vec; titre: texte);`
Affichage du titre, puis lecture du vecteur a d'ordre n.
`PROCEDURE EntrerVecteurE (n: integer; VAR a: vecE; titre: texte);`
Affichage du titre, puis lecture du vecteur entier a d'ordre n.
`PROCEDURE EntrerMatriceSymetrique (n: integer; VAR a: mat; titre: texte);`
Affichage du titre, puis lecture de la partie triangulaire inférieure de la matrice symétrique a d'ordre n. La partie supérieure de a est complétée à la lecture.

`PROCEDURE EcrireMatrice (n: integer; VAR a: mat; titre: texte);`
Affichage du titre, puis de la matrice a d'ordre n.
`PROCEDURE EcrireMatriceRect (m, n: integer; VAR a: mat; titre: texte);`
Affichage du titre, puis de la matrice a d'ordre (m, n).
`PROCEDURE EcrireVecteur (n: integer; VAR a: vec; titre: texte);`
Affichage du titre, puis du vecteur a d'ordre n.
`PROCEDURE EcrireVecteurE (n: integer; VAR a: vecE; titre: texte);`
Affichage du titre, puis du vecteur entier a d'ordre n.

Les opérations simples se font à l'aide de procédures dont voici les en-têtes :

`PROCEDURE MatricePlusMatrice (n: integer; a, b: mat; VAR ab: mat);`
Somme des deux matrices a et b d'ordre n. Résultat dans la matrice ab.
`PROCEDURE MatriceMoinsMatrice (n: integer; a, b: mat; VAR ab: mat);`
Différence des deux matrices a et b d'ordre n. Résultat dans la matrice ab.

```
PROCEDURE MatriceParMatrice (n: integer; a, b: mat; VAR ab: mat);
```
Produit des deux matrices a et b d'ordre n. Résultat dans la matrice ab.
```
PROCEDURE MatriceParMatriceRect (m, n, p: integer; a, b: mat; VAR ab: mat);
```
Produit de la matrice a d'ordre (m, n) par la matrice b d'ordre (n, p). Résultat dans la matrice ab, d'ordre (m, p).
```
PROCEDURE MatriceParVecteur (n: integer; a: mat; x: vec; VAR ax: vec);
```
Produit de la matrice a d'ordre n par le vecteur x. Résultat dans le vecteur ax.
```
PROCEDURE MatriceRectParVecteur (m, n: integer; a: mat; x: vec; VAR ax: vec);
```
Produit de la matrice a d'ordre (m, n) par le vecteur x. Résultat dans le vecteur ax.
```
PROCEDURE VecteurParMatrice (n: integer; x: vec; a: mat; VAR xa: vec);
```
Produit du vecteur ligne x par la matrice a d'ordre n. Résultat dans le vecteur xa.
```
PROCEDURE VecteurPlusVecteur (n: integer; a, b: vec; VAR ab: vec);
```
Somme des deux vecteurs a et b d'ordre n. Résultat dans le vecteur ab.
```
PROCEDURE VecteurMoinsVecteur (n: integer; a, b: vec; VAR ab: vec);
```
Différence des deux vecteurs a et b d'ordre n. Résultat dans le vecteur ab.
```
PROCEDURE VecteurParScalaire (n: integer; x: vec; s: real; VAR sx: vec);
```
Produit du vecteur x d'ordre n par le réel s. Résultat dans le vecteur sx.

D'autres opérations élémentaires sont décrites dans les procédures :

```
PROCEDURE Transposer (n: integer; a: mat; VAR ta: mat);
```
La matrice ta contient la transposée de la matrice a d'ordre n.
```
PROCEDURE TransposerRect (m, n: integer; a: mat; VAR ta: mat);
```
La matrice ta d'ordre (n, m) contient la transposée de la matrice a d'ordre (m, n).
```
FUNCTION ProduitScalaire (n: integer; VAR a, b: vec): real;
```
Donne le produit scalaire des deux vecteurs a et b d'ordre n.
```
FUNCTION Norme (n: integer; VAR a: vec): real;
```
Il s'agit de la norme euclidienne.
```
FUNCTION NormeInfinie (n: integer; VAR a: vec): real;
```
C'est le maximum des valeurs absolues.

Les procédures que voici sont plus difficiles à réaliser (sauf la première); leur écriture sera détaillée dans les sections suivantes :

```
PROCEDURE SystemeTriangulaireSuperieur (n: integer; a: mat; b: vec;
   VAR x: vec);
```
Solution d'un système d'équations triangulaire supérieur. La matrice est supposée inversible.
```
PROCEDURE SystemeParGauss (n: integer; a: mat; b: vec; VAR x: vec;
   VAR inversible: boolean);
```
Solution d'un système d'équations par la méthode de Gauss.
```
PROCEDURE InverseParGauss (n: integer; a: mat; VAR ia: mat;
   VAR inversible: boolean);
```
Calcule la matrice inverse ia de a par la méthode de Gauss, si l'inverse existe.
```
PROCEDURE SystemeParJordan (n: integer; a: mat; b: vec; VAR x: vec;
   VAR inversible: boolean);
```
Solution d'un système d'équations par la méthode de Jordan.

```
PROCEDURE InverseParJordan (n: integer; a: mat; VAR ia: mat;
   VAR inversible: boolean);
```
Calcule la matrice inverse ia de a par la méthode de Jordan, si l'inverse existe.

```
FUNCTION Determinant (n: integer; a: mat): real;
```
Calcule le déterminant par la méthode de Gauss.

Nous donnons maintenant la réalisation de certaines procédures.

```
PROCEDURE InitMatrices;
```
Définition de la matrice unité et de la matrice nulle.

```
   VAR
      i, j: integer;
   BEGIN
      FOR i := 1 TO OrdreMax DO                Définition de la matrice nulle.
         FOR j := 1 TO OrdreMax DO
            MatriceNulle[i, j] := 0;
      MatriceUnite := MatriceNulle;            Définition de la matrice unité.
      FOR i := 1 TO OrdreMax DO
         MatriceUnite[i, i] := 1;
   END; { de "InitMatrices" }
```

```
PROCEDURE EntrerMatrice (n: integer; VAR a: mat; titre: texte);
```
Affichage du titre, puis lecture de la matrice a carrée d'ordre n.

```
   VAR
      i, j: integer;
   BEGIN
      writeln;
      writeln(titre);
      FOR i := 1 TO n DO
         BEGIN
            write('Ligne ', i : 1, ' :  ');
            FOR j := 1 TO n DO
               read(a[i, j]);
         END;
      readln
   END; { de "EntrerMatrice" }
```

Voici une variante de la procédure précédente :

```
PROCEDURE EntrerMatriceSymetrique (n: integer; VAR a: mat; titre: texte);
```
Affichage du titre, puis lecture de la partie triangulaire inférieure de la matrice symétrique a d'ordre n. La partie supérieure de a est complétée à la lecture.

```
   VAR
      i, j: integer;
   BEGIN
      writeln;
      writeln(titre);
      FOR i := 1 TO n DO BEGIN
         write('Ligne ', i : 1, ' :  ');
```

```
        FOR j := 1 TO i DO BEGIN
           read(a[i, j]);
           a[j, i] := a[i, j]
        END;
     END;
     readln
  END; { de "EntrerMatriceSymetrique" }
```

La lecture d'un vecteur se fait sur le même modèle :

```
PROCEDURE EntrerVecteur (n: integer; VAR a: vec; titre: texte);
```
Affichage du titre, puis lecture du vecteur a d'ordre n.
```
  VAR
     i: integer;
  BEGIN
     writeln;
     writeln(titre);
     FOR i := 1 TO n DO
        read(a[i]);
     readln
  END; { de "EntrerVecteur" }
```

Dans les procédures d'affichage, nous faisons appel à une variable globale `precision`, qui indique le nombre de chiffres à imprimer après le point décimal. Ce paramétrage est utile quand on veut se rendre compte de la précision des calculs numériques.

```
PROCEDURE EcrireMatrice (n: integer; VAR a: mat; titre: texte);
```
Affichage du titre, puis de la matrice a d'ordre n.
```
  VAR
     i, j: integer;
  BEGIN
     writeln;
     writeln(titre);
     FOR i := 1 TO n DO BEGIN
        FOR j := 1 TO n DO
           write(a[i, j] : (precision + 5) : precision);
        writeln
     END
  END; { de "EcrireMatrice" }
```

Voici une variante de cette procédure.

```
PROCEDURE EcrireVecteurE (n: integer; VAR a: vecE; titre: texte);
```
Affichage du titre, puis du vecteur entier a d'ordre n.
```
  VAR
     i: integer;
  BEGIN
     writeln;
     writeln(titre);
     FOR i := 1 TO n DO
```

```
         write(a[i] : precision);
      writeln
   END; { de "EcrireVecteurE" }
```

A titre d'exemple de procédures simples de manipulation de matrices, donnons le produit de deux matrices rectangulaires et la transposée d'une matrice carrée.

```
PROCEDURE MatriceParMatriceRect (m, n, p: integer; a, b: mat; VAR ab: mat);
```

Produit de la matrice a d'ordre (m, n) par la matrice b d'ordre (n, p). Résultat dans la matrice ab, d'ordre (m, p).

```
   VAR
      i, j, k: integer;
      somme: real;
   BEGIN
      FOR i := 1 TO m DO
         FOR j := 1 TO p DO BEGIN
            somme := 0;
            FOR k := 1 TO n DO
               somme := somme + a[i, k] * b[k, j];
            ab[i, j] := somme
         END
   END; {de "MatriceParMatriceR" }

PROCEDURE Transposer (n: integer; a: mat; VAR ta: mat);
```

La matrice ta contient la transposée de la matrice a d'ordre n.

```
   VAR
      i, j: integer;
   BEGIN
      FOR i := 1 TO n DO
         FOR j := 1 TO n DO
            ta[i, j] := a[j, i]
   END; { de "Transposer" }
```

Enfin, les procédures de calcul du produit scalaire et des deux normes.

```
FUNCTION ProduitScalaire (n: integer; VAR a, b: vec): real;
   VAR
      i: integer;
      somme: real;
   BEGIN
      somme := 0;
      FOR i := 1 TO n DO
         somme := somme + a[i] * b[i];
      ProduitScalaire := somme
   END; { de "ProduitScalaire" }

FUNCTION Norme (n: integer; VAR a: vec): real;
   BEGIN
         norme := sqrt(ProduitScalaire(n, a, a))
   END; { de "Norme" }
```

```
FUNCTION NormeInfinie (n: integer; VAR a: vec): real;
   VAR
      i: integer;
      r: real;
   BEGIN
      r := 0;
      FOR i := 1 TO n DO
         r := rmax(r, abs(a[i]));
      NormeInfinie := r
   END; { de "NormeInfinie" }
```

1.2 Résolution de systèmes linéaires

Soit à résoudre un système d'équations linéaires :

$$Ax = b$$

où $A = (a_{i,j})$ est une matrice réelle d'ordre n, et x et b sont des vecteurs colonnes. On suppose que A est de rang n. Le cas des systèmes triangulaires est suffisamment simple pour être traité immédiatement.

```
PROCEDURE SystemeTriangulaireSuperieur (n: integer; a: mat; b: vec;
      VAR x: vec);
   VAR
      k, j: integer;
      s: real;
   BEGIN
      FOR k := n DOWNTO 1 DO BEGIN
         s := b[k];
         FOR j := k + 1 TO n DO
            s := s - a[k, j] * x[j];
         x[k] := s / a[k, k]
      END;
   END; { de "SystemeTriangulaireSuperieur" }
```

Pour le cas général, on considère ici deux méthodes voisines, la méthode dite de Gauss et la méthode de Jordan. Une variante de ces méthodes, fondée sur la décomposition LU, sera présentée plus loin.

1.2.1 Méthode de Gauss

La méthode de Gauss ou *méthode du pivot partiel* est composée de deux phases : une phase d'élimination qui transforme le système d'équations en un système triangulaire équivalent et une phase de substitution dans laquelle on résout ce système triangulaire.

Soit $A^{(1)}x = b^{(1)}$ le système original. Après une éventuelle permutation de lignes, on peut supposer que le coefficient $a_{1,1}^{(1)}$ (le «pivot») n'est pas nul, sinon A ne serait pas de rang n. On résout alors la première équation par rapport à x_1 et on élimine cette variable des autres équations, ce qui donne le système équivalent suivant :

$$\begin{array}{rcrcrcrcl} a_{1,1}^{(1)}x_1 & + & a_{1,2}^{(1)}x_2 & + & \cdots & + & a_{1,n}^{(1)}x_n & = & b_1^{(1)} \\ & & a_{2,2}^{(2)}x_2 & + & \cdots & + & a_{2,n}^{(2)}x_n & = & b_2^{(2)} \\ & & \vdots & & & & \vdots & & \vdots \\ & & a_{n,2}^{(2)}x_2 & + & \cdots & + & a_{n,n}^{(2)}x_n & = & b_n^{(2)} \end{array}$$

Après une éventuelle permutation de la deuxième ligne avec une ligne d'indice supérieur, on peut supposer le «pivot» $a_{2,2}^{(2)}$ non nul et recommencer avec ce système. Après k itérations, la matrice $A^{(k+1)}$ a la forme :

$$A^{(k+1)} = \begin{pmatrix} a_{1,1}^{(1)} & a_{1,2}^{(1)} & \cdots & a_{1,k}^{(1)} & & & a_{1,n}^{(1)} \\ 0 & a_{2,2}^{(2)} & \cdots & a_{2,k}^{(2)} & & & a_{2,n}^{(2)} \\ \vdots & & & \vdots & & & \vdots \\ 0 & & & a_{k,k}^{(k)} & & & a_{k,n}^{(k)} \\ 0 & & & 0 & a_{k+1,k+1}^{(k+1)} & \cdots & a_{k+1,n}^{(k+1)} \\ \vdots & & & \vdots & \vdots & & \vdots \\ 0 & & & 0 & a_{n,k+1}^{(k+1)} & & a_{n,n}^{(k+1)} \end{pmatrix}$$

Les «équations de pivotage» pour passer de $A^{(k)}$ à $A^{(k+1)}$ sont

$$\begin{aligned} a_{i,j}^{(k+1)} &= a_{i,j}^{(k)} - a_{i,k}^{(k)} a_{k,j}^{(k)} / a_{k,k}^{(k)} \qquad \text{pour } k+1 \le i, j \le n \\ b_i^{(k+1)} &= b_i^{(k)} - a_{i,k}^{(k)} b_k^{(k)} / a_{k,k}^{(k)} \qquad \text{pour } k+1 \le i \le n \end{aligned} \tag{2.1}$$

et par ailleurs

$$\begin{aligned} a_{i,j}^{(k+1)} &= \begin{cases} 0 & \text{pour } k+1 \le i \le n \text{ et } j = k \\ a_{i,j}^{(k)} & \text{sinon} \end{cases} \\ b_i^{(k+1)} &= b_i^{(k)} \qquad \text{pour } 1 \le i \le k-1 \end{aligned}$$

Notons que le passage de l'équation $A^{(k)}x = b^{(k)}$ à l'équation $A^{(k+1)}x = b^{(k+1)}$ revient à multiplier les deux membres de la première équation par la matrice de Frobenius G_k (une *matrice de Frobenius* est une matrice qui ne diffère de la matrice unité que par une ligne ou colonne) donnée par

$$G_k = \begin{pmatrix} 1 & & 0 & & 0 \\ & \ddots & & & \\ 0 & & 1 & & 0 \\ & & -a_{k+1,k}^{(k)}/a_{k,k}^{(k)} & 1 & 0 \\ & & \vdots & & \ddots \\ 0 & & -a_{n,k}^{(k)}/a_{k,k}^{(k)} & 0 & 1 \end{pmatrix}$$

qui est de déterminant 1, ce qui prouve en particulier que $A^{(k+1)}$ est inversible si et seulement si $A^{(k)}$ l'est. Dans le cas où il y a échange de lignes, il convient de multiplier au préalable la matrice $A^{(k)}$ par la matrice de permutation correspondante, ce qui change le signe du déterminant.

On obtient donc, après la dernière opération de pivotage, un système d'équations triangulaire $A^{(n)}x = b^{(n)}$ qui s'écrit

$$\begin{aligned} a_{1,1}^{(1)}x_1 + a_{1,2}^{(1)}x_2 + \cdots + a_{1,n}^{(1)}x_n &= b_1^{(1)} \\ a_{2,2}^{(2)}x_2 + \cdots + a_{2,n}^{(2)}x_n &= b_2^{(2)} \\ \ddots \qquad \vdots \quad &\quad \vdots \\ a_{n,n}^{(n)}x_n &= b_n^{(n)} \end{aligned}$$

que l'on résout en utilisant la méthode décrite plus haut. L'opération de pivotage, centrale dans cet algorithme, se programme par :

```
PROCEDURE PivoterGauss (k: integer);
   VAR
      i, j: integer;
      g: real;
   BEGIN
      FOR i := k + 1 TO n DO BEGIN
         g := a[i, k] / a[k, k];
         b[i] := b[i] - g * b[k];
         FOR j := k + 1 TO n DO
            a[i, j] := a[i, j] - g * a[k, j]
      END;
   END; { de "PivoterGauss" }
```

Seuls les coefficients qui interviennent dans la suite sont modifiés. La recherche du pivot peut se faire selon deux stratégies. La première est la recherche d'un pivot non nul et s'implémente par :

```
FUNCTION ChercherPivot (k: integer): integer;
```
Cherche un indice i entre k et n tel que $a_{i,k} \neq 0$; si un tel indice n'existe pas, le résultat est $n + 1$.
```
   VAR
      i: integer;
   BEGIN
      i := k;
      WHILE (i <= n) AND EstNul(a[i, k]) DO          { AND séquentiel }
         i := i + 1;
      ChercherPivot := i
   END; { de "ChercherPivot" }
```

Une autre façon de procéder est la recherche systématique du pivot de plus grand module, ce qui peut avoir un effet bénéfique sur les erreurs d'arrondi :

```
FUNCTION PivotMax (k: integer): integer;
```
Détermine un indice p entre k et n tel que $|a_{p,k}|$ est maximal.
```
   VAR
      i, p: integer;
   BEGIN
      p := k;
      FOR i := k + 1 TO n DO
         IF abs(a[i, k]) > abs(a[p, k]) THEN
            p := i;
      PivotMax := p
   END; { de "PivotMax" }
```

La fonction `EstNul` teste si son argument est nul à ε près. Elle est définie comme suit :

```
FUNCTION EstNul (r: real): boolean;
```
Vrai si r est nul à ε près.
```
   BEGIN
      EstNul := abs(r) < epsilon
   END; { de "EstNul" }
```

Bien entendu, le nombre ε est «petit». Le choix d'une valeur convenable pour ε ne peut pas être fait *a priori*. Prendre $\varepsilon = 10^{-8}$ n'a pas de sens lorsque les éléments de la matrice sont tous de l'ordre de 10^{-10}! Il faut choisir ε petit par rapport à une norme de la matrice. Dans la mesure où les exemples que nous utilisons sont des matrices dont les coefficients sont de l'ordre de l'unité, on peut choisir par exemple $\varepsilon = 10^{-5}$.

Pour échanger des parties de lignes de matrices, on utilise la procédure :

```
PROCEDURE EchangerFinLignes (i, j: integer);
   VAR
      m: integer;
   BEGIN
      FOR m := i TO n DO
         Echanger(a[i, m], a[j, m]);
   END; { de "EchangerFinLignes" }
```

qui fait appel à une procédure qui échange deux nombres réels, soit :

```
PROCEDURE Echanger (VAR u, v: real);
   VAR
      w: real;
   BEGIN
      w := u; u := v; v := w
   END; { de "Echanger" }
```

Avec ces procédures, la méthode de Gauss s'écrit :

```
PROCEDURE SystemeParGauss (n: integer; a: mat; b: vec; VAR x: vec;
   VAR inversible: boolean);
```
Solution d'un système d'équations $Ax = b$ par la méthode de Gauss. Si A est inversible, le booléen est vrai et x contient la solution. Sinon, le booléen est faux.

```
VAR
   k, q: integer;
FUNCTION PivotMax (k: integer): integer;              Définie plus haut.
PROCEDURE PivoterGauss (k: integer);                  Définie plus haut.
PROCEDURE EchangerFinLignes (i, j: integer);          Définie plus haut.
BEGIN  { de "SystemeParGauss" }
   k := 1;                                            Triangulation.
   inversible := true;
   WHILE (k <= n) AND inversible DO BEGIN
      q := PivotMax(k);
      inversible := NOT EstNul(a[q, k]);
      IF inversible THEN BEGIN
         IF q > k THEN BEGIN
            EchangerFinLignes(k, q);
            Echanger(b[k], b[q]);
         END;
         PivoterGauss(k);
      END;
      k := k + 1
   END; { du while sur k }
   IF inversible THEN                                 Résolution.
      SystemeTriangulaireSuperieur(n, a, b, x)
END; { de "SystemeParGauss" }
```

Le *déterminant* de la matrice A est, au signe près, le produit des éléments diagonaux de la matrice $A^{(n)}$:

$$\text{dét}\, A = (-1)^m a_{1,1}^{(1)} a_{2,2}^{(2)} \cdots a_{n,n}^{(n)}$$

où m est le nombre d'échanges de lignes faits. On en déduit immédiatement la procédure suivante de calcul du déterminant par la méthode de Gauss :

```
FUNCTION Determinant (n: integer; a: mat): real;
   Calcule le déterminant de a par la méthode de Gauss.
   VAR
      d: real;
      k, q: integer;
      inversible: boolean;
   FUNCTION ChercherPivot (k: integer): integer;
   PROCEDURE EchangerFinLignes (i, j: integer);
   PROCEDURE pivoter (k: integer);                 Variante de PivoterGauss.
      VAR
         i, j: integer;
         g: real;
      BEGIN
         FOR i := k + 1 TO n DO BEGIN
            g := a[i, k] / a[k, k];
            FOR j := k + 1 TO n DO
               a[i, j] := a[i, j] - g * a[k, j]
```

```
         END;
      END; { de "pivoter" }
   BEGIN { de "Determinant" }
      d := 1; k := 1;
      inversible := true;
      WHILE (k <= n) AND inversible DO BEGIN
         q := ChercherPivot(k);
         inversible := q <> n + 1;
         IF inversible THEN BEGIN
            IF q > k THEN BEGIN
               EchangerFinLignes(k, q); d := -d
            END;
            pivoter(k);
         END;
         k := k + 1
      END; { sur k }
      IF inversible THEN BEGIN
         FOR k := 1 TO n DO d := d * a[k, k];
         Determinant := d
      END
      ELSE
         Determinant := 0;
   END; { de "Determinant" }
```

Une autre façon de calculer le déterminant s'inspire directement du développement par rapport à une ligne ou une colonne. Ceci conduit naturellement à une fonction récursive. Dans la réalisation qui suit, on développe par rapport à la dernière colonne.

```
FUNCTION DeterminantRecursif (n: integer; a: mat): real;
   VAR
      b: mat;
      signe, i, k: integer;
      d: real;
   BEGIN
      IF n = 1 THEN
         DeterminantRecursif := a[1, 1]
      ELSE BEGIN
         d := a[n, n] * DeterminantRecursif(n - 1, a);
         signe := 1;
         FOR i := n - 1 DOWNTO 1 DO BEGIN
            signe := -signe; b := a;
            FOR k := i TO n - 1 DO
               b[k] := b[k + 1];
            d := d + signe * a[i, n] * DeterminantRecursif(n - 1, b);
         END;
         DeterminantRecursif := d
      END
   END; { de "DeterminantRecursif" }
```

Cette procédure engendre, de façon implicite, les $n!$ permutations de $\{1, \dots, n\}$ et est donc très coûteuse en temps dès que n dépasse 6 ou 7, à comparer aux n^3 opérations de la méthode de Gauss. Mais pour $n \leq 5$, on a $n! < n^3$ et malgré les recopies et ajustements de tableaux, le calcul est instantané dans ce cas.

1.2.2 Méthode de Jordan

La méthode de Jordan, aussi appelée méthode de Gauss-Jordan ou *méthode du pivot total*, diffère de la méthode du pivot partiel en deux points : d'une part, à chaque étape, la ligne du pivot est divisée par le pivot et, d'autre part, l'expression de la variable x_k de la k-ième équation est reportée dans toutes les autres équations et pas seulement dans les équations subséquentes.

Le nombre d'opérations de substitution est plus important. En échange, on fait l'économie de la résolution d'un système d'équations triangulaire. Plus précisément, si, avant la k-ième étape, on a le système d'équations

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & a_{1,k}^{(k)} & \cdots & a_{1,n}^{(k)} \\ 0 & 1 & & & a_{2,k}^{(k)} & & a_{2,n}^{(k)} \\ \vdots & & \ddots & & \vdots & & \vdots \\ 0 & & & 1 & a_{k-1,k}^{(k)} & & a_{k-1,n}^{(k)} \\ 0 & & & 0 & a_{k,k}^{(k)} & & a_{k,n}^{(k)} \\ & & & & a_{k+1,k}^{(k)} & \cdots & a_{k+1,n}^{(k)} \\ \vdots & & & \vdots & \vdots & & \vdots \\ 0 & & & 0 & a_{n,k}^{(k)} & & a_{n,n}^{(k)} \end{pmatrix} x = \begin{pmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{k-1}^{(k)} \\ b_k^{(k)} \\ b_{k+1}^{(k)} \\ \vdots \\ b_n^{(k)} \end{pmatrix}$$

on peut supposer, quitte à échanger la ligne k avec une ligne d'indice supérieur, que $a_{k,k}^{(k)}$ n'est pas nul, sinon le système ne serait pas de rang n. En tirant x_k de la k-ième équation et en le remplaçant dans toutes les autres, on obtient

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & a_{1,k+1}^{(k+1)} & \cdots & a_{1,n}^{(k+1)} \\ 0 & 1 & & & a_{2,k+1}^{(k+1)} & & a_{2,n}^{(k+1)} \\ \vdots & & \ddots & & \vdots & & \vdots \\ 0 & & & 1 & a_{k,k+1}^{(k+1)} & & a_{k,n}^{(k+1)} \\ \vdots & & & \vdots & \vdots & & \vdots \\ 0 & & & 0 & a_{n,k+1}^{(k+1)} & & a_{n,n}^{(k+1)} \end{pmatrix} x = \begin{pmatrix} b_1^{(k+1)} \\ b_2^{(k+1)} \\ \vdots \\ b_k^{(k+1)} \\ r\vdots \\ b_n^{(k)} \end{pmatrix}$$

avec

$$\begin{aligned} a_{k,j}^{(k+1)} &= a_{k,j}^{(k)} / a_{k,k}^{(k)} \qquad \text{pour } k+1 \leq j \leq n \\ b_k^{(k+1)} &= b_k^{(k)} / a_{k,k}^{(k)} \end{aligned}$$

et pour $i = 1, \ldots, n$ et $i \neq k$

$$\begin{aligned} a_{i,j}^{(k+1)} &= a_{i,j}^{(k)} - a_{i,k}^{(k)} a_{k,j}^{(k)} / a_{k,k}^{(k)} \qquad \text{pour } k+1 \leq j \leq n \\ b_i^{(k+1)} &= b_i^{(k)} - a_{i,k}^{(k)} b_k^{(k)} / a_{k,k}^{(k)} \end{aligned}$$

Après la n-ième étape, le système s'écrit

$$\begin{pmatrix} 1 & & \cdots & 0 \\ & 1 & & \\ \vdots & & \ddots & \vdots \\ 0 & & & 1 \end{pmatrix} x = \begin{pmatrix} b_1^{(n)} \\ b_2^{(n)} \\ \vdots \\ b_n^{(n)} \end{pmatrix}$$

ce qui donne directement la solution. La différence principale avec la méthode du pivot partiel réside dans l'opération de pivotage, dont voici l'écriture :

```
PROCEDURE PivoterJordan (k: integer);
   VAR
      i, j: integer;
   BEGIN
      b[k] := b[k] / a[k, k];                    Division de la ligne k.
      FOR j := k + 1 TO n DO
         a[k, j] := a[k, j] / a[k, k];
      FOR i := 1 TO n DO                         Substitution dans les autres lignes.
         IF i <> k THEN BEGIN
            b[i] := b[i] - a[i, k] * b[k];
            FOR j := k + 1 TO n DO
               a[i, j] := a[i, j] - a[i, k] * a[k, j]
         END;
   END; { de "PivoterJordan" }
```

Voici une procédure qui réalise l'algorithme complet :

```
PROCEDURE SystemeParJordan (n: integer; a: mat; b: vec; VAR x: vec;
   VAR inversible: boolean);
   Solution du système d'équations Ax = b par la méthode de Jordan.
   VAR
      k, q: integer;
   FUNCTION PivotMax (k: integer): integer;
   PROCEDURE EchangerFinLignes (i, j: integer);
   PROCEDURE PivoterJordan (k: integer);
   BEGIN  { de "SystemeParJordan" }
      k := 1; inversible := true;
      WHILE (k <= n) AND inversible DO BEGIN
         q := PivotMax(k);
         inversible := NOT EstNul(a[q, k]);
         IF inversible THEN BEGIN
            IF q > k THEN BEGIN
```

```
                EchangerFinLignes(k, q);
                Echanger(b[k],b[q])
             END;
             PivoterJordan(k);
          END;
          k := k + 1
       END; { du while sur k }
    END; { de "SystemeParJordan" }
```

Le calcul de l'*inverse* d'une matrice A inversible d'ordre n revient à résoudre les n systèmes linéaires

$$Ax = e_i, \qquad i = 1, \ldots, n$$

où e_i est le vecteur colonne dont toutes les coordonnées sont nulles sauf la i-ème qui est égale à 1. On procède bien entendu par résolution simultanée plutôt que successive de ces systèmes. Si, à une étape, un pivot nul ne peut pas être éliminé par échange de lignes, la matrice n'est pas inversible.

Voici une réalisation.

```
PROCEDURE InverseParJordan (n: integer; a: mat; VAR ia: mat;
   VAR inversible: boolean);
   Calcule la matrice inverse ia de a par Jordan, si a est inversible.
   VAR
      k, q, j, m: integer;
      s: real;
   FUNCTION PivotMax (k: integer): integer;
   PROCEDURE EchangerLignes (i, j: integer);
      VAR
         m: integer;
      BEGIN
         FOR m := i TO n DO Echanger(a[i, m], a[j, m]);
         FOR m := 1 TO n DO Echanger(ia[i, m], ia[j, m]);
      END; { de "EchangerLignes" }
   PROCEDURE PivoterJordan (k: integer);          Variante de PivoterJordan,
      VAR                                          définie plus haut.
         i, j: integer;
      BEGIN
         FOR j := k + 1 TO n DO
            a[k, j] := a[k, j] / a[k, k];
         FOR j := 1 TO n DO
            ia[k, j] := ia[k, j] / a[k, k];
         FOR i := 1 TO n DO
            IF i <> k THEN BEGIN
               FOR j := k + 1 TO n DO a[i, j] := a[i, j] - a[i, k] * a[k, j];
               FOR j := 1 TO n DO ia[i, j] := ia[i, j] - a[i, k] * ia[k, j]
            END;
      END; { de "PivoterJordan" }
   BEGIN  { de "InverseParJordan" }
```

```
      ia := MatriceUnite;
      k := 1; inversible := true;
      WHILE (k <= n) AND inversible DO BEGIN
         q := PivotMax(k);
         inversible := NOT EstNul(a[q, k]);
         IF inversible THEN BEGIN
            IF q > k THEN EchangerLignes(k, q);
            PivoterJordan(k);
         END;
         k := k + 1
      END; { du while sur k }
   END; { de "InverseParJordan" }
```

Rappelons que `MatriceUnite` désigne la matrice unité, que l'on suppose définie dans le programme entourant, par exemple par la procédure `InitMatrices`. La procédure précédente peut être raffinée; en particulier, il est possible de stocker l'inverse de a dans le tableau réservé à a.

1.3 Rang d'une matrice

Soit A une matrice réelle d'ordre (m, n). On peut déterminer le rang r de A en calculant le nombre de colonnes linéairement indépendantes de A. Le même procédé donne la dimension de l'espace vectoriel engendré par un ensemble fini de vecteurs; un petit raffinement permet d'extraire, de cet ensemble, une famille libre maximale.

Pour le calcul du rang, on applique la méthode de Gauss à la matrice A. Soit $A^{(k)}$ la matrice obtenue avant la k-ième étape. Les pivots $a_{1,1}^{(1)}, \ldots, a_{k-1,k-1}^{(k-1)}$ étant non nuls, la matrice est de rang au moins $k-1$. Si les coefficients $a_{i,k}^{(k)}$, pour $i = k+1, \ldots, n$, sont tous nuls, la k-ième colonne de $A^{(k)}$ est combinaison linéaire des précédentes; on la supprime alors de $A^{(k)}$ et on procède avec la suivante. S'il ne reste plus de colonne à examiner, le calcul du rang est terminé. Si, en revanche, l'un des coefficients $a_{i,k}^{(k)}$, pour $i = k+1, \ldots, n$, est non nul, on effectue le pivotage.

Dans la réalisation qui suit, la suppression d'une colonne linéairement dépendante des précédentes se fait en échangeant cette colonne avec la dernière colonne. Un compteur de colonnes est alors décrémenté. Lorsqu'il ne reste plus de colonnes, ce compteur est égal au rang.

```
FUNCTION Rang (m, n: integer; a: mat): integer;
```
Calcule le rang de la matrice a d'ordre (m, n). Les procédures de la méthode de Gauss sont légèrement modifiées, pour tenir compte du fait que a n'est pas nécessairement carrée.
```
   VAR
      k, q: integer;
      r: integer;
      independant: boolean;
```

```
FUNCTION ChercherPivot (k: integer): integer;          Variante.
   VAR
      i: integer;
   BEGIN
      i := k;
      WHILE (i <= m) AND EstNul(a[i, k]) DO  { AND séquentiel }
         i := i + 1;
      ChercherPivot := i
   END; { de "ChercherPivot" }
PROCEDURE EchangerFinLignes (i, j: integer);           Variante.
   VAR
      h: integer;
   BEGIN
      FOR h := i TO n DO
         Echanger(a[i, h], a[j, h])
   END; { de "EchangerFinLignes" }
PROCEDURE EchangerColonnes (i, j: integer);
   VAR
      h: integer;
   BEGIN
      FOR h := 1 TO m DO
         Echanger(a[h, i], a[h, j])
   END; { de "EchangerColonnes" }
PROCEDURE PivoterGauss (k: integer);                   Variante.
   VAR
      i, j: integer;
      g: real;
   BEGIN
      FOR i := k + 1 TO m DO BEGIN
         g := a[i, k] / a[k, k];
         FOR j := k + 1 TO n DO
            a[i, j] := a[i, j] - g * a[k, j]
      END;
   END; { de "PivoterGauss" }

BEGIN  { de "Rang" }
   r := n; k := 1;
   WHILE k <= r DO BEGIN
      q := ChercherPivot(k);
      independant := q <> m + 1;
      IF independant THEN BEGIN
         IF q > k THEN EchangerFinLignes(k, q);
         PivoterGauss(k);
         k := k + 1;
      END
      ELSE BEGIN
         EchangerColonnes(k, r);
         r := r - 1
```

```
        END;
      END; { de k }
      Rang := r
    END; { de "Rang" }
```

Voici un exemple d'exécution de la procédure, avec quelques impressions intermédiaires.

```
Entrer la matrice
Ligne 1 :  1 0 2
Ligne 2 :  2 0 4
Ligne 3 :  3 1 6
Ligne 4 :  1 0 2
Ligne 5 :  0 0 0
Matrice lue :
   1.000   0.000   2.000
   2.000   0.000   4.000
   3.000   1.000   6.000
   1.000   0.000   2.000
   0.000   0.000   0.000
k=2  r=3
   1.000   0.000   2.000
   2.000   0.000   0.000
   3.000   1.000   0.000
   1.000   0.000   0.000
   0.000   0.000   0.000
k=3  r=3
   1.000   0.000   2.000
   2.000   1.000   0.000
   3.000   0.000   0.000
   1.000   0.000   0.000
   0.000   0.000   0.000
k=3  r=2
   1.000   0.000   2.000
   2.000   1.000   0.000
   3.000   0.000   0.000
   1.000   0.000   0.000
   0.000   0.000   0.000
Rang = 2
```

Si l'on conserve les indices des colonnes qui ont donné lieu à un pivotage, on obtient une base des colonnes de la matrice A. Il suffit de quelques changements mineurs pour adapter la procédure précédente en ce sens.

```
PROCEDURE BaseDesColonnes (m, n: integer; a: mat; VAR r: integer;
   VAR p: vecE);
```

Calcule le rang r de la matrice a, et un vecteur p tel que les colonnes d'indices dans $p[1], \ldots, p[r]$ forment une base des colonnes de a.

```
   VAR
      k, q: integer;
```

```
      independant: boolean;
   FUNCTION ChercherPivot (k: integer): integer;         Voir Rang.
   PROCEDURE EchangerFinLignes (i, j: integer);          Voir Rang.
   PROCEDURE EchangerColonnes (i, j: integer);           Voir Rang.
   PROCEDURE PivoterGauss (k: integer);                  Voir Rang.
   BEGIN  { de "BaseDesColonnes" }
      r := n;
      FOR k := 1 TO n DO p[k] := k;                      Initialisation de p.
      k := 1;
      WHILE k <= r DO BEGIN
         q := ChercherPivot(k);
         independant := q <> m + 1;
         IF independant THEN BEGIN
            IF q > k THEN EchangerFinLignes(k, q);
            PivoterGauss(k);
            k := k + 1;
         END
         ELSE BEGIN
            EchangerColonnes(k, r);                      Remplacement
            p[k] := r;                                   de colonne.
            r := r - 1
         END;
      END; { de k }
   END; { de "BaseDesColonnes" }
```

Voici un exemple, avec quelques résultats intermédiaires :

```
Entrer la matrice
Ligne 1 :  0 1 1 0 1
Ligne 2 :  0 0 0 0 0
Ligne 3 :  0 1 1 1 1
Matrice lue :
   0.000   1.000   1.000   0.000   1.000
   0.000   0.000   0.000   0.000   0.000
   0.000   1.000   1.000   1.000   1.000
k=1  r=4
   1.000   1.000   1.000   0.000   0.000
   0.000   0.000   0.000   0.000   0.000
   1.000   1.000   1.000   1.000   0.000
k=2  r=4
   1.000   1.000   1.000   0.000   0.000
   0.000   0.000   0.000   0.000   0.000
   1.000   0.000   0.000   1.000   0.000
k=2  r=3
   1.000   0.000   1.000   1.000   0.000
   0.000   0.000   0.000   0.000   0.000
   1.000   1.000   0.000   0.000   0.000
k=3  r=3
```

```
   1.000   0.000   1.000   1.000   0.000
   0.000   1.000   0.000   0.000   0.000
   1.000   0.000   0.000   0.000   0.000
k=3  r=2
   1.000   0.000   1.000   1.000   0.000
   0.000   1.000   0.000   0.000   0.000
   1.000   0.000   0.000   0.000   0.000
Rang = 2
Indices des colonnes de la base
  5  4
```

Bien entendu, il y a d'autres bases de colonnes ; la solution fournie par le programme est due au sens de parcours de la matrice.

Notes bibliographiques

Tous les livres d'analyse numérique exposent les diverses méthodes dites «directes» de résolution de systèmes d'équations linéaires ; les méthodes itératives sont employées pour des systèmes très volumineux, tels qu'ils se présentent dans la résolution d'équations aux dérivées partielles. Pour plus de détails, on pourra consulter :

P.G. Ciarlet, *Introduction à l'analyse numérique matricielle et à l'optimisation*, Paris, Masson, 1988.

Les erreurs d'arrondi sont un problème majeur de toutes les méthodes de résolution numérique. Un «classique» est :

J.H. Wilkinson, *Rounding errors in algebraic processes*, Englewood Cliffs, Prentice-Hall, 1963.

Chapitre 2

Manipulation de matrices

2.1 Pseudo-inverses

2.1.1 Enoncé : pseudo-inverses

Soient $m, n \geq 1$ deux entiers. On note $\mathcal{M}_{m,n}$ l'ensemble des matrices réelles à m lignes et n colonnes. On note ${}^tA$ la transposée d'une matrice A et I_k la matrice carrée unité d'ordre k. On appelle *pseudo-inverse* d'une matrice $A \in \mathcal{M}_{m,n}$ toute matrice $G \in \mathcal{M}_{n,m}$ vérifiant

$$AGA = A \quad GAG = G \quad {}^t(AG) = AG \quad {}^t(GA) = GA$$

1.– Démontrer qu'une matrice A possède au plus une pseudo-inverse.

On admettra pour l'instant (voir ci-dessous) que toute matrice A possède une pseudo-inverse, qui est donc unique et qui est notée A^g.

2.– Soit $A \in \mathcal{M}_{m,n}$ de rang n. Démontrer que $A^g = ({}^tAA)^{-1}\,{}^tA$.

3.– Soit $U \in \mathcal{M}_{n,n}$ une matrice symétrique définie positive. Démontrer qu'il existe une matrice triangulaire inférieure L à éléments diagonaux strictement positifs et une seule telle que $U = L\,{}^tL$.

4.– a) Ecrire une procédure qui prend en argument une matrice carrée symétrique définie positive U et qui calcule la matrice L de la question précédente.

Exemple numérique : $U = \begin{pmatrix} 5 & 15 & 55 \\ 15 & 55 & 225 \\ 55 & 225 & 979 \end{pmatrix}$

b) Ecrire une procédure qui prend en argument une matrice carrée symétrique définie positive U et qui calcule son inverse. Même exemple numérique.

5.– Ecrire une procédure qui prend en argument une matrice $A \in \mathcal{M}_{m,n}$ de rang n et qui calcule sa pseudo-inverse.

Exemple numérique : $A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \end{pmatrix}$

6.– Soit $A \in \mathcal{M}_{m,n}$ de rang r et supposons que A se mette sous la forme $A = (U,\ UK)$, où $U \in \mathcal{M}_{m,r}$ est de rang r et $K \in \mathcal{M}_{r,n-r}$.
a) Démontrer que

$$A^g = \begin{pmatrix} WU^g \\ {}^tKWU^g \end{pmatrix}$$

où $W = (I_r + K{}^tK)^{-1}$.
b) Ecrire une procédure qui prend en argument une matrice $U \in \mathcal{M}_{m,r}$ de rang r et une matrice $K \in \mathcal{M}_{r,n-r}$ et qui calcule la pseudo-inverse de la matrice $(U,\ UK)$.

Exemple numérique : $U = \begin{pmatrix} 1 & 2 \\ 0 & 0 \\ 0 & 0 \\ 2 & 1 \end{pmatrix} \qquad K = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$

7.– Ecrire une procédure qui prend en argument une matrice $A \in \mathcal{M}_{m,n}$ et qui calcule sa pseudo-inverse.

Exemple numérique : $A = \begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 2 & 0 \end{pmatrix}$

8.– Prouver qu'une matrice possède une pseudo-inverse.

Soient $m, n \geq 1$, avec $m \geq n$. On considère une matrice $A \in \mathcal{M}_{m,n}$ et un vecteur colonne $b \in \mathbb{R}^m$. Le *problème des moindres carrés* est la détermination d'un vecteur $x \in \mathbb{R}^n$ tel que

$$\|Ax - b\| = \min_{y \in \mathbb{R}^n} \|Ay - b\| \tag{$*$}$$

où $\|\cdot\|$ désigne la norme euclidienne.

9.– Démontrer que x est solution de $(*)$ si et seulement si

$${}^tAAx = {}^tAb$$

En déduire que $A^g b$ est solution de $(*)$. Démontrer que si A est de rang n, alors $(*)$ a une solution unique.

2.1.2 Solution : pseudo-inverses

On appelle *pseudo-inverse* ou également *inverse généralisée* ou *inverse de Penrose-Moore* d'une matrice $A \in \mathcal{M}_{m,n}$ toute matrice $G \in \mathcal{M}_{n,m}$ vérifiant

$$AGA = A \quad GAG = G \quad {}^t(AG) = AG \quad {}^t(GA) = GA \tag{1.1}$$

Dans le cas d'une matrice A carrée inversible, l'inverse A^{-1} est l'unique pseudo-inverse.

PROPOSITION 2.1.1. *Une matrice possède au plus une pseudo-inverse.*

Preuve. Soient G et G' deux pseudo-inverses d'une matrice A. On a alors

$$AG' = AGAG' = {}^t(AG){}^t(AG') = {}^t(AG'AG) = {}^t(AG) = AG$$

et de même $GA = G'A$. On en déduit que $G = GAG = GAG' = G'AG' = G'$. ∎

Quant à l'existence, on a la proposition suivante.

PROPOSITION 2.1.2. *Toute matrice possède une pseudo-inverse.*

Il y a plusieurs démonstrations de cette proposition. Nous en donnons deux : la première est brève, la deuxième conduit à un algorithme.

Preuve. Munissons $\mathbb{R}^n$ et $\mathbb{R}^m$ du produit scalaire usuel, noté «$\cdot$». Soit $A \in \mathcal{M}_{m,n}$ et soit a l'application linéaire de $\mathbb{R}^n$ dans $\mathbb{R}^m$ dont A est la matrice dans les bases canoniques. Notons p la projection orthogonale de $\mathbb{R}^m$ d'image $\mathrm{Im}(A)$ et q la projection orthogonale de $\mathbb{R}^n$ de noyau $\mathrm{Ker}(A)$. La restriction $\bar{a}$ de a à $\mathrm{Ker}(A)^\perp$ est un isomorphisme de $\mathrm{Ker}(A)^\perp$ sur $\mathrm{Im}(A)$. Posons

$$g = \bar{a}^{-1} \circ q : \mathbb{R}^m \to \mathbb{R}^n$$

Alors $g \circ a = p$, $a \circ g = q$, donc $g \circ a \circ g = g$ et $a \circ g \circ a = a$. De plus, l'endomorphisme p est symétrique, c'est-à-dire $px \cdot y = x \cdot py$ pour tout x, y dans $\mathbb{R}^m$, car $px \cdot y = px \cdot (py + (I-p)y) = px \cdot py$ et de même $x \cdot py = px \cdot py$, d'où l'égalité. Il en résulte que la matrice P de p dans la base canonique est symétrique et, par le même raisonnement, la matrice Q de q est symétrique. Si l'on note G la matrice de g, on a donc ${}^t(GA) = {}^tP = P = GA$ et de même ${}^t(AG) = AG$. Ceci prouve que G est une pseudo-inverse de A. ∎

Nous notons désormais A^g la pseudo-inverse d'une matrice A. Voici une deuxième démonstration, en trois parties.

LEMME 2.1.3. *Si $A \in \mathcal{M}_{m,n}$ est de rang n, alors $({}^tAA)^{-1}\,{}^tA$ est la pseudo-inverse de A.*

Preuve. Posons $A^g = ({}^tAA)^{-1}\,{}^tA$. La matrice ${}^tAA$ est carrée d'ordre n et de rang n, donc inversible. Comme $A^gA = I_n$, la matrice unité d'ordre n, et que AA^g est symétrique, les équations (1.1) sont manifestement vérifiées. ∎

LEMME 2.1.4. *Soit $A \in \mathcal{M}_{m,n}$ de rang r et supposons que A se mette sous la forme $A = (U,\ UK)$, où $U \in \mathcal{M}_{m,r}$ est de rang r et $K \in \mathcal{M}_{r,n-r}$. Alors la pseudo-inverse de A est*

$$\begin{pmatrix} WU^g \\ {}^tKWU^g \end{pmatrix}$$

où $W = (I_r + K\,{}^tK)^{-1}$.

Preuve. Montrons d'abord que la matrice $S = I_r + K{}^tK$ est inversible. Elle est en effet définie positive parce que ${}^txSx = {}^txx + {}^txK{}^tKx = \|x\|^2 + \|{}^tKx\|^2 = 0$ si et seulement si $x = 0$. Notons B la matrice dont on veut prouver qu'elle est la pseudo-inverse de A. On a

$$AB = UWU^g + UK{}^tKWU^g = U(I + K{}^tK)WU^g = UU^g$$

Donc

$$ABA = (UU^gU,\ UU^gUK) = A \qquad {}^t(AB) = {}^t(UU^g) = UU^g = AB$$

Par ailleurs

$$B(AB) = \begin{pmatrix} WU^g \\ {}^tKWU^g \end{pmatrix} UU^g = \begin{pmatrix} WU^gUU^g \\ {}^tKWU^gUU^g \end{pmatrix} = B$$

Enfin, notons qu'en vertu du lemme précédent, $U^gU = ({}^tUU)^{-1}{}^tUU = I_r$. Donc

$$BA = \begin{pmatrix} WU^g \\ {}^tKWU^g \end{pmatrix} (U,\ UK) = \begin{pmatrix} W & WK \\ {}^tKW & {}^tKW \end{pmatrix}$$

Comme S est symétrique, W l'est également, ce qui montre que ${}^t(BA) = BA$ et achève la preuve. ∎

La *deuxième preuve* de la proposition 2.1.2 se fait alors comme suit : Si $A = 0$, alors A est sa propre pseudo-inverse. Sinon, soit r le rang de A. Il existe une matrice de permutation P telle que $AP = (U,\ UK)$, avec $U \in \mathcal{M}_{m,r}$ de rang r et $K \in \mathcal{M}_{r,n-r}$. D'après le lemme 2.1.4, la matrice $H = AP$ a donc une pseudo-inverse H^g. Montrons que $G = PH^g$ est la pseudo-inverse de A. On a en effet $A = HP^{-1}$, $AG = APH^g = HH^g$ et $GA = PH^gHP^{-1}$. Donc on a

$$\begin{array}{ll} AGA = HH^gHP^{-1} = HP^{-1} = A & GAG = PH^gHH^g = PH^g = G \\ {}^t(AG) = {}^t(HH^g) = AG & {}^t(GA) = {}^t(PH^gHP^{-1}) = GA \end{array}$$

la dernière équation étant vraie parce que ${}^tP = P^{-1}$. ∎

L'algorithme pour calculer la pseudo-inverse d'une matrice A est le suivant : dans une première étape, on cherche une matrice de permutation P telle que

$$AP = (U,\ UK)$$

avec $U \in \mathcal{M}_{m,r}$ de rang r et $K \in \mathcal{M}_{r,n-r}$. On détermine alors la pseudo-inverse de AP par les formules du lemme 2.1.4 et on revient à la pseudo-inverse de A en prémultipliant par P. L'algorithme est mis en œuvre dans la section suivante.

Notons que $(AB)^g$ est en général différent de B^gA^g, l'un des exemples les plus simples est probablement

$$A = (1\ 0) \qquad B = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Il n'est pas difficile de vérifier que $(AB)^g = 1$ et $B^gA^g = 1/2$.

Parmi les applications des pseudo-inverses figure une présentation élégante de la solution au problème dit des moindres carrés. Soient $m, n \geq 1$, avec $m \geq n$. On considère une matrice $A \in \mathcal{M}_{m,n}$ et un vecteur colonne $b \in \mathbb{R}^m$. Le *problème des moindres carrés* est la détermination d'un vecteur $x \in \mathbb{R}^n$ tel que

$$\|Ax - b\| = \min_{y \in \mathbb{R}^n} \|Ay - b\| \tag{1.2}$$

où $\|\cdot\|$ désigne la norme euclidienne.

PROPOSITION 2.1.5. *Le vecteur $x \in \mathbb{R}^n$ est solution de* (1.2) *si et seulement si ${}^tAAx = {}^tAb$. Le vecteur $A^g b$ est solution de* (1.2) *et si A est de rang n, alors c'est la seule solution.*

Preuve. Soit $x \in \mathbb{R}^n$. Pour t réel et w dans $\mathbb{R}^n$, considérons

$$\Phi_w(t) = \|A(x + tw) - b\|^2 - \|Ax - b\|^2 \tag{1.3}$$

On a

$$\Phi_w(t) = t^2\|Aw\|^2 + 2t(Ax - b \cdot Aw) \tag{1.4}$$

En vertu de (1.3), x est solution de (1.2) si et seulement si $\Phi_w(t) \geq 0$ pour tout t et pour tout w et, par (1.4), ceci se réalise si et seulement si $(Ax - b \cdot Aw) = 0$ pour tout w, donc si et seulement si ${}^tA(Ax - b) = 0$. Ceci prouve l'équivalence.

Pour vérifier que $A^g b$ est solution de (1.2), il suffit de calculer :

$${}^tAAA^g b = {}^tA^t(AA^g)b = {}^t(AA^gA)b = {}^tAb$$

L'unicité de la solution provient de ce que l'équation ${}^tAAx = {}^tAb$ a une solution unique si A est de rang n, puisqu'alors ${}^tAA$ est inversible. ∎

2.1.3 Programme : pseudo-inverses

Commençons le calcul de la pseudo-inverse par le cas le plus simple, à savoir le cas où la matrice A d'ordre (m, n) est de rang n. Il suffit alors de calculer $({}^tAA)^{-1}{}^tA$. Or, la matrice ${}^tAA$ est symétrique et définie positive, puisque ${}^tx{}^tAAx = \|Ax\|^2 = 0$ seulement si $Ax = 0$ et, comme A est de rang n, cela n'est possible que si $x = 0$. On peut donc utiliser, pour inverser ${}^tAA$, la méthode de Choleski exposée au chapitre suivant. On obtient alors la procédure suivante :

```
PROCEDURE PseudoInverseRangMaximal (m, n: integer; a: mat; VAR g: mat);
   La matrice a d'ordre (m,n) est supposée de rang n. Calcul de sa pseudo-inverse g par
   la formule g = (ᵗaa)⁻¹ᵗa.
   VAR
      ta, u, v: mat;
   BEGIN
      TransposerRect(m, n, a, ta);                  Calcul de ᵗa.
      MatriceParMatriceRect(n, m, n, ta, a, u);     u = ᵗaa.
      InverseDefPositiveSymetrique(n, u, v);        v = u⁻¹ = (ᵗaa)⁻¹.
      MatriceParMatriceRect(n, n, m, v, ta, g);     g = vᵗa.
```

```
   END; { de "PseudoInverseRangMaximal" }
```

La procédure `TransposerRect` a été décrite au chapitre 1. Dans une deuxième étape, on suppose la matrice A donnée sous la forme $(U,\ UK)$, où U est d'ordre (m,r) et de rang r et où K est d'ordre $(r,n-r)$. On utilise alors la formule du lemme 2.1.4, ce qui donne la procédure que voici :

```
PROCEDURE PseudoInverseDecomposee (m, r, n: integer; u, k: mat; VAR b: mat);
   Calcule la pseudo-inverse b d'une matrice a = (u, uk), donnée par la matrice u d'ordre
   (m, r) de rang r et la matrice k d'ordre (r, n − r).
   VAR
      tk, h, w, ligne1, ligne2: mat;
   BEGIN
      TransposerRect(r, n - r, k, tk);                 tk = ᵗk.
      MatriceParMatriceRect(r, n - r, r, k, tk, h);    h = kᵗk.
      MatricePlusMatrice(r, h, MatriceUnite, h);       h := I + h.
      InverseDefPositiveSymetrique(r, h, w);           w = (I + ᵗkk)⁻¹.
      PseudoInverseRangMaximal(m, r, u, b);            b = uᵍ.
      MatriceParMatriceRect(r, r, m, w, b, ligne1);    ligne1 = wb.
      MatriceParMatriceRect(n - r, r, m, tk, ligne1, ligne2);
      ComposerLignes(r, n - r, m, ligne1, ligne2, b)
   END; { de "PseudoInverseDecomposee" }
```

Cette procédure utilise une procédure qui compose deux matrices en une seule. La voici, avec quelques autres procédures utiles :

```
PROCEDURE CopierColonne (m, k: integer; VAR a: mat; j: integer; VAR b: mat);
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO m DO b[i, j] := a[i, k]
   END; { de "CopierColonne" }

PROCEDURE CopierLigne (m, k: integer; VAR a: mat; j: integer; VAR b: mat);
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO m DO b[j, i] := a[k, i]
   END; { de "CopierLigne" }

PROCEDURE ComposerColonnes (m, r, s: integer; VAR u, v, u_v: mat);
   u est une matrice d'ordre (m, r), v est d'ordre (m, s), u_v = (u, v) est d'ordre (m, r + s).
   VAR
      j: integer;
   BEGIN
      u_v := u;
      FOR j := 1 TO s DO CopierColonne(m, j, v, r + j, u_v);
   END; { de "ComposerColonnes" }

PROCEDURE ComposerLignes (r, s, n: integer; VAR u, v, u_v: mat);
```

```
   u est une matrice d'ordre (r,n), v est d'ordre (s,n), u_v = (u over v) est d'ordre (r + s,n).
   VAR
      i: integer;
   BEGIN
      u_v := u;
      FOR i := 1 TO s DO CopierLigne(n, i, v, r + i, u_v);
   END; { de "ComposerLignes" }
```

La décomposition d'une matrice A en un produit $(U,\ UK)$ est toujours possible à une permutation des lignes près. Pour cela, on prend pour U une famille libre des colonnes de A et pour matrice K la matrice qui donne explicitement les coefficients de dépendance linéaire des autres colonnes de A dans la famille U. Or, ces coefficients s'obtiennent comme solution d'équations linéaires et on voit, après un peu de réflexion, que ces coefficients sont fournis, dans la méthode de Gauss, chaque fois que l'on rencontre une colonne qui n'a pas de pivot non nul. Il suffit donc d'adapter légèrement la procédure `BaseDesColonnes` du chapitre 1.

```
PROCEDURE DecompositionSurBaseDesColonnes (m, n: integer; a: mat;
   VAR rang: integer; VAR u, h: mat; VAR p: vecE);
   Calcule deux matrices u et h telles que a = (u, uh). Le rang de u est rang et p est le
   vecteur des indices des colonnes de u et de h.
   VAR
      k, q: integer;
      independant: boolean;

   FUNCTION ChercherPivot (k: integer): integer;       Les mêmes que
   PROCEDURE EchangerFinlignes (i, j: integer);        dans : BaseDesColonnes.
   PROCEDURE EchangerColonnes (i, j: integer);
   PROCEDURE PivoterGauss (k: integer);

   BEGIN  { de "DecompositionSurBaseDesColonnes" }
      u := a;
      FOR k := 1 TO n DO p[k] := k;                    Initialisation de p.
      k := 1; rang := n;
      WHILE k <= rang DO BEGIN
         q := ChercherPivot(k);
         independant := q <> m + 1;
         IF independant THEN BEGIN
            IF q > k THEN
               EchangerFinlignes(k, q);
            PivoterGauss(k);
            k := k + 1;
         END
         ELSE BEGIN
            EchangerColonnes(k, rang);
            EchangerE(p[k], p[rang]);                  Répercussion sur p.
            rang := rang - 1
         END;
         END; { de k }
```

```
      FOR k := 1 TO rang DO
         CopierColonne(m, p[k], u, k, u);
      h := MatriceNulle;
      FOR k := 1 TO n - rang DO
         CopierColonne(rang, rang + k, a, k, h);
   END; { de "DecompositionSurBaseDesColonnes" }
```

Après ces préliminaires, la procédure de calcul de la pseudo-inverse s'écrit :

```
PROCEDURE PseudoInverse (m, n: integer; a: mat; VAR g: mat);
   Calcule la pseudo-inverse g de la matrice a d'ordre (m, n).
   VAR
      u, h, bg: mat;
      p: vecE;
      i, r: integer;
   BEGIN
      DecompositionSurBaseDesColonnes(m, n, a, r, u, h, p);
      PseudoInverseDecomposee(m, r, n, u, h, bg);
      FOR i := 1 TO n DO
         CopierLigne(m, i, bg, p[i], g);
   END; { de "PseudoInverse" }
```

Voici quelques résultats numériques. Considérons d'abord le cas d'une matrice de rang maximal. On obtient :

```
Matrice de départ
   1.000   1.000   1.000
   1.000   2.000   4.000
   1.000   3.000   9.000
   1.000   4.000  16.000
   1.000   5.000  25.000

Sa pseudo-inverse
   1.800   0.000  -0.800  -0.600   0.600
  -1.057   0.329   0.857   0.529  -0.657
   0.143  -0.071  -0.143  -0.071   0.143
```

Si la matrice est donnée sous forme décomposée :

```
Entrer la matrice U
Ligne 1 :  1 2
Ligne 2 :  0 0
Ligne 3 :  0 0
Ligne 4 :  2 1
Entrer la matrice K
Ligne 1 :  0 0 1
Ligne 2 :  0 0 0

Matrice de départ
   1.000   2.000   0.000   0.000   1.000
   0.000   0.000   0.000   0.000   0.000
```

```
  0.000   0.000   0.000   0.000   0.000
  2.000   1.000   0.000   0.000   2.000

Matrice W
  0.500   0.000
  0.000   1.000

La pseudo-inverse
 -0.167   0.000   0.000   0.333
  0.667   0.000   0.000  -0.333
  0.000   0.000   0.000   0.000
  0.000   0.000   0.000   0.000
 -0.167   0.000   0.000   0.333
```

Enfin, une matrice à décomposer d'abord :

```
Entrer la matrice A
Ligne 1 :  0 1 2 1 0
Ligne 2 :  0 0 0 0 0
Ligne 3 :  0 0 0 0 0
Ligne 4 :  0 2 1 2 0

Matrice U
  1.000   2.000
  0.000   0.000
  0.000   0.000
  2.000   1.000

Matrice H
  1.000   0.000   0.000
  0.000   0.000   0.000

Vecteur permutation
 4  3  2  5  1

Matrice W
  0.500   0.000
  0.000   1.000

Pseudo-inverse avant permutation
 -0.167   0.000   0.000   0.333
  0.667   0.000   0.000  -0.333
 -0.167   0.000   0.000   0.333
  0.000   0.000   0.000   0.000
  0.000   0.000   0.000   0.000

La pseudo-inverse de A
  0.000   0.000   0.000   0.000
 -0.167   0.000   0.000   0.333
  0.667   0.000   0.000  -0.333
 -0.167   0.000   0.000   0.333
  0.000   0.000   0.000   0.000
```

2.2 Matrices trigonalisables

2.2.1 Enoncé : matrices trigonalisables

Les matrices considérées dans ce problème sont éléments de K_n, l'anneau des matrices carrées d'ordre n à coefficients complexes. Une matrice $A = (a_{i,j})$ de K_n est *triangulaire* si $a_{i,j} = 0$ pour $i < j$. On dit qu'un ensemble E de matrices de K_n est *trigonalisable* s'il existe une matrice inversible P telle que, pour toute matrice A de E, la matrice $P^{-1}AP$ soit triangulaire. Le but de ce problème est de donner un critère pour qu'un ensemble de matrices soit trigonalisable.

1.– Ecrire une procédure qui prend en argument deux matrices carrées A et B et calcule $AB - BA$. (On pourra se limiter aux matrices d'ordre inférieur ou égal à 3.)

Exemple numérique. On prend

$$A_1 = \begin{pmatrix} -i & 0 & -2 \\ 0 & 1 & 0 \\ i & 0 & 2+i \end{pmatrix} \quad A_2 = \begin{pmatrix} 2-i & 0 & -2i \\ -i & 1 & -1-i \\ -1 & 0 & i \end{pmatrix} \quad A_3 = \begin{pmatrix} 1 & 1-i & -1-i \\ 0 & 1-i & -1 \\ 0 & -1 & 1+i \end{pmatrix}$$

Afficher $A_1A_2 - A_2A_1$ et $A_1A_3 - A_3A_1$.

2.– Ecrire une procédure qui prend en argument un ensemble fini $E = \{A_1, \ldots, A_k\}$ de matrices de K_n et qui affiche la dimension de l'espace vectoriel (E) engendré par E, ainsi qu'une base de (E) extraite de E. (On pourra supposer que $k \leq 15$ et que les matrices sont d'ordre inférieur ou égal à 3.)

Exemple numérique : $E = \{A_1, A_2, A_3\}$.

Un ensemble F de matrices de K_n est appelé une *algèbre de Lie* si F est un espace vectoriel et si, quels que soient $A, B \in F$, on a

$$AB - BA \in F$$

Soit E un ensemble de matrices et soit E_k la suite d'espaces vectoriels définie ainsi : E_0 est l'espace vectoriel engendré par E et E_{k+1} est l'espace vectoriel engendré par l'ensemble $E_k \cup \{AB - BA \mid A, B \in E_k\}$.

3.– a) Démontrer qu'il existe un plus petit entier s tel que $E_s = E_{s+1}$. Démontrer que E_s est la plus petite algèbre de Lie contenant E (on dit que c'est *l'algèbre de Lie engendrée* par E).

b) Ecrire une procédure qui calcule la dimension de l'algèbre de Lie engendrée par un ensemble fini de matrices.

Exemple numérique : $E = \{A_1, A_2, A_3\}$.

On dit qu'une suite croissante de sous-espaces vectoriels de F

$$\{0\} = F_0 \subset F_1 \subset \ldots \subset F_p = F \qquad (*)$$

est une *suite de résolution de longueur* p de F, si, pour $1 \leq i \leq p$ et quels que soient $A, B \in F_i$, on a

$$AB - BA \in F_{i-1}$$

Une algèbre de Lie F est dite *résoluble* (resp. *résoluble de classe p)* si elle possède une suite de résolution (resp. de longueur p).

4.– a) Si F est une algèbre de Lie, on note $L(F)$ l'algèbre de Lie engendrée par les matrices $AB - BA$ telles que $A, B \in F$. On pose ensuite $L^0(F) = F$ et, pour tout $k \geq 0$, $L^{k+1}(F) = L(L^k(F))$. Démontrer que F est une algèbre de Lie résoluble de classe p si et seulement si $L^p(F) = \{0\}$.

b) Ecrire une procédure qui prend en argument un ensemble fini $E = \{A_1, \ldots, A_k\}$ de matrices de K_n et qui teste si l'algèbre de Lie engendrée par E est résoluble.

Exemple numérique : $E = \{A_1, A_2, A_3\}$.

5.– Démontrer que toute algèbre de Lie trigonalisable est résoluble.

Réciproquement, on se propose de démontrer, par récurrence sur p, que toute algèbre de Lie résoluble de classe p est trigonalisable.

6.– a) Démontrer qu'un ensemble de matrices de K_n qui commutent deux à deux ont un vecteur propre (non nul) commun.

b) En déduire que toute algèbre de Lie de classe 1 est trigonalisable.

On suppose $p > 1$ et on suppose (hypothèse de récurrence) que toute algèbre de Lie résoluble de classe inférieure à p est trigonalisable.

7.– Soit F une algèbre de Lie résoluble de classe p. Démontrer qu'il existe un vecteur propre x_0 (non nul) commun à tous les éléments de F_{p-1}. Pour toute matrice A de F_{p-1}, on désignera par $\lambda(A) \in \mathbb{C}$ la valeur propre de A telle que

$$Ax_0 = \lambda(A)x_0$$

Soit $B \in F_p$. On pose, pour tout $k > 0$, $x_k = B^k x_0$ et on note V_B l'espace vectoriel engendré par la famille $(x_k)_{k \geq 0}$.

8.– a) Démontrer que V_B est l'espace vectoriel engendré par la famille $(x_k)_{0 \leq k \leq r}$, où r est le plus grand entier tel que les vecteurs $x_0, x_1, \ldots, x_r$ soient linéairement indépendants.

b) Démontrer que V_B est stable par B (i.e. pour tout $x \in V_B$, $Bx \in V_B$).

9.– a) Démontrer que pour toute matrice A de F_{p-1}, il existe des nombres complexes $\lambda_{i,j}(A)$ $(0 \leq j < i \leq r)$ tels que, pour $0 \leq i \leq r$,

$$Ax_i = \lambda(A)x_i + \lambda_{i,i-1}(A)x_{i-1} + \cdots + \lambda_{i,0}(A)x_0 \qquad (**)$$

b) Démontrer que pour toute matrice A de F_{p-1}, $\lambda(AB - BA) = 0$.

c) En déduire que, pour tout $x \in V_B$, $Ax = \lambda(A)x$.

10.– On pose

$$V = \{x \in \mathbb{C}^n \mid \text{ pour tout } A \in F_{p-1}, Ax = \lambda(A)x\}$$

a) Démontrer que V est un espace vectoriel non nul stable par toute matrice de F_p.
b) Démontrer que, quels que soient $x \in V$ et $B, C \in F_p$, on a

$$BCx = CBx$$

c) En déduire que les matrices de F_p ont au moins un vecteur propre commun dans V.

11.– Conclure la démonstration.

2.2.2 Solution : matrices trigonalisables

Soit K_n l'anneau des matrices carrées d'ordre n à coefficients complexes. On notera $\mathbb{C}^n$ l'espace vectoriel des vecteurs colonnes d'ordre n à coefficients complexes. Une matrice $A = (a_{i,j})$ de K_n est *triangulaire* si $a_{i,j} = 0$ pour $i < j$. On dit qu'un ensemble E de matrices de K_n est *trigonalisable* s'il existe une matrice inversible P telle que, pour toute matrice A de E, la matrice $P^{-1}AP$ soit triangulaire. On se propose de donner une condition nécessaire et suffisante pour qu'un ensemble de matrices soit trigonalisable. Nous présentons d'abord une condition suffisante simple.

PROPOSITION 2.2.1. *Un ensemble de matrices de K_n qui commutent deux à deux est trigonalisable.*

Preuve. La démonstration repose sur le lemme suivant.

LEMME 2.2.2. *Un ensemble d'endomorphismes de $\mathbb{C}^n$ qui commutent deux à deux ont un vecteur propre (non nul) commun.*

Preuve. On raisonne par récurrence sur n, le résultat étant trivial pour $n = 1$. Soit F un ensemble d'endomorphismes commutant deux à deux. Choisissons, parmi les éléments de F, un endomorphisme u possédant un sous-espace propre V de dimension minimale, et soit λ la valeur propre associée à V. On a donc $V = \{x \in \mathbb{C}^n \mid ux = \lambda x\}$. Maintenant, si $v \in F$ et si $x \in V$, on a, puisque u et v commutent,

$$uvx = vux = v(\lambda x) = \lambda(vx)$$

et donc $vx \in V$. Par conséquent, V est stable par F. Si $\dim(V) = n$, le choix de V impose que V soit un sous-espace propre de *tous* les endomorphismes de F. Si $\dim(V) < n$, on applique l'hypothèse de récurrence à la restriction à V des endomorphismes de F pour conclure. ■

Revenons à la preuve de la proposition 2.2.1. On raisonne encore par récurrence sur n, le cas $n = 1$ étant trivial. Soit F l'ensemble des endomorphismes représentés par les matrices commutant deux à deux de l'énoncé. D'après le lemme 2.2.2, on peut trouver un vecteur propre commun aux éléments de F. Soit x ce vecteur et considérons une base contenant x comme premier vecteur. Dans cette base chaque endomorphisme u a une matrice de la forme

$$\begin{pmatrix} \lambda(u) & 0 \\ A'(u) & A(u) \end{pmatrix}$$

avec $\lambda(u) \in \mathbb{C}$. Maintenant, les matrices $A(u)$, pour $u \in F$, commutent deux à deux. Par récurrence, il existe donc une matrice inversible P' telle que toutes les matrices $P'^{-1}A(u)P'$ soient triangulaires, pour tout $u \in F$. Par conséquent, les matrices

$$\begin{pmatrix} 1 & 0 \\ 0 & P'^{-1} \end{pmatrix} \begin{pmatrix} \lambda(u) & 0 \\ A'(u) & A(u) \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix}$$

sont triangulaires, pour tout $u \in F$. ∎

Pour énoncer le théorème principal, qui donne une caractérisation d'un ensemble de matrices trigonalisables, nous aurons besoin de quelques définitions auxiliaires. Le *crochet de Lie* de deux matrices A et B de K_n est la matrice

$$[A, B] = AB - BA$$

On appelle *algèbre de Lie* tout sous-espace vectoriel de K_n fermé par crochet de Lie. L'algèbre de Lie *engendrée par une partie* E de K_n est la plus petite algèbre de Lie contenant E. L'existence d'une telle algèbre est facile à établir : d'une part, K_n est une algèbre de Lie contenant E, et d'autre part, l'intersection d'une famille quelconque d'algèbres de Lie contenant E est encore une algèbre de Lie contenant E. Pour obtenir un procédé constructif, on calcule la suite des E_k ainsi définie : E_0 est l'espace vectoriel engendré par E et E_{k+1} est l'espace vectoriel engendré par l'ensemble

$$E_k \cup \{[A, B] \mid A, B \in E_k\}$$

Par construction, toute algèbre de Lie contenant E doit contenir tous les E_k. Puisque $\dim(E_k) \leq \dim(E_{k+1}) \leq n$, il existe un plus petit entier s tel que $E_s = E_{s+1}$. L'espace vectoriel E_s est alors fermé par crochet de Lie et c'est donc une algèbre de Lie. Par conséquent, E_s est la plus petite algèbre de Lie contenant E.

On dit qu'une suite croissante de sous-espaces vectoriels de F

$$\{0\} = F_0 \subset F_1 \subset \ldots \subset F_p = F$$

est une *suite de résolution de longueur* p de F si, pour $1 \leq i \leq p$, on a

$$A, B \in F_i \Rightarrow [A, B] \in F_{i-1}$$

Une algèbre de Lie est dite *résoluble* (*de classe* p) si elle admet une suite de résolution (de longueur p). On notera que ces notions sont invariantes par automorphisme. Autrement dit, si φ est un automorphisme de K_n, une algèbre de Lie F est résoluble (resp. résoluble de classe p) si et seulement si $\varphi(F)$ est résoluble (resp. résoluble de classe p). La proposition qui suit donne un premier exemple d'algèbre de Lie résoluble.

Proposition 2.2.3. *Toute algèbre de Lie formée de matrices triangulaires de K_n est résoluble.*

Preuve. Soit F une algèbre de Lie formée de matrices triangulaires de K_n. Posons, pour $1 \leq k \leq n$,

$$T_k = \{A = (a_{i,j}) \in F \mid a_{i,j} = 0 \text{ pour } i < j + k\}$$

Les matrices de T_k sont donc de la forme

$$\begin{pmatrix}
0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
a_{k+1,1} & 0 & \cdots & 0 & 0 & \cdots & 0 \\
a_{k+2,1} & a_{k+2,2} & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
a_{n,1} & a_{n,2} & \cdots & a_{n,n-k} & 0 & \cdots & 0
\end{pmatrix}$$

On montre facilement que, si $A, B \in T_{k-1}$, alors $[A, B] \in T_k$. Il en résulte que la suite $F_k = F \cap T_k$ est une suite de résolution de F. ∎

Il n'est pas immédiat, au vu de la définition d'une algèbre de Lie résoluble, de trouver un algorithme pour tester si une algèbre de Lie est résoluble. Il existe heureusement une définition équivalente beaucoup plus constructive. Si F est une algèbre de Lie, on note $L(F)$ l'algèbre de Lie engendrée par les matrices $[A, B]$ telles que $A, B \in F$. On pose ensuite $L^0(F) = F$ et, pour tout $k \geq 0$, $L^{k+1}(F) = L(L^k(F))$.

Proposition 2.2.4. *Une algèbre de Lie F est résoluble de classe p si et seulement si $L^p(F) = \{0\}$.*

Preuve. Si F est une algèbre de Lie, on a $L(F) \subset F$ et donc

$$L^p(F) \subset L^{p-1}(F) \subset \ldots \subset L(F) \subset F$$

Donc si $L^p(F) = \{0\}$, F est résoluble de classe p. Réciproquement, soit F une algèbre de Lie résoluble de classe p et soit

$$\{0\} = F_0 \subset F_1 \subset \ldots \subset F_p = F$$

une suite de résolution de F. On obtient par récurrence $L^i(F) \subset F_{p-i}$ et donc en particulier $L^p(F) = \{0\}$. ∎

Nous en arrivons au théorème principal.

Théorème 2.2.5. *Un ensemble E de matrices de K_n est trigonalisable si et seulement si l'algèbre de Lie engendrée par E est résoluble.*

Preuve. Soit E un ensemble trigonalisable de matrices et soit P une matrice telle que l'ensemble $E^P = \{P^{-1}AP \mid A \in E\}$ soit formé de matrices triangulaires. Or l'application $A \mapsto P^{-1}AP$ est un automorphisme de l'algèbre de Lie K_n. Comme l'algèbre de Lie engendrée par E^P est résoluble d'après la proposition 2.2.3, l'algèbre de Lie engendrée par E l'est également.

Réciproquement, supposons que l'algèbre de Lie F engendrée par E soit résoluble de classe p et soit

$$\{0\} = F_0 \subset F_1 \subset \ldots \subset F_p = F$$

une suite de résolution de F. On démontre par récurrence sur p que F est trigonalisable (ce qui entraîne *a fortiori* que E est trigonalisable). Si $p = 1$, les matrices de F commutent deux à deux et on est ramené à la proposition 2.2.1.

Supposons à présent $p > 1$. La démonstration repose sur plusieurs lemmes intermédiaires (qui reprennent à peu près les questions de l'énoncé).

LEMME 2.2.6. *Toutes les matrices de F_{p-1} ont un vecteur propre x_0 commun.*

Preuve. Comme F_{p-1} est une algèbre de Lie résoluble de classe $p-1$, l'hypothèse de récurrence montre que F_{p-1} est trigonalisable. Soit P une matrice inversible telle que, pour tout $A \in F_{p-1}$, $P^{-1}AP$ soit triangulaire. Notons $\lambda(A)$ le coefficient de $P^{-1}AP$ situé en première ligne et première colonne et posons $e_1 = {}^t(1, 0, \ldots, 0)$ et $x_0 = Pe_1$. On a alors, pour toute matrice $A \in F_{p-1}$,

$$Ax_0 = APe_1 = P(P^{-1}AP)e_1 = P\lambda(A)e_1 = \lambda(A)Pe_1 = \lambda(A)x_0$$

ce qui démontre le lemme. ∎

Soit $B \in F_p$. On pose, pour tout $k > 0$, $x_k = B^k x_0$ et on note V_B l'espace vectoriel engendré par la famille $(x_k)_{k \geq 0}$. Soit r le plus grand entier tel que les vecteurs $x_0, x_1, \ldots, x_r$ soient linéairement indépendants.

LEMME 2.2.7. *L'espace vectoriel V_B est engendré par la famille $(x_i)_{0 \leq i \leq r}$.*

Preuve. Soit E le sous-espace vectoriel de V_B engendré par les vecteurs $x_0, x_1, \ldots, x_r$. On a $x_{r+1} \in E$ par définition de r. Par récurrence, supposons que x_{r+n} soit élément de E. Alors x_{r+n+1} est combinaison linéaire des vecteurs $Bx_0 = x_1$, $Bx_1 = x_2$, $\ldots$, $Bx_r = x_{r+1}$. Donc $x_{r+n+1} \in E$, ce qui conclut la récurrence. ∎

L'étape suivante consiste à démontrer que la restriction à V_B de toutes les matrices de F_{p-1} est une homothétie.

PROPOSITION 2.2.8. *Pour tout $A \in F_{p-1}$ et pour tout $x \in V_B$, on a $Ax = \lambda(A)x$.*

Preuve. La démonstration se fait en deux étapes. On commence par démontrer que chaque matrice de F_{p-1} définit un endomorphisme de V_B dont la matrice dans la base $(x_i)_{0 \leq i \leq r}$ est triangulaire.

Lemme 2.2.9. *Pour tout $A \in F_{p-1}$ et pour tout entier i, il existe des nombres complexes $\lambda_{i,j}(A)$ $(0 \leq j < i)$ tels que*

$$Ax_i = \lambda(A)x_i + \lambda_{i,i-1}(A)x_{i-1} + \cdots + \lambda_{i,0}(A)x_0$$

Preuve. Le résultat est vrai si $i = 0$, puisque $Ax_0 = \lambda(A)x_0$ par hypothèse. Supposons le résultat acquis jusqu'au rang k et évaluons Ax_{k+1}. Il vient

$$Ax_{k+1} = ABx_k = BAx_k + [A, B]x_k$$

et comme $[A, B] \in F_{p-1}$, on obtient

$$\begin{aligned} Ax_{k+1} &= B(\lambda(A)x_k + \lambda_{k,k-1}(A)x_{k-1} + \cdots + \lambda_{k,0}(A)x_0) \\ &\qquad + \lambda([A, B])x_k + \lambda_{k,k-1}([A, B])x_{k-1} + \cdots + \lambda_{k,0}([A, B])x_0 \\ &= \lambda(A)x_{k+1} + (\lambda_{k,k-1}(A) + \lambda([A, B]))x_{k-1} + \cdots \\ &\qquad + (\lambda_{k,0}(A) + \lambda_{k,1}([A, B]))x_1 + \lambda_{k,0}([A, B])x_0 \end{aligned}$$

d'où le résultat, en posant

$$\lambda_{k+1,i}(A) = \begin{cases} \lambda_{k,i-1}(A) + \lambda_{k,i}([A, B]) & \text{si } i \neq 0 \\ \lambda_{k,0}(A) & \text{si } i = 0 \end{cases} \tag{2.1}$$

■

Le lemme qui suit permet de simplifier considérablement ces formules.

Lemme 2.2.10. *Pour tout $A \in F_{p-1}$, on a $\lambda([A, B]) = 0$.*

Preuve. Le lemme 2.2.9 montre que V_B est stable par toutes les matrices de F_{p-1} et V_B est également stable par B par construction. Complétons la base $\{x_0, x_1, \ldots, x_r\}$ de V_B en une base $\{x_0, x_1, \ldots, x_{n-1}\}$ de $\mathbb{C}^n$ et posons $X = (x_0, \ldots, x_{n-1})$. Quel que soit $A \in F_{p-1}$, la matrice $X^{-1}AX$ peut s'écrire sous la forme

$$\begin{pmatrix} A' & 0 \\ A'' & A''' \end{pmatrix} \quad \text{avec} \quad A' = \begin{pmatrix} \lambda(A) & 0 & \ldots & 0 \\ \lambda_{1,0}(A) & \lambda(A) & \ldots & 0 \\ \vdots & \vdots & \ddots & 0 \\ \lambda_{r,0}(A) & \lambda_{r,1}(A) & \ldots & \lambda(A) \end{pmatrix}$$

De même, la matrice $X^{-1}BX$ s'écrit sous la forme $\left(\begin{smallmatrix} B' & 0 \\ B'' & B''' \end{smallmatrix}\right)$ où B' est une matrice carrée de taille $r + 1$. Rappelons que la *trace* d'une matrice M (notée $\mathrm{tr}(M)$) est la somme de ses coefficients diagonaux. On a ici $\mathrm{tr}(A') = (r + 1)\lambda(A)$ et, puisque $[A, B] \in F_{p-1}$ quel que soit $A \in F_{p-1}$, $\mathrm{tr}([A', B']) = (r + 1)\lambda([A, B])$. Or, si A et B sont deux matrices quelconques de K_n, on a $\mathrm{tr}([A, B]) = 0$ (car $\mathrm{tr}([A, B]) = \mathrm{tr}(AB) - \mathrm{tr}(BA) = 0$). Par conséquent, $\lambda([A, B]) = 0$. ■

Preuve de la proposition 2.2.8. Il suffit, en utilisant les formules 2.1, de prouver par récurrence sur k que, pour tout $A \in F_{p-1}$, $\lambda_{k,i}(A) = 0$ pour $0 \leq i < k$ et, pour cela, de démontrer que $\lambda_{1,0}(A) = 0$. Or

$$[A, B]x_0 = \lambda([A, B])x_0 = \lambda_{1,0}(A)x_0 = 0$$

par le lemme 2.2.10, donc $\lambda_{1,0} = 0$ puisque $x_0 \neq 0$. Le lemme 2.2.9 donne alors simplement $Ax_k = \lambda(A)(x_k)$ pour $0 \leq k \leq r$ et donc $Ax = \lambda(A)x$ pour tout $x \in V_B$. ■

Posons

$$V = \{x \in \mathbb{C}^n \mid Ax = \lambda(A)x \text{ pour tout } A \in F_{p-1}\}$$

PROPOSITION 2.2.11. *L'ensemble V est un espace vectoriel non nul stable par toutes les matrices de F_p.*

Preuve. Par construction, V est un espace vectoriel contenant x_0. D'autre part, si $x \in V$ et $B \in F_p$, on a, d'après le lemme 2.2.10,

$$A(Bx) = BAx + [A, B]x = B\lambda(A)x + \lambda([A, B])x = B\lambda(A)x = \lambda(A)(Bx)$$

ce qui montre que $Bx \in V$. ■

Fin de la preuve du théorème 2.2.5. Soient maintenant B et C deux matrices quelconques de F_p. D'après la proposition 2.2.11, V est stable par B et C. Soit $v_1, \ldots, v_k$ une base de V, que l'on complète en une base $v_1, \ldots, v_n$ de $\mathbb{C}^n$ et posons $P = (v_1, \ldots, v_n)$. On a alors

$$P^{-1}BP = \begin{pmatrix} B' & 0 \\ B'' & B''' \end{pmatrix} \quad \text{et} \quad P^{-1}CP = \begin{pmatrix} C' & 0 \\ C'' & C''' \end{pmatrix}$$

où B' et C' sont des matrices carrées de taille k. Puisque $[B, C] \in F_{p-1}$, on a, par définition de V, $[B, C]v = \lambda([B, C])v$ pour tout $v \in V$. Il en résulte en particulier que $[B', C'] = \lambda([B, C])I_k$, ce qui montre que $\lambda([B, C]) = 0$ (cf. l'argument sur les traces utilisé plus haut). Il en résulte que B' et C' commutent. D'après la proposition 2.2.2, les matrices de F_p ont un vecteur propre commun.

La fin de la démonstration est identique à celle de la proposition 2.2.1 : on raisonne par récurrence sur n, le cas $n = 1$ étant trivial. Soit x un vecteur propre commun aux matrices de F_p et soit X une matrice inversible ayant x comme première colonne. Les matrices $X^{-1}AX$ (pour $A \in F_p$) sont donc de la forme

$$\begin{pmatrix} \mu(A) & 0 \\ A'' & A' \end{pmatrix}$$

avec $\mu(A) \in \mathbb{C}$ et $A' \in K_{n-1}$. Maintenant l'ensemble des matrices A', pour $A \in F$, est une algèbre de Lie de K_{n-1} résoluble de classe p, qui, par hypothèse de récurrence, est trigonalisable. Il existe donc une matrice inversible $P \in K_{n-1}$ telle que toutes les matrices $P^{-1}A'P$ soient triangulaires. Il en résulte que les matrices

$$\begin{pmatrix} 1 & 0 \\ 0 & P^{-1} \end{pmatrix} \begin{pmatrix} \mu(A) & 0 \\ A'' & A' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & P \end{pmatrix}$$

sont triangulaires et donc F est trigonalisable. ■

2.2.3 Une bibliothèque de manipulation des matrices complexes

Comme nous manipulons ici des matrices à coefficients complexes, nous allons décrire une bibliothèque de manipulation de matrices à coefficients complexes qui s'inspire bien entendu de la bibliothèque de manipulation des matrices réelles. Elle fait appel à la bibliothèque de manipulation des complexes et à la bibliothèque générale pour les initialisations.

Nous suivons le même ordre de présentation que pour la bibliothèque relative aux matrices réelles, sans reprendre le détail des commentaires. Commençons par les types :

```
CONST
   OrdreMax = 10;                L'ordre maximal des matrices et vecteurs.
TYPE
   vecC = ARRAY[1..OrdreMax] OF complexe;
   matC = ARRAY[1..OrdreMax] OF vecC;
VAR
   MatriceUnite: matC;           Matrice prédéfinie.
   MatriceNulle: matC;           Matrice prédéfinie.
```

Voici les en-têtes des procédures ou fonctions qui constituent la bibliothèque :

```
PROCEDURE InitMatricesC;
PROCEDURE EntrerMatriceC (n: integer; VAR a: matC; titre: texte);
PROCEDURE EntrerMatriceCRect (m, n: integer; VAR a: matC; titre: texte);
PROCEDURE EntrerVecteurC (n: integer; VAR a: vecC; titre: texte);
PROCEDURE EntrerMatriceHermitienne (n: integer; VAR a: matC; titre: texte);
PROCEDURE EcrireMatriceC (n: integer; a: matC; titre: texte);
PROCEDURE EcrireMatriceCRect (m, n: integer; a: matC; titre: texte);
PROCEDURE EcrireVecteurC (n: integer; a: vecC; titre: texte);
PROCEDURE EcrireMatriceHermitienne (n: integer; a: matC; titre: texte);

PROCEDURE MatriceCPlusMatriceC (n: integer; a, b: matC; VAR ab: matC);
PROCEDURE MatriceCMoinsMatriceC (n: integer; a, b: matC; VAR ab: matC);
PROCEDURE MatriceCParMatriceC (n: integer; a, b: matC; VAR ab: matC);
PROCEDURE MatriceCParMatriceCRect (m, n, p: integer; a, b: matC;
   VAR ab: matC);
PROCEDURE MatriceCParVecteurC (n: integer; a: matC; x: vecC;
   VAR ax: vecC);
PROCEDURE MatriceCRectParVecteurC (m, n: integer; a: matC; x: vecC;
   VAR ax: vecC);
PROCEDURE VecteurCParMatriceC (n: integer; x: vecC; a: matC;
   VAR xa: vecC);
PROCEDURE VecteurCPlusVecteurC (n: integer; a, b: vecC; VAR ab: vecC);
PROCEDURE VecteurCMoinsVecteurC (n: integer; a, b: vecC; VAR ab: vecC);
PROCEDURE VecteurCParScalaire (n: integer; x: vecC; s: real;
   VAR sx: vecC);

PROCEDURE TransposerC (n: integer; a: matC; VAR ta: matC);
PROCEDURE TransposerCRect (m, n: integer; a: matC; VAR ta: matC);
```

```
PROCEDURE Adjointe (n: integer; a: matC; VAR aEtoile: matC);
```
La matrice aEtoile contient la matrice adjointe de la matrice a d'ordre n.

```
FUNCTION NormeC (n: integer; VAR a: vecC): real;
FUNCTION NormeCInfinie (n: integer; VAR a: vecC): real;

PROCEDURE SystemeCTriangulaireSuperieur (n: integer; a: matC; b: vecC;
   VAR x: vecC);
PROCEDURE SystemeCParGauss (n: integer; a: matC; b: vecC;
   VAR x: vecC; VAR inversible: boolean);
PROCEDURE InverseCParGauss (n: integer; a: matC; VAR Inva: matC;
   VAR inversible: boolean);
PROCEDURE SystemeCParJordan (n: integer; a: matC; b: vecC;
   VAR x: vecC; VAR inversible: boolean);
PROCEDURE InverseCParJordan (n: integer; a: matC; VAR Inva: matC;
   VAR inversible: boolean);
```

Voici maintenant le détail de quelques-unes de ces procédures. Elles ne diffèrent que fort peu des procédures utilisées pour les matrices réelles, c'est pourquoi nous n'en présentons qu'un échantillon.

```
PROCEDURE InitMatricesC;
   Définition de la matrice unité et de la matrice nulle.
   VAR
      i, j: integer;
   BEGIN
      InitComplexes;
      FOR i := 1 TO OrdreMax DO          Définition de la matrice nulle.
         FOR j := 1 TO OrdreMax DO
            MatriceNulle[i, j] := ComplexeZero;
      MatriceUnite := MatriceNulle;      Définition de la matrice unité.
      FOR i := 1 TO OrdreMax DO
         MatriceUnite[i, i] := ComplexeUn;
   END; { de "InitMatricesC" }

PROCEDURE EntrerMatriceC (n: integer; VAR a: matC; titre: texte);
   Affichage du titre, puis lecture de la matrice a carrée d'ordre n.
   VAR
      i, j: integer;
   BEGIN
      writeln;
      writeln(titre);
      FOR i := 1 TO n DO BEGIN
         write('Ligne ', i : 1, ' :  ');
         FOR j := 1 TO n DO
            EntrerComplexe(a[i, j], '');
      END;
      readln
   END; { de "EntrerMatriceC" }

PROCEDURE EntrerMatriceCHermitienne (n: integer; VAR a: matC;
```

```
    titre: texte);
```
Affichage du titre, puis lecture de la partie triangulaire inférieure de la matrice hermitienne a d'ordre n. La partie supérieure de a est complétée à la lecture.
```
    VAR
       i, j: integer;
       r: real;
    BEGIN
       writeln;
       writeln(titre);
       FOR i := 1 TO n DO BEGIN
          write('Ligne ', i : 1, ' :  ');
          FOR j := 1 TO i - 1 DO BEGIN
             write('Coef.', i : 2, j : 2, ' : ');
             EntrerComplexe(a[i, j], '');
             Conjugue(a[i, j], a[j, i])
          END;
          write('Coef.', i : 2, i : 2, '(reel) : ');
          read(r);
          ReelEnComplexe(a[i, i], r);
       END;
       readln
    END; { de "EntrerMatriceCHermitienne" }

 PROCEDURE EcrireMatriceC (n: integer; a: matC; titre: texte);
```
Affichage du titre, puis de la matrice a d'ordre n.
```
    VAR
       i, j: integer;
    BEGIN
       writeln;
       writeln(titre);
       FOR i := 1 TO n DO BEGIN
          FOR j := 1 TO n DO
             FormaterComplexe(a[i, j], AligneADroite);
          writeln
       END
    END; { de "EcrireMatriceC" }

 PROCEDURE MatriceCPlusMatriceC (n: integer; a, b: matC; VAR ab: matC);
```
Somme des deux matrices a et b d'ordre n. Résultat dans la matrice ab.
```
    VAR
       i, j: integer;
    BEGIN
       FOR i := 1 TO n DO
          FOR j := 1 TO n DO
             ComplexePlusComplexe(a[i, j], b[i, j], ab[i, j]);
    END; {de "MatriceCPlusMatriceC" }

 PROCEDURE MatriceCParMatriceC (n: integer; a, b: matC;
    VAR ab: matC);
```
Produit des deux matrices a et b d'ordre n. Résultat dans la matrice ab.

```
VAR
   i, j, k: integer;
   somme, produit: complexe;
BEGIN
   FOR i := 1 TO n DO
      FOR j := 1 TO n DO BEGIN
         somme := ComplexeZero;
         FOR k := 1 TO n DO BEGIN
            ComplexeParComplexe(a[i, k], b[k, j], produit);
            ComplexePlusComplexe(somme, produit, somme);
         END;
         ab[i, j] := somme
      END
END; {de "MatriceCParMatriceC" }
```

```
PROCEDURE Adjointe (n: integer; a: matC; VAR aEtoile: matC);
```

La matrice aEtoile contient la matrice adjointe de la matrice a d'ordre n.

```
VAR
   i, j: integer;
BEGIN
   FOR i := 1 TO n DO
      FOR j := 1 TO n DO
         Conjugue(a[j, i], aEtoile[i, j])
END; { de "Adjointe" }
```

```
PROCEDURE SystemeCTriangulaireSuperieur (n: integer; a: matC;
   b: vecC; VAR x: vecC);
```

Solution d'un système triangulaire d'équations $ax = b$ où a est une matrice triangulaire supérieure.

```
VAR
   k, j: integer;
   s, p: complexe;
BEGIN
   FOR k := n DOWNTO 1 DO BEGIN
      s := b[k];
      FOR j := k + 1 TO n DO BEGIN
         ComplexeParComplexe(a[k, j], x[j], p);
         ComplexeMoinsComplexe(s, p, s);
      END;
      ComplexeSurComplexe(s, a[k, k], x[k])
   END;
END; { de "SystemeCTriangulaireSuperieur" }
```

```
PROCEDURE SystemeCParGauss (n: integer; a: matC; b: vecC;
   VAR x: vecC; VAR inversible: boolean);
```

Solution d'un système d'équations linéaires à coefficients complexes par la méthode de Gauss.

```
VAR
   k, q: integer;

FUNCTION PivotMax (k: integer): integer;
```

```
   Cherche le pivot maximal entre k et n.
   VAR
      i, p: integer;
   BEGIN
      p := k;
      FOR i := k + 1 TO n DO
         IF Module(a[i, k]) > Module(a[p, k]) THEN
            p := i;
      PivotMax := p
   END; { de "PivotMax" }
PROCEDURE EchangerFinLignes (i, j: integer);
   VAR
      m: integer;
   BEGIN
      FOR m := i TO n DO
         echangerC(a[i, m], a[j, m]);
      echangerC(b[i], b[j]);
   END; { de "EchangerFinLignes" }
PROCEDURE PivoterGauss (k: integer);
   VAR
      i, j: integer;
      g, z: complexe;
   BEGIN
      FOR i := k + 1 TO n DO BEGIN
         ComplexeSurComplexe(a[i, k], a[k, k], g);          g := a_ik/a_kk
         ComplexeParComplexe(g, b[k], z);
         ComplexeMoinsComplexe(b[i], z, b[i]);              b_i := b_i − g b_k
         FOR j := k + 1 TO n DO BEGIN
            ComplexeParComplexe(g, a[k, j], z);
            ComplexeMoinsComplexe(a[i, j], z, a[i, j]);     a_ij := a_ij − g a_kj
         END;
      END;
   END; { de "PivoterGauss" }
BEGIN { de "SystemeCParGauss" }                               Triangulation.
   k := 1;
   inversible := true;
   WHILE (k <= n) AND inversible DO BEGIN
      q := PivotMax(k);
      inversible := NOT EstCNul(a[q, k]);
      IF inversible THEN BEGIN
         IF q > k THEN
            EchangerFinLignes(k, q);
         PivoterGauss(k);
      END;
      k := k + 1
   END; { du while sur k }                                    Résolution.
   IF inversible THEN
```

```
        SystemeCTriangulaireSuperieur(n, a, b, x);
    END; { de "SystemeCParGauss" }
```

2.2.4 Programme : matrices trigonalisables

Le calcul du crochet de Lie ne pose pas de problème particulier.

```
PROCEDURE CrochetDeLie (n: integer; A, B: matC; VAR ALieB: matC);
   ALieB = AB - BA
   VAR
      AB, BA: matC;
   BEGIN
      MatriceCParMatriceC(n, A, B, AB);
      MatriceCParMatriceC(n, B, A, BA);
      MatriceCMoinsMatriceC(n, AB, BA, ALieB);
   END;  { de "CrochetDeLie" }
```

Avec les exemples de la question 1, on trouve

$$A_1A_2 = \begin{pmatrix} 1-2i & 0 & -2-2i \\ -i & 1 & -1-i \\ -1+i & 0 & 1+2i \end{pmatrix} \qquad A_1A_2 - A_2A_1 = 0$$

$$A_1A_3 - A_3A_1 = \begin{pmatrix} -1+i & 0 & 2i \\ i & 0 & 1+i \\ 1 & 0 & 1-i \end{pmatrix}$$

En réalité, on a $A_1 = P^{-1}B_1P$, $A_2 = P^{-1}B_2P$ et $A_3 = P^{-1}B_3P$ avec

$$P = \begin{pmatrix} -1 & 0 & 1-i \\ 0 & 1 & -i \\ 1+i & i & 2 \end{pmatrix} \qquad P^{-1} = \begin{pmatrix} -1 & -1-i & 1-i \\ i-1 & 0 & -i \\ 1+i & i & -1 \end{pmatrix}$$

$$B_1 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad B_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad B_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

ce qui explique les résultats.

Pour calculer la dimension de l'espace vectoriel (E) engendré par un ensemble fini $E = \{A_1, \ldots, A_k\}$ de matrices de K_n et pour déterminer une base de (E), on transforme chaque matrice A_i en un vecteur v_i de dimension n^2, puis on calcule le rang de la matrice rectangulaire V de taille $k \times n^2$ formée par les v_i. On extrait simultanément de V une famille maximale de lignes linéairement indépendantes, qui fournit donc une base de E. La première opération est réalisée par la procédure suivante :

```
PROCEDURE Linearise (n: integer; A: matC; VAR vA: vecC);
   Transforme une matrice carrée d'ordre n en un vecteur à n^2 composantes.
   VAR
      i, j: integer;
   BEGIN
      FOR i := 0 TO n - 1 DO
         FOR j := 1 TO n DO
            vA[i * n + j] := A[i + 1, j];
   END; { de "Linearise" }
```

Pour le calcul du rang et d'un système maximal de lignes indépendantes, le principe est assez simple. On examine les lignes de la matrice dans leur ordre naturel. Si la ligne k est nulle, on lui substitue la dernière ligne courante et on diminue le nombre de lignes d'une unité. Sinon on utilise la méthode du pivot : on sélectionne le premier élément $a_{k,q}$ non nul de la ligne k. On permute éventuellement les colonnes k et q si $q > k$, puis on pivote.

On fait donc subir à la matrice un certain nombre de permutations des lignes et des colonnes et des opérations de pivot sur les lignes. Comme on veut également obtenir un ensemble maximal de lignes indépendantes, on mémorise les permutations des lignes. En revanche, il est inutile de mémoriser les permutations des colonnes et les opérations de pivot, puisque deux matrices déduites l'une de l'autre par permutation des colonnes ou par opération de pivot sur les lignes ont les mêmes ensembles de lignes indépendantes.

```
PROCEDURE BaseDesLignes (m, n: integer; B: matC; VAR rang: integer;
   VAR LignesIndependantes: vecE);
   Calcule le rang d'une matrice B d'ordre (m, n) et en extrait un ensemble maximal de
   lignes indépendantes.
   VAR
      inversible: boolean;
      A: matC;              Obtenue par permutation des lignes de B.
      Lignes: vecE;         Décrit la permutation des lignes définissant A.
      v: vecC;
      i, j, k, pivot, q: integer;
   FUNCTION ChercherPivot (k: integer): integer;
   Cherche un pivot non nul entre k et n; si le résultat est n + 1, il n'y a pas de pivot non
   nul.
      VAR
         i: integer;
      BEGIN
         i := k;
         WHILE (i <= n) AND EstCNul(a[k, i]) DO         { "and" séquentiel }
            i := i + 1;
         ChercherPivot := i
      END; { de "ChercherPivot" }
   PROCEDURE EchangerFinColonnes (i, j: integer);
      VAR
         m: integer;
```

```
      BEGIN
         FOR m := i TO n DO
            EchangerC(a[m, i], a[m, j]);
      END; { de "EchangerColonnes" }
```

La procédure qui permet l'échange de deux lignes est presque identique à la précédente, mais on prend la précaution de mémoriser la permutation.

```
   PROCEDURE EchangerLignes (i, j: integer);
      VAR
         m: integer;
      BEGIN
         FOR m := i TO n DO
            EchangerC(a[i, m], a[j, m]);
         EchangerE(Lignes[i], Lignes[j]);
      END; { de "EchangerLignes" }
   PROCEDURE PivoterGauss (k: integer);
      VAR
         i, j: integer;
         g, z: complexe;
      BEGIN
         FOR j := k + 1 TO n DO BEGIN
            ComplexeSurComplexe(a[k, j], a[k, k], g);
            FOR i := k + 1 TO n DO BEGIN
               ComplexeParComplexe(g, a[i, k], z);
               ComplexeMoinsComplexe(a[i, j], z, a[i, j]);
            END
         END
      END; { de "PivoterGauss" }
   BEGIN { de "BaseDesLignes" }
      A := B;
      rang := 0;
      FOR i := 1 TO m DO
         Lignes[i] := i;                  Au départ, les lignes sont dans l'ordre.
      inversible := true;
      k := 1;
      WHILE (k <= m) DO BEGIN
         q := ChercherPivot(k);
         inversible := q <> n + 1;
         IF inversible THEN BEGIN
            IF q > k THEN
               EchangerFinColonnes(k, q);
            PivoterGauss(k);
            rang := rang + 1;
            LignesIndependantes[rang] := Lignes[k];
            k := k + 1
         END
         ELSE BEGIN
```

```
            EchangerLignes(k, m);
            m := m - 1
         END
      END; { sur k }
   END; { de "BaseDesLignes" }
```

Enfin, la procédure ci-dessous donne la dimension de l'espace vectoriel engendré par un ensemble fini de matrices.

```
FUNCTION dim (CardE, n: integer; E: EnsembleDeMatrices;
   VAR BaseE: EnsembleDeMatrices): integer;
```

Calcule la dimension de l'espace vectoriel engendré par un ensemble E de $CardE$ matrices et extrait de E une base de cet espace vectoriel.

```
   VAR
      i, r: integer;
      L: vecE;
      ELinearise: matC;
   BEGIN
      FOR i := 1 TO CardE DO
         Linearise(n, E[i], ELinearise[i]);
      BaseDesLignes(CardE, n * n, ELinearise, r, L);
      FOR i := 1 TO r DO
         BaseE[i] := E[L[i]];
      dim := r;
   END; {de "dim"}
```

Pour calculer la dimension de l'algèbre de Lie engendrée par un ensemble fini de matrices E, on calcule, pour chaque entier k, une base B_k de l'espace vectoriel E_k défini dans l'énoncé. Pour obtenir B_0, il suffit d'extraire une base de E. Pour passer de B_k à B_{k+1}, on calcule l'ensemble de tous les crochets de Lie des matrices de B_k, puis on extrait une base de cet ensemble. Lorsque les dimensions de B_k et de B_{k+1} sont égales, le calcul est terminé.

```
PROCEDURE LesCrochetsDeLie (VAR CardB: integer; n: integer;
   VAR B: EnsembleDeMatrices);
   VAR
      i, j, k: integer;
   BEGIN
      k := CardB;
      FOR i := 1 TO CardB DO
         FOR j := i + 1 TO CardB DO BEGIN
            k := k + 1;                  On ajoute tous les crochets de Lie.
            CrochetDeLie(n, B[i], B[j], B[k])
         END;
      CardB := k
   END; { de "LesCrochetsDeLie" }
PROCEDURE AlgebreDeLie (CardE, n: integer; E: EnsembleDeMatrices);
```

Calcule la dimension de l'algèbre de Lie engendrée par un ensemble E de $CardE$ matrices d'ordre n.

```
VAR
   i, CardB, dimE: integer;
   B: EnsembleDeMatrices;
BEGIN
   CardB := dim(CardE, n, E, B);
   REPEAT
      dimE := CardB;
      LesCrochetsDeLie(CardB, n, B);
      CardB := dim(CardB, n, B, B);
   UNTIL dimE = CardB;
   Si le rang n'a pas augmenté, on a l'algèbre de Lie engendrée par E.
   writeln('Dimension de l''algèbre de Lie comme espace vectoriel = ',
           CardB : 1);
   writeln('Base de l''algèbre de Lie comme espace vectoriel : ');
   FOR i := 1 TO CardB DO
      EcrireMatriceC(n, B[i], '');
END; { de "AlgebreDeLie" }
```

Pour terminer, voici quelques exemples d'exécution :

```
Voici A1 :
                  - i                      0                 - 2.0
                    0                    1.0                     0
                    i                      0               2.0 + i
rang(A_1) = 3
Voici A2 :
              2.0 - i                      0               - 2.0 i
                  - i                    1.0           - 1.0 - i
                - 1.0                      0                     i
rang(A_2) = 3
Voici A3 :
                  1.0                1.0 - i           - 1.0 - i
                    0                1.0 - i               - 1.0
                    0                  - 1.0             1.0 + i
rang(A_3) = 3
Voici A1A2 :
          1.0 - 2.0 i                      0         - 2.0 - 2.0 i
                  - i                    1.0           - 1.0 - i
            - 1.0 + i                      0           1.0 + 2.0 i
rang (A1A2) = 3
Voici [A1,A2] :
                    0                      0                     0
                    0                      0                     0
                    0                      0                     0
rang ([A1,A2]) = 0
Voici [A1,A3] :
            - 1.0 + i                      0                 2.0 i
```

```
                 i                      0                1.0 + i
               1.0                      0                1.0 - i
rang(A_1A_3) = 1
Dimension de l'espace engendré par E = 3
Dimension de l'algèbre de Lie comme espace vectoriel = 4
Base de l'algèbre de Lie comme espace vectoriel :
               - i                      0                  - 2.0
                 0                    1.0                      0
                 i                      0                2.0 + i
           2.0 - i                      0                - 2.0 i
               - i                    1.0            - 1.0 - i
             - 1.0                      0                      i
               1.0                1.0 - i            - 1.0 - i
                 0                1.0 - i                  - 1.0
                 0                  - 1.0                1.0 + i
       - 1.0 + i                        0                  2.0 i
                 i                      0                1.0 + i
               1.0                      0                1.0 - i
```

Notes bibliographiques

Les pseudo-inverses sont fréquemment définies pour les matrices complexes, la transposée étant remplacée par l'adjointe. Parmi les monographies consacrées aux pseudo-inverses, citons :

R. M. Pringle, A. A. Rayner, *Generalized Inverse Matrices*, London, Griffin, 1971.

Une présentation brève est donnée par :

J. Stoer, *Numerische Mathematik 1*, Berlin, Springer-Verlag, 1989.

Chapitre 3

Décompositions

Une *décomposition* d'une matrice carrée A est un couple (B, C) de deux matrices carrées B et C telles que $A = BC$. Lorsque ces matrices ont une forme particulière, certaines opérations, comme la résolution de systèmes linéaires, s'en trouvent simplifiées. Dans ce chapitre, nous considérons trois décompositions : la décomposition LU, la décomposition de Choleski et la décomposition QR. On utilisera aussi ces décompositions pour le calcul des valeurs propres.

3.1 Décomposition LU

Une *décomposition* LU d'une matrice carrée A d'ordre n est un couple (L, U) de matrices carrées d'ordre n, avec L unitriangulaire inférieure, c'est-à-dire ayant des 1 sur la diagonale, et U triangulaire supérieure, telles que

$$A = LU$$

L'intérêt de disposer d'une telle décomposition est clair. La résolution d'un système d'équations linéaires

$$Ax = b$$

se ramène alors à la résolution des deux systèmes triangulaires

$$Ly = b \qquad Ux = y$$

ce qui est particulièrement simple. De la même manière, si A est inversible, l'inverse $A^{-1} = U^{-1}L^{-1}$ s'obtient en inversant deux matrices triangulaires, ce qui est facile. En fait, une matrice quelconque n'admet pas nécessairement une décomposition LU.

Proposition 3.1.1. *Une matrice inversible A admet une décomposition LU si et seulement si tous ses mineurs principaux sont inversibles; de plus, la décomposition LU est alors unique.*

Preuve. Supposons que A ait deux décompositions $A = LU = L'U'$. Comme A est inversible, L et L' le sont. Alors $L^{-1}L' = UU'^{-1}$. La matrice $L^{-1}L'$ est triangulaire inférieure et UU'^{-1} est triangulaire supérieure. L'égalité implique qu'elles sont toutes deux diagonales et la première, donc aussi la deuxième, est en fait la matrice unité, montrant que $L = L'$ et $U = U'$. Ceci prouve l'unicité, qui d'ailleurs découle aussi de la preuve d'existence qui suit.

L'existence se montre par récurrence sur l'ordre n de A, le cas $n = 1$ étant évident. Considérons alors la décomposition par blocs

$$A = \begin{pmatrix} A' & b \\ b' & a \end{pmatrix}$$

où A' est d'ordre $n-1$. Par récurrence, A' possède une décomposition LU, soit (L', U'). Cherchons alors deux matrices d'ordre n

$$L = \begin{pmatrix} L' & 0 \\ v & 1 \end{pmatrix} \qquad U = \begin{pmatrix} U' & u \\ 0 & r \end{pmatrix}$$

avec $LU = A$. On obtient les équations

$$L'u = b \qquad vU' = b' \qquad vu + r = a$$

Les deux premières permettent de déterminer u et v de façon unique parce que L' et U' sont inversibles et la troisième donne r de façon unique. D'où l'existence.

Réciproquement, si $A = LU$, alors U est inversible et en particulier le produit de ses éléments diagonaux est non nul. Soit $1 \leq k \leq n$ et soit A' le mineur principal de A d'ordre k. Considérons la décomposition par blocs

$$L = \begin{pmatrix} L' & 0 \\ X & L'' \end{pmatrix} \qquad U = \begin{pmatrix} U' & Y \\ 0 & U'' \end{pmatrix}$$

où L' et U' sont d'ordre k. Alors $L'U' = A'$ a un déterminant non nul. ∎

Il n'est pas difficile de vérifier que, si A possède deux décompositions (L, U) et (L', U'), alors on a toujours $U = U'$. En revanche, l'égalité entre L et L' ne peut être assurée; considérer par exemple le cas $U = 0$.

Toute matrice inversible A peut être, par une permutation de ses lignes, transformée en une matrice dont les mineurs principaux sont inversibles. Toute matrice inversible possède donc, à une permutation de ses lignes près, une décomposition LU. La méthode appelée *méthode de Crout* ou *méthode de Gauss modernisée* est un algorithme pour obtenir cette décomposition. Elle consiste en fait en une organisation judicieuse des calculs faits lors de la méthode de Gauss.

Dans la méthode de Gauss, une opération de pivotage peut être interprétée comme la multiplication à gauche par une matrice de Frobenius G_k qui, dans ce cas, est unitriangulaire inférieure. S'il n'y a pas d'échange de lignes, une matrice A d'ordre n est transformée en

$$G_{n-1} \cdots G_2 G_1 A = U$$

la matrice U est triangulaire supérieure et $G = G_{n-1} \cdots G_2 G_1$ est unitriangulaire inférieure. On a donc

$$A = LU$$

avec $L = G^{-1}$. En d'autres termes, la méthode de Gauss contient de façon cachée une méthode de décomposition LU.

S'il y a échange de lignes dans la méthode de Gauss, on a en fait

$$G_{n-1} P_{n-1} \cdots G_2 P_2 G_1 P_1 A = U$$

où les P_k sont des matrices de permutations. Or

$$G_{n-1} P_{n-1} \cdots G_2 P_2 G_1 P_1 = P H_{n-1} \cdots H_2 H_1$$

avec $P = P_{n-1} \cdots P_2 P_1$ et

$$H_k = (P_k \cdots P_1)^{-1} G_k P_k \cdots P_1 \qquad 1 \leq k < n$$

de sorte que la matrice $H = P H_{n-1} \cdots H_2 H_1 P^{-1}$ est unitriangulaire inférieure et $HPA = U$; avec $L = H^{-1}$, on a

$$A = LU$$

Pour la réalisation, nous partons de l'équation $A = LU$, avec $A = (a_{i,j})$, $L = (\ell_{i,j})$, $U = (u_{i,j})$. On suppose $\ell_{i,j} = 0$ pour $j > i$ et $\ell_{i,i} = 1$ et de même $u_{i,j} = 0$ pour $i > j$. On a alors, en identifiant

$$a_{i,k} = \sum_{j=1}^{\min(i,k)} \ell_{i,j} u_{j,k}$$

d'où l'on tire

$$\begin{aligned} \ell_{i,k} &= (a_{i,k} - \sum_{j=1}^{k-1} \ell_{i,j} u_{j,k}) / u_{k,k} \qquad && i > k \\ u_{i,k} &= a_{i,k} - \sum_{j=1}^{i-1} \ell_{i,j} u_{j,k} && i \leq k \end{aligned}$$

Il y a plusieurs stratégies pour résoudre ces équations : on peut procéder ligne par ligne, colonne par colonne, en diagonale, etc. On peut également remarquer que chaque coefficient $a_{i,k}$ n'est utilisé qu'une fois; on peut donc stocker, à la place de $a_{i,k}$, le coefficient $\ell_{i,k}$, si $i > k$, et $u_{i,k}$ quand $i \leq k$ (les éléments diagonaux de L n'ont pas besoin d'être conservés).

Pour pouvoir éliminer un pivot $u_{k,k}$ nul, on constate que les formules donnant $u_{k,k}$ et les $\ell_{i,k}$ sont, à la division près, formellement les mêmes. Ceci montre que tout $\ell_{i,k}$ pour $i > k$ peut jouer le rôle de $u_{k,k}$. On adopte donc la démarche suivante : on calcule, pour chaque colonne k, d'abord des éléments $\ell'_{i,k}$ et $u_{i,k}$ par les formules

$$\begin{aligned} u_{i,k} &= a_{i,k} - \sum_{j=1}^{i-1} \ell_{i,j} u_{j,k} \quad & i \le k \\ \ell'_{i,k} &= a_{i,k} - \sum_{j=1}^{k-1} \ell_{i,j} u_{j,k} \quad & i > k \end{aligned}$$

puis l'on cherche un pivot non nul et on fait la division des $\ell'_{i,k}$ par le pivot. Voici une réalisation :

```
PROCEDURE DecompositionLU (n: integer; VAR a: mat; VAR p: vecE;
   VAR inversible: boolean);
   Calcule, dans a, la décomposition LU de a, à une permutation des lignes près. Cette
   permutation est dans le vecteur p. Si a n'est pas inversible, le booléen est mis à faux,
   mais une partie de a peut être détruite.
   VAR
      k, q: integer;

   FUNCTION PivotMax (k: integer): integer;
      Cherche le pivot maximal entre k et n.
      VAR
         i, p: integer;
      BEGIN
         p := k;
         FOR i := k + 1 TO n DO
            IF abs(a[i, k]) > abs(a[p, k]) THEN
               p := i;
         PivotMax := p
      END; { de "PivotMax" }

   PROCEDURE EchangerLignes (i, j: integer);
      VAR
         m: integer;
      BEGIN
         FOR m := 1 TO n DO echanger(a[i, m], a[j, m]);
      END; { de "EchangerLignes" }

   PROCEDURE Decomposer (k: integer);
      VAR
         i, j: integer;
         somme: real;
      BEGIN
         FOR i := 1 TO n DO BEGIN
            somme := a[i, k];
            FOR j := 1 TO min(i, k) - 1 DO
               somme := somme - a[i, j] * a[j, k];
            a[i, k] := somme
         END
```

```
      END; { de "Decomposer" }

   PROCEDURE Diviser (k: integer);
      VAR
         i: integer;
      BEGIN
         FOR i := k + 1 TO n DO
            a[i, k] := a[i, k] / a[k, k]
      END; { de "Diviser" }

   BEGIN  { de "DecompositionLU" }
      FOR k := 1 TO n DO p[k] := k;
      k := 1;
      inversible := true;
      WHILE (k <= n) AND inversible DO BEGIN
         Decomposer(k);
         q := PivotMax(k);
         inversible := NOT EstNul(a[q, k]);
         IF inversible THEN BEGIN
            IF q > k THEN BEGIN
               EchangerLignes(k, q);
               echangerE(p[q], p[k])
            END;
            Diviser(k)
         END;
         k := k + 1
      END; { du while sur k }
   END; { de "DecompositionLU" }
```

Voici un exemple de décomposition :

```
Voici la matrice lue
   4.000   2.000   1.000   0.000
   2.000   5.000   3.000   5.000
   1.000   1.000   1.000   1.000
   2.000   1.000   4.000   1.000

Matrice L
   1.000   0.000   0.000   0.000
   0.500   1.000   0.000   0.000
   0.500   0.000   1.000   0.000
   0.250   0.125   0.125   1.000

Matrice U
   4.000   2.000   1.000   0.000
   0.000   4.000   2.500   5.000
   0.000   0.000   3.500   1.000
   0.000   0.000   0.000   0.250

Vecteur de permutation des lignes
  1  2  4  3
```

```
Matrice LU
   4.000   2.000   1.000   0.000
   2.000   5.000   3.000   5.000
   2.000   1.000   4.000   1.000
   1.000   1.000   1.000   1.000
```

Les lignes 3 et 4 sont en effet permutées. Comme il est déjà dit plus haut, la décomposition *LU* facilite la résolution d'un système linéaire. Voici une réalisation :

```
PROCEDURE SystemeParCrout (n: integer; a: mat; b: vec; VAR x: vec;
   VAR inversible: boolean);
   Résolution du système d'équations linéaire Ax = b par la méthode de Crout. Si A est
   inversible, x contient la solution, sinon le booléen est faux.
   VAR
      p: vecE;
   PROCEDURE Permuter (b: vec; p: vecE; VAR bp: vec);
      VAR
         k: integer;
      BEGIN
         FOR k := 1 TO n DO bp[k] := b[p[k]];
      END; { de "Permuter" }
   BEGIN
      DecompositionLU(n, a, p, inversible);
      IF inversible THEN BEGIN
         Permuter(b, p, b);
         SystemeUniTriangulaireInferieur(n, a, b, x);
         SystemeTriangulaireSuperieur(n, a, x, x);
      END
   END; { de "SystemeParCrout" }
```

La procédure `SystemeTriangulaireSuperieur` a déjà été présentée au chapitre 1; la procédure `SystemeUniTriangulaireInferieur` ci-dessous tient en plus compte du fait que les éléments diagonaux valent 1 pour ne pas faire de division :

```
PROCEDURE SystemeUniTriangulaireInferieur (n: integer; a: mat; b: vec;
   VAR x: vec);
   Résolution d'un système unitriangulaire inférieur.
   VAR
      k, j: integer;
      s: real;
   BEGIN
      FOR k := 1 TO n DO BEGIN
         s := b[k];
         FOR j := 1 TO k - 1 DO s := s - a[k, j] * x[j];
         x[k] := s                 Pas de division : le coefficient diagonal est 1.
      END;
   END; { de "SystemeUniTriangulaireInferieur" }
```

3.2 Décomposition de Choleski

La décomposition de Choleski ne s'applique qu'à des matrices particulières, à savoir des matrices symétriques définies positives (resp. hermitiennes définies positives). Elle repose sur le résultat de la proposition suivante.

PROPOSITION 3.2.1. *Soit A une matrice symétrique définie positive d'ordre n. Il existe une matrice triangulaire inférieure L à coefficients diagonaux strictement positifs et une seule telle que $A = L\,{}^tL$.*

Preuve. Par récurrence sur n. Si $n = 1$, la matrice est réduite à une entrée positive α et L a comme seul élément le nombre $\sqrt{\alpha}$. Si $n > 1$, posons

$$A = \begin{pmatrix} B & b \\ {}^tb & a \end{pmatrix}$$

avec $B = L_{n-1}{}^tL_{n-1}$ et $L_{n-1} = (\ell_{i,j})$, $\ell_{i,i} > 0$ et $\ell_{i,j} = 0$ pour $j > i$, et soit

$$L = \begin{pmatrix} L_{n-1} & 0 \\ {}^tc & \alpha \end{pmatrix}$$

De $L\,{}^tL = A$, on tire

$$L_{n-1}c = b$$

qui a la solution unique $c = L_{n-1}^{-1}b$, parce que les éléments diagonaux de L_{n-1} sont strictement positifs, donc L_{n-1} est inversible. Or A est définie positive, donc $\mathrm{dét}(A) > 0$. Comme

$$\mathrm{dét}(A) = \mathrm{dét}(L_{n-1})^2\alpha^2 > 0$$

on a $\alpha^2 > 0$ et il y a une solution $\alpha > 0$ unique. ■

De l'équation $A = L\,{}^tL$ on tire, par identification, des formules qui permettent d'obtenir la matrice L. Ces formules sont

$$\begin{aligned} \ell_{i,i}^2 &= a_{i,i} - \sum_{k=1}^{i-1} \ell_{i,k}^2 \\ \ell_{i,j} &= \left(a_{i,j} - \sum_{k=1}^{j-1} \ell_{i,k}\ell_{k,j} \right) / \ell_{i,i} \qquad i > j \end{aligned}$$

Voici une réalisation :

```
PROCEDURE Choleski (n: integer; a: mat; VAR l: mat);
   a est une matrice symétrique définie positive. Calcul de ℓ tel que a = ℓ tℓ.
   VAR
      i, j, k: integer;
      s: real;
   BEGIN
```

```
      FOR i := 1 TO n DO BEGIN
         FOR j := 1 TO i - 1 DO BEGIN        Calcul des éléments non diagonaux.
            s := a[i, j];
            FOR k := 1 TO j - 1 DO
               s := s - l[i, k] * l[j, k];
            l[i, j] := s / l[j, j]
         END;
         s := a[i, i];                       Calcul de l'élément diagonal.
         FOR k := 1 TO i - 1 DO
            s := s - sqr(l[i, k]);
         l[i, i] := sqrt(s)
      END  { de la ligne i }
   END; { de "Choleski" }
```

L'inverse d'une matrice ainsi décomposée est particulièrement facile à calculer, puisqu'il suffit de calculer l'inverse de la matrice triangulaire L, par exemple comme suit :

```
PROCEDURE InverseTriangulaireInferieure (n: integer; a: mat; VAR b: mat);
   a est une matrice triangulaire inférieure inversible. On calcule son inverse dans b, par
   résolution des systèmes triangulaires.
   VAR
      i, j, k: integer;
      s: real;
   BEGIN
      b := MatriceNulle;
      FOR i := 1 TO n DO                     Eléments diagonaux.
         b[i, i] := 1 / a[i, i];
      FOR i := 2 TO n DO                     Eléments hors diagonaux.
         FOR j := 1 TO i - 1 DO BEGIN
            s := 0;
            FOR k := j TO i - 1 DO s := s + a[i, k] * b[k, j];
            b[i, j] := -s * b[i, i]
         END
   END; { de "InverseTriangulaireInferieure" }
```

Ensuite, on fait le produit de cette matrice par sa transposée. Ainsi on obtient la procédure suivante :

```
PROCEDURE InverseDefPositiveSymetrique (n: integer; a: mat; VAR ia: mat);
   Calcule l'inverse ia d'une matrice symétrique définie positive a.
   VAR
      k, tk, l: mat;
   BEGIN
      Choleski(n, a, l);                            a = ℓ tℓ
      InverseTriangulaireInferieure(n, l, k);       k = ℓ^-1
      Transposer(n, k, tk);
      MatriceParMatrice(n, tk, k, ia)               a^-1 = (ℓ tℓ)^-1 = tk k
   END;{ de "InverseDefPositiveSymetrique" }
```

3.3 Décomposition QR

Toute matrice carrée réelle A est, comme nous le verrons, produit d'une matrice orthogonale Q par une matrice triangulaire supérieure R. Une telle factorisation est appelée une *décomposition QR*. Il existe plusieurs méthodes pour obtenir une telle décomposition. La plus connue est le procédé d'orthogonalisation de Gram-Schmidt. Nous exposons ici deux autres méthodes qui sont numériquement plus stables : la méthode de Givens et la méthode de Householder. On verra, dans le chapitre suivant, comment les mêmes procédés permettent de mettre une matrice sous forme tridiagonale.

3.3.1 Enoncé : décomposition QR (méthode de Givens)

Soit $n \geq 1$ un entier. Les matrices considérées sont réelles, carrées d'ordre n. Pour p, q dans $\{1, \ldots, n\}$, $p \neq q$, on note $G(q,p) = (g_{i,j})$ toute matrice de rotation de la forme

$$G(q,p) = \begin{pmatrix} 1 & & & & & & 0 \\ & \ddots & & & & & \\ & & c & & s & & \\ & & & \ddots & & & \\ & & -s & & c & & \\ & & & & & \ddots & \\ 0 & & & & & & 1 \end{pmatrix}$$

avec $c^2 + s^2 = 1$, qui ne diffère de la matrice unité que par les éléments d'indice (p,p), (p,q), (q,p), (q,q). Plus précisément,

$$\begin{array}{ll} g_{i,i} = 1 & i \neq p \text{ et } i \neq q \\ g_{i,j} = 0 & i \neq j, \quad (i,j) \neq (p,q) \text{ et } (i,j) \neq (q,p) \\ g_{p,p} = g_{q,q} = c & \\ g_{p,q} = -g_{q,p} = s & \end{array}$$

1.– Etant donnés une matrice A et $p < q$, montrer qu'il existe c et s tels que l'élément $b_{q,p}$ de la matrice $B = (b_{i,j}) = G(q,p)A$ soit nul.

2.– On définit une suite de matrices $B^{(q,p)} = (b_{i,j}^{(q,p)})$ pour $1 \leq p \leq q \leq n$ par

$$\begin{array}{ll} B^{(1,1)} = A & \\ B^{(q,p)} = G(q,p)B^{(q-1,p)} & (1 \leq p < q \leq n) \\ B^{(p,p)} = B^{(n,p-1)} & (2 \leq p \leq n) \end{array}$$

où $G(q,p)$ est construit de manière telle que $b_{q,p}^{(q,p)} = 0$. On pose $R = B^{(n,n-1)}$. Montrer que R est une matrice triangulaire et qu'elle s'écrit sous la forme $R = UA$ où U est une matrice orthogonale.

3.– Ecrire un programme qui prend en argument une matrice A et qui calcule U et R comme définis ci-dessus. Ajouter une procédure qui calcule le déterminant de A.

4.– Compléter ce programme pour qu'ensuite, il lise un vecteur b et donne la solution du système $Ax = b$, la matrice A étant supposée inversible.

Exemple numérique :

$$A = \begin{pmatrix} 4 & 1 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 \\ 0 & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 1 & 4 \end{pmatrix} \qquad b = \begin{pmatrix} 5 \\ 6 \\ 6 \\ 6 \\ 5 \end{pmatrix}$$

5.– On appelle *décomposition QR* d'une matrice A un couple formé d'une matrice orthogonale Q et d'une matrice triangulaire supérieure R telles que $A = QR$. Montrer que si A est inversible, alors la décomposition QR de A est unique si les signes des éléments diagonaux de R sont fixés.

3.3.2 Solution : décomposition QR (méthode de Givens)

La méthode de Givens de calcul d'une décomposition QR d'une matrice A consiste à prémultiplier A par une suite de matrices de rotation qui annulent successivement les coefficients en dessous de la diagonale de A. La matrice obtenue est donc triangulaire supérieure et le produit des matrices de rotation est l'inverse de la matrice orthogonale cherchée.

Soit donc A une matrice carrée. Pour p, q dans $\{1, \ldots, n\}$, $p \neq q$, on note $G(q, p) = (g_{i,j})$ toute matrice de rotation définie par

$$\begin{array}{ll} g_{i,i} = 1 & i \neq p \text{ et } i \neq q \\ g_{i,j} = 0 & i \neq j, \quad (i,j) \neq (p,q) \text{ et } (i,j) \neq (q,p) \\ g_{p,p} = g_{q,q} = c & \\ g_{p,q} = -g_{q,p} = s & \end{array}$$

avec $c^2 + s^2 = 1$. Soit $p < q$. Pour que l'élément

$$b_{q,p} = -s a_{p,p} + c a_{q,p}$$

de la matrice $B = (b_{i,j}) = G(q, p)A$ soit nul, il suffit de définir (c, s) par

$$\begin{array}{lll} c = 1 & s = 0 & \text{si } a_{p,p}^2 + a_{q,p}^2 = 0 \\ c = a_{p,p}/r & s = a_{q,p}/r & \text{si } a_{p,p}^2 + a_{q,p}^2 = r^2 \neq 0 \end{array}$$

On définit une suite de matrices $B^{(q,p)} = (b_{i,j}^{(q,p)})$ pour $1 \leq p \leq q \leq n$ par

$$\begin{array}{ll} B^{(1,1)} = A & \\ B^{(q,p)} = G(q,p)B^{(q-1,p)} & (1 \leq p < q \leq n) \\ B^{(p,p)} = B^{(n,p-1)} & (2 \leq p \leq n) \end{array}$$

où $G(q,p)$ est construit de manière telle que $b^{(q,p)}_{q,p} = 0$.

LEMME 3.3.1. *On a* $b^{(q,p)}_{i,j} = 0$ *pour tout* $i > j$ *tels que* ($j < p$ *et* $1 \leq i \leq n$) *ou* ($j = p$ *et* $p < i \leq q$).

Preuve. Soient p, q avec $1 \leq p < q \leq n$. La matrice $B^{(q,p)}$ ne diffère de $B^{(q-1,p)}$ que par les éléments des lignes d'indice p et q et on a, pour $j = 1, \ldots, n$:

$$\begin{aligned} b^{(q,p)}_{p,j} &= g_{p,p} b^{(q-1,p)}_{p,j} + g_{p,q} b^{(q-1,p)}_{q,j} \\ b^{(q,p)}_{q,j} &= g_{q,p} b^{(q-1,p)}_{p,j} + g_{q,q} b^{(q-1,p)}_{q,j} \end{aligned}$$

En raisonnant par récurrence, on a $b^{(q,p)}_{i,j} = 0$ si $b^{(q-1,p)}_{i,j} = 0$ et si $j < p$, ou si $j = p$ et $p < i < q$. Mais par construction, $b^{(q,p)}_{q,p} = 0$, d'où le lemme. ∎

COROLLAIRE 3.3.2. *La matrice* $R = B^{(n,n-1)}$ *est triangulaire supérieure.* ∎

PROPOSITION 3.3.3. *Toute matrice carrée* A *admet une décomposition* $A = QR$, *où* Q *est une matrice orthogonale et* R *est une matrice triangulaire supérieure.*

Preuve. On a, avec les notations précédentes,

$$R = G(n, n-1)G(n, n-2)G(n-1, n-2) \cdots G(n, 1)G(n-1, 1) \cdots G(2, 1)A$$

Notons U le produit des matrices de rotation. Alors U est orthogonale et $A = {}^tUR$. ∎

PROPOSITION 3.3.4. *Soit* A *une matrice inversible. Alors la décomposition* QR *de* A *est unique si les signes des éléments diagonaux de* R *sont fixés.*

Preuve. Supposons que A admette deux décompositions :

$$A = QR = Q'R'$$

Comme A est inversible, R et R' le sont et on a donc $Q^{-1}Q' = RR'^{-1}$. La matrice $Q^{-1}Q'$ est orthogonale et triangulaire supérieure, donc c'est une matrice diagonale. De plus, les éléments diagonaux de $Q^{-1}Q'$ sont de valeur absolue égale à 1. Posons $R = (r_{i,j})$ et $R' = (r'_{i,j})$. Alors $r_{i,i}/r'_{i,i} = \pm 1$. Par hypothèse, les signes des éléments diagonaux de R et de R' sont les mêmes, donc les quotients valent 1. Il en résulte que $Q^{-1}Q'$ est la matrice identité, donc que RR'^{-1} l'est également, et $R = R'$. ∎

3.3.3 Programme : décomposition QR (méthode de Givens)

La détermination des coefficients des matrices de rotation $G(q,p)$ se traduit par la procédure que voici :

```
PROCEDURE CoefficientsGivens (p, q: integer; VAR c, s: real; VAR a: mat);
   Calcul des réels c et s tels que b(q, p) = 0.
   VAR
      norme: real;
   BEGIN
      norme := sqrt(sqr(a[p, p]) + sqr(a[q, p]));
      IF EstNul(norme) THEN BEGIN
         c := 1;
         s := 0
      END
      ELSE BEGIN
         c := a[p, p] / norme;
         s := a[q, p] / norme
      END
   END; { de "CoefficientsGivens" }
```

On peut construire les matrices $G(q, p)$ explicitement puis faire le produit des matrices. Mais comme cette matrice ne modifie que peu de coefficients, il est plus économique de faire ces quelques opérations directement. On a alors :

```
PROCEDURE Premultiplier (n, p, q: integer; c, s: real; VAR a: mat);
   Prémultiplication de la matrice a par la matrice de rotation G(q, p).
   VAR
      j: integer;
      v, w: real;
   BEGIN
      FOR j := 1 TO n DO BEGIN
         v := a[p, j]; w := a[q, j];
         a[p, j] := c * v + s * w;
         a[q, j] := -s * v + c * w;
      END
   END; { de "Premultiplier" }
```

La méthode de Givens consiste à itérer cette opération pour les couples (p, q), avec $1 \leq p < q \leq n$. La matrice obtenue, si l'on part de A, est la matrice R. Si l'on fait le produit des matrices de transformation orthogonales, on obtient une matrice U telle que $R = UA$ et la matrice cherchée Q est égale à $U^{-1} = {}^tU$. Les deux procédures suivantes réalisent ces calculs.

```
PROCEDURE IterationGivens (n: integer; VAR a, u: mat);
   On annule successivement les coefficients d'indice (p, q), pour 1 ≤ p < q ≤ n. Le produit
   des matrices de rotation est l'inverse de la matrice orthogonale cherchée.
   VAR
      p, q: integer;
      c, s: real;
   BEGIN
      FOR p := 1 TO n - 1 DO
         FOR q := p + 1 TO n DO BEGIN
            CoefficientsGivens(p, q, c, s, a);
```

```
            Premultiplier(n, p, q, c, s, u);
            Premultiplier(n, p, q, c, s, a);
         END
   END; { de "IterationGivens" }

PROCEDURE DecompositionQRparGivens (n: integer; a: mat; VAR q, r: mat);
   Décomposition QR de la matrice a par la méthode de Givens.
   BEGIN
      r := a;
      q := MatriceUnite;          Le couple de matrices (A, I) est transformé
      IterationGivens(n, r, q);   en un couple (R, Q), puis Q est transposée
      Transposer(n, q, q)         pour donner la matrice cherchée.
   END; {de "DecompositionQRparGivens" }
```

Voici un exemple d'exécution. Les impressions intermédiaires montrent comment la matrice de départ est progressivement transformée en matrice triangulaire supérieure.

```
Voici la matrice a
   4.000   2.000   1.000   0.000
   2.000   5.000   3.000   5.000
   1.000   1.000   1.000   1.000
   2.000   1.000   4.000   1.000

Matrice a pour p = 1, q = 2
   4.472   4.025   2.236   2.236
   0.000   3.578   2.236   4.472
   1.000   1.000   1.000   1.000
   2.000   1.000   4.000   1.000
Matrice a pour p = 1, q = 3
   4.583   4.146   2.400   2.400
   0.000   3.578   2.236   4.472
   0.000   0.098   0.488   0.488
   2.000   1.000   4.000   1.000
Matrice a pour p = 1, q = 4
   5.000   4.200   3.800   2.600
   0.000   3.578   2.236   4.472
   0.000   0.098   0.488   0.488
  -0.000  -0.742   2.706  -0.044

Matrice a pour p = 2, q = 3
   5.000   4.200   3.800   2.600
   0.000   3.579   2.249   4.484
   0.000  -0.000   0.427   0.366
  -0.000  -0.742   2.706  -0.044
Matrice a pour p = 2, q = 4
   5.000   4.200   3.800   2.600
   0.000   3.655   1.652   4.399
   0.000  -0.000   0.427   0.366
  -0.000  -0.000   3.106   0.867
```

```
Matrice a pour p = 3, q = 4
   5.000   4.200   3.800   2.600
   0.000   3.655   1.652   4.399
  -0.000  -0.000   3.135   0.909
  -0.000   0.000  -0.000  -0.244
Décomposition QR de a :
Voici la matrice q
   0.800  -0.372  -0.455   0.122
   0.400   0.908  -0.007   0.122
   0.200   0.044   0.053  -0.977
   0.400  -0.186   0.889   0.122
Voici la matrice r
   5.000   4.200   3.800   2.600
   0.000   3.655   1.652   4.399
  -0.000  -0.000   3.135   0.909
  -0.000   0.000  -0.000  -0.244
Vérification: produit q*r
   4.000   2.000   1.000   0.000
   2.000   5.000   3.000   5.000
   1.000   1.000   1.000   1.000
   2.000   1.000   4.000   1.000
```

Voici un deuxième exemple :

```
Voici la matrice a
   4.000   1.000   0.000   0.000   0.000
   1.000   4.000   1.000   0.000   0.000
   0.000   1.000   4.000   1.000   0.000
   0.000   0.000   1.000   4.000   1.000
   0.000   0.000   0.000   1.000   4.000
Décomposition QR de a
Voici la matrice q
   0.970  -0.234   0.062  -0.017   0.005
   0.243   0.935  -0.248   0.066  -0.018
   0.000   0.265   0.929  -0.249   0.069
   0.000   0.000   0.268   0.928  -0.258
   0.000   0.000   0.000   0.268   0.963
Voici la matrice r
   4.123   1.940   0.243   0.000   0.000
   0.000   3.773   1.996   0.265   0.000
   0.000   0.000   3.736   2.000   0.268
   0.000  -0.000   0.000   3.732   2.000
   0.000   0.000  -0.000  -0.000   3.596
```

Comme $A = QR$ et que Q est orthogonale, on a $\text{dét}(A) = \text{dét}(R)$ et en particulier A est inversible si et seulement si R l'est. De plus, le système d'équations $Ax = b$ équivaut aux systèmes

$$Qy = b \qquad Rx = y$$

Le premier se résout immédiatement parce que $y = {}^tQb$ et le deuxième est un système triangulaire. D'où :

```
PROCEDURE SystemeParQR (n: integer; a: mat; b: vec; VAR x: vec;
   VAR inversible: boolean);
   Calcul de la solution du système linéaire Ax = b par la décomposition A = QR. Si R est
   inversible, on résout d'abord le système Qy = b par y = tQb, puis le système triangulaire
   Rx = y.
   VAR
      i: integer;
      q, r: mat;
      y: vec;
   BEGIN
      DecompositionQR(n, a, q, r);
      inversible := true; i := 1;
      WHILE (i <= n) AND inversible DO BEGIN
         inversible := NOT EstNul(r[i, i]); i := i + 1
      END;
      IF inversible THEN BEGIN
         VecteurParMatrice(n, b, q, y);
         SystemeTriangulaireSuperieur(n, r, y, x);
      END
   END; { de "SystemeParQR" }
```

3.3.4 Décomposition QR (méthode de Householder)

Nous présentons maintenant une autre méthode de calcul d'une décomposition QR. On pourra aussi consulter l'énoncé correspondant de mise sous forme tridiagonale. Les matrices considérées sont réelles, carrées d'ordre n. On note I la matrice unité d'ordre n.

LEMME 3.3.5. *Soit $w \in \mathbb{R}^n$ un vecteur colonne de norme euclidienne* 1 *et soit $P = I - 2w\,{}^tw$. Alors ${}^tP = P^{-1} = P$.*

En effet, on a ${}^tP = I - 2\,{}^t(w\,{}^tw) = P$ et

$$\begin{aligned} P^2 &= (I - 2w\,{}^tw)(I - 2w\,{}^tw) \\ &= I - 4w\,{}^tw + 4w\,{}^tww\,{}^tw = I \end{aligned}$$

car ${}^tw\,w = \|w\| = 1$. ∎

Pour $k \in \{1, \ldots, n\}$ et pour $x \in \mathbb{R}^n$, on note $v = v(k, x) \in \mathbb{R}^n$ le vecteur défini par

$$v_i = \begin{cases} 0 & \text{si } 1 \le i < k \\ x_k + signe(x_k)\alpha & \text{si } i = k \\ x_i & \text{si } k < i \le n \end{cases}$$

où $\alpha = \sqrt{x_k^2 + \cdots + x_n^2}$ et $signe(s) = 1, 0, -1$ selon que $s > 0$, $s = 0$, $s < 0$. On définit

$$P = P(k, x) = I - \beta v\, {}^t v$$

avec $\beta = 2/\|v\|^2$ si $v \neq 0$, $\beta = 2$ sinon. Posons $y = P(k, x)x$.

Lemme 3.3.6. *On a $\|v\|^2 = 2 \cdot {}^t vx$ et $y = x - v$.*

Preuve. On a

$${}^t vx = (x_k + signe(x_k)\alpha)x_k + \sum_{i=k+1}^{n} x_i^2 = signe(x_k)x_k\alpha + \alpha^2$$

et

$$\|v\|^2 = (x_k + signe(x_k)\alpha)^2 + \sum_{i=k+1}^{n} x_i^2 = 2\alpha^2 + 2signe(x_k)x_k\alpha$$

d'où la première égalité. Par ailleurs, si $v \neq 0$,

$$\beta v\, {}^t vx = \frac{v}{\|v\|^2} 2\, {}^t vx = v$$

donc $y = x - \beta v\, {}^t vx = x - v$. ∎

En d'autres termes,

$$y_i = \begin{cases} x_i & \text{si } 1 \leq i < k \\ -signe(x_k)\alpha & \text{si } i = k \\ 0 & \text{si } k < i \leq n \end{cases}$$

et en particulier les coefficients de y d'indices plus grands que k sont nuls. Notons $A^{(j)}$ la j-ième colonne de la matrice A.

Proposition 3.3.7. *Soit A une matrice d'ordre n, soit $A_0 = A$ et posons*

$$P_h = P(h, A_{h-1}^{(h)}) \quad A_h = P_h A_{h-1} \qquad h = 1, \ldots, n-1$$

Alors $P_{n-1} \cdots P_1 A = A_{n-1}$; de plus, la matrice A_{n-1} est triangulaire supérieure et la matrice $P_{n-1} \cdots P_1$ est orthogonale.

On a donc

$$A = QR$$

avec $Q = {}^t(P_{n-1} \cdots P_1)$ et $R = A_{n-1}$.

Preuve. D'après ce qui précède, si l'on prend pour x la j-ième colonne $A^{(j)}$ de A, alors la j-ième colonne de $P(k, A^{(j)})A$ a ses coefficients nuls dans les lignes d'indices plus grands que k. De plus, on a, avec $v = v(k, A^{(j)})$,

$$P(k, A^{(j)})A = A - v\,{}^t p \qquad \text{avec } p = \beta\, {}^t A v \tag{3.1}$$

ce qui montre que les colonnes d'indices inférieurs à k de A ne sont pas modifiées par la prémultiplication par P. La proposition en résulte. ∎

Pour la réalisation de cette méthode de décomposition, il convient d'abord de calculer le vecteur $v(k, A^{(j)})$, ce qui se fait par la procédure que voici :

```
PROCEDURE VecteurHouseholder (n: integer; VAR a: mat; k, j: integer;
   VAR v: vec);
   Calcule le vecteur v(k, a^(j)), où a^(j) est la j-ième colonne de a.
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO k - 1 DO v[i] := 0;
      FOR i := k TO n DO v[i] := a[i, j];
      v[k] := v[k] + signe(v[k]) * norme(n, v);
   END; { de "VecteurHouseholder" }
```

L'application de la formule (3.1) conduit à la procédure que voici, qui calcule dans Q le produit de P par Q :

```
 PROCEDURE TransformationHouseholder (n, h: integer; VAR q, a: mat);
   VAR
      i, j: integer;
      beta, normeV2: real;
      v, z, p: vec;
   BEGIN  { de "TransformationHouseholder" }
      VecteurHouseholder(n, a, h, h, v);
      normeV2 := ProduitScalaire(n, v, v);
      IF EstNul(normeV2) THEN
         beta := 2
      ELSE
         beta := 2 / normeV2;
      VecteurParMatrice(n, v, a, p);
      VecteurParScalaire(n, p, beta, p);         p = βvA
      VecteurParMatrice(n, v, q, z);
      VecteurParScalaire(n, z, beta, z);         z = βvQ
      FOR i := 1 TO n DO
         FOR j := 1 TO n DO BEGIN
            a[i, j] := a[i, j] - v[i] * p[j];
            q[i, j] := q[i, j] - v[i] * z[j];
         END
   END; { de "TransformationHouseholder" }
```

La décomposition QR de A s'obtient enfin par :

```
PROCEDURE DecompositionQRparHouseholder (n: integer; a: mat; VAR q, r: mat);
   VAR
      h: integer;
   BEGIN
      q := MatriceUnite;
      r := a;
      FOR h := 1 TO n - 1 DO
         TransformationHouseholder(n, h, q, r);
      Transposer(n, q, q)
   END; { de "DecompositionQRparHouseholder" }
```

Voici un exemple d'exécution, avec des impressions intermédiaires :

```
Voici la matrice lue
   4.000   2.000   1.000   0.000
   2.000   5.000   3.000   5.000
   1.000   1.000   1.000   1.000
   2.000   1.000   4.000   1.000
Matrice après transformation de la colonne 1
  -5.000  -4.200  -3.800  -2.600
   0.000   3.622   1.933   4.422
   0.000   0.311   0.467   0.711
   0.000  -0.378   2.933   0.422
Matrice après transformation de la colonne 2
  -5.000  -4.200  -3.800  -2.600
   0.000  -3.655  -1.652  -4.399
   0.000   0.000   0.313   0.334
   0.000   0.000   3.119   0.880
Matrice après transformation de la colonne 3
  -5.000  -4.200  -3.800  -2.600
   0.000  -3.655  -1.652  -4.399
   0.000   0.000  -3.135  -0.909
   0.000   0.000  -0.000  -0.244
Matrice r
  -5.000  -4.200  -3.800  -2.600
   0.000  -3.655  -1.652  -4.399
   0.000   0.000  -3.135  -0.909
   0.000   0.000  -0.000  -0.244
Matrice q
  -0.800   0.372   0.455   0.122
  -0.400  -0.908   0.007   0.122
  -0.200  -0.044  -0.053  -0.977
  -0.400   0.186  -0.889   0.122
```

On pourra comparer la décomposition à celle obtenue par la méthode de Givens : les matrices R n'ont pas les mêmes diagonales.

Notes bibliographiques

Les décompositions présentées ici sont des plus classiques. Il en existe bien d'autres, en particulier la *singular value decomposition* qui est d'une grande utilité. On pourra consulter :

G. H. Golub, C. F. van Loan, *Matrix computations*, Baltimore, John Hopkins University Press, 1985.

Chapitre 4

Matrices tridiagonales

Une matrice carrée $A = (a_{i,j})$ est *tridiagonale* si $a_{i,j} = 0$ pour $|i-j| > 1$. Les procédures des chapitres précédents prennent des formes particulières pour les matrices tridiagonales et elles peuvent surtout être réalisées de manière beaucoup plus efficace. Les matrices tridiagonales se rencontrent dans le calcul des valeurs et vecteurs propres de matrices. Dans ce cas, une première étape consiste à mettre la matrice donnée sous forme tridiagonale. Les matrices tridiagonales interviennent aussi dans le calcul des splines cubiques et surtout dans la discrétisation d'équations aux dérivées partielles.

4.1 Opérations sur les matrices tridiagonales

4.1.1 Système tridiagonal d'équations

Une matrice tridiagonale

$$A = \begin{pmatrix} a_1 & b_1 & 0 & & \dots & 0 \\ c_2 & a_2 & b_2 & 0 & & \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & c_{n-2} & a_{n-2} & b_{n-2} & 0 \\ & & & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \dots & & 0 & c_n & a_n \end{pmatrix}$$

est déterminée par trois vecteurs a, b et c qui contiennent respectivement la diagonale, la sur-diagonale et la sous-diagonale de A. Le système d'équations linéaires

$$Ax = d$$

s'écrit

$$\begin{array}{llll} a_1x_1 & + b_1x_2 & & = d_1 \\ c_ix_{i-1} & + a_ix_i & + b_ix_{i+1} & = d_i \qquad i = 2, \dots, n-1 \\ & c_nx_{n-1} & + a_nx_n & = d_n \end{array}$$

Par élimination, on le met sous la forme

$$\begin{aligned} a'_1x_1 + b_1x_2 &= d'_1 \\ a'_ix_i + b_ix_{i+1} &= d'_i \qquad i = 2, \ldots, n-1 \\ a'_nx_n &= d_n \end{aligned}$$

avec

$$\begin{aligned} a'_1 &= a_1 & d'_1 &= d_1 \\ a'_i &= a_i - c_ib_{i-1}/a'_{i-1} & d'_i &= d_i - c_id_{i-1}/a'_{i-1} \qquad i = 2, \ldots, n \end{aligned}$$

Le système obtenu est triangulaire et se résout donc simplement. La seule difficulté provient d'un coefficient a'_i nul. Ceci est dû à un mineur principal ayant un déterminant nul. Dans ce cas, le système n'est pas résoluble de cette manière, et il faut donc revenir à une méthode plus coûteuse et employer par exemple la méthode de Gauss générale. En effet, un pivotage perturbe le caractère tridiagonal du système.

Voici une réalisation de l'algorithme :

```
PROCEDURE SystemeTridiagonal (n: integer; a, b, c, d: vec; VAR x: vec;
   VAR soluble: boolean);
   a est la diagonale, b la sur-diagonale, c la sous-diagonale et d le second membre. On
   résout le système sans pivotage; ce n'est possible que si les mineurs principaux sont
   inversibles. Le booléen indique succès ou échec.
   VAR
      i: integer;
   BEGIN
      soluble := NOT EstNul(a[1]);
      i := 2;
      WHILE (i <= n) AND soluble DO BEGIN
         a[i] := a[i] - c[i] / a[i - 1] * b[i - 1];
         d[i] := d[i] - c[i] / a[i - 1] * d[i - 1];
         soluble := NOT EstNul(a[i]);
         i := i + 1
      END;
      IF soluble THEN BEGIN
         x[n] := d[n] / a[n];
         FOR i := n - 1 DOWNTO 1 DO
            x[i] := (d[i] - b[i] * x[i + 1]) / a[i]
      END
   END; { de "SystemeTridiagonal" }
```

4.1.2 Décomposition LU

La décomposition LU (comme d'ailleurs la décomposition QR) peut être réalisée sur une matrice tridiagonale

$$A = \begin{pmatrix} a_1 & b_1 & 0 & & \dots & 0 \\ c_2 & a_2 & b_2 & 0 & & \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & c_{n-2} & a_{n-2} & b_{n-2} & 0 \\ & & & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \dots & & 0 & c_n & a_n \end{pmatrix}$$

Il est intéressant de constater que les matrices L et U sont aussi tridiagonales; plus précisément, posons

$$A = LU = \begin{pmatrix} 1 & & & & 0 \\ \ell_2 & 1 & & & \\ & \ell_3 & 1 & & \\ & & \ddots & \ddots & \\ 0 & & & \ell_n & 1 \end{pmatrix} \begin{pmatrix} u_1 & v_1 & & & 0 \\ & u_2 & v_2 & & \\ & & \ddots & \ddots & \\ & & & & v_{n-1} \\ 0 & & & & u_n \end{pmatrix}$$

L'identification donne $v = b$ et $u_1 = a_1$, et

$$\begin{aligned} \ell_i &= c_i/u_{i-1} \\ u_i &= a_i - \ell_i b_{i-1} \end{aligned}$$

pour $i = 2, \dots, n$, ce qui permet d'écrire la procédure suivante :

```
PROCEDURE DecompositionLUtridiagonal (n: integer; a, b, c: vec;
   VAR u, l: vec);
   Calcule la décomposition LU d'une matrice tridiagonale.
   VAR
      i: integer;
   BEGIN
      u[1] := a[1];
      FOR i := 2 TO n DO BEGIN
         l[i] := c[i] / u[i - 1];
         u[i] := a[i] - l[i] * b[i - 1]
      END
   END; { de "DecompositionLUtridiagonal" }
```

Dans le cas particulier où la matrice de départ est une matrice *de Lanczos* (c'est-à-dire lorsque le vecteur b a tous ses coefficients égaux à 1), la procédure se simplifie encore et devient

```
PROCEDURE LanczosLU (n: integer; a, c: vec; VAR u, l: vec);
   VAR
```

```
      i: integer;
   BEGIN
      u[1] := a[1];
      FOR i := 2 TO n DO BEGIN
         l[i] := c[i] / u[i - 1];
         u[i] := a[i] - l[i]
      END
   END; { de "LanczosLU" }
```

On utilisera ces procédures dans la méthode de Rutishauser de calcul des valeurs propres. Dans les deux cas, la matrice UL est à nouveau tridiagonale.

4.1.3 Décomposition de Choleski

La décomposition de Choleski d'une matrice A symétrique définie positive est, elle aussi, très simple si A est tridiagonale. Posons en effet

$$A = \begin{pmatrix} a_1 & b_1 & & & 0 \\ & a_2 & b_2 & & \\ & & \ddots & \ddots & \\ & & & & b_{n-1} \\ 0 & & & & a_n \end{pmatrix} \qquad \text{et} \qquad L = \begin{pmatrix} d_1 & & & & 0 \\ \ell_2 & d_2 & & & \\ & \ell_3 & d_3 & & \\ & & \ddots & \ddots & \\ 0 & & & \ell_n & d_n \end{pmatrix}$$

L'équation $A = L^tL$ donne, par identification,

$$\begin{aligned} d_1 &= \sqrt{a_1} \\ \ell_i &= b_{i-1}/d_{i-1} \\ d_i &= \sqrt{a_i - \ell_i^2} \qquad i = 2, \ldots, n \end{aligned}$$

Ce qui se traduit dans la procédure que voici :

```
PROCEDURE CholeskiTridiagonal (n: integer; a, b: vec; VAR d, l: vec);
   VAR
      i: integer;
   BEGIN
      d[1] := sqrt(a[1]);
      FOR i := 2 TO n DO BEGIN
         l[i] := b[i - 1] / d[i - 1];
         d[i] := sqrt(a[i] - sqr(l[i]))
      END
   END; { de "CholeskiTridiagonal" }
```

4.2 Tridiagonalisation

La simplicité de manipulation des matrices tridiagonales est manifeste. Il est donc souhaitable de se ramener à des matrices de cette forme avant de procéder à des calculs poussés comme ceux des valeurs et des vecteurs propres. Deux méthodes de transformation orthogonale d'une matrice en une matrice tridiagonale sont exposées ici : la méthode de Givens et la méthode de Householder.

4.2.1 Enoncé : tridiagonalisation (méthode de Givens)

Soit $n \geq 2$ un entier. Les matrices considérées sont réelles, d'ordre n. On note ${}^tP$ la transposée d'une matrice P.

Pour p, q dans $\{1, \ldots, n\}$, $p \neq q$, on note $G(q,p) = (g_{i,j})$ toute matrice de rotation de la forme

$$G(q,p) = \begin{pmatrix} 1 & & & & & & 0 \\ & \ddots & & & & & \\ & & c & & s & & \\ & & & \ddots & & & \\ & & -s & & c & & \\ & & & & & \ddots & \\ 0 & & & & & & 1 \end{pmatrix}$$

avec $c^2 + s^2 = 1$, qui ne diffère de la matrice unité que par les éléments d'indice (p,p), (p,q), (q,p), (q,q). Plus précisément,

$$\begin{array}{ll} g_{i,i} = 1 & i \neq p \text{ et } i \neq q \\ g_{i,j} = 0 & i \neq j, \quad (i,j) \neq (p,q) \text{ et } (i,j) \neq (q,p) \\ g_{p,p} = g_{q,q} = c & \\ g_{p,q} = -g_{q,p} = s & \end{array}$$

1.– Etant donnés une matrice A et deux indices p et q avec $2 \leq p < q$, démontrer qu'il existe c et s tels que l'élément $b_{q,p-1}$ de la matrice $B = (b_{i,j}) = {}^tG(q,p)AG(q,p)$ soit nul.

2.– Soit A une matrice symétrique. On définit une suite de matrices $B^{(q,p)} = (b_{i,j}^{(q,p)})$ pour $2 \leq p \leq q \leq n$ par

$$\begin{array}{ll} B^{(2,2)} = A & \\ B^{(q,p)} = {}^tG(q,p)B^{(q-1,p)}G(q,p) & (2 \leq p < q \leq n) \\ B^{(p,p)} = B^{(n,p-1)} & (3 \leq p \leq n) \end{array}$$

où $G(q,p)$ est construit de manière telle que $b_{q,p-1}^{(q,p)} = 0$. On pose $R = (r_{i,j}) = B^{(n,n-1)}$. Démontrer que R est *tridiagonale*, c'est-à-dire que $r_{i,j} = 0$ pour $|i-j| > 1$.

3.– Ecrire une procédure qui prend en donnée une matrice symétrique A d'ordre n et qui affiche comme résultat la matrice R sous la forme de deux vecteurs $d \in \mathbb{R}^n$ et $h \in \mathbb{R}^{n-1}$ contenant respectivement ses éléments diagonaux et hors-diagonaux.

Exemple numérique :

$$A = \begin{pmatrix} 4 & 1 & 2 & 3 & 4 \\ 1 & 4 & 1 & 5 & 1 \\ 2 & 1 & 10 & 1 & 7 \\ 3 & 5 & 1 & 14 & 6 \\ 4 & 1 & 7 & 6 & 14 \end{pmatrix}$$

4.– On suppose que la matrice A est de plus définie positive. Démontrer qu'il existe alors une matrice triangulaire inférieure $L = (l_{i,j})$ telle que $R = L\,{}^tL$. Démontrer que L vérifie $l_{i,j} = 0$ pour $i - j > 1$. Ecrire une procédure qui prend comme donnée une matrice tridiagonale définie positive, sous la forme de deux vecteurs comme décrit à la question précédente, et qui détermine la matrice L sous forme de deux vecteurs également.

4.2.2 Solution : tridiagonalisation (méthode de Givens)

Soit $n \geq 2$ un entier. Les matrices considérées sont réelles, d'ordre n. On note ${}^tP$ la transposée d'une matrice P. Pour p, q dans $\{1, \ldots, n\}$, $p \neq q$, on note $G(q,p) = (g_{i,j})$ toute matrice de rotation définie par

$$\begin{array}{ll} g_{i,i} = 1 & i \neq p \text{ et } i \neq q \\ g_{i,j} = 0 & i \neq j, \quad (i,j) \neq (p,q) \text{ et } (i,j) \neq (q,p) \\ g_{p,p} = g_{q,q} = c & \\ g_{p,q} = -g_{q,p} = s & \end{array}$$

avec $c^2 + s^2 = 1$.

Etant donnés une matrice A et deux indices p et q avec $2 \leq p < q$, il est facile de déterminer deux nombres réels c et s tels que l'élément $b_{q,p-1}$ de la matrice $B = (b_{i,j}) = {}^tG(q,p)AG(q,p)$ soit nul. Il suffit en effet d'évaluer le coefficient correspondant. Comme

$$b_{q,p-1} = s a_{p,p-1} + c a_{q,p-1}$$

on pose

$$\begin{array}{lll} c = 1 & s = 0 & \text{si } a_{p,p-1}^2 + a_{q,p-1}^2 = 0 \\ c = a_{p,p-1}/r & s = a_{q,p-1}/r & \text{si } a_{p,p-1}^2 + a_{q,p-1}^2 = r^2 \neq 0 \end{array}$$

Soit maintenant A une matrice symétrique et considérons la suite de matrices $B^{(q,p)} = (b_{i,j}^{(q,p)})$ pour $2 \leq p \leq q \leq n$ définie par

$$\begin{array}{ll} B^{(2,2)} = A & \\ B^{(q,p)} = {}^tG(q,p)B^{(q-1,p)}G(q,p) & (2 \leq p < q \leq n) \\ B^{(p,p)} = B^{(n,p-1)} \qquad (3 \leq p \leq n) & \end{array}$$

où $G(q,p)$ est construit de manière telle que $b_{q,p-1}^{(q,p)} = 0$. On pose $R = (r_{i,j}) = B^{(n,n-1)}$.

Proposition 4.2.1. *La matrice R est tridiagonale.*

Preuve. Soit A une matrice symétrique et soient p, q avec $2 \leq p < q$. Soit $G(q,p)$ telle que l'élément $b_{q,p-1}$ de la matrice $B = (b_{i,j}) = {}^tG(q,p)AG(q,p)$ soit nul. On a $b_{i,j} = a_{i,j}$ si $i \neq p, q$ et pour $j < p$:

$$\begin{aligned} b_{p,j} &= ca_{p,j} - sa_{q,j} \\ b_{q,j} &= sa_{p,j} + ca_{q,j} \end{aligned}$$

Il en résulte que si $a_{i,j} = 0$ pour $1 \leq j < p$, $j + 2 \leq i \leq n$, il en est de même pour $b_{i,j}$. Il en résulte par récurrence que R est tridiagonale. ∎

4.2.3 Programme : tridiagonalisation (méthode de Givens)

Par identification, on obtient des expressions pour c et s qui conduisent à la procédure que voici, très proche de la procédure `CoefficientsGivens` du chapitre 3.

```
PROCEDURE CoefficientsGivensE (p, q: integer; VAR c, s: real; VAR a: mat);
   Calcul des réels c et s tels que b(q, p − 1) = 0.
   VAR
      norme: real;
   BEGIN
      norme := sqrt(sqr(a[p, p - 1]) + sqr(a[q, p - 1]));
      IF EstNul(norme) THEN BEGIN
         c := 1;
         s := 0
      END
      ELSE BEGIN
         c := a[p, p - 1] / norme;
         s := -a[q, p - 1] / norme
      END
   END; { de "CoefficientsGivensE" }
```

Le calcul de la matrice ${}^tG(q,p)AG(q,p)$ se fait simplement et de façon économique, en évaluant les quelques coefficients modifiés de la matrice A. Voici une procédure qui réalise cette transformation :

```
PROCEDURE TransformationGivens (n, p, q: integer; c, s: real; VAR a: mat);
   Multiplication tgag où g est la matrice de rotation dont les coefficients sont déterminés
   par c et s.
   VAR
      j: integer;
      u, v, w: real;
   BEGIN
      FOR j := 1 TO n DO
         IF (j <> p) AND (j <> q) THEN BEGIN
            v := a[p, j]; w := a[q, j];
            a[p, j] := c * v - s * w; a[j, p] := a[p, j];
            a[q, j] := s * v + c * w; a[j, q] := a[q, j]
```

```
      END;
    v := a[p, p]; u := a[p, q]; w := a[q, q];
    a[p, p] := c * c * v - 2 * c * s * u + s * s * w;
    a[q, q] := s * s * v + 2 * c * s * u + c * c * w;
    a[p, q] := (c * c - s * s) * u + c * s * (v - w);
    a[q, p] := a[p, q]
  END; { de "TransformationGivens" }
```

Notons qu'il n'y a que $2n$ coefficients modifiés, donc $O(n)$ opérations arithmétiques, alors que le calcul de ${}^tG(q,p)AG(q,p)$ par les procédures générales d'évaluation de matrices aurait coûté $O(n^3)$ opérations. La différence est appréciable.

Il n'y a plus qu'à utiliser cette procédure pour $2 \leq p < q \leq n$:

```
PROCEDURE TridiagonalParGivens (n: integer; VAR a: mat);
  On annule successivement les coefficients d'indices (p, q), 2 ≤ p < q ≤ n. Le produit des
  matrices de rotation est l'inverse de la matrice orthogonale cherchée.
  VAR
    p, q: integer;
    c, s: real;
  BEGIN
    FOR p := 2 TO n - 1 DO
      FOR q := p + 1 TO n DO BEGIN
        CoefficientsGivensE(p, q, c, s, a);
        TransformationGivens(n, p, q, c, s, a);
      END
  END; { de "TridiagonalParGivens" }
```

Une variante de cette procédure permet d'obtenir aussi la matrice orthogonale qui est le produit des matrices de rotations. Pour cela, on accumule les matrices de rotations dans une matrice qui, initialement, est la matrice unité et qui est postmultipliée, au fur et à mesure, avec les matrices de rotations. La procédure de postmultiplication, c'est-à-dire du calcul du produit $BG(q,p)$, est la suivante :

```
PROCEDURE Postmultiplier (n, p, q: integer; c, s: real; VAR a: mat);
  Postmultiplication de la matrice a par la matrice de rotation G(q, p).
  VAR
    i: integer;
    v, w: real;
  BEGIN
    FOR i := 1 TO n DO BEGIN
      v := a[i, p]; w := a[i, q];
      a[i, p] := c * v - s * w;
      a[i, q] := s * v + c * w;
    END
  END; { de "Postmultiplier" }
```

La procédure suivante calcule donc, dans B, le produit $G(3,2) \cdots G(n, n-1)$.

```
PROCEDURE TridiagonalParGivens2 (n: integer; VAR a, g: mat);
```

On annule successivement les coefficients d'indices (p, q), pour $2 \leq p < q \leq n$. Le produit des matrices de rotation est l'inverse de la matrice orthogonale cherchée.

```
VAR
   p, q: integer;
   c, s: real;
BEGIN
   g := MatriceUnite;
   FOR p := 2 TO n - 1 DO
      FOR q := p + 1 TO n DO BEGIN
         CoefficientsGivensE(p, q, c, s, a);
         Postmultiplier(n, p, q, c, s, g);
         TransformationGivens(n, p, q, c, s, a)
      END
END; { de "TridiagonalParGivens2" }
```

Voici un exemple d'exécution de cette procédure, avec quelques impressions intermédiaires :

```
Donner l'ordre n :   5
Entrer la matrice (partie inférieure)
Ligne 1 :  4
Ligne 2 :  1 4
Ligne 3 :  2 1 10
Ligne 4 :  3 5 1 14
Ligne 5 :  4 1 7 6 14
Voici la matrice lue
   4.000   1.000   2.000   3.000   4.000
   1.000   4.000   1.000   5.000   1.000
   2.000   1.000  10.000   1.000   7.000
   3.000   5.000   1.000  14.000   6.000
   4.000   1.000   7.000   6.000  14.000
Matrice A pour p = 2, q = 3
   4.000   2.236   0.000   3.000   4.000
   2.236   9.600   1.800   3.130   6.708
   0.000   1.800   4.400  -4.025   2.236
   3.000   3.130  -4.025  14.000   6.000
   4.000   6.708   2.236   6.000  14.000
Matrice A pour p = 2, q = 4
   4.000   3.742   0.000   0.000   4.000
   3.742  15.429  -2.151   1.214   8.820
   0.000  -2.151   4.400  -3.849   2.236
   0.000   1.214  -3.849   8.171  -1.793
   4.000   8.820   2.236  -1.793  14.000
Matrice A pour p = 2, q = 5
   4.000   5.477   0.000   0.000   0.000
   5.477  23.467   0.163  -0.480  -1.301
   0.000   0.163   4.400  -3.849   3.099
```

```
    0.000  -0.480  -3.849   8.171  -2.111
    0.000  -1.301   3.099  -2.111   5.962
Matrice A pour p = 3, q = 4
    4.000   5.477  -0.000   0.000   0.000
    5.477  23.467   0.507   0.000  -1.301
   -0.000   0.507  10.127   1.901   2.997
    0.000   0.000   1.901   2.444   2.254
    0.000  -1.301   2.997   2.254   5.962
Matrice A pour p = 3, q = 5
    4.000   5.477  -0.000   0.000  -0.000
    5.477  23.467   1.396   0.000  -0.000
   -0.000   1.396   4.483  -1.409  -0.796
    0.000   0.000  -1.409   2.444   2.589
   -0.000  -0.000  -0.796   2.589  11.606
Matrice A pour p = 4, q = 5
    4.000   5.477  -0.000  -0.000   0.000
    5.477  23.467   1.396   0.000   0.000
   -0.000   1.396   4.483   1.619  -0.000
   -0.000   0.000   1.619   6.879   5.260
    0.000   0.000  -0.000   5.260   7.171
Matrice G
    1.000   0.000   0.000  -0.000   0.000
    0.000   0.183   0.201   0.849  -0.453
    0.000   0.365   0.663  -0.475  -0.449
    0.000   0.548   0.340   0.196   0.739
    0.000   0.730  -0.636  -0.122  -0.216
Matrice A tridiagonalisée (A’)
    4.000   5.477  -0.000  -0.000   0.000
    5.477  23.467   1.396   0.000   0.000
   -0.000   1.396   4.483   1.619  -0.000
   -0.000   0.000   1.619   6.879   5.260
    0.000   0.000  -0.000   5.260   7.171
Matrice GA’tG
    4.000   1.000   2.000   3.000   4.000
    1.000   4.000   1.000   5.000   1.000
    2.000   1.000  10.000   1.000   7.000
    3.000   5.000   1.000  14.000   6.000
    4.000   1.000   7.000   6.000  14.000
```

4.3 Tridiagonalisation (méthode de Householder)

On pourra se reporter également à la décomposition QR par la méthode de Householder, décrite au chapitre 3.

4.3.1 Enoncé : tridiagonalisation (méthode de Householder)

On note ${}^tP$ la transposée d'une matrice P. Soit $n \geq 1$ un entier. Les matrices considérées sont réelles. Un vecteur v de $\mathbb{R}^n$ est considéré comme une matrice à une colonne et on note $\|v\|$ sa norme euclidienne. On note I la matrice unité d'ordre n. On se propose de construire, pour toute matrice A d'ordre n symétrique, une matrice semblable $B = (b_{i,j})$ qui soit *tridiagonale*, c'est-à-dire telle que $b_{i,j} = 0$ pour $|i - j| > 1$.

1.– Soient $w \in \mathbb{R}^n$ un vecteur de norme euclidienne 1 et P la matrice $I - 2w\,{}^tw$. Démontrer que ${}^tP = P^{-1} = P$.

2.– Pour $k \in \{2, \ldots, n\}$ et pour $x \in \mathbb{R}^n$, on note $v = v(k, x) \in \mathbb{R}^n$ le vecteur défini par

$$v_i = \begin{cases} 0 & \text{si } 1 \leq i < k \\ x_k + signe(x_k)\alpha & \text{si } i = k \\ x_i & \text{si } k < i \leq n \end{cases}$$

où $\alpha = \sqrt{x_k^2 + \cdots + x_n^2}$ et $signe(s) = 1, 0, -1$ selon que $s > 0$, $s = 0$, $s < 0$. On pose

$$P = P(k, x) = I - \beta v\,{}^tv \qquad (*)$$

avec $\beta = 2/\|v\|^2$ si $v \neq 0$, $\beta = 2$ sinon. Démontrer que $y = P(k, x)x$ vérifie

$$y_i = \begin{cases} x_i & \text{si } 1 \leq i < k \\ -signe(x_k)\alpha & \text{si } i = k \\ 0 & \text{si } k < i \leq n \end{cases}$$

Soit A une matrice symétrique d'ordre n.

3.– Démontrer que $PAP = A - q\,{}^tv - v\,{}^tq$, où P et v sont comme dans $(*)$, et où $p = \beta Av$, $2K = \beta\,{}^tvp$ et $q = p - Kv$.

4.– Soient $A_1, \ldots, A_{n-2}$, $P_1, \ldots, P_{n-2}$ les matrices définies par

$$\begin{aligned} &P_1 = P(2, A^{(1)}) \quad A_1 = P_1AP_1 \\ &P_r = P(r + 1, A_{r-1}^{(r)}) \quad A_r = P_rA_{r-1}P_r \qquad r = 2, \ldots, n-2 \end{aligned}$$

où pour une matrice M, on note $M^{(j)}$ la j-ième colonne de M. Démontrer que la matrice A_{n-2} est symétrique et tridiagonale.

5.– Ecrire un programme qui prend en argument une matrice symétrique A d'ordre n et qui affiche comme résultat la matrice A_{n-2} sous la forme de deux vecteurs $d \in \mathbb{R}^n$ et $e \in \mathbb{R}^{n-1}$ contenant respectivement ses éléments diagonaux et hors-diagonaux.

Exemple numérique :

$$A = \begin{pmatrix} 4 & 1 & 2 & 3 & 4 \\ 1 & 4 & 1 & 5 & 1 \\ 2 & 1 & 0 & 1 & 7 \\ 3 & 5 & 1 & 4 & 6 \\ 4 & 1 & 7 & 6 & 4 \end{pmatrix}$$

6.– Soit B une matrice carrée inversible, symétrique, tridiagonale d'ordre n donnée par deux vecteurs d et e comme à la question précédente. Ecrire une procédure pour résoudre le système linéaire $Bx = b$ et l'appliquer à la matrice tridiagonale obtenue sur l'exemple numérique, avec

$$b = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$$

4.3.2 Solution : tridiagonalisation (méthode de Householder)

Les matrices considérées sont réelles, carrées d'ordre n. On note I la matrice unité d'ordre n. Rappelons les notations de la section 3.3.4 :

Pour $k \in \{1, \dots, n\}$ et pour $x \in \mathbb{R}^n$, le vecteur $v = v(k, x) \in \mathbb{R}^n$ est défini par

$$v_i = \begin{cases} 0 & \text{si } 1 \le i < k \\ x_k + signe(x_k)\alpha & \text{si } i = k \\ x_i & \text{si } k < i \le n \end{cases}$$

où $\alpha = \sqrt{x_k^2 + \cdots + x_n^2}$ et $signe(s) = 1, 0, -1$ selon que $s > 0$, $s = 0$, $s < 0$. Avec $P = P(k, x) = I - \beta v\,{}^t v$ et $\beta = 2/\|v\|^2$ si $v \neq 0$, $\beta = 2$ sinon, le vecteur $y = P(k, x)x$ vérifie alors $y = x - v$ parce que $2 \cdot {}^t v x = \|v\|^2$; en fait, on a

$$y_i = \begin{cases} x_i & \text{si } 1 \le i < k \\ -signe(x_k)\alpha & \text{si } i = k \\ 0 & \text{si } k < i \le n \end{cases}$$

et en particulier les coefficients de y d'indices plus grands que k sont nuls. Le procédé de tridiagonalisation est très voisin de celui qui mène à la décomposition QR. Dans la tridiagonalisation, on annule successivement les colonnes *et* les lignes, à l'exception des coefficients en dehors de la diagonale et des sous- et sur-diagonales, alors que pour la décomposition QR, on n'annule que la partie triangulaire strictement inférieure, mais *toute* cette partie. La méthode repose sur la proposition qui suit.

Proposition 4.3.1. *Soit A une matrice symétrique d'ordre n et soient $A_1, \dots, A_{n-2}$, $P_1, \dots, P_{n-2}$ les matrices définies par*

$$\begin{aligned} &P_1 = P(2, A^{(1)}) \quad A_1 = P_1 A P_1 \\ &P_r = P(r+1, A_{r-1}^{(r)}) \quad A_r = P_r A_{r-1} P_r \qquad r = 2, \dots, n-2 \end{aligned}$$

Alors la matrice A_{n-2} est symétrique et tridiagonale.

Preuve. Chaque matrice A_r est évidemment symétrique. Montrons que les matrices A_r sont «tridiagonales sur leurs r premières lignes et colonnes», c'est-à-dire que les coefficients $a_{i,j}^{(r)}$ de A_r sont nuls si $|i-j| \geq 2$ et si $i < r$ ou $j < r$. Ceci est bien entendu vrai pour $r = 0$, en posant $A = A_0$. Nous avons déjà remarqué que la matrice $P_r A_{r-1}$ est tridiagonale sur les $r-1$ premières colonnes parce que A_{r-1} l'est. De plus, les coefficients d'indice i, r sont nuls pour $i = r+2, \ldots, n$. Par symétrie, $A_{r-1}P_r$ est symétrique sur les $r-1$ premières lignes. Or, la matrice P_r s'écrit par bloc sous la forme

$$P_r = \begin{pmatrix} I_r & 0 \\ 0 & P'_r \end{pmatrix}$$

où I_r est la matrice unité d'ordre r. Par conséquent, $A_r = P_r A_{r-1} P_r$ a les mêmes r premières colonnes que $P_r A_{r-1}$ et les mêmes r premières lignes que $A_{r-1}P_r$. ∎

Pour le calcul, il est judicieux de grouper les expressions. La formule à utiliser est donnée dans la proposition suivante.

Proposition 4.3.2. *Soit A une matrice symétrique, soit v un vecteur et soit $P = I - \beta v\,{}^t v$ avec $\beta = 2/\|v\|^2$ si $v \neq 0$, $\beta = 2$ sinon. Alors*

$$PAP = A - q\,{}^t v - v\,{}^t q$$

où $p = \beta A v$, $2K = \beta\,{}^t v p$ et $q = p - Kv$.

Preuve. Posons $\gamma = \beta/2$. On a

$$\begin{aligned} PAP &= (I - \beta v\,{}^t v)A(I - \beta v\,{}^t v) \\ &= A - \beta v\,{}^t v A - \beta A v\,{}^t v + \beta^2 v\,{}^t v A v\,{}^t v \\ &= A - v\,{}^t p - p\,{}^t v + \beta v\,{}^t v p\,{}^t v \\ &= A - v\,{}^t p - p\,{}^t v + 2Kv\,{}^t v \end{aligned}$$

d'où l'égalité annoncée. ∎

4.3.3 Programme : tridiagonalisation (méthode de Householder)

Rappelons la procédure :

```
PROCEDURE VecteurHouseholder (n: integer; VAR a: mat; k, j: integer;
   VAR v: vec);
```

du chapitre 3, qui calcule le vecteur $v = v(k, A^{(j)})$, où $A^{(j)}$ est la j-ième colonne de A utilisée pour la transformation de Househoulder. Cette transformation elle-même est ici un peu différente, puisqu'on applique la matrice de Householder à la matrice et à sa transposée. Voici la procédure ; on a noté `w` le vecteur noté q dans l'énoncé, pour ne pas confondre ce vecteur avec la matrice Q produit des transformations orthogonales.

```
PROCEDURE TransformationHouseholder (n, r: integer; VAR a: mat);
  VAR
     beta, K, normeV2: real;
```

```
      i, j: integer;
      v, p, w: vec;
   BEGIN  { de "TransformationHouseholder" }
      VecteurHouseholder(n, a, r + 1, r, v);           v = v(r+1, A^(r)_(r-1))
      normeV2 := ProduitScalaire(n, v, v);
      IF EstNul(normeV2) THEN beta := 2
      ELSE beta := 2 / normeV2;
      MatriceParVecteur(n, a, v, p);
      VecteurParScalaire(n, p, beta, p);               p = βAv
      K := beta / 2 * ProduitScalaire(n, v, p);        K = ½β ᵗvp
      VecteurParScalaire(n, v, -K, w);
      VecteurPlusVecteur(n, p, w, w);                  q = p − Kv
      FOR i := 1 TO n DO
         FOR j := 1 TO n DO
            a[i, j] := a[i, j] - v[i] * w[j] - w[i] * v[j];
   END; { de "TransformationHouseholder" }
```

et voici la procédure qui réalise la tridiagonalisation :

```
PROCEDURE TridiagonalParHouseholder (n: integer; VAR a: mat);
   VAR
      r: integer;
   BEGIN
      FOR r := 1 TO n - 2 DO
         TransformationHouseholder(n, r, a);
   END; { de "TridiagonalParHouseholder" }
```

La variante ci-dessous des deux dernières procédures permet de calculer en même temps le produit des matrices de transformation. Plus précisément, à la fin de la procédure `TridiagonalParHouseholder2`, la matrice Q est égale à

$$Q = P_{n-2} \cdots P_1$$

de sorte que l'on peut retrouver la matrice de départ à partir de la matrice A' obtenue en faisant le produit ${}^tQA'Q$.

```
PROCEDURE TransformationHouseholder2 (n, r: integer; VAR a, q: mat);
   VAR
      beta, K, normeV2: real;
      i, j: integer;
      v, p, w, z: vec;
   BEGIN  { de "TransformationHouseholder2" }
      VecteurHouseholder(n, a, r + 1, r, v);           v = v(r+1, A^(r)_(r-1))
      normeV2 := ProduitScalaire(n, v, v);
      IF EstNul(normeV2) THEN
         beta := 2
      ELSE
         beta := 2 / normeV2;
      MatriceParVecteur(n, a, v, p);
```

```
    VecteurParScalaire(n, p, beta, p);            p = βAv
    K := beta / 2 * ProduitScalaire(n, v, p);     K = ½β^t vp
    VecteurParScalaire(n, v, -K, w);
    VecteurPlusVecteur(n, p, w, w);               q = p − Kv
    VecteurParMatrice(n, v, q, z);
    VecteurParScalaire(n, z, beta, z);            z = βvQ
    FOR i := 1 TO n DO
       FOR j := 1 TO n DO BEGIN
          a[i, j] := a[i, j] - v[i] * w[j] - w[i] * v[j];
          q[i, j] := q[i, j] - v[i] * z[j];
       END;
 END; { de "TransformationHouseholder2" }
```

Bien entendu, on s'en sert dans une variante appropriée :

```
PROCEDURE TridiagonalParHouseholder2 (n: integer; VAR a, q: mat);
   VAR
      r: integer;
   BEGIN
      q := MatriceUnite;
      FOR r := 1 TO n - 2 DO
         TransformationHouseholder2(n, r, a, q);
      END; { de "TridiagonalParHouseholder2" }
```

Voici un exemple d'utilisation de ces procédures :

```
Donner l'ordre n :   5

Entrer la matrice (partie inférieure)
Ligne 1 :  4
Ligne 2 :  1 4
Ligne 3 :  2 1 10
Ligne 4 :  3 5 1 14
Ligne 5 :  4 1 7 6 14

Voici la matrice lue
   4.000   1.000   2.000   3.000   4.000
   1.000   4.000   1.000   5.000   1.000
   2.000   1.000  10.000   1.000   7.000
   3.000   5.000   1.000  14.000   6.000
   4.000   1.000   7.000   6.000  14.000

Vecteur v
   0.000   6.477   2.000   3.000   4.000
Vecteur w
   1.000  -1.503   0.748   1.521   0.919
Matrice A pour r = 1
   4.000  -5.477   0.000   0.000   0.000
  -5.477  23.467  -0.839  -0.345   1.061
   0.000  -0.839   7.009  -4.286   2.172
   0.000  -0.345  -4.286   4.873  -2.840
```

```
   0.000   1.061   2.172  -2.840   6.652
Vecteur v
   0.000   0.000  -2.235  -0.345   1.061
Vecteur w
   0.000   1.000  -0.565   2.065  -0.518
Matrice A pour r = 2
   4.000  -5.477   0.000   0.000   0.000
  -5.477  23.467   1.396   0.000  -0.000
   0.000   1.396   4.483   0.135   1.613
   0.000   0.000   0.135   6.298  -5.211
   0.000  -0.000   1.613  -5.211   7.752
Vecteur v
   0.000   0.000   0.000   1.753   1.613
Vecteur w
   0.000  -0.000   1.000  -0.166   0.180
Matrice A pour r = 3
   4.000  -5.477   0.000   0.000   0.000
  -5.477  23.467   1.396   0.000  -0.000
   0.000   1.396   4.483  -1.619   0.000
   0.000   0.000  -1.619   6.879  -5.260
   0.000  -0.000   0.000  -5.260   7.171
Matrice A tridiagonalisée (A’)
   4.000  -5.477   0.000   0.000   0.000
  -5.477  23.467   1.396   0.000  -0.000
   0.000   1.396   4.483  -1.619   0.000
   0.000   0.000  -1.619   6.879  -5.260
   0.000  -0.000   0.000  -5.260   7.171
Matrice Q de transformation
   1.000   0.000   0.000   0.000   0.000
   0.000  -0.183  -0.365  -0.548  -0.730
   0.000  -0.201  -0.663  -0.340   0.636
   0.000   0.849  -0.475   0.196  -0.122
   0.000   0.453   0.449  -0.739   0.216
Matrice tQA’Q
   4.000   1.000   2.000   3.000   4.000
   1.000   4.000   1.000   5.000   1.000
   2.000   1.000  10.000   1.000   7.000
   3.000   5.000   1.000  14.000   6.000
   4.000   1.000   7.000   6.000  14.000
```

Chapitre 5

Valeurs et vecteurs propres

Le calcul des valeurs propres d'une matrice A peut se faire en calculant d'abord son polynôme caractéristique, puis en déterminant les zéros de ce polynôme. Une autre famille de méthodes travaille directement sur la matrice : on transforme A progressivement en une matrice semblable à la matrice d'origine et qui a une forme particulière, sur laquelle les valeurs propres se «lisent» directement, comme une matrice diagonale ou triangulaire. L'accumulation des matrices de transformation permet d'obtenir également les vecteurs propres.

5.1 Méthode de Jacobi

La méthode de Jacobi est un procédé simple à mettre en œuvre pour le calcul des valeurs et vecteurs propres d'une matrice réelle symétrique.

5.1.1 Enoncé : méthode de Jacobi

Soit $n \geq 2$ un entier. Les matrices considérées sont réelles, carrées d'ordre n. Pour p, q dans $\{1, \ldots, n\}$, $p \neq q$, on note $G(q,p) = (g_{i,j})$ toute matrice de rotation de la forme

$$G(q,p) = \begin{pmatrix} 1 & & & & & & 0 \\ & \ddots & & & & & \\ & & c & & s & & \\ & & & \ddots & & & \\ & & -s & & c & & \\ & & & & & \ddots & \\ 0 & & & & & & 1 \end{pmatrix}$$

avec $c^2 + s^2 = 1$, qui ne diffère de la matrice unité que par les éléments d'indice (p, p), (p, q), (q, p), (q, q). Plus précisément,

$$\begin{array}{ll} g_{i,i} = 1 & i \neq p \text{ et } i \neq q \\ g_{i,j} = 0 & i \neq j \quad (i,j) \neq (p,q) \text{ et } (i,j) \neq (q,p) \\ g_{p,p} = g_{q,q} = c & \\ g_{p,q} = -g_{q,p} = s & \end{array}$$

7.– Soit $A = (a_{i,j})$ une matrice symétrique et soit $p \neq q$. Démontrer que $B = (b_{i,j}) = {}^tG(q, p)AG(q, p)$ vérifie

$$S(B) = S(A) - 2a_{p,q}^2 + 2b_{p,q}^2$$

où

$$S(A) = \sum_{\substack{i,j=1 \\ i \neq j}}^{n} a_{i,j}^2$$

(On note ${}^tM$ la transposée d'une matrice M.)

8.– Démontrer que l'on peut choisir $c \geq \sqrt{2}/2$ et s tels que

$$b_{p,q} = b_{q,p} = 0 \tag{$*$}$$

Démontrer que si de plus on choisit p, q tels que

$$|a_{p,q}| = \max_{1 \leq i < j \leq n} |a_{i,j}| \tag{$**$}$$

alors

$$S(B) \leq \left(1 - \frac{2}{n(n-1)}\right) S(A)$$

9.– En itérant le procédé de construction d'une matrice B à partir de A avec les choix $(**)$ et $(*)$, on obtient une suite de matrices $A^{(1)} = A, A^{(2)}, \ldots, A^{(k)}, \ldots$. Démontrer que $A^{(k)} = (a_{i,j}^{(k)})$ tend vers une matrice diagonale dont les valeurs propres sont les valeurs propres de la matrice A. (On pourra montrer que pour $i = 1, \ldots, n$,

$$|a_{i,i}^{(k+1)} - a_{i,i}^{(k)}| \leq \sqrt{S(A^{(k)})}$$

et que la suite $(a_{i,i}^{(k)})_{k \geq 1}$ est de Cauchy.)

10.– Ecrire un programme qui prend en argument la partie triangulaire inférieure d'une matrice symétrique et qui met en œuvre cet algorithme. On pourra arrêter les itérations lorsque $S(B) < \varepsilon$, pour $\varepsilon = 10^{-3}$.

Exemple numérique :

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

11.– On suppose de plus que les valeurs propres de la matrice A sont distinctes. Soient $G_1, G_2, \ldots$ les matrices de rotations employées dans l'algorithme. Démontrer que la suite

$$G_1 G_2 \cdots G_k \qquad (k \geq 1)$$

converge vers une limite G lorsque k tend vers l'infini. Démontrer que les colonnes de G sont des vecteurs propres associés aux valeurs propres de la matrice A. Donner une procédure qui calcule ces vecteurs propres pour la matrice donnée en exemple.

5.1.2 Solution : méthode de Jacobi

Soit $n \geq 2$ un entier. Les matrices considérées sont réelles, carrées d'ordre n. Pour p, q dans $\{1, \ldots, n\}$, $p \neq q$, on note $G(q,p) = (g_{i,j})$ toute matrice de rotation définie par

$$\begin{array}{ll} g_{i,i} = 1 & i \neq p \text{ et } i \neq q \\ g_{i,j} = 0 & i \neq j \quad (i,j) \neq (p,q) \text{ et } (i,j) \neq (q,p) \\ g_{p,p} = g_{q,q} = c & \\ g_{p,q} = -g_{q,p} = s & \end{array}$$

avec $c^2 + s^2 = 1$.

Soit $A = (a_{i,j})$ une matrice symétrique et soient p et q deux indices tels que $p \neq q$. Soit $B = (b_{i,j}) = {}^tG(q,p)AG(q,p)$. Alors les coefficients de B sont :

$$\begin{array}{ll} b_{i,j} = a_{i,j} & i \neq p,\, i \neq q \quad j \neq p,\, j \neq q \\ b_{p,j} = ca_{p,j} - sa_{q,j} & j \neq p,\, j \neq q \\ b_{q,j} = ca_{q,j} + sa_{p,j} & j \neq p,\, j \neq q \\ b_{p,p} = c^2 a_{p,p} + s^2 a_{q,q} - 2sca_{p,q} & \\ b_{q,q} = s^2 a_{p,p} + c^2 a_{q,q} + 2sca_{p,q} & \\ b_{p,q} = (c^2 - s^2)a_{p,q} + sc(a_{p,p} - a_{q,q}) & \end{array}$$

Proposition 5.1.1. *Si $a_{p,q} \neq 0$, il existe un réel $\theta \in\ ]-\pi/4, \pi/4]$, $\theta \neq 0$ unique tel qu'en posant $c = \cos\theta$ et $s = \sin\theta$, on ait $b_{p,q} = 0$.*

Preuve. Soit θ tel que

$$\tau = \operatorname{cotan} 2\theta = \frac{a_{q,q} - a_{p,p}}{2a_{p,q}}$$

et soit t la racine de plus petit module du trinôme

$$t^2 + 2\tau t - 1 = 0$$

(avec $t = 1$ si $\tau = 0$). Alors $t = \tan\theta$ et, en posant $c = \cos\theta$ et $s = \sin\theta$, on a $b_{p,q} = 0$. ∎

Un calcul rapide montre que t s'exprime simplement :

$$t = \frac{\varepsilon(\tau)}{|\tau| + \sqrt{1 + \tau^2}}$$

où $\varepsilon(u) = -1$ si $u < 0$ et $\varepsilon(u) = 1$ si $u \geq 0$. Comme

$$c = \frac{1}{\sqrt{1+t^2}} \qquad s = t\,c$$

aucun calcul de fonction trigonométrique n'est nécessaire pour évaluer c et s.

L'introduction de θ est utile pour prouver la convergence. Le paramètre convenable est le carré de la norme euclidienne. Pour toute matrice symétrique A, soit

$$S(A) = \sum_{\substack{i,j=1 \\ i\neq j}}^{n} a_{i,j}^2 = \operatorname{tr}(A^2) - \sum_{i=1}^{n} a_{i,i}^2$$

où $\operatorname{tr}(M)$ est la *trace* de la matrice M.

LEMME 5.1.2. *Soit A une matrice symétrique et soit $B = {}^t\!G(q,p)AG(q,p)$. Alors $S(B) = S(A) - 2a_{p,q}^2 + 2b_{p,q}^2$.*

Preuve. On a $B^2 = {}^t\!G(q,p)A^2G(q,p)$, donc $\operatorname{tr}(B^2) = \operatorname{tr}(G(q,p){}^t\!G(q,p)A^2) = \operatorname{tr}(A^2)$, soit

$$S(B) + \sum_{i=1}^{n} b_{i,i}^2 = S(A) + \sum_{i=1}^{n} a_{i,i}^2$$

Par ailleurs, l'expression des coefficients de B en fonction des coefficients de A montre que l'on a

$$\begin{pmatrix} b_{p,p} & b_{p,q} \\ b_{q,p} & b_{q,q} \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a_{p,p} & a_{p,q} \\ a_{q,p} & a_{q,q} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

A nouveau, les traces des carrés de ces matrices sont égales et celles-ci s'écrivent :

$$b_{p,p}^2 + b_{q,q}^2 + 2b_{p,q}^2 = a_{p,p}^2 + a_{q,q}^2 + 2a_{p,q}^2$$

Comme $b_{i,i} = a_{i,i}$ pour $i \neq p, q$, on obtient la relation. ∎

THÉORÈME 5.1.3. *Soit A une matrice symétrique et soit $(A^{(k)})$ la suite de matrice obtenue par $A^{(1)} = A$ et $A^{(k+1)} = {}^t\!G(q_k,p_k)A^{(k)}G(q_k,p_k)$, où le couple (q_k,p_k) est choisi à chaque étape tel que*

$$\left|a_{p_k,q_k}^{(k)}\right| = \max_{1\leq i<j\leq n} \left|a_{i,j}^{(k)}\right|$$

Alors la suite $A^{(k)}$ tend vers une matrice diagonale dont les éléments diagonaux sont les valeurs propres de A.

Rappelons que la convergence d'une suite de matrices d'ordre n fini est indépendante de la norme choisie. Elle équivaut à la convergence des n^2 suites de scalaires formées par les éléments des matrices.

Preuve. Soit A une matrice symétrique et soit $B = {}^tG(q,p)AG(q,p)$, où (q,p) sont choisis tels que

$$|a_{p,q}| = \max_{1\le i<j\le n} |a_{i,j}|$$

Alors on a, en sommant, $n(n-1)a_{p,q}^2 \ge S(A)$ et, comme $b_{p,q} = 0$, le lemme ci-dessus montre que

$$S(B) = S(A) - 2a_{p,q}^2 \le \left(1 - \frac{2}{n(n-1)}\right) S(A)$$

Par ailleurs et avec les mêmes notations que pour la preuve de la proposition 5.1.1

$$a_{q,q} - b_{q,q} = (a_{q,q} - a_{p,p})s^2 - 2a_{p,q}sc = 2a_{p,q}(\tau s^2 - sc) = -a_{p,q}t$$

Comme $|t| \le 1$ en vertu du choix de θ, on a $|a_{q,q} - b_{q,q}| \le |a_{p,q}|$ et de même $|a_{p,p} - b_{p,p}| \le |a_{p,q}|$. En particulier

$$|a_{i,i} - b_{i,i}| \le \sqrt{S(A)} \quad \text{pour tout } i = 1, \ldots, n. \tag{1.1}$$

Considérons maintenant la suite $(A^{(k)})$. On a

$$S(A^{(k)}) \le r^2 S(A^{(k-1)}) \quad \text{avec } r = \sqrt{1 - \frac{2}{n(n-1)}} < 1$$

donc $S(A^{(k)})$ tend vers 0 ; en conséquence, pour tout i, j avec $i \ne j$, la suite $|a_{i,j}^{(k)}|$ tend vers 0. Restent les éléments diagonaux. Or d'après (1.1),

$$|a_{i,i}^{(k+1)} - a_{i,i}^{(k)}| \le \sqrt{S(A^{(k)})} \le r^k \sqrt{S(A)}$$

et la suite $(a_{i,i}^{(k)})_{k\ge 0}$ converge donc pour tout i. Comme les matrices $A^{(k)}$ sont toutes semblables, elles ont toutes les mêmes valeurs propres, qui sont aussi les valeurs propres de la matrice limite. Sa diagonale est donc formée des valeurs propres de A. Ceci achève la preuve. ∎

Posons

$$\Lambda = \lim_{k\to\infty} A^{(k)}$$

Proposition 5.1.4. *Supposons les valeurs propres de A distinctes et, avec les notations du théorème, posons*

$$\Gamma_k = G(q_1, p_1)G(q_2, p_2) \cdots G(q_k, p_k)$$

Alors $G = \lim_{k\to\infty} \Gamma_k$ existe et ses colonnes sont des vecteurs propres associés aux valeurs propres de A ; plus précisément,

$$AG = G\Lambda$$

Preuve. Soient λ_i $(1 \le i \le n)$ les valeurs propres de A. Comme elles sont distinctes, le nombre

$$m = \frac{1}{2} \min_{1 \le i < j \le n} |\lambda_i - \lambda_j|$$

est non nul et, comme les suites $(a_{i,i}^{(k)})_{k \ge 0}$ convergent, on a, pour k assez grand et $i \neq j$,

$$\left| a_{q_k,q_k}^{(k)} - a_{q_k,q_k}^{(k)} \right| \ge m$$

Par ailleurs,

$$\left(a_{q_k,q_k}^{(k)} - a_{q_k,q_k}^{(k)} \right) \tan 2\theta_k = 2a_{p_k,q_k}^{(k)}$$

ce qui montre que $\tan 2\theta_k$ tend vers 0 et donc θ_k tend vers 0. On peut même être plus précis :

$$m|\tan 2\theta_k| \le 2 \left| a_{p_k,q_k}^{(k)} \right| \le r^{2k} \sqrt{S(A)}$$

donc θ_k tend vers zéro géométriquement.
Considérons maintenant la norme euclidienne $\|G(q_k, p_k) - I\|_E$. On a

$$\|G(q_k, p_k) - I\|_E = 4(1 - \cos\theta_k) \sim 2\theta_k$$

donc

$$\|\Gamma_k - \Gamma_{k-1}\|_E \le \|\Gamma_k\|_E \|G(q_k, p_k) - I\|_E \sim 2\theta_k$$

(parce que Γ_k est unitaire). Comme la suite (θ_k) tend vers 0 géométriquement, cela prouve la convergence de la suite (Γ_k). En passant à la limite dans l'expression

$$A^{(k+1)} = {}^t\Gamma_k A \Gamma_k$$

on obtient ${}^tGAG = \Lambda$, d'où la relation cherchée. ∎

5.1.3 Programme : méthode de Jacobi

Pour programmer la méthode de Jacobi, mettons d'abord en place le calcul des matrices de rotations. On n'en calcule que les coefficients non nuls, c'est-à-dire les nombres c et s. Ceci se fait comme suit, en utilisant les formules de la section précédente :

```
PROCEDURE CoefficientsJacobi (p, q: integer; VAR c, s: real; VAR a: mat);
   Calcule les nombres c et s tels que b_{p,q} = 0. On suppose que a_{p,q} ≠ 0.
   VAR
      tau, t: real;
   BEGIN
      tau := (a[q, q] - a[p, p]) / (2 * a[p, q]);
      IF EstNul(tau) THEN
         t := 1
      ELSE
```

```
        t := signe(tau) / (abs(tau) + sqrt(1 + sqr(tau)));
      c := 1 / sqrt(1 + sqr(t));
      s := t * c
    END; { de "CoefficientsJacobi" }
```

La stratégie à employer pour le choix des indices p, q s'écrit comme suit :

```
  PROCEDURE IndicesMax (n: integer; VAR a: mat; VAR p, q: integer);
    Calcule un couple (p, q), avec p < q, tel que |a_{p,q}| = max_{i≠j} |a_{i,j}|.
    VAR
      i, j: integer;
      maxlocal: real;
    BEGIN
      p := 1; q := 2; maxlocal := 0;
      FOR i := 1 TO n - 1 DO FOR j := i + 1 TO n DO
        IF abs(a[i, j]) > maxlocal THEN BEGIN
          maxlocal := abs(a[i, j]);
          p := i; q := j
        END;
    END; { de "IndicesMax" }
```

On peut alors transformer la matrice par la matrice de rotation.

```
  PROCEDURE TransformationJacobi (n, p, q: integer; c, s: real; VAR a: mat);
    On applique la matrice de rotation G(p, q) à la matrice A.
    VAR
      j: integer;
      u, v, w: real;
    BEGIN
      FOR j := 1 TO n DO
        IF (j <> p) AND (j <> q) THEN BEGIN
          v := a[p, j]; w := a[q, j];
          a[p, j] := c * v - s * w; a[j, p] := a[p, j];
          a[q, j] := s * v + c * w; a[j, q] := a[q, j]
        END;
      v := a[p, p]; u := a[p, q]; w := a[q, q];          La bande des quatre.
      a[p, p] := c * c * v - 2 * c * s * u + s * s * w;
      a[q, q] := s * s * v + 2 * c * s * u + c * c * w;
      a[p, q] := 0; a[q, p] := 0
    END; { de "TransformationJacobi" }
```

Comme les matrices sont symétriques, la moitié des coefficients sont inutiles et il aurait suffit de travailler uniquement sur la partie triangulaire supérieure par exemple. En fait, la programmation est plus simple avec la matrice entière et les opérations supplémentaires sont simplement des affectations.

Une étape de la méthode de Jacobi est réalisée en groupant les opérations précédentes. Dans la procédure qui suit, on met à jour également le nombre $S(A)$ (noté `sigma`) :

```
  PROCEDURE Jacobi (n: integer; VAR a: mat; VAR sigma: real);
```

```
   Réalise une itération de la méthode de Jacobi : choix des indices p, q, calcul de la matrice
   de transformation et application de la transformation. La somme des carrés des éléments
   non diagonaux est mise à jour dans sigma.
   VAR
      p, q: integer;
      c, s: real;
   BEGIN
      IndicesMax(n, a, p, q);
      CoefficientsJacobi(p, q, c, s, a);
      sigma := sigma - 2 * sqr(a[p, q]);
      TransformationJacobi(n, p, q, c, s, a)
   END; { de "Jacobi" }
```

Quand arrêter les itérations ? Une bonne indication est donnée par le fait que $S(A)$ ne change plus beaucoup. C'est ce critère que nous appliquons, avec en plus un test qui permet de sortir lorsque le nombre d'itérations dépasse une quantité fixée. La variable `MaxIteration` est à définir dans le programme entourant.

```
PROCEDURE ValeursPropresParJacobi (n: integer; VAR a: mat;
   VAR NIteration: integer);
   Remplace a par une matrice diagonale contenant les valeurs propres de a. Le nombre
   d'itérations est également rapporté.
   VAR
      sigma, sigmaN: real;
   BEGIN
      sigmaN := SommeDesCarres(n, a);
      NIteration := 0;
      REPEAT
         sigma := sigmaN;
         NIteration := NIteration + 1;
         Jacobi(n, a, sigmaN);
      UNTIL EstNul(sigmaN - sigma) OR (NIteration > MaxIteration);
   END; { de "ValeursPropresParJacobi" }
```

On initialise donc le processus en calculant le nombre $S(A)$ par :

```
FUNCTION SommeDesCarres (n: integer; VAR a: mat): real;
   Calcule la somme des carrés des coefficients non diagonaux d'une matrice symétrique a.
   VAR
      s: real;
      i, j: integer;
   BEGIN
      s := 0;
      FOR i := 1 TO n - 1 DO FOR j := i + 1 TO n DO
         s := s + sqr(a[i, j]);
      SommeDesCarres := 2 * s
   END; { de "SommeDesCarres" }
```

Voici un exemple d'exécution, avec quelques impressions intermédiaires :

```
Entrer la matrice
Ligne 1 :  1
Ligne 2 :  1 4
Ligne 3 :  0 1 4
Ligne 4 :  0 0 1 4
Voici la matrice lue
   1.000   1.000   0.000   0.000
   1.000   4.000   1.000   0.000
   0.000   1.000   4.000   1.000
   0.000   0.000   1.000   4.000
S = 6.00000000
Matrice a pour p = 1, q = 2
   0.697  -0.000  -0.290   0.000
  -0.000   4.303   0.957   0.000
  -0.290   0.957   4.000   1.000
   0.000   0.000   1.000   4.000
S = 4.00000000
Matrice a pour p = 3, q = 4
   0.697  -0.000  -0.205  -0.205
  -0.000   4.303   0.677   0.677
  -0.205   0.677   3.000   0.000
  -0.205   0.677   0.000   5.000
S = 2.00000000
Matrice a pour p = 2, q = 3
   0.697  -0.080  -0.189  -0.205
  -0.080   4.591   0.000   0.623
  -0.189   0.000   2.712  -0.265
  -0.205   0.623  -0.265   5.000
S = 1.08397484
Matrice a pour p = 2, q = 4
   0.697   0.055  -0.189  -0.213
   0.055   4.140   0.155  -0.000
  -0.189   0.155   2.712  -0.215
  -0.213  -0.000  -0.215   5.451
S = 0.30834481
Matrice a pour p = 3, q = 4
   0.697   0.055  -0.205  -0.198
   0.055   4.140   0.155  -0.012
  -0.205   0.155   2.695  -0.000
  -0.198  -0.012  -0.000   5.468

   ...

S = 0.00000005
Après 13 itérations,
voici la matrice "diagonale"
   0.667   0.000  -0.000   0.000
```

```
   0.000   4.156  -0.000   0.000
  -0.000  -0.000   2.701  -0.000
   0.000   0.000  -0.000   5.476
```

Voici un deuxième exemple :

```
Voici la matrice lue
   1.000   1.000   0.000   0.000   0.000
   1.000   4.000   1.000   0.000   0.000
   0.000   1.000   4.000   1.000   0.000
   0.000   0.000   1.000   4.000   1.000
   0.000   0.000   0.000   1.000   4.000
Après 23 itérations,
voici la matrice "diagonale"
   0.667  -0.000  -0.000   0.000  -0.000
  -0.000   3.512   0.000  -0.000  -0.000
  -0.000   0.000   2.449  -0.000  -0.000
   0.000  -0.000  -0.000   5.652  -0.000
  -0.000  -0.000  -0.000  -0.000   4.720
```

Dans les deux cas, $\varepsilon = 10^{-5}$. Venons-en au calcul des vecteurs propres. Il se fait simplement en accumulant les matrices de rotation. On emploie donc la procédure `PostMultiplier` du chapitre 4. La procédure principale prend alors la forme que voici :

```
PROCEDURE Jacobi2 (n: integer; VAR a, g: mat; VAR sigma: real);
   Réalise une itération de la méthode de Jacobi et accumule la matrice de transformation.
   VAR
      p, q: integer;
      c, s: real;
   BEGIN
      IndicesMax(n, a, p, q);
      CoefficientsJacobi(p, q, c, s, a);
      sigma := sigma - 2 * sqr(a[p, q]);
      TransformationJacobi(n, p, q, c, s, a);
      PostMultiplier(n, p, q, c, s, g);
   END; { de "Jacobi2" }
```

Finalement, on obtient la procédure que voici :

```
PROCEDURE ElementsPropresParJacobi (n: integer; VAR a, g: mat;
   VAR NIteration: integer);
   Remplace a par une matrice diagonale contenant les valeurs propres de a. La matrice
   g contient les vecteurs propres correspondants. Le nombre d'itérations est également
   rapporté.
   VAR
      sigma, sigmaN: real;
   BEGIN
      sigmaN := SommedesCarres(n, a);
      NIteration := 0;
```

```
      g := MatriceUnite;
      REPEAT
         sigma := sigmaN;
         NIteration := NIteration + 1;
         Jacobi2(n, a, g, sigmaN);
      UNTIL EstNul(sigmaN - sigma) OR (NIteration > MaxIteration);
   END; { de "ElementsPropresParJacobi" }
```

Voici deux exemples :

```
Voici la matrice lue
   1.000   1.000   1.000
   1.000   2.000   2.000
   1.000   2.000   3.000
Après 6 itérations,
voici la matrice "diagonale" D
   0.643   0.000   0.000
   0.000   0.308  -0.000
   0.000  -0.000   5.049
Matrice G des vecteurs propres
   0.737  -0.591   0.328
   0.328   0.737   0.591
  -0.591  -0.328   0.737
AG
   0.474  -0.182   1.656
   0.211   0.227   2.984
  -0.380  -0.101   3.721
GD
   0.474  -0.182   1.656
   0.211   0.227   2.984
  -0.380  -0.101   3.721
Voici la matrice lue
   1.000   1.000   0.000   0.000   0.000
   1.000   4.000   1.000   0.000   0.000
   0.000   1.000   4.000   1.000   0.000
   0.000   0.000   1.000   4.000   1.000
   0.000   0.000   0.000   1.000   4.000
Après 23 itérations,
voici la matrice "diagonale" D
   0.667  -0.000   0.000   0.000  -0.000
  -0.000   3.512   0.000  -0.000   0.000
   0.000   0.000   2.449  -0.000  -0.000
   0.000  -0.000  -0.000   5.652  -0.000
  -0.000   0.000  -0.000  -0.000   4.720
Matrice G des vecteurs propres
   0.943   0.209  -0.177   0.090  -0.166
  -0.314   0.526  -0.257   0.419  -0.619
   0.105  -0.466   0.575   0.603  -0.279
```

```
   -0.034  -0.298  -0.635   0.576   0.418
    0.010   0.611   0.410   0.349   0.580
 AG
    0.629   0.735  -0.434   0.509  -0.785
   -0.210   1.846  -0.629   2.370  -2.922
    0.070  -1.636   1.409   3.405  -1.318
   -0.023  -1.048  -1.557   3.254   1.972
    0.007   2.147   1.003   1.970   2.739
 GD
    0.629   0.735  -0.434   0.509  -0.785
   -0.210   1.846  -0.629   2.370  -2.922
    0.070  -1.636   1.409   3.405  -1.318
   -0.023  -1.048  -1.557   3.254   1.972
    0.007   2.147   1.003   1.970   2.739
```

5.2 Méthode QR

La méthode QR de calcul des valeurs propres d'une matrice est une méthode itérative fondée sur la décomposition QR (voir chapitre 3). C'est une méthode couramment employée.

Soit A une matrice carrée d'ordre n. On forme la suite (A_k) de matrices par $A_1 = A$ et

$$A_{k+1} = R_k Q_k = Q_{k+1} R_{k+1} \quad k \geq 1$$

où $A_k = Q_k R_k$ est une décomposition QR de la matrice A_k.

Posons $\mathcal{Q}_k = Q_1 Q_2 \cdots Q_k$ et $\mathcal{R}_k = R_k \cdots R_2 R_1$. Alors on a

$$A_{k+1} = {}^t\mathcal{Q}_k A \mathcal{Q}_k$$

de sorte que les matrices A_k sont toutes semblables à A. Nous allons montrer (sous des conditions assez restrictives) que les matrices A_k «deviennent» triangulaires supérieures en ce sens que les éléments au-dessous de la diagonale tendent vers zéro et que la diagonale converge (vers les valeurs propres de la matrice A).

Théorème 5.2.1. *Soit A une matrice carrée inversible dont les valeurs propres sont toutes de module différent. Soit P une matrice inversible telle que $A = P\Lambda P^{-1}$, avec $\Lambda = diag(\lambda_1, \ldots, \lambda_n)$ et $|\lambda_1| > \cdots > |\lambda_n| > 0$. On suppose de plus que la matrice P^{-1} possède une décomposition LU. Alors*

$$\begin{aligned} &\lim_{k\to\infty} (A_k)_{i,i} = \lambda_i \quad && 1 \leq i \leq n \\ &\lim_{k\to\infty} (A_k)_{i,j} = 0 \quad && 1 \leq j < i \leq n \end{aligned}$$

Avant de passer à la preuve, remarquons que les conclusions du théorème demeurent valables dans des cas plus généraux. Toutefois, les démonstrations sont plus compliquées que celle du théorème, qui est déjà assez longue.

Preuve. Considérons la puissance k-ième de la matrice A. On a

$$A^k = \mathcal{Q}_k \mathcal{R}_k$$

car $A^k = (Q_1R_1)^k = Q_1(R_1Q_1)^{k-1}R_1 = Q_1(Q_2R_2)^{k-1}R_1$ et plus généralement $A_j^\ell = Q_j A_{j+1}^{\ell-1} R_j$. Appelons

$$P = QR \qquad P^{-1} = LU$$

les factorisations QR et LU des matrices P et P^{-1}. Comme la matrice A est inversible, on a

$$A^k = P\Lambda^k P^{-1} = QR(\Lambda^k L \Lambda^{-k})\Lambda^k U$$

Or, la matrice L étant unitriangulaire inférieure, la matrice $\Lambda^k L \Lambda^{-k}$ est de même nature et de plus

$$(\Lambda^k L \Lambda^{-k})_{i,j} = \left(\frac{\lambda_i}{\lambda_j}\right)^k L_{i,j} \qquad i > j$$

et en vertu des hypothèses sur les valeurs propres de A, on a

$$\lim_{k\to\infty} \Lambda^k L \Lambda^{-k} = I$$

Posons

$$\Lambda^k L \Lambda^{-k} = I + F_k \qquad \text{avec } \lim_{k\to\infty} F_k = 0$$

Alors

$$R(\Lambda^k L \Lambda^{-k}) = (I + RF_kR^{-1})R$$

Pour des valeurs suffisamment grandes de k, les matrices $I + RF_kR^{-1}$ sont inversibles, puisque $\|RF_kR^{-1}\| < 1$, quelle que soit la norme. Elles admettent donc une factorisation QR

$$I + RF_kR^{-1} = \tilde{Q}_k \tilde{R}_k$$

qui est unique si l'on impose par exemple que $(\tilde{R}_k)_{i,i} > 0$, pour $1 \leq i \leq n$.

Montrons que les suites $\tilde{Q}_k$ et $\tilde{R}_k$ tendent vers la matrice unité. Les matrices $\tilde{Q}_k$ étant unitaires, la suite $(\tilde{Q}_k)$ est bornée (parce que $\|\tilde{Q}_k\|_2 = 1$). On peut donc en extraire une suite $(\tilde{Q}_{k'})$ qui converge vers une matrice $\tilde{Q}$, également unitaire. Comme

$$\tilde{R}_{k'} = {}^t\tilde{Q}_{k'}(I + RF_{k'}R^{-1})$$

la suite $(\tilde{R}_{k'})$ converge également, vers une matrice $\tilde{R}$ triangulaire supérieure, et telle que $(\tilde{R})_{i,i} \geq 0$ pour $1 \leq i \leq n$. En passant à la limite sur la suite extraite, on a

$$I = \tilde{Q}\tilde{R}$$

Ceci impose $(\tilde{R})_{i,i} > 0$ pour $1 \leq i \leq n$ et l'unicité de la factorisation QR montre que $\tilde{Q} = \tilde{R} = I$. Le même raisonnement vaut pour toute suite extraite. L'unicité des limites montre que les suites de départ convergent et que

$$\lim_{k\to\infty} \tilde{Q}_k = I \qquad \lim_{k\to\infty} \tilde{R}_k = I$$

En conclusion, on a

$$A^k = (Q\tilde{Q}_k)(\tilde{R}_k R \Lambda^k U) = \mathcal{Q}_k \mathcal{R}_k$$

et comme la matrice $Q\tilde{Q}_k$ est unitaire et $\tilde{R}_k R \Lambda^k U$ est triangulaire supérieure, nous voici en présence de deux décompositions QR de A^k. Il suffit de déterminer une matrice diagonale D_k dont les éléments diagonaux valent ± 1 et telle que $D_k \tilde{R}_k R \Lambda^k U$ ait ses éléments diagonaux positifs, pour pouvoir conclure à l'égalité des deux factorisations. On a donc

$$\mathcal{Q}_k = Q\tilde{Q}_k D_k$$

Revenons aux matrices A_k. Comme $A = QR\Lambda R^{-1}Q^{-1}$, on a

$$A_{k+1} = {}^t\mathcal{Q}_k A \mathcal{Q}_k = D_k B_k D_k$$

avec $B_k = {}^t\tilde{Q}_k R\Lambda R^{-1}\tilde{Q}_k$ et comme $\lim_{k\to\infty} \tilde{Q}_k = I$, on en conclut que

$$\lim_{k\to\infty} B_k = R\Lambda R^{-1}$$

De plus, cette matrice est triangulaire supérieure et a sur sa diagonale les valeurs propres $\lambda_1, \ldots, \lambda_n$ dans cet ordre. Maintenant

$$(A_{k+1})_{i,j} = (D_k)_{i,i}(D_k)_{j,j}(B_k)_{i,j}$$

de sorte que $(A_{k+1})_{i,i} = (B_k)_{i,i}$ pour $1 \le i \le n$ et $\lim_{k\to\infty}(A_{k+1})_{i,j} = 0$ pour $i > j$. Ceci achève la vérification. ■

La méthode est simple à programmer. Une itération consiste à décomposer la matrice et à faire le produit des facteurs dans l'ordre inverse. Voici une réalisation :

```
PROCEDURE IterationQR (n: integer; VAR a: mat);
        VAR
                q, r: mat;
        BEGIN
                DecompositionQR(n, a, q, r);
                MatriceParMatrice(n, r, q, a)
        END; { de "IterationQR" }
```

Dans cette procédure, `DecompositionQR` est l'une des procédures de décomposition qui ont été présentées au chapitre 3. Voici quelques exemples numériques, où les matrices imprimées sont les A_k, pour diverses valeurs de k.

```
Voici la matrice lue
   4.000   1.000   0.000   0.000   0.000
   1.000   4.000   1.000   0.000   0.000
   0.000   1.000   4.000   1.000   0.000
   0.000   0.000   1.000   4.000   1.000
   0.000   0.000   0.000   1.000   4.000
Matrice après 5 itérations :
   5.326   0.497   0.000  -0.000   0.000
```

```
   0.497   4.732   0.718   0.000  -0.000
   0.000   0.718   4.228   0.667  -0.000
  -0.000   0.000   0.667   3.312   0.326
   0.000  -0.000  -0.000   0.326   2.402
Matrice après 10 itérations :
   5.608   0.282  -0.000   0.000  -0.000
   0.282   5.035   0.298   0.000  -0.000
   0.000   0.298   4.064   0.158  -0.000
  -0.000   0.000   0.158   3.016   0.077
   0.000  -0.000  -0.000   0.077   2.276
Matrice après 30 itérations :
   5.731   0.021  -0.000   0.000  -0.000
   0.021   5.001   0.003   0.000  -0.000
   0.000   0.003   4.000   0.000  -0.000
  -0.000   0.000   0.000   3.000   0.000
   0.000  -0.000  -0.000   0.000   2.268
Matrice après 40 itérations :
   5.732   0.005  -0.000   0.000  -0.000
   0.005   5.000   0.000   0.000  -0.000
   0.000   0.000   4.000   0.000  -0.000
  -0.000  -0.000   0.000   3.000   0.000
   0.000   0.000  -0.000   0.000   2.268
```

La méthode QR a fait l'objet de beaucoup d'études et il existe des techniques variées d'accélération de la convergence. Elle ne converge pas toujours. En voici un exemple :

```
Voici la matrice lue
   0.000   7.000  -6.000
  -1.000   4.000   0.000
   0.000   2.000  -2.000
Matrice après 5 itérations :
   2.080   9.004   3.937
  -0.016  -0.156  -2.993
  -0.000  -0.322   0.076
     ...
Matrice après 40 itérations :
   2.000  -6.261   7.612
  -0.000  -0.800   0.131
   0.000   2.750   0.800
Matrice après 45 itérations :
   2.000   9.058   3.885
  -0.000  -0.098  -2.954
  -0.000  -0.335   0.098
Matrice après 50 itérations :
   2.000  -6.261   7.612
  -0.000  -0.800   0.131
```

```
     0.000   2.750   0.800
  Matrice après 55 itérations :
     2.000   9.058   3.885
    -0.000  -0.098  -2.954
    -0.000  -0.335   0.098
```

Notons que cette matrice a deux valeurs propres de même module, donc ne vérifie pas les hypothèses du théorème!

5.3 Valeurs propres de matrices tridiagonales

5.3.1 Enoncé : valeurs propres de matrices tridiagonales

Soit $n \geq 1$ un entier. On considère une matrice carrée symétrique tridiagonale

$$A = \begin{pmatrix} a_1 & b_2 & 0 & \cdots & 0 \\ b_2 & a_2 & b_3 & & \vdots \\ 0 & b_3 & a_3 & \ddots & 0 \\ \vdots & & \ddots & \ddots & b_n \\ 0 & \cdots & 0 & b_n & a_n \end{pmatrix}$$

telle que $b_i \neq 0$ pour $i = 2, \ldots, n$. On note $p_r(x) = \text{dét}\,(A_r - xI_r)$, où I_r est la matrice unité d'ordre r, le polynôme caractéristique de la matrice

$$A_r = \begin{pmatrix} a_1 & b_2 & & 0 \\ b_2 & & \ddots & \\ & \ddots & \ddots & b_r \\ 0 & & b_r & a_r \end{pmatrix}$$

et on pose $p_0(x) = 1$.

1.– Vérifier que $p_1(x) = a_1 - x$ et

$$p_r(x) = (a_r - x)p_{r-1}(x) - b_r^2 p_{r-2}(x) \qquad 2 \leq r \leq n$$

2.– Démontrer que pour $2 \leq r \leq n$, les zéros des $p_r(x)$ sont réels, simples et que

$$x_1 < y_1 < x_2 < y_2 < \cdots < y_{r-1} < x_r$$

où $x_1, \ldots, x_r$ sont les zéros de $p_r(x)$ et $y_1, \ldots, y_{r-1}$ sont les zéros de $p_{r-1}(x)$, et donner une majoration des valeurs absolues des zéros de $p_r(x)$ en fonction des éléments de A_r.

3.– Ecrire une procédure qui calcule les valeurs propres de la matrice A en calculant successivement des valeurs approchées des zéros des polynômes $p_r(x)$, pour $r = 1, \ldots, n$.

Exemple numérique :

$$A = \begin{pmatrix} 4 & 1 & 0 & 0 & 0 \\ 1 & 5 & 2 & 0 & 0 \\ 0 & 2 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 4 \end{pmatrix}$$

4.– Quelles sont les modifications à apporter dans le cas où l'un des coefficients b_i dans la matrice A serait nul ?

5.– On suppose que la matrice tridiagonale A est de plus définie positive. Démontrer qu'il existe alors une matrice triangulaire inférieure L telle que $A = L\,{}^tL$ (On note ${}^tM$ la transposée d'une matrice M).

6.– On suppose de plus que les valeurs propres de A sont distinctes. On définit une suite A_r de matrices par $A_1 = A$ et $A_{r+1} = {}^tL_rL_r$ où L_r est une matrice triangulaire inférieure telle que $A_r = L_r\,{}^tL_r$. Démontrer que A_r est tridiagonale symétrique. On montre que A_r tend vers une matrice diagonale lorsque r tend vers l'infini. Ecrire une procédure qui calcule la suite A_r de matrices pour une matrice A tridiagonale symétrique définie positive et comparer la diagonale aux valeurs propres obtenues dans l'exemple numérique ci-dessus.

5.3.2 Solution : valeurs propres de matrices tridiagonales

Soient

$$A = \begin{pmatrix} a_1 & b_2 & 0 & \cdots & 0 \\ b_2 & a_2 & b_3 & & \vdots \\ 0 & b_3 & a_3 & \ddots & 0 \\ \vdots & & \ddots & \ddots & b_n \\ 0 & \cdots & 0 & b_n & a_n \end{pmatrix} \qquad A_r = \begin{pmatrix} a_1 & b_2 & & 0 \\ b_2 & & \ddots & \\ & \ddots & \ddots & b_r \\ 0 & & b_r & a_r \end{pmatrix}$$

pour $1 \le r \le n$. On note $p_r(x) = \text{dét}\,(A_r - xI_r)$, où I_r est la matrice unité d'ordre r, le polynôme caractéristique de la matrice et on pose $p_0(x) = 1$. On suppose que les b_r sont non nuls.

Proposition 5.3.1. *On a* $p_1(x) = a_1 - x$ *et*

$$p_r(x) = (a_r - x)p_{r-1}(x) - b_r^2 p_{r-2}(x) \qquad 2 \le r \le n$$

De plus, les zéros des $p_r(x)$ *sont réels, simples et pour* $2 \le r \le n$,

$$x_1 < y_1 < x_2 < y_2 < \cdots < y_{r-1} < x_r$$

où $x_1, \ldots, x_r$ *sont les zéros de* $p_r(x)$ *et* $y_1, \ldots, y_{r-1}$ *sont les zéros de* $p_{r-1}(x)$.

Preuve. La formule de récurrence sur les polynômes p_r s'obtient en développant le déterminant par rapport à sa dernière ligne. Quant à la propriété d'entrelacement des zéros, elle est un cas particulier de la proposition 7.1.5. ∎

Pour majorer les zéros, on peut utiliser par exemple l'inégalité de Hadamard qui affirme, rappelons-le, que, pour toute matrice $A = (a_{i,j})$ d'ordre n et toute valeur propre λ, on a

$$|\lambda| \leq \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_{i,j}|$$

Lorsqu'un des éléments b_r de la matrice est nul, la matrice se décompose par blocs et on est ramené à deux sous-problèmes de taille plus petite.

5.3.3 Programme : calcul par dichotomie

Une première façon de calculer les valeurs propres de A, consiste à calculer les zéros de p_n. Pour cela, on doit d'abord en calculer les coefficients, puis en localiser les zéros. Nous allons voir deux méthodes pour ce faire : une première, par dichotomie et une deuxième, fondée sur les suites de Sturm. Nous avons vu et nous verrons d'autres méthodes pour calculer les valeurs propres sans passer par le polynôme caractéristique.

Les relations de récurrence de la proposition 5.3.1 permettent de calculer immédiatement les coefficients des polynômes p_r. Nous utilisons librement la bibliothèque de manipulation de polynômes décrite dans l'annexe A. Voici une réalisation :

```
PROCEDURE PolynomeCaracteristique (r: integer; a, b: real; p1, p2: pol;
   VAR p: pol);
```
Calcule $p(x) = (a - x)p_1(x) - b^2 p_2(x)$. Les degrés de p, p_1, p_2 sont respectivement r, $r - 1$ et $r - 2$.
```
   VAR
      i: integer;
   BEGIN
      p := PolynomeNul;
      p[0] := a * p1[0] - sqr(b) * p2[0];
      FOR i := 1 TO r DO
         p[i] := a * p1[i] - p1[i - 1] - sqr(b) * p2[i]
   END; { de "PolynomeCaracteristique" }
```

Les polynômes sont rangés dans une table de type :

```
TYPE
   SuitePol = ARRAY[0..OrdreMax] OF pol;
```

Ceci est fait par la procédure :

```
PROCEDURE LesPolynomesCaracteristiques (n: integer; a, b: vec;
   VAR p: SuitePol);
```

Calcule la suite des polynômes p_r, pour $r = 0, \dots, n$. Le dernier est le polynôme caractéristique de la matrice.
```
VAR
   r: integer;
BEGIN
   p[0] := PolynomeUnite;                    p[0] = 1.
   p[1] := PolynomeNul;                      p[1] = a[1] - X.
   p[1][0] := a[1];
   p[1][1] := -1;
   FOR r := 2 TO n DO
      PolynomeCaracteristique(r, a[r], b[r], p[r - 1], p[r - 2], p[r]);
END; { de "LesPolynomesCaracteristiques" }
```

Pour calculer les zéros de ces polynômes, et en particulier ceux de p_n, on suppose que $b_i \neq 0$ pour $i = 2, \dots, n$ et on utilise la démarche suivante : si l'on connaît les zéros de p_{r-1}, soit $z_1, \dots, z_{r-1}$ et si, de plus, on connaît un minorant strict z_0 et un majorant strict z_r des zéros de p_r, on sait que le polynôme p_r a un et un seul zéro dans chacun des intervalles ouverts $]z_{k-1}, z_k[$, pour $k = 1, \dots, r$. En particulier, les signes de $p_r(z_{k-1})$ et de $p_r(z_k)$ sont opposés et on peut donc appliquer la méthode de recherche dichotomique du zéro de p_r dans cet intervalle. On utilise pour cela la procédure ci-dessous. Elle fait appel à la procédure `Valeurd(d, p, x)` qui évalue le polynôme p de degré d au point x et qui permet d'accélérer les calculs (voir la discussion dans l'annexe A).

```
FUNCTION ZeroParDichotomie (a, b: real; p: pol): real;
```
On sait que $p(a)p(b) < 0$ et qu'il n'y a qu'un seul zéro de p dans l'intervalle $[a, b]$. On le calcule par dichotomie et on s'arrête lorsque $|a - b| < \varepsilon$.
```
VAR
   Va, Vb, Vm, m: real;
   d: integer;
BEGIN
   d := degre(p);
   Va := Valeurd(d, p, a);
   Vb := Valeurd(d, p, b);
   WHILE NOT EstNul(a - b) DO BEGIN
      m := (a + b) / 2; Vm := Valeur(d, p, m);
      IF Va * Vm < 0 THEN BEGIN
         b := m; Vb := Vm
      END
      ELSE BEGIN
         a := m; Va := Vm
      END
   END;
   ZeroParDichotomie := a
END; { de "ZeroParDichotomie" }
```

Par commodité, nous définissons le nouveau type `vec0` qui a une coordonnée supplémentaire, d'indice 0, par rapport au type `vec`.

```
TYPE
   vec0 = ARRAY[0..OrdreMax] OF real;
```

La procédure suivante calcule les zéros de p (supposés tous réels), en connaissant pour chaque zéro un encadrement.

```
PROCEDURE LesZerosParDichotomie (r: integer; I: vec0; p: pol; VAR z: vec0);
```
Calcule les zéros du polynôme p de degré r. On suppose que p a exactement un zéro dans l'intervalle $]I_{k-1}, I_k[$, pour $1 \leq k \leq r$ et que p prend des valeurs de signes opposés aux bornes de cet intervalle. On calcule ce zéro et on le range dans z_k.
```
   VAR
      k: integer;
   BEGIN
      FOR k := 1 TO r DO z[k] := ZeroParDichotomie(I[k - 1], I[k], p)
   END; { de "LesZerosParDichotomie" }
```

Il ne reste donc plus qu'à calculer des majorants. Pour cela, on utilise l'inégalité de Hadamard qui affirme que le rayon spectral (c'est-à-dire le module maximal des valeurs propres) d'une matrice $A = (a_{i,j})$ est majoré par

$$\max_{1 \leq i \leq n} \sum_{j=1}^{n} |a_{i,j}|$$

On calcule l'expression pour chaque matrice A_r. Notons-la d_r ; on sait alors que les zéros de p_r sont compris entre $-d_r$ et d_r. La procédure que voici calcule les majorants. Le nombre s est, à tout moment, le maximum des $\sum_{j=1}^{n} |a_{i,j}|$ pour $1 \leq i \leq r-1$. Evidemment, `rmax` calcule le maximum de deux réels.

```
PROCEDURE MajorantsParHadamard (n: integer; a, b: vec; VAR d: vec);
   VAR
      k: integer;
      s: real;
   BEGIN
      d[1] := abs(a[1]);
      d[2] := abs(b[2]) + rMax(abs(a[1]), abs(a[2]));
      s := abs(a[1]) + abs(b[2]);
      FOR k := 3 TO n DO BEGIN
         s := rmax(s, abs(b[k - 1]) + abs(a[k - 1]) + abs(b[k]));
         d[k] := rmax(s, abs(b[k]) + abs(a[k]))
      END;
   END; { de "MajorantsParHadamard" }
```

Ces procédures, mises ensemble, donnent finalement la procédure recherchée, qui calcule directement les valeurs propres par évaluation du polynôme caractéristique et recherche de ses zéros :

```
PROCEDURE ValeursPropresParDichotomie (n: integer; a, b: vec; VAR vp: vec);
```
Calcule les valeurs propres d'une matrice tridiagonale symétrique d'ordre n. La diagonale est dans le vecteur a, la sous-diagonale dans b. On suppose qu'aucun coefficient de b n'est nul. Les valeurs propres sont rangées dans vp.

```
VAR
   r: integer;
   tp: SuitePol;
   z: vec0;
   d: vec;
BEGIN
   LesPolynomesCaracteristiques(n, a, b, tp);   Les polynômes p_r.
   MajorantsParHadamard(n, a, b, d);            Calcul des majorants.
   z[1] := a[1];                                Le seul zéro de p_1.
   FOR r := 2 TO n DO BEGIN
      z[0] := -d[r]; z[r] := d[r];              Mise en place des majorants.
    LesZerosParDichotomie(r, z, tp[r], z);      Calcul des zéros de p_r.
   END;
   FOR r := 1 TO n DO
      vp[r] := z[r]                             Recopie des résultats.
END; { de "ValeursPropresParDichotomie" }
```

Voici un exemple de déroulement, avec quelques impressions intermédiaires :

```
Donner l'ordre n :   5
Donner la diagonale
4 5 3 1 4
Donner la sous-diagonale
1 2 1 1
Voici la matrice lue
   4.000   1.000   0.000   0.000   0.000
   1.000   5.000   2.000   0.000   0.000
   0.000   2.000   3.000   1.000   0.000
   0.000   0.000   1.000   1.000   1.000
   0.000   0.000   0.000   1.000   4.000
Pour r = 2
Le polynôme est :
   X^2 - 9.000 X + 19.000
Les intervalles de localisation sont :
  -6.000   4.000   6.000
et les zéros du polynôme sont :
   3.382   5.617
Pour r = 3
Le polynôme est :
   - X^3 + 12.000 X^2 - 42.000 X + 41.000
Les intervalles de localisation sont :
  -8.000   3.382   5.617   8.000
et les zéros du polynôme sont :
   1.638   3.833   6.529
Pour r = 4
Le polynôme est :
   X^4 - 13.000 X^3 + 53.000 X^2 - 74.000 X + 22.000
```

```
Les intervalles de localisation sont :
  -8.000   1.638   3.833   6.529   8.000
et les zéros du polynôme sont :
   0.402   2.145   3.883   6.569
Pour r = 5
Le polynôme est :
   - X^5 + 17.000 X^4 - 104.000 X^3 + 274.000 X^2 - 276.000 X + 47.000
Les intervalles de localisation sont :
  -8.000   0.402   2.145   3.883   6.569   8.000
et les zéros du polynôme sont :
   0.211   2.028   3.847   4.340   6.572
Les valeurs propres sont
   0.211   2.028   3.847   4.340   6.572
Le polynôme caractéristique est
   - X^5 + 17.000 X^4 - 104.000 X^3 + 274.000 X^2 - 276.000 X + 47.000
On a
p( 0.21077 ) =  0.05816
p( 2.02791 ) = -0.01254
p( 3.84672 ) =  0.00564
p( 4.34024 ) = -0.00692
p( 6.57180 ) =  0.10053
```

Notons la grande taille des coefficients du polynôme caractéristique. Ce phénomène est encore plus frappant dans l'exemple suivant :

```
Donner l'ordre n :   5
Donner la diagonale
4 4 4 4 4
Donner la sous-diagonale
1 1 1 1
Voici la matrice lue
   4.000   1.000   0.000   0.000   0.000
   1.000   4.000   1.000   0.000   0.000
   0.000   1.000   4.000   1.000   0.000
   0.000   0.000   1.000   4.000   1.000
   0.000   0.000   0.000   1.000   4.000
Les valeurs propres sont
   2.268   3.000   4.000   5.000   5.731
Le polynôme caractéristique est
  - X^5 + 20.000 X^4 - 156.000 X^3 + 592.000 X^2 - 1091.000 X + 780.000
On a
p( 2.268 ) =  0.00411
p( 3.000 ) = -0.00099
p( 4.000 ) =  0.00082
p( 5.000 ) = -0.00106
p( 5.731 ) =  0.00889
```

5.3.4 Programme : calcul par suites de Sturm

La suite de polynômes définie par

$$\begin{aligned} &p_0(X) = 1 \\ &p_1(X) = a_1 - X \\ &p_r(X) = (a_r - X)p_{r-1} - b_r^2 p_{r-2}(X) \qquad 2 \leq r \end{aligned}$$

vérifie les hypothèses de la proposition 7.1.5. Il en résulte directement que chaque $p_n(X)$ a n zéros réels simples. La suite est aussi une suite de Sturm par la proposition 7.1.6. Avec les notations de la section 7.1, le nombre de zéros de p_n dans l'intervalle $]-\infty, \alpha]$ est égal à

$$V(p_n, \ldots, p_0; \alpha) - V(p_n, \ldots, p_0; -\infty)$$

Notons $W(\alpha)$ ce nombre. La fonction $\alpha \to W(\alpha)$ croît de 0 à n lorsque α varie de $-\infty$ à ∞. Pour déterminer les zéros de p_n et donc les valeurs propres cherchées, on est donc ramené au calcul des points de discontinuité de W.

Soient $x_1 < \ldots < x_n$ les zéros de p_n. Au voisinage de x_i, on a $W(\alpha) = i - 1$, si $\alpha \leq x_i$ et $W(\alpha) = i$, si $\alpha > x_i$. L'algorithme sera donc le suivant : partant d'un intervalle $[\alpha, \beta]$ tel que $W(\alpha) = i - 1$ et $W(\beta) = i$, on procède par dichotomie en coupant l'intervalle en deux intervalles $[\alpha, \gamma]$ et $[\gamma, \beta]$ jusqu'à obtenir un intervalle de longueur suffisamment petite. Selon que $W(\gamma) = i - 1$ ou non, c'est le premier ou le second des deux intervalles qui est conservé.

La valeur initiale de β est fournie par une majoration du rayon spectral de la matrice. Comme valeur initiale de α on prendra l'extrémité supérieure de l'intervalle de l'itération précédente et au départ l'opposé de la majoration du rayon spectral.

La procédure suivante est proche de celle de même nom de la section 7.1. Le calcul de la variation de la suite de Sturm en un point est fait fréquemment. C'est pourquoi les carrés b_r^2 ne sont évalués qu'une seule fois et rangés dans un tableau `b2`. On obtient :

```
FUNCTION Variation (n: integer; VAR a, b2: vec; x: real): integer;
   Calcule la variation V(x) au point x.
   VAR
      i, s, changements: integer;
      p, q, r: real;
   BEGIN
      changements := 0;                 Compteur de changements de signe.
      p := 1;                           Initialisation p = a0.
      s := 1;                           Signe de p = a0.
      r := 0;
      FOR i := 1 TO n DO BEGIN
         q := (a[i] - x) * p - b2[i] * r;  q = pi(x).
         r := p;
         p := q;
         IF signe(q) = -s THEN BEGIN    Ainsi, on saute les termes nuls.
            s := -s;
```

```
            changements := changements + 1
         END;
      END;
    Variation := changements
   END; { de "Variation" }
```

L'algorithme dichotomique s'écrit alors comme suit :

```
PROCEDURE ZeroParVariation (i, n, VMoinsInfini: integer; VAR a, b2: vec;
   VAR alpha, beta: real);
   Calcul de la i-ième valeur propre.
   VAR
      gamma: real;
   BEGIN
      WHILE NOT EstNul(alpha - beta) DO BEGIN
         gamma := (alpha + beta) / 2;
         IF Variation(n, a, b2, gamma) - VMoinsInfini = i - 1 THEN
            alpha := gamma
         ELSE
            beta := gamma
      END;
   END; { de "ZeroParVariation" }
```

Le calcul important est celui du nombre $W(\gamma) = V(\gamma) - V(-\infty)$. Selon la valeur de ce nombre, on réduit l'extrémité gauche ou l'extrémité droite de l'intervalle. La procédure précédente est mise en œuvre par :

```
PROCEDURE ValeursPropresParVariation (n: integer; a, b: vec; VAR vp: vec);
   VAR
      i, r: integer;
      b2: vec;
      infini, alpha, beta: real;
      VMoinsInfini: integer;
   BEGIN
      b2[1] := 0;
      FOR i := 2 TO n DO
         b2[i] := sqr(b[i]);
      infini := MajorantRayonSpectral(n, a, b);
      VMoinsInfini := Variation(n, a, b2, -infini);
      beta := -infini;
      FOR r := 1 TO n DO BEGIN
         alpha := beta;
         beta := infini;
         ZeroParVariation(r, n, VMoinsInfini, a, b2, alpha, beta);
         vp[r] := (alpha + beta) / 2;
      END;
   END; { de "ValeursPropresParVariation" }
```

La majoration du rayon spectral s'obtient en utilisant l'inégalité de Hadamard :

```
FUNCTION MajorantRayonSpectral (n: integer; a, b: vec): real;
   VAR
      k: integer;
      s: real;
   BEGIN
      s := abs(a[1]) + abs(b[2]);
      FOR k := 2 TO n - 1 DO
         s := rMax(s, abs(b[k]) + abs(a[k]) + abs(b[k + 1]));
      MajorantRayonSpectral := rMax(s, abs(b[n]) + abs(a[n]));
   END; { de "MajorantRayonSpectral" }
```

Voici un exemple de l'exécution de cette procédure, avec des impressions intermédiaires :

```
Voici la matrice lue
   4.000   1.000   0.000   0.000   0.000
   1.000   5.000   2.000   0.000   0.000
   0.000   2.000   3.000   1.000   0.000
   0.000   0.000   1.000   1.000   1.000
   0.000   0.000   0.000   1.000   4.000
r=1
  alpha=  -8.000  beta=   8.000  gamma=   0.000  W(gamma)=  0
  alpha=   0.000  beta=   8.000  gamma=   4.000  W(gamma)=  3
  alpha=   0.000  beta=   4.000  gamma=   2.000  W(gamma)=  1
  alpha=   0.000  beta=   2.000  gamma=   1.000  W(gamma)=  1
  alpha=   0.000  beta=   1.000  gamma=   0.500  W(gamma)=  1
  alpha=   0.000  beta=   0.500  gamma=   0.250  W(gamma)=  1
  alpha=   0.000  beta=   0.250  gamma=   0.125  W(gamma)=  0
  alpha=   0.125  beta=   0.250  gamma=   0.188  W(gamma)=  0
  alpha=   0.188  beta=   0.250  gamma=   0.219  W(gamma)=  1
  alpha=   0.188  beta=   0.219  gamma=   0.203  W(gamma)=  0
  alpha=   0.203  beta=   0.219  gamma=   0.211  W(gamma)=  0
  alpha=   0.211  beta=   0.219  gamma=   0.215  W(gamma)=  1
  alpha=   0.211  beta=   0.215  gamma=   0.213  W(gamma)=  1
  alpha=   0.211  beta=   0.213  gamma=   0.212  W(gamma)=  1
Intervalle : 0.21094 0.21191
r=2
  alpha=   0.212  beta=   8.000  gamma=   4.106  W(gamma)=  3
  alpha=   0.212  beta=   4.106  gamma=   2.159  W(gamma)=  2
    ...
  alpha=   2.026  beta=   2.030  gamma=   2.028  W(gamma)=  1
  alpha=   2.028  beta=   2.030  gamma=   2.029  W(gamma)=  2
Intervalle : 2.02774 2.02869
r=3
  alpha=   2.029  beta=   8.000  gamma=   5.014  W(gamma)=  4
  alpha=   2.029  beta=   5.014  gamma=   3.522  W(gamma)=  2
    ...
  alpha=   3.845  beta=   3.848  gamma=   3.847  W(gamma)=  2
  alpha=   3.847  beta=   3.848  gamma=   3.847  W(gamma)=  2
```

```
Intervalle : 3.84735 3.84807

r=4
  alpha=   3.848  beta=   8.000  gamma=   5.924  W(gamma)=  4
  alpha=   3.848  beta=   5.924  gamma=   4.886  W(gamma)=  4
    ...
  alpha=   4.341  beta=   4.343  gamma=   4.342  W(gamma)=  4
  alpha=   4.341  beta=   4.342  gamma=   4.341  W(gamma)=  4
Intervalle : 4.34071 4.34122

r=5
  alpha=   4.341  beta=   8.000  gamma=   6.171  W(gamma)=  4
  alpha=   6.171  beta=   8.000  gamma=   7.085  W(gamma)=  5
    ...
  alpha=   6.571  beta=   6.574  gamma=   6.573  W(gamma)=  5
  alpha=   6.571  beta=   6.573  gamma=   6.572  W(gamma)=  4
Intervalle : 6.57168 6.57257

Voici les valeurs propres
   0.211   2.028   3.848   4.341   6.572
```

5.4 Méthode LR de Rutishauser

Dans cette section, nous présentons une méthode, très simple à mettre en œuvre, pour calculer les valeurs propres d'une matrice, lorsqu'elles sont toutes réelles. Les preuves de convergence sont délicates et nous renvoyons à la littérature pour tous les aspects théoriques.

Soit $n \geq 1$ un entier, et soit A une matrice carrée réelle d'ordre n dont les valeurs propres sont réelles. La méthode de Rutishauser est fondée sur la décomposition LU, que nous avons décrite au chapitre 3 (et que Rutishauser appelle LR, d'où le nom). On supposera en particulier que toutes les matrices considérées possèdent une décomposition LU. On définit alors une suite de matrices (A_k) par $A_1 = A$ et

$$A_{k+1} = U_k L_k \qquad (A_k = L_k U_k)$$

où le couple (L_k, U_k) est la décomposition LU de A_k, c'est-à-dire que L_k est unitriangulaire inférieure, et U_k est triangulaire supérieure. On peut montrer que, sous certaines hypothèses, les suites (L_k) et (U_k) convergent et que L_k tend vers la matrice unité d'ordre n ; alors U_k tend vers une matrice U dont la diagonale contient les valeurs propres de A.

La mise en œuvre de cet algorithme est particulièrement simple — et c'est le cas que nous considérons — lorsque la matrice A est, dès le départ, tridiagonale. En effet, comme nous l'avons vu au chapitre 4, la décomposition LU est alors très facile à écrire et l'opération complémentaire, qui à partir d'une décomposition $A = LU$ calcule $A' = UL$, l'est encore plus ; comme la matrice A' est à nouveau tridiagonale, le processus peut continuer.

Une matrice tridiagonale A s'écrit sous la forme

$$A = \begin{pmatrix} a_1 & b_1 & 0 & & \dots & 0 \\ c_2 & a_2 & b_2 & 0 & & \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & c_{n-2} & a_{n-2} & b_{n-2} & 0 \\ & & & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \dots & & 0 & c_n & a_n \end{pmatrix}$$

donc est représentée économiquement par les trois vecteurs a, b, c. Si $A = LU$, les matrices L et U sont aussi tridiagonales; plus précisément, posons

$$A = LU = \begin{pmatrix} 1 & & & & 0 \\ \ell_2 & 1 & & & \\ & \ell_3 & 1 & & \\ & & \ddots & \ddots & \\ 0 & & & \ell_n & 1 \end{pmatrix} \begin{pmatrix} u_1 & v_1 & & & 0 \\ & u_2 & v_2 & & \\ & & \ddots & \ddots & \\ & & & & v_{n-1} \\ 0 & & & & u_n \end{pmatrix}$$

L'identification donne $v = b$ et $u_1 = a_1$, et

$$\begin{aligned} \ell_i &= c_i/u_{i-1} \\ u_i &= a_i - \ell_i b_{i-1} \end{aligned}$$

pour $i = 2, \dots, n$. Ainsi, la décomposition LU est représentée de façon compacte par les trois vecteurs ℓ, u et v. De plus, $v = b$. Rappelons la procédure suivante :

```
PROCEDURE DecompositionLUtridiagonal (n: integer; a, b, c: vec;
   VAR u, l: vec);
```

du chapitre 4 qui calcule la décomposition LU d'une matrice tridiagonale. La recomposition, qui calcule la matrice tridiagonale UL, est réalisée par

```
PROCEDURE ULTridiagonal (n: integer; u, l: vec; VAR a, b, c: vec);
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO n - 1 DO a[i] := u[i] + b[i] * l[i + 1];
      a[n] := u[n];
      FOR i := 2 TO n DO c[i] := u[i] * l[i];
   END; { de "ULTridiagonal" }
```

Ainsi, l'étape qui consiste à passer de A_k à A_{k+1} s'écrit simplement :

```
PROCEDURE IterationLUTridiagonal (n: integer; VAR a, b, c: vec);
   VAR
      u, l: vec;
   BEGIN
      DecompositionLUTridiagonal(n, a, b, c, u, l);
```

```
      ULTridiagonal(n, u, l, a, b, c);
   END; { de "IterationLUTridiagonal" }
```

Un bon critère pour l'arrêt des itérations porte sur la norme infinie de la sous-diagonale, c'est-à-dire du vecteur c, puisqu'il doit tendre vers 0.

Mentionnons deux cas particuliers. Le premier concerne les matrices (tridiagonales) sous forme de Lanczos (c'est-à-dire telles que la sur-diagonale a tous ses coefficients égaux à 1). Dans ce cas, la décomposition et la recomposition sont encore plus simples. La procédure de décomposition a été donnée au chapitre 4. Voici son en-tête :

```
PROCEDURE LanczosLU (n: integer; a, c: vec; VAR u, l: vec);
```

La recomposition se fait par :

```
PROCEDURE LanczosUL (n: integer; u, l: vec; VAR a, c: vec);
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO n - 1 DO a[i] := u[i] + l[i + 1];
      a[n] := u[n];
      FOR i := 2 TO n DO c[i] := u[i] * l[i];
   END; { de "LanczosUL" }
```

Comme ci-dessus, une itération de la méthode de Rutishauser pour ces matrices combine les deux procédures.

```
PROCEDURE IterationLULanczos (n: integer; VAR a, c: vec);
   VAR
      u, l: vec;
   BEGIN
      LanczosLU(n, a, c, u, l);
      LanczosUL(n, u, l, a, c);
   END; { de "IterationLULanczos" }
```

Le deuxième cas est plutôt une variante. Si A est une matrice définie positive, on peut, au lieu de faire une décomposition LU, faire la décomposition de Choleski $A = L\,{}^tL$ (qui n'est pas LU puisque la matrice L n'a pas nécessairement des 1 sur la diagonale). On peut alors itérer l'application

$$A = L\,{}^tL \;\mapsto\; A' = {}^tL\,L$$

Là encore, la nouvelle matrice est tridiagonale si A l'est et elle est bien sûr définie positive. S'il y a convergence, L tend vers une matrice diagonale et on obtient donc dans $L\,{}^tL$ une matrice diagonale formée des valeurs propres de A. Plus précisément, si l'on pose

$$A = \begin{pmatrix} a_1 & b_1 & & & 0 \\ b_1 & a_2 & b_2 & & \\ & b_2 & \ddots & \ddots & \\ & & \ddots & & b_{n-1} \\ 0 & & & b_{n-1} & a_n \end{pmatrix} \qquad \text{et} \qquad L = \begin{pmatrix} d_1 & & & & 0 \\ \ell_2 & d_2 & & & \\ & \ell_3 & d_3 & & \\ & & \ddots & \ddots & \\ 0 & & & \ell_n & d_n \end{pmatrix}$$

on obtient, par identification,

$$\begin{aligned} d_1 &= \sqrt{a_1} \\ \ell_i &= b_{i-1}/d_{i-1} \\ d_i &= \sqrt{a_i - \ell_i^2} \qquad i = 2, \ldots, n \end{aligned}$$

Maintenant, soit $A' = {}^tLL$. Alors

$$A' = \begin{pmatrix} a'_1 & b'_1 & & & 0 \\ b'_1 & a'_2 & b'_2 & & \\ & b'_2 & \ddots & \ddots & \\ & & \ddots & & b'_{n-1} \\ 0 & & & b'_{n-1} & a'_n \end{pmatrix}$$

et on obtient

$$\begin{aligned} a'_i &= d_i^2 + \ell_{i+1}^2 \\ b'_i &= d_{i+1}\ell_{i+1} \qquad i = 1, \ldots, n-1 \\ a'_n &= d_n^2 \end{aligned}$$

Sur le même schéma que ci-dessus, on a donc les trois procédures suivantes :

```
PROCEDURE CholeskiTridiagonal (n: integer; a, b: vec; VAR d, l: vec);
```

Cette procédure a déjà été donnée au chapitre 4.

```
PROCEDURE TridiagonalCholeski (n: integer; d, l: vec; VAR a, b: vec);
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO n - 1 DO BEGIN
         a[i] := sqr(l[i + 1]) + sqr(d[i]);
         b[i] := d[i + 1] * l[i + 1];
      END;
      a[n] := sqr(d[n]);
   END; { de "TridiagonalCholeski" }
```

Ces deux procédures sont combinées en

```
PROCEDURE IterationCholeski (n: integer; VAR a, b: vec);
   VAR
      d, l: vec;
   BEGIN
      CholeskiTridiagonal(n, a, b, d, l);
      TridiagonalCholeski(n, d, l, a, b);
   END; { de "IterationCholeski" }
```

Voici quelques résultats numériques :

```
Donner l'ordre n :   5
Donner la diagonale
4 4 4 4 4
Donner la sous-diagonale
1 1 1 1
Voici la matrice lue
   4.000   1.000   0.000   0.000   0.000
   1.000   4.000   1.000   0.000   0.000
   0.000   1.000   4.000   1.000   0.000
   0.000   0.000   1.000   4.000   1.000
   0.000   0.000   0.000   1.000   4.000
```

Nous appliquons à cette matrice tridiagonale, qui est à la fois une matrice de Lanczos et une matrice définie positive, les deux méthodes correspondantes.

```
Lanczos
Itérations          Diagonale                               Norme sous-diagonale
    10       5.326   4.732   4.228   3.312   2.402               0.51508206
    20       5.608   5.035   4.064   3.016   2.276               0.08852942
    30       5.697   5.025   4.008   3.001   2.268               0.02442379
    40       5.723   5.008   4.001   3.000   2.268               0.00663863
    50       5.730   5.002   4.000   3.000   2.268               0.00172361
    60       5.731   5.001   4.000   3.000   2.268               0.00044166
    70       5.732   5.000   4.000   3.000   2.268               0.00011277
    80       5.732   5.000   4.000   3.000   2.268               0.00002877
Choleski
Itérations          Diagonale                               Norme sous-diagonale
    10       5.326   4.732   4.228   3.312   2.402               0.71769243
    20       5.608   5.035   4.064   3.016   2.276               0.29753917
    30       5.697   5.025   4.008   3.001   2.268               0.15628137
    40       5.723   5.008   4.001   3.000   2.268               0.08147795
    50       5.730   5.002   4.000   3.000   2.268               0.04151653
    60       5.731   5.001   4.000   3.000   2.268               0.02101571
    70       5.732   5.000   4.000   3.000   2.268               0.01061958
    80       5.732   5.000   4.000   3.000   2.268               0.00536382
```

On obtient les mêmes résultats en sensiblement le même nombre d'itérations. Ce nombre est considérable, mais chaque itération demande très peu de calculs. Voici une autre matrice symétrique :

```
Donner l'ordre n :   5
Donner la diagonale
4 5 3 1 4
Donner la sous-diagonale
1 2 1 1
Voici la matrice lue
   4.000   1.000   0.000   0.000   0.000
   1.000   5.000   2.000   0.000   0.000
   0.000   2.000   3.000   1.000   0.000
   0.000   0.000   1.000   1.000   1.000
   0.000   0.000   0.000   1.000   4.000
Tridiagonale
Itérations          Diagonale                             Norme sous-diagonale
    10       6.461   4.012   4.170   2.146   0.211           0.28426918
    20       6.572   3.992   4.197   2.028   0.211           0.02520046
    30       6.572   4.134   4.055   2.028   0.211           0.02966949
    40       6.572   4.253   3.935   2.028   0.211           0.01782500
    50       6.572   4.311   3.877   2.028   0.211           0.00696057
    60       6.572   4.332   3.857   2.028   0.211           0.00227160
    70       6.572   4.338   3.850   2.028   0.211           0.00069793
    80       6.572   4.340   3.848   2.028   0.211           0.00021045
    90       6.572   4.341   3.848   2.028   0.211           0.00006310
   100       6.572   4.341   3.847   2.028   0.211           0.00001889
   110       6.572   4.341   3.847   2.028   0.211           0.00000565
Choleski
Itérations          Diagonale                             Norme sous-diagonale
    10       6.461   4.012   4.170   2.146   0.211           0.53316951
    20       6.572   3.992   4.197   2.028   0.211           0.22450070
    40       6.572   4.253   3.935   2.028   0.211           0.18881191
    60       6.572   4.332   3.857   2.028   0.211           0.06740393
    80       6.572   4.340   3.848   2.028   0.211           0.02051641
   100       6.572   4.341   3.847   2.028   0.211           0.00614636
   120       6.572   4.341   3.847   2.028   0.211           0.00183871
   160       6.572   4.341   3.847   2.028   0.211           0.00016451
   200       6.572   4.341   3.847   2.028   0.211           0.00001472
```

Notes bibliographiques

L'ouvrage de référence sur les méthodes de calcul des valeurs propres est :

J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford, Clarendon Press, 1965.

De nombreux compléments se trouvent dans :

G. H. Golub, C. F. van Loan, *Matrix Computations*, Baltimore, John Hopkins University Press, 1985.

Voir également :

P. G. Ciarlet, *Introduction à l'analyse numérique matricielle et à l'optimisation*, Paris,Masson, 1988,

et le volume d'exercices correspondant, de :

P. G. Ciarlet, B. Miara et J.-M. Thomas, Paris, Masson, 1987.

Un exposé orienté vers la programmation, avec de nombreux programmes, est :

W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes*, Cambridge, Cambridge University Press, 1988.

Chapitre 6

Matrices en combinatoire

6.1 Matrices unimodulaires

6.1.1 Enoncé : matrices unimodulaires

Soient p et q des entiers positifs. Dans tout le problème, $A = (a_{ij})_{1\leq i\leq p, 1\leq j\leq q}$ désigne une matrice réelle d'ordre (p, q) à coefficients 0, 1 ou -1. Une *sous-matrice* de A est une matrice de la forme $A' = (a_{ij})_{i\in I, j\in J}$, où $I \subset \{1, \cdots, p\}$ et $J \subset \{1, \cdots, q\}$.

On dit que A est *unimodulaire* si et seulement si A est de rang p et si toutes les sous-matrices carrées d'ordre (p, p) ont un déterminant égal à 0, 1 ou -1. On dit que A est *totalement unimodulaire* si et seulement si le déterminant de toutes les sous-matrices carrées de A vaut 0, 1 ou -1.

1.– Ecrire une procédure qui prend en argument une matrice carrée à coefficients 0, 1 ou -1, et qui calcule la valeur absolue de son déterminant (on pourra supposer $p, q \leq 5$).

Exemple numérique :

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 & 1 \\ 1 & 1 & -1 & 1 & 1 \\ 1 & 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 \end{pmatrix}$$

2.– Ecrire une procédure qui teste si une matrice A à coefficients 0, 1 ou -1 est unimodulaire (on pourra supposer $p \leq 5$ et $q \leq 10$).

Exemples numériques : tester si les matrices suivantes sont unimodulaires.

$$A_1 = \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & -1 & 0 \\ 0 & -1 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix} \quad \text{et} \quad A_2 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

3.– Démontrer que si chaque ligne de A contient exactement un $+1$ et un -1 (les autres coefficients étant nuls), alors A est totalement unimodulaire. Ceci fournit des exemples de matrices totalement unimodulaires.

4.– Démontrer que A est totalement unimodulaire si et seulement si la matrice

$$(A, I_p) = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1q} & 1 & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & a_{2q} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{p1} & a_{p2} & \dots & a_{pq} & 0 & 0 & \dots & 1 \end{pmatrix}$$

est unimodulaire.

5.– Ecrire une procédure qui teste si A est totalement unimodulaire (on pourra supposer $p, q \leq 5$). Exemples numériques : tester si les matrices A_1 et A_2 ci-dessus sont totalement unimodulaires.

Pour $1 \leq j \leq q$, on note C_j la j-ième colonne de A. Une partie K de $\{1, \dots, q\}$ est dite *équilibrée* dans A s'il existe des sous-ensembles disjoints K_1 et K_2 de K, d'union K et tels que les coefficients du vecteur colonne

$$\sum_{k \in K_1} C_k - \sum_{k \in K_2} C_k$$

soient égaux à 0, 1 ou -1. On dit qu'une matrice A est *équilibrée* si toute partie K de $\{1, \dots, q\}$ est équilibrée dans A.

6.– On se propose de démontrer que toute matrice équilibrée est totalement unimodulaire. Soit A une matrice équilibrée.

a) Démontrer que les coefficients de A sont égaux à 0, 1 ou -1.

b) Soit k un entier tel que $1 \leq k \leq \min(p, q) - 1$. On suppose que les sous-matrices carrées d'ordre k de A ont un déterminant égal à 0, 1 ou -1. Soit $B = (b_{ij})_{i \in I, j \in J}$ une sous-matrice carrée d'ordre $k+1$, de déterminant d non nul. Démontrer que la matrice $B^* = dB^{-1}$ est à coefficients 0, 1 ou -1.

c) Soit $b = (b_{1,j})_{j \in J}$ la première colonne de B^*. Démontrer qu'il existe un vecteur $x = (x_j)_{j \in J}$ à coefficients 0, 1 ou -1 tel que $x_j \equiv b_j \bmod 2$ pour tout j et tel que $Bx = {}^t(1, \underbrace{0, \dots, 0}_{k \text{ fois}})$. En déduire que $|d| = 1$.

d) Démontrer que A est totalement unimodulaire.

7.– Soient $I_1, \dots, I_p$ des intervalles de $\mathbb{R}$ et soient $x_1, \dots, x_q$ des nombres réels distincts. Démontrer que la matrice A définie par

$$a_{ij} = \begin{cases} 1 & \text{si } x_j \in I_i \\ 0 & \text{si } x_j \notin I_i \end{cases}$$

est équilibrée.

8.– On dit qu'une matrice A à coefficients égaux à 0, 1 ou -1 est *élémentaire* si, pour $1 \leq i \leq p$, la i-ième ligne de A a exactement deux coefficients non nuls, $a_{i,f(i)}$ et $a_{i,g(i)}$.
a) Démontrer qu'une matrice élémentaire A est équilibrée si et seulement si l'ensemble $\{1, \ldots, q\}$ peut être partagé en deux classes J_1 et J_2 telles que

$$\sum_{j \in J_1} C_j = \sum_{j \in J_2} C_j$$

b) En déduire qu'une matrice élémentaire A est équilibrée si et seulement si l'ensemble $\{1, \ldots, q\}$ peut être partagé en deux classes de façon que, pour tout i, les colonnes d'indice $f(i)$ et $g(i)$ soient dans la même classe si et seulement si $a_{i,f(i)}$ et $a_{i,g(i)}$ sont de signes distincts.

9.– Décrire un algorithme qui utilise le critère donné en (8b) pour tester si une matrice élémentaire est équilibrée. Ecrire la procédure correspondante. (La procédure devra pouvoir tester des matrices d'ordre (p, q) lorsque $p, q < 10$).

Remarque (hors énoncé). *On peut en fait démontrer la réciproque de la question 6 : toute matrice totalement unimodulaire est équilibrée.*

6.1.2 Solution : matrices unimodulaires

Soient p et q des entiers positifs. Dans toute la suite, $A = (a_{ij})_{1 \leq i \leq p, 1 \leq j \leq q}$ désigne une matrice réelle d'ordre (p, q) à coefficients 0, 1 ou -1. Une *sous-matrice* de A est une matrice de la forme $A' = (a_{ij})_{i \in I, j \in J}$ où $I \subset \{1, \cdots, p\}$ et $J \subset \{1, \cdots, q\}$.

On dit que A est *unimodulaire* si et seulement si A est de rang p et si toutes les sous-matrices carrées d'ordre (p, p) ont un déterminant égal à 0, 1 ou -1. On dit que A est *totalement unimodulaire* si et seulement si le déterminant de toutes les sous-matrices carrées de A vaut 0, 1 ou -1.

Les matrices unimodulaires et totalement unimodulaires, outre leur intérêt combinatoire, sont très utilisées en programmation linéaire *en nombres entiers* (recherche des solutions en nombres entiers de systèmes d'inéquations linéaires à coefficients entiers). Ces matrices admettent de très nombreuses caractérisations dont nous donnons deux exemples ci-dessous. D'un point de vue algorithmique, tester si une matrice est totalement unimodulaire semble requérir a priori un temps exponentiel en fonction de la taille de la matrice (l'algorithme suggéré dans le problème, qui constitue une amélioration par rapport à l'algorithme le plus naïf, est néanmoins exponentiel). Il existe cependant un algorithme polynomial pour ce problème, découvert par Seymour. Il s'agit là d'un résultat très difficile, fondé sur un théorème de décomposition qui permet de décrire toutes les matrices totalement unimodulaires à l'aide d'opérations élémentaires en partant de matrices particulières.

Pour nous familiariser avec les matrices totalement unimodulaires, le mieux est d'en donner tout de suite quelques exemples.

PROPOSITION 6.1.1. *Si chaque ligne de A contient exactement un $+1$ et un -1 (les autres coefficients étant nuls), alors A est totalement unimodulaire.*

Preuve. Démontrons par récurrence sur n que toute sous-matrice carrée d'ordre n de A a un déterminant égal à 0, 1 ou -1. C'est clair si $n = 1$. Prenons une sous-matrice carrée d'ordre n. Si l'une des lignes est nulle, le déterminant vaut 0. Si l'une des lignes contient un seul coefficient non nul, on peut développer le déterminant par rapport à cette ligne et on conclut par récurrence. Reste le cas où toutes les lignes contiennent au moins (et donc exactement) deux coefficients non nuls, qui sont nécessairement 1 et -1. Dans ce cas la somme des colonnes est nulle et le déterminant est nul. ∎

Le calcul (cf. la partie programme) démontre que les deux matrices suivantes sont totalement unimodulaires :

$$B_1 = \begin{pmatrix} 1 & -1 & 0 & 0 & -1 \\ -1 & 1 & -1 & 0 & 0 \\ 0 & -1 & 1 & -1 & 0 \\ 0 & 0 & -1 & 1 & -1 \\ -1 & 0 & 0 & -1 & 1 \end{pmatrix} \quad \text{et} \quad B_2 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Ces deux matrices jouent un rôle très important dans le théorème de décomposition de Seymour évoqué plus haut. Le lien entre matrices unimodulaires et totalement unimodulaires est précisé ci-dessous :

PROPOSITION 6.1.2. *Une matrice A est totalement unimodulaire si et seulement si la matrice*

$$(A, I_p) = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1q} & 1 & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & a_{2q} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{p1} & a_{p2} & \dots & a_{pq} & 0 & 0 & \dots & 1 \end{pmatrix}$$

est unimodulaire.

Preuve. Supposons A totalement unimodulaire et soit $B = (A, I_p)$. Il est clair que B est de rang p. D'autre part, si C est une sous-matrice carrée de taille (p, p) de B, son déterminant est celui d'une sous-matrice de A : pour le voir, il suffit de développer le déterminant suivant les colonnes de C d'indice supérieur ou égal à p. Comme A est totalement unimodulaire, ce déterminant est égal à 0, 1 ou -1.

Réciproquement, supposons B unimodulaire et soit

$$A' = \begin{pmatrix} a_{i_1,j_1} & \dots & a_{i_1,j_r} \\ \vdots & \ddots & \vdots \\ a_{i_r,j_1} & \dots & a_{i_r,j_r} \end{pmatrix}$$

une sous-matrice carrée de A d'ordre (r, r). Considérons la matrice carrée d'ordre (p, p)

$$B' = \begin{pmatrix} a_{1,j_1} & \dots & a_{1,j_r} & 1 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & & & \\ a_{p,j_1} & \dots & a_{p,j_r} & 0 & 0 & \dots & 1 \end{pmatrix}$$

où, pour $i = 1, \dots, p - r$, la $r + i$-ième colonne de B' contient un 1 en position $r + i, j$ où j est le i-ième élément de l'ensemble (ordonné) $\{1, \dots, p\} \setminus \{i_1, \dots, i_r\}$, et des zéros ailleurs. Par exemple, si $\{i_1, i_2, i_3\} = \{2, 4, 7\}$ et si $p = 7$, on a

$$B' = \begin{pmatrix} a_{1,j_1} & a_{1,j_2} & a_{1,j_3} & 1 & 0 & 0 & 0 \\ a_{2,j_1} & a_{2,j_2} & a_{2,j_3} & 0 & 0 & 0 & 0 \\ a_{3,j_1} & a_{3,j_2} & a_{3,j_3} & 0 & 1 & 0 & 0 \\ a_{4,j_1} & a_{4,j_2} & a_{4,j_3} & 0 & 0 & 0 & 0 \\ a_{5,j_1} & a_{5,j_2} & a_{5,j_3} & 0 & 0 & 1 & 0 \\ a_{6,j_1} & a_{6,j_2} & a_{6,j_3} & 0 & 0 & 0 & 1 \\ a_{7,j_1} & a_{7,j_2} & a_{7,j_3} & 0 & 0 & 0 & 0 \end{pmatrix}$$

La matrice B' a même déterminant que A' et comme B est unimodulaire, ce déterminant vaut 0, 1 ou -1. ∎

Nous en venons aux caractérisations les plus intéressantes des matrices totalement unimodulaires, dont nous ne donnerons qu'une démonstration très partielle. Commençons par une définition commode. Pour $1 \le j \le q$, on note C_j la j-ième colonne de A. Une partie K de $\{1, \dots, q\}$ est dite *équilibrée* dans A s'il existe des sous-ensembles disjoints K_1 et K_2 de K, d'union K, et tels que les coefficients du vecteur colonne

$$\sum_{k \in K_1} C_k - \sum_{k \in K_2} C_k$$

soient égaux à 0, 1 ou -1. On dit qu'une matrice A est *équilibrée*si toute partie K de $\{1, \dots, q\}$ est équilibrée dans A.

Proposition 6.1.3. *Soit A une matrice. Les conditions suivantes sont équivalentes :*

(1) *A est totalement unimodulaire,*
(2) *A est équilibrée,*
(3) *A est à coefficients* 0, 1 *ou* -1 *et ne contient aucune sous-matrice carrée de déterminant* 2 *ou* -2.

Nous démontrerons seulement que toute matrice équilibrée est totalement unimodulaire.

Preuve. Soit A une matrice équilibrée. Montrons par récurrence sur k que les sous-matrices carrées d'ordre k de A ont un déterminant égal à 0, 1 ou -1.

Si on prend $J = \{j\}$, on a nécessairement $\{J_1, J_2\} = \{J, \emptyset\}$ et donc les coefficients de la colonne C_j sont égaux à 0, 1 ou -1. Supposons le résultat acquis pour les sous-matrices de taille inférieure ou égale à k, avec $1 \le k \le \min(p, q) - 1$, et soit $B = (b_{ij})_{i \in I, j \in J}$ une sous-matrice carrée d'ordre $k + 1$, de déterminant d non nul. Posons $B^* = dB^{-1}$. Cette matrice est la «transposée de la matrice des cofacteurs» de B. Comme les déterminants de tous les mineurs d'ordre $\le k$ sont égaux à 0, 1 ou -1, les coefficients de B^* sont égaux à 0, 1 ou -1. Soit $b = (b_j)_{j \in J}$ la première colonne de B^*. On a donc $Bb = de_1$ où $e_1 = {}^t(1, \underbrace{0, \dots, 0}_{k \text{ fois}})$. On pose

$$K = \{k \in J \mid b_k \neq 0\}$$

Puisque A est équilibrée, il existe des sous-ensembles disjoints K_1 et K_2 de K, d'union K, et tels que les coefficients du vecteur colonne

$$\sum_{k\in K_1} C_k - \sum_{k\in K_2} C_k$$

soient égaux à 0, 1 ou -1. Soit $x = (x_k)_{k\in J}$ le vecteur défini par

$$x_k = \begin{cases} 1 & \text{si } k \in K_1, \\ -1 & \text{si } k \in K_2, \\ 0 & \text{si } k \notin K \end{cases}$$

Par construction, on a, quel que soit $k \in J$, $x_k \equiv b_k \bmod 2$ et donc aussi $(Bx)_k \equiv (Bb)_k \bmod 2$. D'autre part

$$Bx = \sum_{k\in K} C_k x_k = \sum_{k\in K_1} C_k - \sum_{k\in K_2} C_k$$

est à coefficients 0, 1 ou -1. Il en résulte que les k derniers coefficients de Bx sont nécessairement nuls. De plus, comme B est inversible, b n'est pas nul, x n'est pas nul et donc Bx n'est pas nul. Par conséquent, Bx est égal soit à e_1, soit à $-e_1$ et on a $B^*Bx = dx = \pm B^*e_1$. Comme les vecteurs x et B^*e_1 sont à coefficients 0, 1 ou -1, on a nécessairement $|d| = 1$, ce qui conclut la démonstration par récurrence. ∎

La proposition 6.1.3 permet de donner d'autres exemples de matrices totalement unimodulaires.

Proposition 6.1.4. *Soient $I_1, \ldots, I_p$ des intervalles de $\mathbb{R}$ et soient $x_1, \ldots, x_q$ des nombres réels distincts. Alors la matrice A définie par*

$$a_{ij} = \begin{cases} 1 & \text{si } x_j \in I_i \\ 0 & \text{si } x_j \notin I_i \end{cases}$$

est équilibrée.

Preuve. Soit K une partie de $\{1, \ldots, q\}$. On numérote les réels x_k pour $k \in K$ d'après leur position sur la droite et on pose

$$K_1 = \{k \mid x_k \text{ est de rang pair}\} \quad \text{et} \quad K_2 = \{k \mid x_k \text{ est de rang impair}\}.$$

Quel que soit $k \in \{1, \ldots, q\}$, le vecteur

$$b = \sum_{k\in K_1} C_k - \sum_{k\in K_2} C_k$$

est à coefficients 0, 1 ou -1. En effet b_k représente le nombre de points de rang pair contenus dans le segment I_k, moins le nombre de points de rang impair contenus dans I_k. Ce nombre est donc toujours égal à 0, 1 ou -1. ∎

On se propose maintenant de décrire toutes les matrices totalement unimodulaires contenant exactement deux coefficients non nuls dans chaque ligne (la proposition 6.1.1 fournit des exemples de telles matrices). On dira qu'une matrice A à coefficients 0, 1 ou -1 est *élémentaire* si, pour $1 \leq i \leq p$, la i-ième ligne de A a exactement deux coefficients non nuls, que l'on notera $a_{i,f(i)}$ et $a_{i,g(i)}$.

PROPOSITION 6.1.5. *Une matrice élémentaire A est équilibrée si et seulement si l'ensemble $\{1, \ldots, q\}$ peut être partagé en deux classes J_1 et J_2 telles que*

$$\sum_{j \in J_1} C_j = \sum_{j \in J_2} C_j \tag{1.1}$$

Preuve. Démontrons d'abord qu'une matrice élémentaire équilibrée A vérifie (1.1). Puisque A est équilibrée, l'ensemble $\{1, \ldots, q\}$ peut être partagé en deux classes J_1 et J_2 telles que

$$\sum_{j \in J_1} C_j - \sum_{j \in J_2} C_j$$

soit à coefficients 0, 1 ou -1. Mais puisque A est élémentaire, la somme des coefficients de chaque ligne est congrue à 0 modulo 2. Donc

$$\sum_{j \in J_1} C_j = \sum_{j \in J_2} C_j$$

Réciproquement, si une matrice élémentaire vérifie (1.1), l'ensemble $\{1, \ldots, q\}$ peut être partagé en deux classes J_1 et J_2 telles que (1.1) soit vérifié. Soit J une partie de $\{1, \ldots, q\}$ et posons $J_1' = J \cap J_1$ et $J_2' = J \cap J_2$. Alors le vecteur

$$V = \sum_{j \in J_1'} C_j - \sum_{j \in J_2'} C_j$$

est à coefficients 0, 1 ou -1. En effet, fixons un indice i. Si $\mathrm{Card}(J \cap \{f(i), g(i)\}) \leq 2$, alors $|V_i| \leq 1$. Supposons maintenant que J contient $f(i)$ et $g(i)$. Si $f(i)$ et $g(i)$ sont de signe contraire, alors ils sont nécessairement dans la même classe (J_1' ou J_2'), car sinon, $|V_i| = 2$. Donc $C_i = 0$ dans ce cas. Si $f(i)$ et $g(i)$ sont de même signe, ils sont, par le même argument, dans deux classes différentes et là encore, $V_i = 0$. ■

COROLLAIRE 6.1.6. *Une matrice élémentaire A est équilibrée si et seulement si l'ensemble $\{1, \ldots, q\}$ peut être partagé en deux classes de façon que, pour tout i, les colonnes d'indice $f(i)$ et $g(i)$ soient dans la même classe si et seulement si $a_{i,f(i)}$ et $a_{i,g(i)}$ sont de signes distincts.*

Preuve. On vient de voir que $f(i)$ et $g(i)$ sont de signe contraire si et seulement si ils sont dans la même classe. ■

On peut démontrer directement qu'une matrice vérifiant (1.1) est totalement unimodulaire. Soient J_1 et J_2 les deux classes de $\{1, \ldots, q\}$. Multiplions toutes les colonnes d'indice dans J_1 par -1. La matrice B obtenue est totalement unimodulaire si et seulement si A est totalement unimodulaire. Démontrons que B contient exactement un $+1$ et un -1 par ligne. En effet, soit i un indice de colonne. Si $a_{i,f(i)}$ et $a_{i,g(i)}$ ont même signe, ils sont dans deux classes distinctes et donc $b_{i,f(i)}$ et $b_{i,g(i)}$ auront des signes différents. Si maintenant $a_{i,f(i)}$ et $a_{i,g(i)}$ ont des signes différents, $f(i)$ et $g(i)$ sont dans la même classe et donc ou bien $a_{i,f(i)}$ et $a_{i,g(i)}$ ne changent pas de signe, ou bien ils changent de signe simultanément. Dans les deux cas, $b_{i,f(i)}$ et $b_{i,g(i)}$ auront des signes différents. D'après la proposition 6.1.1, B, et donc A, est totalement unimodulaire.

Le corollaire 6.1.6 permet de donner un algorithme efficace pour tester si une matrice élémentaire est équilibrée. Cet algorithme sera décrit dans la section suivante.

6.1.3 Programme : Matrices unimodulaires

Le programme fait appel aux bibliothèques `General` et `Matrices`. En particulier, on utilisera le type `vecE = ARRAY[1..OrdreMax] OF integer` déjà déclaré dans `General`. On définit de façon analogue un type spécial pour les matrices à coefficients entiers :

```
TYPE
   matE = ARRAY[1..OrdreMax] OF vecE;
```

On adapte les procédures d'entrée-sortie de la bibliothèque `Matrices` à ce nouveau type :

```
PROCEDURE EcrireMatE (m, n: integer; VAR a: matE; titre: texte);
```
Affichage du titre, puis de la matrice a d'ordre (m, n).
```
   VAR
      i, j: integer;
   BEGIN
      writeln;
      writeln(titre);
      FOR i := 1 TO m DO BEGIN
         FOR j := 1 TO n DO
            write(A[i, j] : 3);
         writeln
      END
   END; { de "EcrireMatE" }
```

Comme nous n'aurons à utiliser que des matrices à coefficients 0, 1 ou -1, la procédure d'entrée se simplifie ainsi :

```
PROCEDURE EntrerMatE (VAR m, n: integer; VAR a: matE; titre: texte);
```
Affichage du titre, puis lecture de la matrice A d'ordre (m, n) à coefficients 0, 1 ou -1. Les entrées sont codées par 0, $+$ et $-$
```
   VAR
      ch: char;
      i, j: integer;
   BEGIN
      writeln;
```

```
      writeln(titre);
      write('Nombre de lignes : '); readln(m);
      write('Nombre de colonnes : '); readln(n);
      FOR i := 1 TO m DO BEGIN
         writeln('ligne', i : 2, ' :   ');
         FOR j := 1 TO n DO BEGIN
            readln(ch);
            CASE ch OF
               '+':
                  a[i, j] := 1;
               '-':
                  a[i, j] := -1;
               OTHERWISE
                  a[i, j] := 0;
            END;
         END;
         writeln;
      END
   END; { de "EntrerMatE" }
```

Pour le calcul du déterminant, on se contente de convertir la matrice de type `matE` au type `mat` et on utilise la procédure `determinant` de la bibliothèque. On pourrait éventuellement utiliser une autre procédure pour ne pas avoir à passer en réels (en adaptant par exemple la procédure récursive de calcul du déteminant, qui est efficace pour des matrices carrées d'ordre inférieur ou égal à 5).

```
FUNCTION det (VAR A: matE; p: integer): integer;
   Calcule le déterminant de A.
   VAR
      i, j: integer;
      R: mat;
   BEGIN
      FOR i := 1 TO p DO
         FOR j := 1 TO p DO
            R[i, j] := A[i, j];
      det := round(determinant(p, R));
   END; { de "det" }
```

Pour tester si une matrice d'ordre (p, q) est unimodulaire, on calcule le déterminant de toutes ses sous-matrices carrées d'ordre p. Pour cela, on engendre toutes les parties à p éléments de l'ensemble $\{1, \ldots, q\}$. Ces procédures sont détaillées au chapitre 8.

```
CONST
   LongueurSuite = 12;
TYPE
   suite = ARRAY[0..LongueurSuite] OF integer;
PROCEDURE PremierePartieRestreinte (VAR x: suite; n, k: integer);
PROCEDURE PartieSuivanteRestreinte (VAR x: suite; n, k: integer;
```

```
    VAR derniere:boolean);
```

Un sous-ensemble de p colonnes étant retenu, on extrait la sous-matrice de A correspondante à l'aide de la procédure suivante :

```
PROCEDURE ExtraitSousMatrice (SousEnsemble: suite; p, q: integer; A: matE;
   VAR C: matE);
```

Extrait de A la matrice carrée C d'ordre p obtenue en sélectionnant les colonnes d'indice i tel que SousEnsemble$[i] = 1$.

```
   VAR
      i, j, k: integer;
   BEGIN
      k := 0;
      FOR j := 1 TO q DO
         IF SousEnsemble[j] = 1 THEN BEGIN
            k := k + 1;
            FOR i := 1 TO p DO
               C[i, k] := A[i, j]
         END
   END; { de "ExtraitSousMatrice" }
```

Pour éviter de faire des calculs de déterminant inutiles, on utilise une procédure qui teste si une matrice ne contient pas de colonne nulle.

```
FUNCTION UneColonneEstNulle (A: matE; p, q: integer): boolean;
```

Teste si A contient une colonne nulle.

```
   VAR
      i, j: integer;
      DebutColonneNul, PasDeColonneNulle: boolean;
   BEGIN
      j := 0;
      PasDeColonneNulle := true;
      WHILE (j < q) AND PasDeColonneNulle DO BEGIN
         j := j + 1;
         i := 0;
         DebutColonneNul := true;
         WHILE (i < p) AND DebutColonneNul DO BEGIN
            i := i + 1;
            DebutColonneNul := A[i, j] = 0
         END;
         PasDeColonneNulle := NOT DebutColonneNul;
      END;
      UneColonneEstNulle := NOT PasDeColonneNulle
   END; { de "UneColonneEstNulle" }
```

Tout est en place et on teste maintenant facilement si une matrice est unimodulaire à l'aide de la procédure suivante :

```
FUNCTION EstUnimodulaire (A: matE; p, q: integer): boolean;
```

```
    Teste si A est unimodulaire.
    VAR
       d, i: integer;
       LeRangEstp, CetaitLaDerniere: boolean;
       C: matE;
       SousEnsemble: suite;
    BEGIN
       IF UneColonneEstNulle(A, p, q) OR (q < p) THEN
          Si q < p, la matrice ne peut être unimodulaire.
          EstUnimodulaire := false
       ELSE BEGIN
          PremierePartieRestreinte(SousEnsemble, q, p);
          LeRangEstp := false;
          REPEAT
             ExtraitSousMatrice(SousEnsemble, p, q, A, C);
             d := Det(C, p);
             Si on trouve une sous-matrice de rang p, le rang vaut p.
             LeRangEstp := d <> 0;
             PartieSuivanteRestreinte(SousEnsemble, q, p, CetaitLaDerniere)
          UNTIL (abs(d) > 1) OR CetaitLaDerniere;
          EstUnimodulaire := LeRangEstp AND (abs(d) <= 1)
       END
    END; { de "EstUnimodulaire" }
```

Le test pour les matrices totalement unimodulaires est encore plus simple à programmer et repose sur la proposition 6.1.2.

```
FUNCTION EstTotalementUnimodulaire (A: matE; p, q: integer): boolean;
    Teste si A est totalement unimodulaire. On se contente de tester si la matrice [A, I] est
    unimodulaire
    VAR
       i, j: integer;
    BEGIN
       FOR i := 1 TO p DO
          FOR j := q + 1 TO q + p DO
             A[i, j] := 0;
       FOR i := 1 TO p DO
          A[i, q + i] := 1;
       EstTotalementUnimodulaire := EstUnimodulaire(A, p, p + q);
    END; { de "EstTotalementUnimodulaire" }
```

Dans le cas des matrices élémentaires, le corollaire 6.1.6 permet de donner un algorithme efficace pour tester si une matrice est totalement unimodulaire. On commence par coder la ième ligne d'une matrice élémentaire par le triplet $(f(i), g(i), \varepsilon)$, où ε vaut -1 si $a_{i,f(i)}$ et $a_{i,g(i)}$ ont même signe et vaut 1 sinon. On construit ensuite le graphe étiqueté sur l'ensemble $\{1, \ldots, q\}$ dont les arêtes sont les triplets $(f(i), \varepsilon, g(i))$. Chaque ligne de la matrice définit une arête, chaque colonne un sommet. Par exemple, si on part de la matrice

$$A = \begin{pmatrix} 0 & 1 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

on obtient le graphe de la figure 6.1.1.

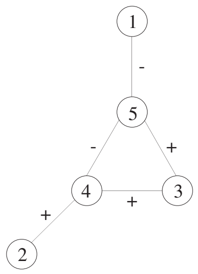


Figure 6.1.1: *Le graphe associé à la matrice A.*

La procédure qui suit permet de réaliser ce codage :

```
PROCEDURE Conversion (A: matE; p, q: integer; VAR M: MatElementaire);
   VAR
      i, j: integer;
   BEGIN
      FOR i := 1 TO p DO BEGIN
         j := 1;
         WHILE A[i, j] = 0 DO
            j := j + 1;
         M[i, 1] := j;                  j = f(i).
         j := j + 1;
         WHILE A[i, j] = 0 DO
            j := j + 1;
         M[i, 2] := j;                  j = g(i).
         M[i, 3] := -A[i, M[i, 1]] * A[i, M[i, 2]];
      END;
   END; { de "Conversion" }
```

Il s'agit de savoir si l'on peut attacher à chaque sommet du graphe une étiquette $+1$ ou -1 compatible avec l'étiquetage des arêtes. De façon plus précise, il faut que pour toute arête (i, ε, j) du graphe, on ait la relation

$$\text{étiquette}(i) = \varepsilon \cdot \text{étiquette}(j)$$

A cet effet, on part de l'étiquetage qui attribue l'étiquette $+1$ à chaque sommet et on modifie cet étiquetage en examinant l'une après l'autre les arêtes du graphe (c'est-à-dire les lignes de la matrice). On tient simultanément à jour une fonction `Pere` de l'ensemble des sommets dans lui-même, dont le rôle intuitif est de mémoriser la suite des déductions.

```
TYPE
   Chaque ligne d'une matrice élémentaire est déterminée, au signe près, par le triplet
   (f(i), g(i), ε).
   LigneElementaire = ARRAY[1..3] OF integer;
Une matrice élémentaire est représentée par un vecteur de LigneElementaire.
   MatElementaire = ARRAY[1..OrdreMax] OF LigneElementaire;

FUNCTION ElementaireEstTU (VAR M: MatElementaire; p, q: integer): boolean;
   Teste si une matrice élémentaire est totalement unimodulaire.
   VAR
      Pere, Signe: vecE;
      i1, i2, j, s: integer;
      b: boolean;
   Il s'agit de tester si on peut décomposer les colonnes en deux classes. On utilise un
   algorithme « Union-Find » modifié.
   BEGIN
      b := true;
      FOR j := 1 TO q DO       Initialisation des fonctions Père et Signe.
         BEGIN
         Pere[j] := 0;
         Signe[j] := 1;
      END;
      j := 0;
      WHILE (j < p) AND b DO BEGIN
         j := j + 1;
         i1 := M[j, 1];
         i2 := M[j, 2];
         s := M[j, 3];                                   Signe.
         WHILE pere[i1] > 0 DO BEGIN
            s := s * Signe[i1];
            i1 := pere[i1];
         END;
         WHILE pere[i2] > 0 DO BEGIN
            s := s * Signe[i2];
            i2 := pere[i2];
         END;
         IF i1 <> i2 THEN BEGIN
            Signe[i2] := s;
            pere[i2] := i1
         END
         ELSE
            b := s = 1;
      END;
      Elementaire := b;
   END; { de "ElementaireEstTU" }
```

Sur l'exemple précédent, le déroulement pas à pas du programme donne le résultat suivant (on note `Ancetre` la fonction obtenue par itération de la fonction `Pere`) :

```
j = 1     i1 = 2     i2 = 4
Ancetre(i1) = 2     Ancetre(i2) = 4     s = 1
j = 2     i1 = 1     i2 = 5
Ancetre(i1) = 1     Ancetre(i2) = 5     s = -1
j = 3     i1 = 4     i2 = 5
Ancetre(i1) = 2     Ancetre(i2) = 1     s = 1
j = 4     i1 = 3     i2 = 4
Ancetre(i1) = 3     Ancetre(i2) = 2     s = 1
j = 5     i1 = 3     i2 = 5
Ancetre(i1) = 3     Ancetre(i2) = 3     s = -1
La matrice n'est pas totalement unimodulaire
```

Voici quelques résultats numériques. Le déterminant de la matrice A de l'énoncé est égal à 16. Les deux matrices A_1 et A_2 de l'énoncé sont totalement unimodulaires.

6.2 Matrices irréductibles

6.2.1 Enoncé : matrices irréductibles

Soit $n \geq 1$ un entier, et $S = \{1, \ldots, n\}$. On note $\mathcal{M}$ l'ensemble des matrices carrées, d'ordre n, à coefficients dans $\{0, 1\}$. Soit $A = (a_{i,j}) \in \mathcal{M}$ et soient $i, j \in S$. Un *chemin* de i vers j de *longueur* $m \geq 0$ (dans A) est une suite $(i_0, \ldots, i_m)$ d'éléments de S telle que $i_0 = i$, $i_m = j$ et $a_{i_{k-1}, i_k} = 1$ pour $k = 1, \ldots, m$. Un *cycle* autour de i est un chemin de i vers i de longueur non nulle. Un élément $i \in S$ est *cyclique* s'il existe un cycle autour de i.

Sur l'ensemble $\mathcal{M}$, on définit une addition notée $\oplus$ et une multiplication notée $\otimes$, comme suit : soient $A, B, C, D \in \mathcal{M}$. Alors $C = (c_{i,j}) = A \oplus B$ est définie par

$$c_{i,j} = \begin{cases} 0 & \text{si } a_{i,j} = b_{i,j} = 0 \\ 1 & \text{sinon} \end{cases}$$

et $D = (d_{i,j}) = A \otimes B$ est définie par

$$d_{i,j} = \begin{cases} 0 & \text{si } a_{i,k} b_{k,j} = 0 \text{ pour tout } k \in S \\ 1 & \text{sinon} \end{cases}$$

On définit la puissance $A^{\otimes k}$ par $A^{\otimes 0} = I$ (I est la matrice unité) et $A^{\otimes k+1} = A \otimes A^{\otimes k}$ pour $k \geq 0$. On note $a_{i,j}^{(k)}$ l'élément d'indice (i, j) de $A^{\otimes k}$.

1.– Ecrire deux procédures qui, étant données deux matrices A et B dans $\mathcal{M}$, calculent respectivement $A \oplus B$ et $A \otimes B$. Ecrire une procédure qui prend en argument une matrice A et qui calcule $(I \oplus A)^{\otimes n-1}$.

Exemple numérique : $n = 5$ et $A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$

2.– Montrer qu'il existe un chemin de i vers j si et seulement si l'élément d'indice (i, j) de la matrice $(I \oplus A)^{\otimes n-1}$ est égal à 1.

On dit que j est *accessible* de i s'il existe un chemin de i vers j.

3.– Ecrire une procédure qui prend en argument un entier $i \in S$ et qui calcule, pour chaque élément j qui est accessible de i, un chemin de i vers j.

Même exemple numérique, avec $i = 1$.

4.– Ecrire une procédure qui calcule les éléments cycliques de S et qui affiche un cycle pour chacun de ces éléments. Exemple numérique de la question 1.

Une matrice $A \in \mathcal{M}$ est dite *acyclique* si l'ensemble de ses éléments cycliques est vide.

5.– Montrer que A est acyclique si et seulement si $A^{\otimes n} = 0$.

6.– On suppose dans cette question que A est acyclique. On appelle *rang* de $i \in S$ l'entier

$$r(i) = \max\{m \mid \text{il existe un chemin vers } i \text{ de longueur } m\}$$

Ecrire une procédure qui calcule le rang des éléments de S.

Exemple numérique : $n = 6$ et $A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$

7.– On appelle *période* d'un élément cyclique $i \in S$ de A et on note $d(i)$, le pgcd des entiers $k > 0$ tels que $a_{i,i}^{(k)} = 1$. Montrer que si i est accessible de j et j est accessible de i, alors $d(i) = d(j)$.

On note U la matrice de $\mathcal{M}$ dont tous les coefficients sont égaux à 1. Une matrice $A \in \mathcal{M}$ est *irréductible* si $(I \oplus A)^{\otimes n-1} = U$, elle est *primitive* s'il existe un entier $k \geq 0$ tel que $A^{\otimes k} = U$. Si A est primitive, le plus petit entier k tel que $A^{\otimes k} = U$ est noté $\gamma(A)$.

8.– a) Montrer qu'une matrice primitive est irréductible.
b) Une matrice est *apériodique* si elle possède un élément cyclique de période 1. Montrer qu'une matrice irréductible et apériodique est primitive.

9.– Montrer que si A est primitive et symétrique, alors $\gamma(A) \leq 2(n-1)$.

10.– Montrer qu'une matrice de permutation n'est pas primitive si $n > 1$. Caractériser les matrices de permutation qui sont irréductibles.

11.– On suppose $n > 1$. Soit A une matrice primitive. Pour $i \in S$, on pose

$$h_i = \min\{k \in \mathbb{N} \mid a_{i,j}^{(k)} = 1 \text{ pour tout } j \in S\}$$

a) Montrer que $\gamma(A) = \max\{h_1, \dots, h_n\}$.
b) Soit s la longueur du plus court cycle de A. Montrer que $\gamma(A) \leq n + s(n-2)$.
c) Montrer que $\gamma(A) \leq n^2 - 2n + 2$.
d) Donner, pour chaque valeur de n, un exemple d'une matrice A telle que $\gamma(A) = n^2 - 2n + 2$.

6.2.2 Solution : matrices irréductibles

Soit $n \geq 1$ un entier, et $S = \{1, \dots, n\}$. On note $\mathcal{M}$ l'ensemble des matrices carrées, d'ordre n, à coefficients dans $\{0, 1\}$. Soit $A = (a_{i,j}) \in \mathcal{M}$ et soient $i, j \in S$. Un *chemin* de i vers j de *longueur* $m \geq 0$ (dans A) est une suite $(i_0, \dots, i_m)$ d'éléments de S telle que $i_0 = i$, $i_m = j$ et $a_{i_{k-1}, i_k} = 1$ pour $k = 1, \dots, m$. Un *cycle* autour de i est un chemin de i vers lui-même de longueur non nulle. Un élément $i \in S$ est *cyclique* s'il existe un cycle autour de i.

Cette terminologie est inspirée de la théorie des graphes : un *graphe* est un couple (S, U), où S est un ensemble de *sommets* et $U \subset S \times S$ un ensemble d'*arcs*. La matrice $A = (a_{i,j}) \in \mathcal{M}$ définie par $a_{i,j} = 1$ si et seulement si $(i, j) \in U$ est appelée la *matrice d'adjacence* du graphe. Réciproquement, toute matrice $A \in \mathcal{M}$ définit un graphe.

Sur l'ensemble $\mathcal{M}$, on définit une addition notée $\oplus$ et une multiplication notée $\otimes$, comme suit : soient $A, B, C, D \in \mathcal{M}$. Alors $C = (c_{i,j}) = A \oplus B$ est définie par

$$c_{i,j} = \begin{cases} 0 & \text{si } a_{i,j} = b_{i,j} = 0 \\ 1 & \text{sinon} \end{cases}$$

et $D = (d_{i,j}) = A \otimes B$ est définie par

$$d_{i,j} = \begin{cases} 0 & \text{si } a_{i,k} b_{k,j} = 0 \text{ pour tout } k \in S \\ 1 & \text{sinon} \end{cases}$$

(On peut considérer les matrices de $\mathcal{M}$ à coefficients booléens; l'addition est alors le «ou» booléen et la multiplication le «et»; les formules ci-dessus reviennent à étendre ces opérations aux matrices.) Il résulte immédiatement de la définition que l'addition est idempotente : $A \oplus A = A$, que la matrice identité est neutre pour la multiplication et que la multiplication est distributive par rapport à l'addition. L'ensemble $\mathcal{M}$, muni de ces deux opérations, est un *semi-anneau*.

On définit la puissance $A^{\otimes k}$ par $A^{\otimes 0} = I$ (I est la matrice unité) et $A^{\otimes k+1} = A \otimes A^{\otimes k}$ pour $k \geq 0$. On note $a_{i,j}^{(k)}$ l'élément d'indice (i, j) de $A^{\otimes k}$. L'idempotence de l'addition entraîne que

$$(I \oplus A)^{\otimes k} = I \oplus A \oplus A^2 \oplus \cdots \oplus A^k \tag{2.1}$$

LEMME 6.2.1. *Soient i, j dans S; il existe un chemin de i vers j de longueur k si et seulement si $a_{i,j}^{(k)} = 1$.*

Preuve. Le lemme est évident pour $k = 0$. Soit $k \geq 1$. Alors $a_{i,j}^{(k)} = 1$ si et seulement s'il existe $\ell \in S$ tel que $a_{i,\ell}^{(k-1)} = a_{\ell,j} = 1$, donc par récurrence si et seulement s'il existe un ℓ, un chemin de i à ℓ de longueur $k-1$ et un arc de ℓ à j. ∎

Corollaire 6.2.2. *Il existe un chemin de i vers j si et seulement si le coefficient d'indice (i, j) de la matrice $(I \oplus A)^{\otimes n-1}$ est égal à 1.*

Preuve. La condition est évidemment suffisante, en vertu de (2.1). Réciproquement, s'il existe un chemin de i vers j, il en existe aussi un de longueur au plus $n-1$, en supprimant les cycles éventuels. ∎

On dit que j est *accessible* de i s'il existe un chemin de i vers j. Pour déterminer les éléments accessibles, il suffit donc de calculer la matrice $(I \oplus A)^{\otimes n-1}$; les éléments accessibles à partir de i sont les indices de colonnes des coefficients non nuls sur la ligne d'indice i.

Une matrice $A \in \mathcal{M}$ est dite *acyclique* si l'ensemble de ses éléments cycliques est vide.

Proposition 6.2.3. *Une matrice A est acyclique si et seulement si $A^{\otimes n} = 0$.*

Preuve. Si A est acyclique, tous les chemins ne contiennent que des éléments distincts et sont donc de longueur au plus $n-1$ et par conséquent $A^{\otimes n} = 0$. Réciproquement, si A contient un cycle, elle contient un cycle de longueur $k \leq n$, donc des chemins de toute longueur supérieure ou égale à k et par conséquent $A^{\otimes n} \neq 0$. ∎

Soit i un élément cyclique de S et soit $K(i) = \{k > 0 \mid a_{i,i}^{(k)} = 1\}$ l'ensemble des longueurs des cycles de i à i. On appelle *période* d'un élément cyclique $i \in S$ de A et on note $d(i)$, le pgcd des éléments de $K(i)$. Ce pgcd $d(i)$ est en fait le pgcd d'un nombre *fini* d'éléments de $K(i)$: considérons en effet la suite des pgcds d'une énumération des éléments de $K(i)$; cette suite est décroissante, donc est stationnaire à partir d'un certain rang. La période est indépendante de l'élément choisi sur le cycle.

Proposition 6.2.4. *Si i est accessible de j et j est accessible de i, alors $d(i) = d(j)$.*

Preuve. Soit $K(i) = \{k > 0 \mid a_{i,i}^{(k)} = 1\}$ l'ensemble des longueurs des cycles de i à i. Montrons que $d(j)$ divise tout $k \in K(i)$ (d'où il découle que $d(j)$ divise $d(i)$).

Soit en effet $k \in K(i)$ et soient p et q les longueurs de chemins de j à i et de i à j respectivement. Alors $p+q$ et $p+k+q$ appartiennent à $K(j)$, donc sont divisibles par $d(i)$; la différence de ces deux nombres, soit k, est aussi divisible par $d(j)$. ∎

On note U la matrice de $\mathcal{M}$ dont tous les coefficients sont égaux à 1.

Une matrice A carrée d'ordre n (à coefficients complexes) est *réductible* s'il existe une partition $\{1, \ldots, n\} = I \cup J$ telle que $a_{i,j} = 0$ pour $i \in I$ et $j \in J$. Elle est *irréductible* dans le cas contraire. Cette condition se vérifie simplement lorsque A est dans $\mathcal{M}$.

Proposition 6.2.5. *Une matrice A de $\mathcal{M}$ est irréductible si et seulement si elle vérifie $(I \oplus A)^{\otimes n-1} = U$.*

Preuve. Si A est réductible, il existe un couple (i, j) d'éléments de S tels qu'il n'y ait pas de chemin de i vers j, donc la matrice $B = (I \oplus A)^{\otimes n-1}$ n'est pas égale à U. Réciproquement, si $B = (b_{i,j}) \neq U$, soient i, j tels que $b_{i,j} = 0$. On a $i \neq j$. Soient alors $J = \{k \in S \mid b_{k,j} = 1\}$ et $I = S \setminus J$. Supposons qu'il existe $i' \in I$ et $j' \in J$ tels que $b_{i',j'} = 1$, donc qu'il existe un chemin de i' vers j' ; comme il existe un chemin de j' vers j, il existe donc un chemin de i' vers j, donc $i \in J$, ce qui est impossible. ∎

Une matrice A est *primitive* s'il existe un entier $k \geq 0$ tel que $A^{\otimes k} = U$. Bien entendu, une matrice primitive est *a fortiori* irréductible et la réciproque n'est pas vraie, comme le montre la matrice

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Plus généralement, une matrice de permutation n'est jamais primitive dès que $n > 1$, puisque toute puissance n'a qu'un seul 1 par ligne. En revanche, une matrice de permutation est irréductible si et seulement si la permutation est formée d'un seul cycle. Considérons un autre exemple. Une matrice est *apériodique* si elle possède un élément cyclique i de période $d(i) = 1$. Notons que cela n'implique pas l'existence d'un cycle de longueur 1.

PROPOSITION 6.2.6. *Une matrice irréductible et apériodique est primitive.*

Preuve. On montre d'abord qu'il existe un entier N tel que pour tout i, l'ensemble $K(i)$ contienne tous les entiers supérieurs ou égaux à N.

Soit $i \in S$ de période 1. Nous montrons d'abord que, pour tout entier ℓ assez grand, il existe un cycle de longueur ℓ de i à i. Soit $K(i)$ l'ensemble des longueurs des cycles de i à i. La période $d(i)$ est le pgcd d'un nombre *fini* d'éléments de $K(i)$. Par l'identité de Bezout, il existe $k_1, \ldots, k_r \in K(i)$ et $\lambda_1, \ldots, \lambda_r \in \mathbb{Z}$ tels que

$$1 = \sum_{t=1}^{r} \lambda_r k_r$$

Posons

$$a = \sum_{\lambda_r > 0} \lambda_r k_r \qquad b = -\sum_{\lambda_r < 0} \lambda_r k_r$$

Alors $a - b = 1$. On pose $N = b^2$. Tout entier $m \geq N$ s'écrit, par division euclidienne par b, sous la forme $m = ub + v$, avec $0 \leq v < b$ et $b \leq u$, d'où

$$m = ua + v = ub + v(a - b) = (u - v)b + va$$

Or $a, b \in K(i)$, donc $m \in K(i)$. Ainsi, $K(i)$ contient tous les entiers supérieurs ou égaux à N.

Soient maintenant p, q dans S. Soit N_p la longueur d'un chemin de p vers i et soit $M(q)$ la longueur d'un chemin de i vers q. Ces chemins existent parce que A est irréductible. Il existe donc un chemin de p vers q de longueur $L(p, q) = N(p) + N + M(q)$ et en fait de toute longueur supérieure ou égale à $L(p,q)$. Posons $L = \max L(p,q)$. Alors pour tout p, q, il existe un chemin de p vers q de longueur L, donc $A^L = U$. ■

Si A est primitive, le plus petit entier k tel que $A^{\otimes k} = U$ est noté $\gamma(A)$. Nous allons étudier ce nombre. Considérons d'abord un cas simple.

Proposition 6.2.7. *Si A est primitive et symétrique, alors $\gamma(A) \leq 2(n-1)$.*

Preuve. Si A est symétrique, alors $B = A^{\otimes 2}$ a ses coefficients diagonaux non nuls, donc $B = I \oplus B$ et, comme B est irréductible, $B^{\otimes n-1} = U$. ■

Théorème 6.2.8. *Pour toute matrice primitive A d'ordre $n > 1$, on a $\gamma(A) \leq n^2 - 2n + 2$.*

Preuve. Pour $i \in S$, on pose

$$h_i = \min\{k \in \mathbb{N} \mid a_{i,j}^{(k)} = 1 \text{ pour tout } j \in S\}$$

Comme A est irréductible, on a $a_{i,j}^{(h)} = 1$ pour tout $j \in S$ et tout $h \geq h_i$. Par conséquent $\gamma(A) = \max\{h_1, \ldots, h_n\}$.

Soit s la longueur du plus court cycle de A. Nous allons montrer que $\gamma(A) \leq n + s(n-2)$. Pour cela, montrons d'abord que, pour tout $i \in S$, il existe un entier $p_i \leq n - s$ tel que, pour tout j, il y a un chemin de i vers j de longueur exactement $p_i + s(n-1)$. Dans A, il existe un chemin de longueur au plus $n - s$ de i à un des éléments du cycle minimal. En effet, tout chemin sans élément répété, partant de i et de longueur $n - s$, contient $n - s + 1$ éléments. Il ne peut donc éviter les s éléments du cycle. Notons p_i la longueur de ce chemin.

Dans A^s, les éléments du cycle ont des boucles (i. e. des cycles de longueur 1). Comme A^s est encore irréductible, il existe dans A^s un chemin de chacun des éléments du cycle à tout élément $j \in S$ et, comme chacun des éléments du cycle comporte une boucle, il existe un chemin de longueur $n - 1$ exactement de chacun des éléments du cycle à chaque élément de S. Revenons à A. Il existe donc un chemin de longueur $s(n-1)$ de chaque élément du cycle minimal à chaque élément de S. Ceci prouve l'existence pour tout j, d'un chemin de longueur exactement $p_i + s(n-1)$ de i vers j. Par conséquent $h_i \leq p_i + s(n-1) \leq n - s + s(n-1) = n + s(n-2)$.

Il reste à prouver que $s \leq n - 1$. Si l'on avait $s = n$, la matrice A serait une matrice de permutation, ce qui est exclu puisqu'une telle matrice n'est pas primitive. ■

Il n'est pas difficile de voir que la matrice

$$\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & & \cdots & 1 \\ 1 & 1 & 0 & \cdots & 0 \end{pmatrix}$$

réalise l'égalité. A une permutation près, c'est la seule matrice qui réalise l'égalité.

6.2.3 Programme : matrices irréductibles

Les matrices considérées sont à coefficients 0 ou 1. On peut donc choisir de les représenter comme matrices à coefficients entiers ou à coefficients booléens. Nous choisissons la première solution, car elle va nous permettre de coder diverses informations supplémentaires dans les éléments des matrices. On définit donc

```
TYPE
        vecE = ARRAY[1..OrdreMax] OF integer;
        matE = ARRAY[1..OrdreMax] OF vecE;
```

où `OrdreMax` est une constante convenable. Nous avons besoin de la matrice unité :

```
VAR
        MatriceEUnite: matE;                          Matrice prédéfinie.
```

Les procédures de lecture, d'écriture et d'initialisation de la matrice unité sont en tout point analogues à celles déjà vues pour les matrices réelles. Nous ne donnons que les en-têtes :

```
PROCEDURE InitMatricesE;
PROCEDURE EntrerMatriceE (n: integer; VAR a: matE; titre: texte);
PROCEDURE EcrireMatriceE (n: integer; VAR a: matE; titre: texte);
```

Les deux opérations, notées $\oplus$ et $\otimes$, se réalisent comme suit :

```
PROCEDURE MatriceOPlusMatrice (n: integer; a, b: matE; VAR c: matE);
   Calcule la «somme» a ⊕ b de deux matrices a et b d'ordre n. Résultat dans la matrice c.
   VAR
      i, j: integer;
   BEGIN
      FOR i := 1 TO n DO
         FOR j := 1 TO n DO
            c[i, j] := max(a[i, j], b[i, j]);
   END; {de "MatriceOPlusMatrice" }

PROCEDURE MatriceOParMatrice (n: integer; a, b: matE; VAR ab: matE);
   Calcule le «produit» a ⊗ b de deux matrices a et b d'ordre n. Résultat dans la matrice
   ab.
   VAR
      i, j, k: integer;
      s: integer;
   BEGIN
      FOR i := 1 TO n DO
         FOR j := 1 TO n DO BEGIN
            s := 0;
            FOR k := 1 TO n DO s := max(s, a[i, k] * b[k, j]);
            ab[i, j] := s
         END
   END; {de "MatriceOParMatrice" }
```

Avec ces opérations, il est facile de calculer la matrice $(I \oplus A)^{\otimes n-1}$:

```
PROCEDURE Acces (n: integer; a: matE; VAR c: matE);
   VAR
      UnPlusA: matE;
      i: integer;
   BEGIN
      MatriceOPlusMatrice(n, MatriceEUnite, a, UnPlusA);
      c := UnPlusA;                                        C = I ⊕ A.
      FOR i := 2 TO n - 1 DO
         MatriceOParMatrice(n, c, UnPlusA, c);
   END; { de "Acces" }
```

Voici un exemple :

```
Voici la matrice a :
  1  0  1  0  0
  0  0  0  0  1
  0  1  0  1  0
  1  0  0  0  0
  0  1  0  0  0
Voici la matrice d'accès :
  1  1  1  1  1
  0  1  0  0  1
  1  1  1  1  1
  1  1  1  1  1
  0  1  0  0  1
```

On voit que tous les sommets sont accessibles à partir de $1, 3$ ou 4. Maintenant, considérons le problème de calculer non seulement les éléments accessibles, mais aussi un chemin pour les éléments accessibles à partir d'un élément i donné. Nous allons faire un peu plus et calculer un chemin de longueur positive dans le cas où i est cyclique. Pour cela, nous calculons

$$A \oplus A^{\otimes 2} \oplus \cdots \oplus A^{\otimes n}$$

et au lieu de calculer seulement cette matrice, nous allons conserver des informations supplémentaires : si i et j sont reliés par un chemin de longueur p, il existe un k tel que $a_{i,k} = 1$ et $a^{(p-1)}_{k,j} = 1$. Nous conservons l'entier k et savons ainsi qu'il existe un chemin de i vers j commençant par l'arc (i, k). Le codage demande une légère modification de la matrice au départ, pour ne pas confondre le sommet 1 avec l'existence d'un arc. L'algorithme est réalisé comme suit :

```
PROCEDURE Chemins (n: integer; a: matE; VAR c: matE);
```

Calcule une matrice c contenant, de façon compacte, les chemins entre éléments accessibles. Si $c_{i,j} = k > 0$, il existe un chemin de i vers j qui commence par (i, k). Si $k = 0$, le chemin est réduit au couple (i, j), si $k < 0$, il n'y a pas de chemin.

```
   VAR
      i, j, k: integer;
   BEGIN
      FOR i := 1 TO n DO                Diminution de tous les coefficients de a.
```

```
      FOR j := 1 TO n DO
         a[i, j] := a[i, j] - 1;
   c := a;
   FOR k := 2 TO n DO             Calcul des puissances de a.
      MatriceXMatriceE(n, a, c)
 END; { de "Chemins" }
```

Il utilise la procédure :

```
PROCEDURE MatriceXMatriceE (n: integer; a: matE; VAR c: matE);
```

Produit «cumulé» de deux matrices a et c d'ordre n. L'absence d'arc est repérée par un nombre négatif. Le résultat est dans la matrice c. Si l'on note c' la valeur de la matrice à la sortie de la procédure, on a $c'_{i,j} = k$ si $c_{i,j} < 0$, $a_{i,k} \geq 0$ et $c_{k,j} \geq 0$.

```
   VAR
      i, j, k: integer;
      b : matE;
   BEGIN
      b := c;
      FOR i := 1 TO n DO
         FOR j := 1 TO n DO
            FOR k := 1 TO n DO
               IF (a[i, k] >= 0) AND (b[k, j] >= 0) AND (c[i, j] < 0) THEN
                  c[i, j] := k
   END; {de "MatriceXMatriceE" }
```

Voici un exemple de calcul :

```
Voici la matrice a :
  1  0  1  0  0
  0  0  0  0  1
  0  1  0  1  0
  1  0  0  0  0
  0  1  0  0  0

Voici la matrice des chemins :
  0  3  0  3  3
 -1  5 -1 -1  0
  4  0  4  0  2
  0  1  1  1  1
 -1  0 -1 -1  2
```

Notons c cette matrice des chemins. La valeur $c_{4,2} = 1$ dit qu'un chemin de 4 à 2 débute par $(4, 1)$. La valeur $c_{1,2} = 3$ permet de continuer ce chemin ; on obtient $(4, 1, 3)$. Enfin, $c_{3,2} = 0$, donc le chemin complet est $(4, 1, 3, 2)$. La procédure suivante réalise ce calcul :

```
PROCEDURE CalculerChemin (i, j, n: integer; VAR c: matE;
   VAR longueur: integer; VAR chemin: vecE);
```

Extrait un chemin de i vers j de la matrice c. La longueur du chemin et les sommets qui le composent sont calculés.

```
   VAR
```

```
      k: integer;
   BEGIN
      IF c[i, j] < 0 THEN                Il n'y a pas de chemin.
         longueur := -1
      ELSE BEGIN
         k := 1;
         chemin[k] := i;                 Premier élément : i.
         WHILE c[i, j] > 0 DO BEGIN
            i := c[i, j];                Elément suivant.
            k := k + 1;
            chemin[k] := i;
         END;
         chemin[k + 1] := j;             Dernier élément.
         longueur := k                   Longueur k, et k + 1 éléments.
      END;
   END; { de "CalculerChemin" }
```

Voici les résultats obtenus avec la matrice précédente :

```
Chemin de 1 à 1    1  1
Chemin de 1 à 2    1  3  2
Chemin de 1 à 3    1  3
Chemin de 1 à 4    1  3  4
Chemin de 1 à 5    1  3  2  5
Chemin de 2 à 1
Chemin de 2 à 2    2  5  2
Chemin de 2 à 3
Chemin de 2 à 4
Chemin de 2 à 5    2  5
Chemin de 3 à 1    3  4  1
Chemin de 3 à 2    3  2
Chemin de 3 à 3    3  4  1  3
Chemin de 3 à 4    3  4
Chemin de 3 à 5    3  2  5
Chemin de 4 à 1    4  1
Chemin de 4 à 2    4  1  3  2
Chemin de 4 à 3    4  1  3
Chemin de 4 à 4    4  1  3  4
Chemin de 4 à 5    4  1  3  2  5
Chemin de 5 à 1
Chemin de 5 à 2    5  2
Chemin de 5 à 3
Chemin de 5 à 4
Chemin de 5 à 5    5  2  5
```

Les éléments cycliques et les cycles se calculent facilement sur la matrice. Il convient de noter que l'algorithme présenté ici, s'il est simple, n'est pas très efficace en temps. L'emploi de structures de données plus élaborées permet de calculer tous ces chemins en un temps proportionnel à leur longueur.

Tester qu'une matrice est acyclique est facile. On peut calculer les rangs des éléments de diverses manières : la plus simple est d'employer une méthode récursive, en observant que le rang d'un élément i est 1 plus le maximum des rangs des éléments j tels que $a_{i,j} = 1$. Voici comment on écrit cette procédure :

```
FUNCTION rang (i, n: integer; VAR a: matE): integer;
   VAR
      m, j: integer;
   BEGIN
      m := 0;                                    m est le maximum partiel.
      FOR j := 1 TO n DO
         IF a[j, i] = 1 THEN
            m := max(m, 1 + rang(j, n, a));      Ici calcul récursif.
      rang := m
   END; { de "rang" }
```

Comme souvent quand on a affaire à des procédures récursives, l'écriture est simple. On peut aussi calculer les rangs en observant que le rang de i est m si m est le plus petit entier tel que $A^{\otimes m+1}$ a sa i-ième colonne nulle. On obtient la procédure suivante :

```
PROCEDURE LesRangs (n: integer; a: matE; VAR rang: vecE);
   VAR
      i, j, m, h: integer;
      b: matE;
   BEGIN
      FOR i := 1 TO n DO rang[i] := -1;     Initialisation du tableau.
      b := a;                               B = A^{⊗m+1}.
      FOR m := 0 TO n - 1 DO BEGIN
         FOR i := 1 TO n DO
            IF rang[i] < 0 THEN BEGIN       Si le rang de i est au moins m et...
               h := 0;
               FOR j := 1 TO n DO h := max(h, b[j, i]);
               IF h = 0 THEN                si la colonne d'indice i est nulle,
                  rang[i] := m              alors le rang est m.
            END;
         MatriceOParMatrice(n, b, a, b)     Calcul de la puissance suivante de a.
      END;
   END; { de "LesRangs" }
```

La procédure présentée est très lente, en $O(n^4)$, à cause du calcul des puissances. Pour accélérer la détermination des rangs, nous allons calculer, pour chaque i, le nombre d_i de coefficients $a_{j,i}$ égaux à 1, pour $j = 1, \ldots, n$. Si $d_i = 0$, le rang de i est nul. On «supprime» alors i et on recommence. Au lieu de supprimer physiquement l'élément i, on met à jour le tableau d. Comme plusieurs éléments peuvent avoir un coefficient d_i nul, on doit gérer convenablement cet ensemble. On range ces éléments dans une *file* ou *queue*, représentée par un intervalle dans un tableau. La procédure s'écrit comme suit :

```
PROCEDURE LesRangsRapides (n: integer; a: matE; VAR rang: vecE);
```

```
VAR
   SommetsEnAttente: vecE;                        La queue.
   DebutQueue, FinQueue: integer;                 Le début et la fin de la queue.
   i, j, s: integer;
   d: vecE;                                       Le vecteur d.
PROCEDURE Prendre (VAR i: integer);               Prend le premier élément
   BEGIN                                          dans la queue.
      i := SommetsEnAttente[DebutQueue];
      DebutQueue := 1 + DebutQueue
   END; { de "Prendre" }
PROCEDURE Inserer (i: integer);                   Ajoute un élément à la queue.
   BEGIN
      SommetsEnAttente[FinQueue] := i;
      FinQueue := 1 + FinQueue
   END; { de "Inserer" }
FUNCTION EnAttente: boolean;                      Teste si la queue est vide.
   BEGIN
      EnAttente := FinQueue > DebutQueue
   END; { de "EnAttente" }
BEGIN
   DebutQueue := 1; FinQueue := 1;                Initialise la queue.
   FOR i := 1 TO n DO rang[i] := -1;              Initialise le tableau des rangs.
   FOR i := 1 TO n DO BEGIN                       Calcul du tableau d.
      s := 0;
      FOR j := 1 TO n DO s := s + a[j, i];
      d[i] := s;
      IF s = 0 THEN BEGIN
         inserer(i); rang[i] := 0                 Des éléments de rang 0.
      END;
   END;
   WHILE EnAttente DO BEGIN                       S'il reste des éléments,
      Prendre(i);                                 on en prend un,
      FOR j := 1 TO n DO
         IF a[i, j] = 1 THEN BEGIN                on supprime les arcs,
            d[j] := d[j] - 1;
            IF d[j] = 0 THEN                      et si plus aucun arc n'arrive à j,
               inserer(j);                        on ajoute j à la queue.
            rang[j] := max(rang[j], 1 + rang[i])     On met à jour le rang.
         END
   END;
END; { de "LesRangsRapides" }
```

Cette procédure a un temps d'exécution en $O(n^2)$. En effet, la boucle `WHILE` est parcourue une fois pour chaque sommet. Voici un exemple de résultat obtenu :

```
Voici la matrice :
  0  1  0  1  0  0
  0  0  0  1  0  0
  0  1  0  0  0  1
```

```
  0  0  0  0  0  0
  1  1  0  0  0  0
  0  0  0  0  0  0

Voici les rangs  :
rang(1) =   1
rang(2) =   2
rang(3) =   0
rang(4) =   3
rang(5) =   0
rang(6) =   1
```

Dans le cas où la matrice n'est pas acyclique, les deux procédures itératives se terminent, alors que la procédure récursive boucle. Dans le tableau des rangs, les éléments cycliques conservent leur valeur -1 et on peut donc ainsi les trouver d'une autre manière.

Notes bibliographiques

Il existe une abondante littérature sur les matrices totalement unimodulaires. On pourra consulter les chapitres 19, 20 et 21 et la bibliographie (très complète) du livre suivant :

A. Schrijver, *Theory of Linear and Integer Programming*, New-York, John Wiley and Sons, 1986

L'algorithme utilisé dans la procédure `ElementaireEstTU` est du type «Union-Find». On trouvera une discussion détaillée de ce type d'algorithme dans le livre :

R. Sedgewick, *Algorithms*, Reading, Addison-Wesley, 1983.

Le théorème 6.2.8 est dû à Wielandt. Pour les liens entre matrices à coefficients 0 et 1 et les graphes, on pourra consulter par exemple :

C. Froidevaux, M. C. Gaudel, M. Soria, *Types de données et algorithmes*, Paris, McGraw-Hill, 1990.

Partie II

Polynômes

Chapitre 7

Polynômes

7.1 Suites de Sturm

Les suites de Sturm constituent un moyen efficace pour déterminer les zéros réels d'un polynôme à coefficients réels. Cette méthode est un peu lente, mais sûre et est particulièrement intéressante lorsque l'on sait par ailleurs que tous les zéros sont réels.

7.1.1 Enoncé : suites de Sturm

Soit $p(X) = a_n X^n + \cdots + a_0$ un polynôme de degré n, à coefficients réels, sans zéros multiples. On cherche à *localiser* les zéros réels de $p(X)$, c'est-à-dire à calculer des intervalles deux à deux disjoints contenant chacun un et un seul zéro réel de $p(X)$.

Dans cet énoncé, on appelle *suite de Sturm* associée à $p(X)$ la suite de polynômes définie par $p_0(X) = p(X)$, $p_1(X) = p'(X)$ et

$$p_{i-2}(X) = p_{i-1}(X)q_{i-1}(X) - p_i(X) \qquad (2 \leq i \leq N)$$

le polynôme $-p_i(X)$ étant le reste de la division euclidienne de $p_{i-2}(X)$ par $p_{i-1}(X)$. L'entier N est déterminé par $p_N \neq 0$, $p_{N+1} = 0$. On pose $q_N(X) = p_{N-1}(X)/p_N(X)$.

Soit y un nombre réel qui n'est pas un zéro de $p(X)$. La *variation* de $p(X)$ en y, notée $V(y)$, est le nombre de changements de signe dans la suite $p_0(y), p_1(y), \ldots, p_N(y)$. Les termes nuls de cette suite sont ignorés dans le décompte. Par exemple, pour la suite -1, 2, 1, 0, -2, la variation est 2. On rappelle le théorème de Sturm :

Si a et b sont deux nombres réels tels que $a < b$ et si $p(X)$ ne s'annule ni en a ni en b, alors le nombre de zéros réels de $p(X)$ dans l'intervalle $[a, b]$ est $V(a) - V(b)$.

1.– Montrer que la connaissance des quotients $q_1(y), \ldots, q_N(y)$ et de p_N suffit pour calculer la variation $V(y)$ de $p(X)$ en y.

2.– Ecrire une procédure qui prend en argument un polynôme p et qui calcule les quotients $q_1(X), \ldots, q_N(X)$ et p_N. Ecrire une procédure qui calcule $V(y)$ en fonction de y.

On admet que si α est un zéro de p, on a

$$|\alpha| \le 2 \max\left(\left|\frac{a_{n-1}}{a_n}\right|, \left|\frac{a_{n-2}}{a_n}\right|^{1/2}, \left|\frac{a_{n-3}}{a_n}\right|^{1/3}, \ldots, \left|\frac{a_0}{a_n}\right|^{1/n}\right) \qquad (*)$$

3.– Déduire de la formule $(*)$ une minoration du module des zéros d'un polynôme $q(X)$ tel que $q(0) \neq 0$. En déduire une procédure qui, pour un zéro réel x_0 du polynôme $p(X)$, calcule un réel $\epsilon > 0$ tel que $p(X)$ n'ait pas d'autre zéro dans l'intervalle $]x_0 - \epsilon, x_0 + \epsilon[$. (On pourra faire un changement de variable dans $p(X)$.)

4.– Ecrire un programme de localisation des zéros réels de $p(X)$. Pour cela, partant d'un intervalle $[c, d]$, on calcule $V(c) - V(d)$; si $V(c) - V(d) > 1$ alors on calcule $V(e)$ pour $e = (c + d)/2$ et on considère séparément le cas où $p(e) = 0$ et le cas où $p(e) \neq 0$.

Exemple numérique : $p(X) = X^3 - 6X^2 + 11X - 6$.

5.– Compléter ce programme pour qu'il accepte des polynômes ayant des zéros multiples.

6.– On suppose que $p(X)$ est sans zéros multiples. Démontrer que si $p_i(x) = 0$ pour un $i > 0$ et un réel x, alors $p_{i-1}(x)p_{i+1}(x) < 0$.

7.– Soient a et b deux nombres réels tels que $a < b$ et $p(a)p(b) \neq 0$. Soit $x_0 \in]a, b[$ un zéro de $p(X)$. Démontrer que si $\epsilon > 0$ est suffisamment petit, alors $V(x_0 - \epsilon) - V(x_0 + \epsilon) = 1$ et en déduire le théorème de Sturm.

7.1.2 Solution : suites de Sturm

Donnons d'abord la définition générale des suites de Sturm, plus générale que celle de l'énoncé. Soit $p(X)$ un polynôme (à coefficients réels) et soient a et b deux nombres réels avec $a < b$. Une suite de polynômes

$$p_0, p_1, \ldots, p_m$$

avec $p_0 = p$ est une *suite de Sturm* pour p sur $[a, b]$ lorsque les conditions suivantes sont vérifiées :

(i) $p(a)p(b) \neq 0$;
(ii) la fonction polynôme p_m ne s'annule pas sur $[a, b]$;
(iii) si $p_j(c) = 0$ pour un $c \in]a, b[$ et pour un indice j tel que $0 < j < m$, alors

$$p_{j-1}(c)p_{j+1}(c) < 0$$

On considérera le plus souvent des suites qui vérifient la condition supplémentaire :

(iv) si $p(c) = 0$ pour un $c \in]a, b[$, alors $p(x)p_1(x)(x - c) > 0$ au voisinage de c, ou la condition opposée :

(iv') si $p(c) = 0$ pour un $c \in]a, b[$, alors $p(x)p_1(x)(c - x) > 0$ au voisinage de c.

A une suite de Sturm et à un point y de l'intervalle $]a, b[$, on associe la *variation* de la suite en y, nombre noté $V(p_0, \ldots, p_m; y)$ ou plus simplement $V(y)$, qui est le nombre de changements de signes dans la suite de nombres $p_0(y), \ldots, p_m(y)$. Les termes nuls de cette suite sont ignorés dans le décompte. Soit $(a_0, \ldots, a_m)$ une suite de nombres réels; on définit

$$C(a_0, \ldots, a_m) = \mathrm{Card}\{(i, j) \mid 0 \leq i < j \leq m,\ a_i a_j < 0 \text{ et } a_k = 0 \text{ si } i < k < j\}$$

Par exemple, $C(-1, 2, 1, 0, -2) = 2$. Avec cette notation,

$$V(p_0, \ldots, p_m; y) = V(y) = C(p_0(y), \ldots, p_m(y))$$

Théorème 7.1.1. *Soit p un polynôme à coefficients réels et soient a et b deux nombres réels tels que $a < b$. Soit $p_0, \ldots, p_m$ une suite de Sturm pour p sur $[a, b]$ et posons*

$$\begin{aligned} r_+ &= \mathrm{Card}\{c \in [a, b] \mid p(c) = 0 \textit{ et } p(x)p_1(x)(x - c) > 0 \textit{ au voisinage de } c\} \\ r_- &= \mathrm{Card}\{c \in [a, b] \mid p(c) = 0 \textit{ et } p(x)p_1(x)(c - x) > 0 \textit{ au voisinage de } c\} \end{aligned}$$

Alors $V(a) - V(b) = r_+ - r_-$.

Preuve. Remarquons d'abord que $V(y)$ est constant sur tout intervalle contenu dans $[a, b]$ et sur lequel aucune fonction p_i ne s'annule. Comme les p_i n'ont qu'un nombre fini de zéros, il suffit d'examiner la modification de $V(y)$ en ces points.

Supposons d'abord que $c \in]a, b[$ soit un zéro de p et montrons que $p_1(c) \neq 0$. En effet, sinon on aurait $m > 1$ par la condition (ii) et alors, par (iii) avec $j = 1$, on aurait $p(c) \neq 0$, contrairement à l'hypothèse. Donc $p_1(c) \neq 0$ et, au voisinage de c, la fonction $p_1(x)$ ne change pas de signe.

Si $p(x)p_1(x)(x - c) > 0$ au voisinage de c, alors, pour $\varepsilon > 0$ assez petit, les nombres $p_1(c - \varepsilon)$ et $p(c - \varepsilon)$ sont de signe opposé, alors que $p_1(c + \varepsilon)$ et $p(c + \varepsilon)$ sont de même signe. Donc $C(p(c - \varepsilon), p_1(c - \varepsilon)) = 1$ et $C(p(c + \varepsilon), p_1(c + \varepsilon)) = 0$.

Si $p(x)p_1(x)(c - x) > 0$ au voisinage de c, alors le même argument montre que $C(p(c - \varepsilon), p_1(c - \varepsilon)) = 0$ et $C(p(c + \varepsilon), p_1(c + \varepsilon)) = 1$.

Supposons maintenant que $p_i(c) = 0$ pour un $c \in]a, b[$ avec $i > 0$ (ce cas peut se produire simultanément avec le cas précédent). Alors $i < m$ en vertu de (ii), et par (iii), on a $p_{i-1}(c)p_{i+1}(c) < 0$. Il en résulte que

$$C(p_{i-1}(c - \varepsilon), p_i(c - \varepsilon), p_{i+1}(c - \varepsilon)) = C(p_{i-1}(c + \varepsilon), p_i(c + \varepsilon), p_{i+1}(c + \varepsilon)) = 1$$

Par conséquent, $V(c - \varepsilon) - V(c + \varepsilon) = 1$, si $p(x)p_1(x)(x - c) > 0$ et $V(c - \varepsilon) - V(c + \varepsilon) = -1$, si $p(x)p_1(x)(c - x) > 0$. En sommant sur les zéros de l'intervalle $[a, b]$, on obtient $V(a) - V(b) = r_+ - r_-$. ∎

Corollaire 7.1.2. *Soit p un polynôme à coefficients réels et soient a et b deux nombres réels tels que $a < b$. Si $p_0, \ldots, p_m$ est une suite de Sturm pour p sur $[a, b]$, vérifiant la condition (iv) (resp. (iv')), alors le nombre de zéros distincts de p dans l'intervalle $[a, b]$ est $V(a) - V(b)$ (resp. $V(b) - V(a)$).*

Preuve. Si la condition (iv) est vérifiée, alors $r_- = 0$. De plus, la condition (iv) montre que r_+ compte le nombre de zéros distincts de p dans l'intervalle. ∎

Nous considérons maintenant l'existence de suites de Sturm.

Proposition 7.1.3. *Soit p un polynôme à coefficients réels et soient a et b deux nombres réels tels que $a < b$. Considérons la suite $p_0, \ldots, p_m$ de polynômes définie par $p_0 = p$, $p_1 = p'$, où le polynôme $-p_i$ est le reste de la division euclidienne de p_{i-2} par p_{i-1} pour $2 \leq i \leq m$ et où m est le plus grand entier tel que $p_m \neq 0$. Alors p_m divise chacun des p_i et si $p(a)p(b) \neq 0$, la suite*

$$g_i = p_i/p_m \qquad i = 0, \ldots, m$$

est une suite de Sturm pour g_0 sur $[a, b]$ vérifiant la condition (iv).

Preuve. Vérifions les propriétés (i) à (iv) ci-dessus pour la suite g_i. Notons que p_m est le pgcd de p et de p', au signe près, donc en particulier g_0 et g_1 n'ont pas de zéro commun. Puisque $p(a)p(b) \neq 0$, on a $g_0(a)g_0(b) \neq 0$, ce qui prouve (i). La condition (ii) provient du fait que $g_m = 1$. Pour (iii), considérons les divisions euclidiennes

$$p_{i-1}(X) = q_i(X)p_i(X) - p_{i+1}(X) \qquad 0 < i < m$$

Alors $g_{i-1}(X) = q_i(X)g_i(X) - g_{i+1}(X)$, pour $0 < i < m$. Deux polynômes g_i d'indices consécutifs n'ont pas de zéro commun dans $[a, b]$, sinon de proche en proche, p_m aurait un zéro dans $[a, b]$. Donc $g_{i-1}(c)g_{i+1}(c) < 0$ quand $g_i(c) = 0$. Enfin, si $g_0(c) = 0$, alors $g_1(c) \neq 0$ et on obtient (iv) grâce au théorème de Rolle. ∎

Corollaire 7.1.4. *Soit p un polynôme à coefficients réels et soient a et b deux nombres réels tels que $a < b$. Considérons la suite $p_0, \ldots, p_m$ de polynômes définie par $p_0 = p$, $p_1 = p'$, où le polynôme $-p_i$ est le reste de la division euclidienne de p_{i-2} par p_{i-1} pour $2 \leq i \leq m$ et où m est le plus grand entier tel que $p_m \neq 0$. Si $p(a)p(b) \neq 0$, alors le nombre de zéros distincts de p dans l'intervalle $[a, b]$ est égal à $V(a) - V(b)$.*

Preuve. Avec les notations de la proposition précédente, on a $p_i = p_m g_i$, pour $i = 0, \ldots, m$. Il en résulte que

$$V(p_0, \ldots, p_m; y) = V(g_0, \ldots, g_m; y)$$

pour tout y tel que $p_m(y) \neq 0$. Or tout zéro de p_m est un zéro de p, donc $p_m(a)p_m(b) \neq 0$. Le résultat découle alors directement de la proposition précédente. ∎

On considère maintenant trois suites de nombres réels $(a_n)_{n\geq 0}$, $(b_n)_{n\geq 1}$ et $(c_n)_{n\geq 2}$ sujettes aux deux conditions suivantes :

(i) $a_0 \neq 0$, $b_1 \neq 0$,
(ii) $b_n b_{n-1} c_n > 0$ pour tout $n \geq 2$,
et on définit une suite p_n de polynômes à coefficients réels par

$$\begin{aligned}
&p_0(X) = a_0 \\
&p_1(X) = a_1 + b_1 X \\
&p_n(X) = (a_n + b_n X) p_{n-1}(X) - c_n p_{n-2} \qquad n \geq 2
\end{aligned}$$

Comme $b_n \neq 0$ pour tout $n > 0$, chaque polynôme p_n est de degré n.

PROPOSITION 7.1.5. *Les zéros de p_n sont tous réels, simples et, pour $n \geq 2$, on a*

$$x_1 < y_1 < x_2 < \cdots < y_{n-1} < x_n$$

où $x_1, \ldots, x_n$ sont les zéros de p_n et $y_1, \ldots, y_{n-1}$ sont les zéros de p_{n-1}.

Preuve. Observons d'abord que deux polynômes d'indices consécutifs n'ont pas de zéro commun ou, de manière équivalente, sont premiers entre eux. En effet, si $p_n(x) = p_{n-1}(x) = 0$, alors par la définition de p_n, on a $p_{n-2}(x) = 0$ et, de proche en proche, on obtient $p_0(x) = 0$. Comme $a_0 \neq 0$, ceci est impossible.

Raisonnons par récurrence sur n et supposons la propriété établie pour $n - 1$. Si y est un zéro de p_{n-1}, on a $p_n(y)p_{n-2}(y) = -c_n(p_{n-2})^2$, donc les nombres

$$p_n(y_1)p_{n-2}(y_1), \ldots, p_n(y_{n-1})p_{n-2}(y_{n-1})$$

sont tous du signe de $-c_n$.

Les nombres $p_{n-2}(y_1), \ldots, p_{n-2}(y_{n-1})$ sont de signes alternés. En effet, si $p_{n-2}(y_i)$ et $p_{n-2}(y_{i+1})$ étaient de même signe, le nombre de zéros de $p_{n-2}(X)$ dans l'intervalle $]y_i, y_{i+1}[$ serait pair. Or, p_{n-2} a un seul zéro simple dans cet invervalle par récurrence.

Il résulte des deux remarques précédentes que les nombres $p_n(y_1), \ldots, p_n(y_{n-1})$ sont de signes alternés et donc que p_n a un nombre impair de zéros dans chacun des intervalles $]y_i, y_{i+1}[$ pour $i = 1, \ldots, n-2$. Comme $p_n(y_i) = -c_n p_{n-2}(y_i)$, le polynôme $q(X) = p_n(X)p_{n-2}(X)$ est du signe de $-c_n$ en tout point y_i. Or, pour $x \to \pm\infty$, on a $q(x) \sim x^2 b_n b_{n-1}$, donc $q(x)$ est du même signe que c_n. Ceci montre que $q(x)$, et donc aussi $p_n(x)$, change de signe dans l'intervalle $]-\infty, y_1[$ et dans l'intervalle $]y_{n-1}, \infty[$, et donc que p_n a un zéro dans chacun de ces intervalles, en plus des $n - 2$ zéros déjà mis en évidence. ∎

PROPOSITION 7.1.6. *Si $c_n > 0$ pour tout $n \geq 2$, alors pour tout $m > 0$, la suite*

$$(p_m, p_{m-1}, \ldots, p_0)$$

est une suite de Sturm pour p_m dans l'intervalle $[a, b]$ pour des nombres réels $a < b$ qui ne sont pas des zéros de p_m ; de plus, le nombre de zéros de p_m dans l'intervalle $[a, b]$ est $V(a) - V(b)$ si $b_1 > 0$ et $V(b) - V(a)$ sinon.

Preuve. Notons d'abord que la condition sur les polynômes p_n implique que, si $c_n > 0$, tous les b_n sont du même signe, celui de b_1.

Comme p_0 est un polynôme constant et comme $p_{n+1}(c) = -c_{n+1}p_{n-1}(c)$ pour tout $n > 0$ et tout zéro c de p_n, la condition que c_n est positif entraîne que la suite $(p_m, p_{m-1}, \ldots, p_0)$ est une suite de Sturm.

Pour montrer la deuxième partie de l'énoncé, nous allons prouver que la suite vérifie la condition (iv) respectivement (iv') des suites de Sturm et appliquer le théorème. Soient $x_1 < \cdots < x_m$ les zéros de p_m. Comme ces zéros sont simples, la fonction polynôme p_m est alternativement croissante et décroissante au voisinage de ces zéros. Par ailleurs, p_{m-1} ayant ses zéros entre deux zéros de p_m, la fonction p_{m-1} est alternativement positive et négative au voisinage des zéros de p_m. Il en résulte que les fonctions

$$p_m(x)p_{m-1}(x)(x - x_i) \quad i = 1, \ldots, m$$

sont soit toutes positives, soit toutes négatives aux voisinages de x_i. Il suffit donc de déterminer le signe de l'une d'entre elles. Or, pour $x > x_m$, les fonctions ne changent plus de signe et lorsque $x \to \infty$, le signe de $p_m(x)p_{m-1}(x)$ est celui de b_m. ∎

Il existe de nombreuses inégalités permettant de localiser les zéros d'un polynôme en fonction de paramètres divers. Nous considérons quelques inégalités bien classiques, portant sur les polynômes à coefficients complexes. Il suffit de considérer des polynômes *unitaires*, c'est-à-dire dont le coefficient du terme de plus haut degré est 1.

Théorème 7.1.7. *Soit $p(X) = X^n + \cdots + a_0$ un polynôme unitaire à coefficients complexes de degré n et soit α un zéro de p. Alors*

(*i*) $$|\alpha| \le \max\Bigl(1, \textstyle\sum_{i=0}^{n-1} |a_i|\Bigr)$$

(*ii*) $$|\alpha| \le 1 + \max_{0\le i\le n-1} |a_i|$$

(*iii*) $$|\alpha| \le \max_{0\le i\le n-1} |na_i|^{1/i}$$

(*iv*) $$|\alpha| \le 2 \max_{0\le i\le n-1} |a_i|^{1/i}$$

Pour la preuve, nous établissons d'abord le lemme suivant :

Lemme 7.1.8. *Soit $p(X) = X^n + a_{n-1}X^{n-1} + \cdots + a_0$ un polynôme unitaire à coefficients complexes. Soient $c_0, \ldots, c_{n-1}$ des nombres réels strictement positifs vérifiant $\sum_{i=0}^{n-1} c_i \le 1$. Soit enfin*

$$M = \max_{0\le i\le n-1} \left(\frac{|a_i|}{c_i}\right)^{1/n-i}$$

Alors tout zéro α de p vérifie $|\alpha| \le M$.

Preuve. Soit $z \in \mathbb{C}$ tel que $|z| > M$. Alors pour $0 \le i \le n-1$, on a

$$|z|^{n-i} > \frac{|a_i|}{c_i}$$

donc $|z|^n c_i > |a_i||z|^i$. Il en résulte que

$$|f(z)| \;\geq\; |z|^n - \sum_{i=0}^{n-1} |a_i||z|^i \;\geq\; |z|^n \sum_{i=0}^{n-1} c_i - \sum_{i=0}^{n-1} |a_i||z|^i = \sum_{i=0}^{n-1} \big(|z|^n c_i - |a_i||z|^i\big) > 0 \qquad \blacksquare$$

Preuve du théorème.
(i) Soit $r = \sum_{i=0}^{n-1} |a_i|$. Si $r \leq 1$, on pose $c_i = |a_i|$ dans le lemme et on a $M = 1$. Sinon, on pose $c_i = |a_i|/r$ et comme $r > 1$, on a $M = r$.
(ii) Soit $A = \max_{0 \leq i \leq n-1} |a_i|$ et posons

$$c_i = \frac{|a_i|}{(1+A)^{n-i}}$$

dans le lemme. Comme

$$\sum_{i=0}^{n-1} c_i = \frac{1}{(1+A)^n} \sum_{i=0}^{n-1} |a_i|(1+A)^i \leq 1$$

les hypothèses du lemme sont vérifiées et on obtient $M = 1 + A$.
(iii) Posons $c_i = 1/n$ dans le lemme. Alors $M = \max_{0 \leq i \leq n-1} |na_i|^{1/n-i}$.
(iv) Posons $c_i = 2^{i-n}$ dans le lemme. On trouve

$$\left(\frac{|a_i|}{c_i}\right)^{1/n-i} = 2|a_i|^{1/n-i}$$

d'où le résultat. $\blacksquare$

7.1.3 Programme : suites de Sturm

Le calcul de la suite de Sturm d'un polynôme p ne pose pas de problème, lorsque l'on dispose des procédures de base de manipulation des polynômes (consulter à ce propos l'annexe A). Le résultat étant une suite de polynômes, il convient d'introduire un type approprié, à savoir

```
TYPE
 SuitePol = ARRAY[0..LongueurSuite] OF pol;
```

La procédure est alors :

```
PROCEDURE SuiteDeSturm (p: pol; VAR sp: SuitePol; VAR m: integer);
   Calcule la suite de Sturm p0,...,pm du polynôme p, dans les éléments sp[0],...,sp[m].
   Le polynôme p est supposé n'avoir que des zéros simples.
   VAR
      q, r: pol;
```

```
    BEGIN
       m := 0;
       sp[0] := p;
       PolynomeDerive(p, q);
       WHILE NOT (EstPolNul(q)) DO BEGIN
          m := m + 1; sp[m] := q;
          PolynomeModPolynome(p, q, r);
          PolynomeOppose(r, r);
          p := q; q := r
       END;
    END; { de "SuiteDeSturm" }
```

Voici un exemple de calcul :

```
Voici le polynôme p :
 X^4 - 10.000 X^3 + 35.000 X^2 - 50.000 X + 24.000
Voici la suite de Sturm :
p_0
 X^4 - 10.000 X^3 + 35.000 X^2 - 50.000 X + 24.000
p_1
 4.000 X^3 - 30.000 X^2 + 70.000 X - 50.000
p_2
 1.250 X^2 - 6.250 X + 7.250
p_3
 3.200 X - 8.000
p_4
 0.562
```

Si le polynôme p à examiner a des zéros multiples, il faut les éliminer auparavant. Pour cela, il suffit de diviser p par le polynôme pgcd de p et du polynôme dérivé p'. Ceci est fait dans la procédure que voici :

```
PROCEDURE EliminationZerosMultiples (p: pol; VAR q: pol);
   BEGIN
      PolynomeDerive(p, q);
      PolynomePgcd(p, q, q);
      PolynomeDivPolynome(p, q, q)
   END; { de "EliminationZerosMultiples" }
```

Le calcul de la variation d'une suite de Sturm $p_0, \ldots, p_m$ en y se fait en évaluant les polynômes en y et en notant les changements de signe. Comme on suppose p_0 sans zéro multiple, p_m est une constante non nulle, donc de signe non nul. Les changements de signe se repèrent par le fait que le signe est l'opposé du signe précédent. Voici une réalisation :

```
FUNCTION Variation (m: integer; VAR sp: SuitePol; y: real) : integer;
   Calcule la variation V(y) au point y.
   VAR
      i, s, changements: integer;
```

```
      r: real;
   BEGIN
      changements := 0;                    Compte les changements de signe.
      s := signe(sp[m][0]);                Signe de p_m, non nul.
      FOR i := m - 1 DOWNTO 0 DO BEGIN
         r := Valeur(sp[i], y);            r = p_i(y).
         IF signe(r) = -s THEN BEGIN       Ainsi, on saute les termes nuls.
            s := -s;
            changements := changements + 1
         END;
      END;
      Variation := changements
   END; { de "Variation" }
```

La majoration des modules des zéros d'un polynôme $p(X) = a_n X^n + \cdots + a_0$ se fait en appliquant l'une des formules du théorème. Pour la minoration, il suffit de considérer le polynôme réciproque $X^n p(1/X) = a_0 X^n + \cdots + a_n$ et d'en majorer les zéros. Voici une réalisation :

```
FUNCTION MajorationDesZeros (p: pol): real;
   VAR
      r: real;
      i, n: integer;
   BEGIN
      n := Degre(p);
      r := 0;
      FOR i := 1 TO n DO
         r := rmax(r, rPuiss(abs(p[n - i] / p[n]), 1 / i));
      MajorationDesZeros := 2 * r
   END; { de "MajorationDesZeros" }

FUNCTION MinorationDesZeros (p: pol): real;
   VAR
      r: real;
      i, n: integer;
   BEGIN
      n := Degre(p);
      r := 0;
      FOR i := 1 TO n DO
         r := rmax(r, rPuiss(abs(p[i] / p[0]), 1 / i));
      MinorationDesZeros := 1 / (2 * r)
   END; { de "MinorationDesZeros" }
```

En fait, on est intéressé par la minoration des zéros dans la situation suivante : lorsque l'on a découvert un zéro réel c de p, on veut déterminer un intervalle autour de c qui ne contient aucun autre zéro de p. Pour cela, on considère le polynôme $q(X) = p(X+c)/X$. Une minoration δ du module des zéros de q donne un intervalle $]c-\delta, c+\delta[$ dans lequel il n'y a aucun zéro de p à l'exception de c. Le changement de variable indiqué se réalise par :

```
PROCEDURE ChangementVariable (p: pol; c: real; VAR q: pol);
   Calcule le polynôme q(X) = p(X + c)/X.
   VAR
      j, k, n: integer;
      s: real;
   BEGIN
      q := PolynomeNul;
      n := Degre(p);
      FOR k := 1 TO n DO BEGIN
         s := 0;
         FOR j := k TO n DO
            s := s + p[j] * Binomial(j, k) * Puiss(c, j - k);
         q[k - 1] := s
      END
   END; { de "ChangementVariable" }
```

Les calculs des coefficients binomiaux et des puissances se font au moyen de procédures simples, expliquées dans l'annexe A. Le nombre δ (que nous appelons écart) se calcule simplement par :

```
FUNCTION Ecart (p: pol; c: real): real;
   VAR
      q: pol;
   BEGIN
      ChangementVariable(p, c, q);
      Ecart := MinorationDesZeros(q)
   END; { de "Ecart" }
```

L'algorithme de localisation des zéros opère par dichotomie. On détermine d'abord un intervalle ouvert dans lequel se trouvent tous les zéros réels du polynôme p. Etant donné un intervalle ouvert contenant au moins un zéro, on le coupe en deux au milieu, puis on travaille séparément sur chaque intervalle. On détermine le nombre de zéros qu'il contient. Si ce nombre est nul, on élimine l'intervalle, s'il est égal à 1, on le range parmi les intervalles trouvés, s'il est plus grand que 1, on itère le procédé sur cet intervalle. Il se peut que le milieu de l'intervalle soit un zéro; dans ce cas, on continue non pas avec chacun des deux intervalles moitié, mais avec les intervalles diminués de l'écart qui sépare nécessairement le zéro trouvé d'un autre zéro. Le résultat de l'algorithme est une suite d'intervalles (éventuellement réduits à un point) contenant chacun un et un seul zéro du polynôme de départ.

Pour mettre en pratique cet algorithme, on doit gérer des intervalles et, en particulier, mettre de côté l'un des deux intervalles résultant d'une dichotomie pour le traiter ultérieurement. Ceci se fait en conservant ces intervalles dans une suite que nous réalisons sous forme de *pile*, mais toute autre structure convient. Il s'avère économique de gérer, en même temps que les intervalles, les variations à leurs extrémités, pour ne pas les recalculer. On procède comme suit :

```
TYPE
   Intervalle = ARRAY[0..1] OF real;
   SuiteIntervalles = ARRAY[1..NombreIntervalles] OF Intervalle;
   CoupleEntier = ARRAY[0..1] OF integer;
   SuiteCouplesEntier = ARRAY[1..NombreCouples] OF CoupleEntier;
```

Ces types permettent de gérer d'une part les intervalles par leurs extrémités et d'autre part les couples de variations aux extrémités. On utilise alors trois variables :

```
VAR
   EnAttente: integer;
   IntervalleEnAttente: SuiteIntervalles;
   VariationsEnAttente: SuiteCouplesEntier;
```

pour les intervalles en attente et leurs variations, l'entier `EnAttente` étant le nombre d'intervalles en attente. Deux opérations doivent être réalisées. D'une part, mettre en attente un intervalle et les variations correspondantes, d'autre part la récupération d'un tel objet mis en attente. C'est le propos des deux procédures que voici :

```
PROCEDURE MettreEnAttente (u, v: real; Vu, Vv: integer);
   BEGIN
      EnAttente := EnAttente + 1;
      IntervalleEnAttente[EnAttente][0] := u;
      IntervalleEnAttente[EnAttente][1] := v;
      VariationsEnAttente[EnAttente][0] := Vu;
      VariationsEnAttente[EnAttente][1] := Vv;
   END; { de "MettreEnAttente" }
PROCEDURE Retirer (VAR u, v: real; VAR Vu, Vv: integer);
   BEGIN
      u := IntervalleEnAttente[EnAttente][0];
      v := IntervalleEnAttente[EnAttente][1];
      Vu := VariationsEnAttente[EnAttente][0];
      Vv := VariationsEnAttente[EnAttente][1];
      EnAttente := EnAttente - 1;
   END; { de "Retirer" }
```

L'algorithme lui-même produit une suite d'intervalles. Chaque intervalle contient exactement un zéro réel du polynôme. Lorsque l'on a trouvé un tel intervalle, on le range dans une suite — appelons-la `Localise` – par la procédure :

```
PROCEDURE RangerIntervalleTrouve (u, v: real);
   BEGIN
      n := n + 1;
      Localise[n][0] := u;
      Localise[n][1] := v;
   END; { de "RangerIntervalleTrouve" }
```

où `n` est le compteur du nombre d'intervalles trouvé.

Pour déterminer le sort d'un intervalle, on utilise une procédure qui soit le met en attente, soit le range dans les intervalles trouvés, soit l'ignore, et ceci en fonction du nombre de zéros que cet intervalle contient.

```
PROCEDURE Repartir (a, b: real; Va, Vb: integer);
   BEGIN
      IF Va = Vb + 1 THEN                    Un zéro dans ]a, b[.
         RangerIntervalleTrouve(a, b)
      ELSE IF Va > Vb + 1 THEN               Au moins deux zéros dans ]a, b[.
         MettreEnAttente(a, b, Va, Vb)
   END; { de "Repartir" }
```

Voici donc la procédure qui met en œuvre ces opérations :

```
PROCEDURE LocalisationIntervalles (p: pol; VAR Localise: SuiteIntervalles;
   VAR n: integer);
   p est un polynôme n'ayant que des zéros simples. Localise contient une suite de n inter-
   valles. Chaque intervalle contient exactement un zéro réel de p.
   VAR
      sp: SuitePol;                 Pour ranger la suite de Sturm.
      m: integer;                   Indice du dernier élément de la suite de Sturm.
      EnAttente: integer;
      IntervalleEnAttente: SuiteIntervalles;
      VariationsEnAttente: SuiteCouplesEntier;
      E: real;
      a, b, c: real;                Des extrémités d'intervalles
      Va, Vb, Vc: integer;          et les variations correspondantes.
   PROCEDURE MettreEnAttente (u, v: real; Vu, Vv: integer);
   PROCEDURE Retirer (VAR u, v: real; VAR Vu, Vv: integer);
   PROCEDURE RangerIntervalleTrouve (u, v: real);
   PROCEDURE Repartir (a, b: real; Va, Vb: integer);
   BEGIN
      SuiteDeSturm(p, sp, m);
      b := (1 + epsilon) * MajorationDesZeros(p);
      a := -b;                                   Intervalle initial.
      Va := Variation(m, sp, a);
      Vb := Variation(m, sp, b);
      n := 0;                                    Au début, aucun intervalle
      EnAttente := 0;                            trouvé et aucun en attente.
      IF Va - Vb > 0 THEN BEGIN                  S'il y a au moins un zéro
         Repartir(a, b, Va, Vb);                 dans [a, b], on le traite.
         WHILE EnAttente > 0 DO BEGIN            Tant qu'il reste un intervalle
            Retirer(a, b, Va, Vb);               en attente, on en prend un,
            c := (a + b) / 2;                    on le coupe au milieu.
            IF EstNul(Valeur(p, c)) THEN BEGIN   Si c'est un zéro,
               RangerIntervalleTrouve(c, c);     on le range soigneusement,
               E := Ecart(p, c);                 puis on calcule la distance qui le
               sépare des zéros les plus proches, et on recommence
```

```
               Repartir(a, c - E, Va, Variation(m, sp, c - E));   sur [a, c − E]
               Repartir(c + E, b, Variation(m, sp, c + E), Vb);   et [c + E, b].
            END
            ELSE BEGIN                                    Si ce n'est pas un zéro,
               Vc := Variation(m, sp, c);                 on calcule sa variation
               Repartir(a, c, Va, Vc);                    et on recommence
               Repartir(c, b, Vc, Vb);                    des deux côtés.
            END
         END { du while }
      END
   END; { de "LocalisationIntervalles" }
```

Il reste à expliquer la présence du facteur $1+\varepsilon$ dans le calcul des extrémités du premier intervalle. Les inégalités dans les majorations des modules des zéros étant larges, les bornes peuvent être atteintes; or, pour pouvoir interpréter correctement les variations calculées, elles doivent l'être en des points qui ne sont pas des zéros. C'est pourquoi on élargit l'intervalle.

Voici quelques exemples de déroulement de cette procédure, avec quelques impressions intermédiaires :

```
Voici le polynôme lu
 X^3 - X
Intervalle de départ [a,b]= [  -2.002,   2.002]  Variations :3  0
Nombre de zéros réels :  3
Mis en attente   :  [  -2.002   2.002]   Variations :3  0
Retiré d'attente :  [  -2.002   2.002]
Trouvé :  [   0.000   0.000]
Trouvé :  [  -2.002  -0.500]
Trouvé :  [   0.500   2.002]
Un zéro trouvé :    0.000
Un zéro dans l'intervalle ]-2.002,-0.500[
Un zéro dans l'intervalle ] 0.500, 2.002[

Voici le polynôme lu
 X^4 - 10.000 X^3 + 35.000 X^2 - 50.000 X + 24.000
Intervalle de départ [a,b]= [ -20.020,  20.020]  Variations :4  0
Nombre de zéros réels :  4
Mis en attente   :  [ -20.020  20.020]   Variations :4  0
Retiré d'attente :  [ -20.020  20.020]
Mis en attente   :  [   0.000  20.020]   Variations :4  0
Retiré d'attente :  [   0.000  20.020]
Mis en attente   :  [   0.000  10.010]   Variations :4  0
Retiré d'attente :  [   0.000  10.010]
Mis en attente   :  [   0.000   5.005]   Variations :4  0
Retiré d'attente :  [   0.000   5.005]
Mis en attente   :  [   0.000   2.503]   Variations :4  2
Mis en attente   :  [   2.503   5.005]   Variations :2  0
```

```
Retiré d'attente :  [   2.503   5.005]
Trouvé :  [   2.503   3.754]
Trouvé :  [   3.754   5.005]
Retiré d'attente :  [   0.000   2.503]
Trouvé :  [   0.000   1.251]
Trouvé :  [   1.251   2.503]
Un zéro dans l'intervalle ] 2.503, 3.754[
Un zéro dans l'intervalle ] 3.754, 5.005[
Un zéro dans l'intervalle ] 0.000, 1.251[
Un zéro dans l'intervalle ] 1.251, 2.503[
```

Bien entendu, il est facile, à partir des intervalles ainsi obtenus, de calculer les zéros réels, au moyen par exemple d'une recherche dichotomique.

7.2 Polynômes symétriques

7.2.1 Enoncé : polynômes symétriques

On rappelle que tout polynôme $f \in \mathbb{Z}[X_1, \ldots, X_n]$ peut s'écrire de façon unique sous la forme

$$f = \sum_{(r_1, r_2, \ldots, r_n) \in \mathbb{N}^n} a_{r_1, r_2, \ldots, r_n} X_1^{r_1} X_2^{r_2} \cdots X_n^{r_n}$$

où les $a_{r_1, r_2, \ldots, r_n}$ sont des entiers relatifs tels que l'ensemble $\{(r_1, r_2, \ldots, r_n) \in \mathbb{N}^n \mid a_{r_1, r_2, \ldots, r_n} \neq 0\}$ soit fini. Le *degré* de f, noté $\deg(f)$, est le plus grand entier d tel qu'il existe des entiers $r_1, \ldots, r_n$ vérifiant $a_{r_1, \ldots, r_n} \neq 0$ et $r_1 + \cdots + r_n = d$. Par convention, le degré du polynôme nul est $-\infty$.

On dit qu'un polynôme $f \in \mathbb{Z}[X_1, \ldots, X_n]$ est *symétrique* si l'on a

$$f(X_{\sigma(1)}, \ldots, X_{\sigma(n)}) = f(X_1, \ldots, X_n)$$

pour toute permutation σ de l'ensemble $\{1, \ldots, n\}$.

On pourra représenter un monôme $aX_1^{r_1} \ldots X_n^{r_n}$, avec $a \neq 0$, par un tableau contenant les valeurs $a, r_1, \ldots, r_n$, et un polynôme par un tableau de ses monômes. On supposera que $n \leq 4$ et que tout polynôme est somme d'au plus 100 monômes.

1.– Ecrire des procédures d'addition, de multiplication par une constante et de test d'égalité des polynômes de $\mathbb{Z}[X_1, \ldots, X_n]$. Exemple numérique : on prend $n = 3$, $p_1 = X_1^2X_2^2 - X_1^2X_3^2 + X_2^3X_3$ et $p_2 = 2X_1^2X_3^2 + X_2^2X_3^2 - X_2^3X_3$. Afficher p_1, p_2, $p_1 + p_2$ et $-5p_1$.

2.– Ecrire une procédure qui prend en argument un polynôme $f \in \mathbb{Z}[X_1, \ldots, X_n]$ et qui teste si f est symétrique (on admettra sans démonstration que si $n \geq 2$, le groupe des permutations de l'ensemble $\{1, \ldots, n\}$ est engendré par une permutation circulaire et par une transposition).

Exemple numérique : tester si p_1, p_2 et $p_1 + p_2$ sont symétriques.

On appelle *fonctions symétriques élémentaires*de $X_1, \ldots, X_n$ les polynômes

$$s_k(X_1, \ldots, X_n) = \sum_{1 \leq i_1 < i_2 \cdots < i_k \leq n} X_{i_1} X_{i_2} \cdots X_{i_k}$$

où $1 \leq k \leq n$. On se propose de démontrer que pour tout polynôme symétrique $f(X_1, \ldots, X_n)$ il existe un polynôme p_f et un seul tel que

$$f(X_1, \ldots, X_n) = p_f(s_1(X_1, \ldots, X_n), \; \ldots \; , s_n(X_1, \ldots, X_n)) \qquad (*)$$

3.– On démontre d'abord l'existence de p_f par récurrence sur l'entier $c(f) = n + \deg(f)$. On pose, pour tout polynôme $f(X_1, \ldots, X_n)$,

$$f^0(X_1, \ldots, X_{n-1}) = f(X_1, \ldots, X_{n-1}, 0)$$

On suppose le résultat démontré pour les polynômes symétriques $f'(X_1, \ldots, X_{n'})$ tels que $c(f') < c(f)$.

a) On suppose $n \geq 2$. Démontrer l'existence du polynôme p_{f^0}, puis démontrer que le polynôme

$$g(X_1, \ldots, X_n) = f(X_1, \ldots, X_n) - p_{f^0}(s_1(X_1, \ldots, X_n), \; \ldots \; , s_{n-1}(X_1, \ldots, X_n))$$

peut s'écrire sous la forme

$$g(X_1, \ldots, X_n) = X_1 \cdots X_n h(X_1, \ldots, X_n)$$

où h est un polynôme symétrique.

b) Compléter la démonstration de l'existence de p_f.

4.– Démontrer que pour tout polynôme symétrique $f(X_1, \ldots, X_n)$ il existe un *unique* polynôme p_f vérifiant $(*)$.

5.– On suppose, dans cette question uniquement, que $n = 2$.

a) Démontrer, en utilisant les questions précédentes, que si $f(X_1, X_2)$ est un polynôme symétrique, on a

$$p_f(Y_1, Y_2) = \sum_{k \geq 0} Y_2^k f_k^0(Y_1) \qquad (**)$$

où les polynômes f_k sont définis par les relations de récurrence

$$f_0 = f \quad \text{et} \quad f_{k+1} = \frac{f_k - f_k^0(X_1 + X_2)}{X_1 X_2}$$

b) Ecrire une procédure qui prend pour argument un polynôme symétrique $f(X_1, X_2)$ et qui calcule le polynôme p_f donné par la formule $(*)$.

Exemple numérique : Calculer p_f lorsque $f = X_1^{10} + X_2^{10}$.

6.– On pose $\sigma_0(X_1, \ldots, X_n) = n$ et, pour $k > 0$, $\sigma_k(X_1, \ldots, X_n) = X_1^k + \cdots + X_n^k$.

a) Démontrer que les polynômes σ_k vérifient les relations suivantes :

$$\begin{aligned} \sigma_k - s_1\sigma_{k-1} + s_2\sigma_{k-2} + \cdots + (-1)^{k-1}s_{k-1}\sigma_1 + (-1)^k k s_k &= 0 \text{ pour } 0 < k \leq n \\ \sigma_k - s_1\sigma_{k-1} + \cdots + (-1)^n s_n\sigma_{k-n} &= 0 \text{ pour } 0 < n \leq k \end{aligned}$$

b) Démontrer que

$$\sigma_k = \sum_{r_1+2r_2+\cdots+nr_n=k} a_{r_1,r_2,\ldots,r_n} s_1^{r_1} s_2^{r_2} \cdots s_n^{r_n} \qquad (***)$$

où

$$a_{r_1,r_2,\ldots,r_n} = (-1)^{r_2+r_4+\cdots+r_{2[n/2]}} \frac{(r_1 + 2r_2 + \cdots + nr_n)(r_1 + \cdots + r_n - 1)!}{r_1! \ \cdots \ r_n!}$$

c) Ecrire une procédure qui prend en argument des entiers k et n et qui calcule le polynôme p_{σ_k} à l'aide de la formule $(***)$. Application numérique : $n = 3$, $k = 10$.

7.– Démontrer que si $2n$ nombres entiers $x_1, \ldots, x_n, y_1, \ldots, y_n$ vérifient

$$\sum_{1\leq i\leq n} x_i^k = \sum_{1\leq i\leq n} y_i^k \text{ pour } 1 \leq k \leq n$$

alors il existe une permutation ρ de $\{1, \ldots, n\}$ telle que $x_{\rho(i)} = y_i$ pour $1 \leq k \leq n$.

7.2.2 Solution : polynômes symétriques

Soient X_1, X_2,..., X_n des indéterminées sur un anneau commutatif unitaire K. Un polynôme $f \in K[X_1, \ldots, X_n]$ peut s'écrire de façon unique sous la forme

$$f = \sum_{(r_1,r_2,\ldots,r_n)\in K^n} a_{r_1,r_2,\ldots,r_n} X_1^{r_1} X_2^{r_2} \cdots X_n^{r_n}$$

où les $a_{r_1,r_2,\ldots,r_n}$ sont des éléments de K presque tous nuls. Les *monômes de* f sont les monômes $a_{r_1,r_2,\ldots,r_n} X_1^{r_1} X_2^{r_2} \cdots X_n^{r_n}$ tels que $a_{r_1,r_2,\ldots,r_n} \neq 0$.
On dit qu'un polynôme $f \in K[X_1, \ldots, X_n]$ est *symétrique* si l'on a

$$f(X_{\sigma(1)}, \ldots, X_{\sigma(n)}) = f(X_1, \ldots, X_n)$$

pour toute permutation σ de l'ensemble $\{1, \ldots, n\}$.
On appelle *fonctions symétriques élémentaires* de $X_1, \ldots, X_n$ les polynômes

$$s_k(X_1, \ldots, X_n) = \sum_{1\leq i_1<i_2<\cdots<i_k\leq n} X_{i_1} X_{i_2} \cdots X_{i_k}$$

où $1 \leq k \leq n$. En particulier,

$$\begin{aligned} s_1 &= X_1 + X_2 + \ldots + X_n \\ s_2 &= X_1X_2 + X_2X_3 + \ldots + X_{n-1}X_n \\ &\vdots \\ s_n &= X_1X_2 \ldots X_n \end{aligned}$$

Soit $m = aX_1^{r_1}X_2^{r_2} \ldots X_n^{r_n}$ un monôme non nul de $K[X_1, \ldots, X_n]$. On appelle respectivement *degré* et *poids* de m les entiers

$$\deg(m) = r_1 + r_2 + \cdots + r_n \quad \text{et} \quad \text{poids}(m) = r_1 + 2r_2 + \cdots + nr_n$$

Le degré (resp. le poids) d'un polynôme non nul est égal au maximum des degrés (resp. poids) de ses monômes. Par convention, le degré et le poids du polynôme nul sont égaux à $-\infty$.

Notre premier objectif est de démontrer le résultat suivant.

Théorème 7.2.1. *Pour tout polynôme symétrique f de l'anneau $K[X_1, \ldots, X_n]$, il existe un polynôme p_f de l'anneau $K[X_1, \ldots, X_n]$ et un seul tel que*

$$f(X_1, \ldots, X_n) = p_f(s_1(X_1, \ldots, X_n), \ \ldots \ , s_n(X_1, \ldots, X_n))$$

De plus, le poids de p_f est égal au degré de f.

Preuve. Commençons par démontrer l'unicité de p_f. Si p_f et q_f sont deux solutions du problème, leur différence $u = p_f - q_f$ vérifie la formule

$$u(s_1, \ \ldots \ , s_n) = 0$$

Le résultat découle donc de l'énoncé suivant.

Proposition 7.2.2. *Les fonctions symétriques élémentaires sont algébriquement indépendantes sur K ; autrement dit, un polynôme u de l'anneau $K[X_1, \ldots, X_n]$ est nul si et seulement si $u(s_1, \ \ldots \ , s_n) = 0$.*

Preuve. Supposons qu'il existe un entier n et un polynôme non nul u vérifiant $u(s_1, \ \ldots \ , s_n) = 0$. Choisissons u avec $\deg(u) + n$ minimal et posons

$$u = u_0 + u_1X_n + \cdots + u_dX_n^d$$

avec, pour $1 \leq i \leq d$, $u_i \in K[X_1, \ldots, X_{n-1}]$. On a nécessairement $n \geq 2$. Si $u_0 = 0$, u peut s'écrire sous la forme $u = X_ng(X_1, \ldots, X_n)$, d'où, en substituant,

$$u(s_1, \ldots, s_n) = s_ng(s_1, \ldots, s_n) = 0$$

et donc $g(s_1 \ldots s_n) = 0$, ce qui contredit la minimalité de $\deg(u) + n$.

Donc $u_0 \neq 0$. Maintenant, on a par hypothèse

$$u(s_1(X_1, \ldots, X_n), \ldots, s_n(X_1, \ldots, X_n)) = 0$$

Il en résulte, en prenant $X_n = 0$

$$u_0(s_1(X_1, \ldots, X_{n-1}, 0), \ldots, s_{n-1}(X_1, \ldots, X_{n-1}, 0)) = 0$$

et donc

$$u_0(\hat{s}_1, \ldots, \hat{s}_{n-1}) = 0$$

où les $\hat{s}_i$ sont les fonctions élémentaires de $X_1, \ldots X_{n-1}$, ce qui contredit la minimalité de $\deg(u) + n$. Par conséquent, la condition $u(s_1, \ \ldots \ , s_n) = 0$ entraîne $u = 0$. ∎

Démontrons à présent l'existence d'un polynôme p_f de poids égal à $\deg(f)$ vérifiant l'égalité de l'énoncé. On raisonne par récurrence sur l'entier $c(f) = n + \deg(f)$, en observant que, si $n = 1$, il suffit de prendre $p_f = f$. Supposons donc $n > 1$ et associons à tout polynôme symétrique f de $K[X_1, \ldots, X_n]$ le polynôme $f^0 \in K[X_1, \ldots, X_{n-1}]$ défini par

$$f^0(X_1, \ldots, X_{n-1}) = f(X_1, \ldots, X_{n-1}, 0)$$

Ce polynôme est symétrique par construction et vérifie $c(f^0) \leq c(f) - 1$. En lui appliquant l'hypothèse de récurrence, on obtient un polynôme p_{f^0} de poids égal à $\deg(f^0)$. Posons

$$g(X_1, \ldots, X_n) = f(X_1, \ldots, X_n) - p_{f^0}(s_1(X_1, \ldots, X_n), \ \ldots \ , s_{n-1}(X_1, \ldots, X_n))$$

Le polynôme g est symétrique et s'annule pour $X_n = 0$. Donc, si

$$g = \sum_{(r_1, r_2, \ldots, r_n) \in K^n} a_{r_1, r_2, \ldots, r_n} X_1^{r_1} X_2^{r_2} \cdots X_n^{r_n}$$

on a $a_{r_1, r_2, \ldots, r_n} = 0$ si $r_n = 0$ et, par symétrie, si $r_1 = 0$, $r_2 = 0 \ \ldots$ ou $r_{n-1} = 0$. Par conséquent g s'écrit sous la forme

$$g(X_1, \ldots, X_n) = X_1 \cdots X_n h(X_1, \ldots, X_n)$$

où h est un polynôme symétrique. Puisque $\deg(g) \leq \deg(f)$ et $\deg(h) = \deg(g) - n$, l'hypothèse de récurrence s'applique à h. Donc

$$h(X_1, \ldots, X_n) = p_h(s_1(X_1, \ldots, X_n), \ \ldots \ , s_n(X_1, \ldots, X_n))$$

et

$$f = s_n p_h(s_1, \ \ldots \ , s_n) + p_{f^0}(s_1, \ \ldots \ , s_{n-1})$$

d'où finalement

$$p_f = X_n p_h + p_{f^0}$$

On a de plus

$$\begin{aligned}\text{poids}(p_f) &\leq \max(\text{poids}(X_n p_h), \text{poids}(p_{f^0})) \leq \max(n + \text{poids}(p_h), \text{poids}(p_{f^0}))\\ &\leq \max(n + \deg(h), \deg(f^0)) \leq \max(\deg(g), \deg(f^0)) \leq \deg(f)\end{aligned}$$

Mais d'autre part, la formule

$$f(X_1, \ldots, X_n) = p_f(s_1(X_1, \ldots, X_n),\ \ldots\ , s_n(X_1, \ldots, X_n))$$

entraîne $\deg(f) \leq \text{poids}(p_f)$ et donc finalement $\deg(f) = \text{poids}(p_f)$, ce qui conclut la récurrence et la démonstration du théorème. ∎

Dans le cas de deux indéterminées, il est facile de calculer p_f à partir de f. La démonstration ci-dessus montre que si f est un polynôme symétrique de $K[X_1, X_2]$, il existe un polynôme symétrique h de $K[X_1, X_2]$ tel que $X_1X_2h = f - p_{f^0}(X_1 + X_2)$. Mais puisque p_{f^0} est un polynôme à une indéterminée, il est symétriqueet $p_{f^0} = f^0$. On a donc

$$X_1X_2h = f - f^0(X_1 + X_2)$$

et on peut définir une suite de polynômes symétriques $(f_k)_{k\geq 0}$ de $K[X_1, X_2]$ en posant

$$f_0 = f \quad \text{et} \quad f_{k+1} = \frac{f_k - f_k^0(X_1 + X_2)}{X_1X_2}$$

On en déduit

$$f_k = s_2 f_{k+1} + f_k^0(s_1)$$

et donc, par récurrence

$$\begin{aligned}f_0(X_1, X_2) &= s_2 f_1 + f_0^0(s_1)\\ &= s_2(s_2 f_2 + f_1^0(s_1)) + f_0^0(s_1) = s_2^2 f_2 + s_2 f_1^0(s_1) + f_0^0(s_1)\\ &\ \vdots\\ &= s_2^{r+1} f_{r+1} + \sum_{0\leq k\leq r} s_2^k f_k^0(s_1)\end{aligned}$$

Or comme le degré des f_n est décroissant, on a $f_{r+1} = 0$ pour un r assez grand. Par conséquent, p_f est donné par la formule

$$p_f(X_1, X_2) = \sum_{k\geq 0} X_2^k f_k^0(X_1)$$

On appelle *sommes de Newton* les polynômes de $K[X_1, \ldots, X_n]$ définis par les formules

$$\sigma_0(X_1, \ldots, X_n) = n \text{ et, pour } k > 0,\ \sigma_k(X_1, \ldots, X_n) = X_1^k + X_2^k + \cdots + X_n^k$$

Comme ces polynômes sont symétriques, on peut les exprimer à l'aide des fonctions symétriques élémentaires. Cette expression peut être obtenue à l'aide de formules de récurrence.

Proposition 7.2.3. *Les sommes de Newton vérifient les relations suivantes :*

$$\begin{aligned}\sigma_k - s_1\sigma_{k-1} + s_2\sigma_{k-2} + \cdots + (-1)^{k-1}s_{k-1}\sigma_1 + (-1)^k k s_k &= 0 \text{ pour } 0 < k \le n\\ \sigma_k - s_1\sigma_{k-1} + \cdots + (-1)^n s_n\sigma_{k-n} &= 0 \text{ pour } 0 < n \le k\end{aligned}$$

Preuve. Nous donnerons deux démonstrations de ce résultat.

Première démonstration. Soit $A = K[X_1, \ldots, X_n]$ et soit T une nouvelle indéterminée. On travaille dans l'anneau $A[[T]]$ des séries à coefficients dans A. Considérons la série

$$s = \prod_{1\le i\le n}(1 + X_iT) = 1 + s_1T + \cdots + s_nT^n$$

Puisque le terme constant de s est égal à 1, s est inversible dans $A[[T]]$ (une série à coefficients dans un anneau A est inversible si et seulement si son terme constant est inversible dans A). Pour la même raison, chaque série $1 + X_iT$ est inversible. On en déduit la suite d'égalités :

$$\begin{aligned}\frac{s'}{s} &= \sum_{1\le i\le n}\frac{X_i}{1+X_iT} = \sum_{1\le i\le n} X_i\Big(\sum_{k=0}^{\infty}(-X_iT)^k\Big)\\ &= \sum_{1\le i\le n}\sum_{k=0}^{\infty}(-1)^k X_i^{k+1}T^k\\ &= \sum_{k=0}^{\infty}(-1)^k\sigma_{k+1}T^k\\ &= \sum_{k=1}^{\infty}(-1)^{k-1}\sigma_kT^{k-1}\end{aligned}$$

On obtient finalement, puisque $s' = s_1 + 2s_2T + \cdots + ns_nT^{n-1}$,

$$s_1 + 2s_2T + \cdots + ns_nT^{n-1} = (1 + s_1T + \cdots + s_nT^n)\Big(\sum_{k=1}^{\infty}(-1)^{k-1}\sigma_kT^{k-1}\Big)$$

d'où les formules cherchées par identification des deux membres.

Deuxième démonstration. Partons de l'identité

$$\sum_{\substack{i_2<i_3<\cdots<i_q\\ i_1\notin\{i_2,\ldots,i_q\}}} X_{i_1}^r X_{i_2}\cdots X_{i_q} = \begin{cases} s_q\sigma_{r-1} - \displaystyle\sum_{\substack{i_2<i_3<\cdots<i_{q+1}\\ i_1\notin\{i_2,\ldots,i_{q+1}\}}} X_{i_1}^{r-1}X_{i_2}\cdots X_{i_{q+1}} & \text{si } q<n \text{ et } r>1\\ s_q\sigma_{r-1} & \text{si } q=n \text{ et } r\ge 1\\ qs_q & \text{si } r=1\end{cases}$$

On a successivement

$$\sigma_k = s_1\sigma_{k-1} - \sum_{i_1\neq i_2} X_{i_1}^{k-1}X_{i_2} \qquad (q=1,\ r=k)$$
$$\sum_{i_1\neq i_2} X_{i_1}^{k-1}X_{i_2} = s_2\sigma_{k-2} - \sum_{\substack{i_2<i_3\\ i_1\notin\{i_2,i_3\}}} X_{i_1}^{k-2}X_{i_2}X_{i_3} \qquad (q=2,\ r=k-1)$$
$$\sum_{\substack{i_2<i_3\\ i_1\notin\{i_2,i_3\}}} X_{i_1}^{k-2}X_{i_2}X_{i_3} = s_3\sigma_{k-3} - \sum_{\substack{i_2<i_3<i_4\\ i_1\notin\{i_2,i_3,i_4\}}} X_{i_1}^{k-2}X_{i_2}X_{i_3}X_{i_4} \quad (q=3,\ r=k-2)$$
$$\vdots$$

Si $k<n$, la dernière égalité est obtenue pour $q=k$ et $r=1$. Elle s'écrit

$$\sum_{\substack{i_2<i_3<\cdots<i_k\\ i_1\notin\{i_2,\ldots,i_k\}}} X_{i_1}X_{i_2}\cdots X_{i_k} = ks_k$$

Si $k>n$, la dernière égalité est obtenue pour $q=n$ et $r=k-n+1$. Elle s'écrit

$$\sum_{\substack{i_2<i_3<\cdots<i_k\\ i_1\notin\{i_2,\ldots,i_k\}}} X_{i_1}^{k-n+1}X_{i_2}\cdots X_{i_n} = s_n\sigma_{k-n}$$

Les formules cherchées s'en déduisent facilement. ■

On peut préciser la proposition 7.2.3 en donnant explicitement l'expression des sommes de Newton.

Proposition 7.2.4. *On a les formules :*

$$\sigma_k = \sum_{r_1+2r_2+\cdots+nr_n=k} a_{r_1,r_2,\ldots,r_n} s_1^{r_1}s_2^{r_2}\cdots s_n^{r_n}$$

où

$$a_{r_1,r_2,\ldots,r_n} = (-1)^{r_2+r_4+\cdots+r_{2[n/2]}}\frac{(r_1+2r_2+\cdots+nr_n)(r_1+\cdots+r_n-1)!}{r_1!\ \cdots\ r_n!}$$

Comme nous n'avons fait aucune hypothèse sur la caractéristique de l'anneau K, il peut sembler curieux de voir apparaître des divisions dans l'énoncé ci-dessus. En réalité les coefficients $a_{r_1,r_2,\ldots,r_n}$ sont des *entiers relatifs* et non des éléments de K. La première ligne de l'énoncé utilise donc la convention habituelle qui consiste à noter, pour tout entier naturel n, nx la somme de n éléments égaux à x et $-nx$ la somme de n éléments égaux à $-x$.

Preuve. Posons

$$\sigma_k = \sum_{(r_1,\ldots,r_n)\in\mathbb{N}^n} c_k(r_1,,\ \ldots\ ,r_n)s_1^{r_1}s_2^{r_2}\cdots s_n^{r_n}$$

Il s'agit de prouver la formule

$$c_k(r_1,\dots,r_n) = \begin{cases} a_{r_1,\dots,r_n} & \text{si } r_1+2r_2+\cdots+nr_n = k \\ 0 & \text{sinon} \end{cases}$$

par récurrence sur k. C'est évident pour $k=1$. Supposons la formule vraie jusqu'à $k-1$. Les formules de la proposition 7.2.3 montrent immédiatement que $c_k(r_1,\dots,r_n)$ est nul si $r_1+2r_2+\cdots+nr_n \neq k$. On suppose donc désormais $r_1+2r_2+\cdots+nr_n = k$. Pour faciliter l'écriture des récurrences, il est commode de poser, pour tout n-uplet d'entiers $r_1,\dots,r_n$ tel que l'un des r_i soit négatif, $a_{r_1,\dots,r_n} = 0$. Si $k < n$, la proposition 7.2.3 donne

$$\sigma_k = s_1\sigma_{k-1} - s_2\sigma_{k-2} + \cdots + (-1)^k s_{k-1}\sigma_1 + (-1)^{k+1} k s_k \tag{2.1}$$

ce qui permet de calculer $c_{r_1,\dots,r_n}$. Tout d'abord, si $r_1 = 0$, $r_2 = 0$, …, $r_{k-1} = 0$, $r_k = 1$, $r_{k+1} = 0$, …, $r_n = 0$, on a

$$c_k(r_1,\dots,r_n) = (-1)^{k+1}k$$

ce qui démontre le résultat dans ce cas, puisque

$$(-1)^{r_2+r_4+\cdots+r_{2[n/2]}} = \begin{cases} -1 & \text{si } k \text{ est pair} \\ 1 & \text{si } k \text{ est impair} \end{cases}$$

Dans les autres cas (toujours pour $k < n$), les équations (2.1) donnent

$$\begin{aligned} c_k(r_1,\dots,r_n) &= a_{r_1-1,r_2,\dots,r_n} - a_{r_1,r_2-1,\dots,r_n} + \cdots + (-1)^k a_{r_1,r_2,\dots,r_{k-1}-1,r_k,\dots,r_n} \\ &= \frac{(r_1+\cdots+r_n-2)!}{r_1!\ \cdots\ r_n!}\,(r_1(k-1)+r_2(k-2)+\ \cdots\ +r_{k-1}) \end{aligned}$$

et, en observant que les conditions $r_1+2r_2+\cdots+nr_n = k$ et $k \leq n$ imposent $r_k = r_{k+1} = \ldots = r_n = 0$, on trouve

$$\begin{aligned} &(r_1(k-1)+r_2(k-2)+\cdots+r_{k-1}(k-(k-1)) \\ &\qquad = k(r_1+\cdots+r_n) - (r_1+2r_2+\cdots+nr_n) = k(r_1+\cdots+r_n-1) \end{aligned}$$

ce qui donne bien $c_k(r_1,\dots,r_n) = a_{r_1,\dots,r_n}$.
Dans le cas où $k \geq n$, les équations (2.1) donnent

$$\begin{aligned} c_k(r_1,\dots,r_n) &= a_{r_1-1,r_2,\dots,r_n} - a_{r_1,r_2-1,\dots,r_n} + \cdots + (-1)^{n+1} a_{r_1,r_2,\dots,r_n-1} \\ &= \frac{(r_1+\cdots+r_n-2)!}{r_1!\ \cdots\ r_n!}\,(r_1(k-1)+r_2(k-2)+\cdots+r_n(k-n) \\ &= \frac{(r_1+\cdots+r_n-2)!}{r_1!\ \cdots\ r_n!}\,(k(r_1+\cdots+r_n)-(r_1+2r_2+\cdots+nr_n)) \\ &= \frac{(r_1+\cdots+r_n-2)!}{r_1!\ \cdots\ r_n!}\,k(r_1+\cdots+r_n-1) \\ &= a_{r_1,r_2,\dots,r_n} \end{aligned}$$

■

Voici une application classique des formules de Newton.

Proposition 7.2.5. *Soit K un anneau intègre de caractéristique* 0 *et soient* $(x_i)_{1\le i\le n}$ *et* $(y_i)_{1\le i\le n}$ *deux familles de* n *éléments de* K *tels que*

$$\sum_{1\le i\le n} x_i^k = \sum_{1\le i\le n} y_i^k \text{ pour } 1 \le k \le n$$

Il existe alors une permutation ρ *de* $\{1,\ldots,n\}$ *telle que* $x_{\rho(i)} = y_i$ *pour* $1 \le k \le n$.

Preuve. On a par hypothèse $\sigma_k(x_1,\ldots,x_n) = \sigma_k(y_1, \ldots ,y_n)$ pour $1 \le k \le n$. Montrons par récurrence sur k que $s_k(x_1,\ldots,x_n) = s_k(y_1, \ldots ,y_n)$ pour tout $k \le n$. Le résultat est clair pour $k = 1$, puisque $\sigma_1 = s_1$. Si le résultat est vrai jusqu'au rang $k-1 < n$, les équations 2.1 donnent

$$ks_k(x_1,\ldots,x_n) = ks_k(y_1,\ldots,y_n)$$

Comme l'anneau K est intègre et de caractéristique nulle, cela conclut la récurrence. Il en résulte que $x_1,\ldots,x_n$, (respectivement $y_1,\ldots,y_n$) sont les n racines de l'équation

$$X^n - s_1X^{n-1} + s_2X^{n-2} + \cdots + (-1)^n s_n = 0$$

Par conséquent, il existe une permutation ρ de $\{1,\ldots,n\}$ telle que $x_{\rho(i)} = y_i$. ∎

Notons que les deux hypothèses faites sur K sont nécessaires. En effet, dans $\mathbb{Z}/2\mathbb{Z}$ on a $\sigma_1(0,0) = \sigma_1(1,1) = 0$ et $\sigma_2(0,0) = \sigma_2(1,1) = 0$ et dans l'anneau produit $\mathbb{Z}\times\mathbb{Z}$, on a $\sigma_k((1,1),(0,0)) = (1,1) = \sigma_k((1,0),(0,1))$ pour tout $k > 0$.

7.2.3 Programme : polynômes symétriques

Appelons *multidegré* d'un monôme $aX_1^{r_1}\ldots X_n^{r_n}$ le n-uplet d'entiers $r_1,\ldots,r_n$. Comme le suggère l'énoncé du problème, on peut représenter un monôme $aX_1^{r_1}\ldots X_n^{r_n}$, avec $a \neq 0$, par un tableau contenant les valeurs $a, r_1,\ldots,r_n$, et un polynôme comme un tableau de monômes. Nous appelerons *taille du polynôme* le nombre de monômes figurant dans la table. On ordonnera les monômes suivant l'ordre lexicographique de leurs multidegrés : le monôme $aX_1^{r_1}\ldots X_n^{r_n}$ précède le monôme $a'X_1^{r'_1}\ldots X_n^{r'_n}$ s'il existe un indice k tel que $r_1 = r'_1$, $r_2 = r'_2$, …, $r_{k-1} = r'_{k-1}$ et $r_k < r'_k$.

```
CONST
   DegreMax = 20;
   NbreMaxMonomes = 100;
   NbreMaxIndeterminees = 8;
```

Un monôme $aX^iY^jZ^k$ est représenté par le vecteur (a,i,j,k). Un polynôme p est représenté par un tableau de monômes. La taille du polynôme est donnée par $p[0,0]$ et les monômes sont ordonnés suivant l'ordre lexicographique de leurs multidegrés.

```
TYPE
   monome = ARRAY[0..NbreMaxIndeterminees] OF integer;
   polynome = ARRAY[0..NbreMaxMonomes] OF monome;
```

Pour manipuler facilement la taille d'un polynôme, on utilise les deux procédures suivantes :

```
FUNCTION Taille (p: polynome): integer;
```
Donne le nombre de monômes composant le polynôme.
```
   BEGIN
      Taille := p[0, 0]
   END; { de "Taille" }
PROCEDURE FixerTaille (VAR p: polynome; t: integer);
   BEGIN
      p[0, 0] := t
   END; { de "FixerTaille" }
```

On définit ensuite les polynômes 0 et 1.

```
VAR
   PolynomeNul, PolynomeUnite: polynome;
PROCEDURE InitPolynomes;
```
Définition du polynôme nul et du polynôme unité.
```
   VAR
      i: integer;
   BEGIN
      FixerTaille(PolynomeNul, 0);
      FixerTaille(PolynomeUnite, 1);
      PolynomeUnite[1, 0] := 1;
      FOR i := 1 TO NbreMaxIndeterminees DO
         PolynomeUnite[1, i] := 0;
   END; { de "InitPolynomes" }
```

Comme les monômes sont rangés suivant leur multidegré, il est utile de pouvoir comparer le multidegré de deux monômes.

```
FUNCTION CompareDegreMonome (n: integer; m1, m2: monome): integer;
```
Cette fonction compare les deux monômes m_1 et m_2 dans l'ordre lexicographique des multidegrés. Le résultat est 1 si $m_2 < m_1$, -1 si $m_1 < m_2$, et 0 si m_1 et m_2 ont le même multidegré.
```
   VAR
      i: integer;
   BEGIN
      i := 1;
      WHILE (i <= n) AND (m1[i] = m2[i]) DO   { "and" séquentiel }
         i := i + 1;
      IF i = n + 1 THEN
         CompareDegreMonome := 0
      ELSE
         CompareDegreMonome := Signe(m1[i] - m2[i])
   END; { de "CompareDegreMonome"}
```

Le test d'égalité de deux polynômes peut être réalisé ainsi :

```
FUNCTION SontEgauxPolynomes (n: integer; p1, p2: polynome): boolean;
```
Teste si $p_1 = p_2$.

```
   VAR
      i, t: integer;
      SontEgaux: boolean;
   BEGIN
      IF Taille(p1) <> Taille(p2) THEN
         SontEgauxPolynomes := false
      ELSE BEGIN
         t := Taille(p1);
         i := 0;
         SontEgaux := true;
         WHILE (i <= t) AND SontEgaux DO BEGIN
            SontEgaux := (CompareDegreMonome(n, p1[i], p2[i]) = 0);
            i := i + 1
         END;
         SontEgauxPolynomes := SontEgaux;
      END
   END;  { de "SontEgauxPolynomes" }
```

La procédure qui suit permet d'ajouter un monôme m à un tableau de monômes déjà ordonné. Si aucun monôme du tableau ne possède le même multidegré que m, m est inséré à la place voulue. Si, au contraire, m a le même multidegré qu'un monôme m' du tableau, on substitue à m' le monôme $m + m'$ si celui-ci est non nul. Sinon m' est supprimé du tableau.

```
PROCEDURE MonomePlusPolynome (n: integer; m: monome; VAR q: polynome);
   Ajoute le monôme m au polynôme ordonné q.
   VAR
      i, j, t: integer;
   BEGIN
      t := Taille(q);
      i := 1;
      WHILE (i <= t) AND (CompareDegreMonome(n, q[i], m) < 0) DO
         i := i + 1;
      IF (i <= t) AND (CompareDegreMonome(n, q[i], m) = 0) THEN BEGIN
         Le monôme q_i a même degré que m.
         IF q[i, 0] = -m[0] THEN BEGIN        Le monôme q_i est −m.
            t := t - 1;                       Un monôme de moins ...
            FOR j := i TO t DO                On élimine 0 dans les monômes de q.
               q[j] := q[j + 1];
         END
         ELSE
            q[i, 0] := q[i, 0] + m[0]
      END
      ELSE BEGIN                     m n'apparaît pas dans q.
         FOR j := t DOWNTO i DO      On intercale m dans les monômes de q.
            q[j + 1] := q[j];
         t := t + 1;                 Un monôme de plus ...
         q[i] := m;
```

```
      END;
      FixerTaille(q, t);
   END;  { de "MonomePlusPolynome"}
```

Pour les entrées-sorties, on utilise les procédures suivantes :

```
PROCEDURE EntrerPolynome (n: integer; VAR p: polynome; titre: texte);
```

Affichage du titre, puis lecture du polynôme p à n indéterminées.

```
   i, t: integer;
   m: monome;
   PROCEDURE LireMonome (n: integer; VAR m: monome);
      VAR
         j: integer;
      BEGIN
         write('Coefficient du monome : ');
         readln(m[0]);
         FOR j := 1 TO n DO BEGIN
            write('Degr de X', j : 1, ' : ');
            readln(m[j]);
         END
      END;
   BEGIN { de "EntrerPolynome" }
      writeln(titre);
      p := PolynomeNul;
      write('Combien de monomes ? ');
      readln(t);
      writeln('Donner la suite des monomes.');
      FOR i := 1 TO t DO BEGIN
         LireMonome(n, m);
         MonomePlusPolynome(n, m, p)
      END
   END; { de "EntrerPolynome" }
PROCEDURE EcrireMonome (n: integer; m: monome; indeterminee: char);
```

Affichage du monôme $m(x)$, où x est l'indéterminée.

```
   VAR
      j: integer;
      EstUnOuMoinsUn: boolean;
   BEGIN
      EstUnOuMoinsUn := abs(m[0]) = 1;
      IF NOT EstUnOuMoinsUn THEN
         write(abs(m[0]) : 1);
      FOR j := 1 TO n DO BEGIN
         IF m[j] > 0 THEN BEGIN
            write(indeterminee, '_', j : 1);
            EstUnOuMoinsUn := false;
            IF m[j] > 1 THEN
               write('^', m[j] : 1);
         END
```

```
      END;
      IF EstUnOuMoinsUn THEN
         write('1')
   END; { de "EcrireMonome" }
PROCEDURE EcrirePolynome (n: integer; p: polynome; indeterminee: char;
   titre: texte);
```
Affichage du titre, puis du polynôme $p(x)$, où x est l'indéterminée.
```
   VAR
      i, NbreMonomesParLigne: integer;
   BEGIN
      write(titre);
      CASE n OF
         1:
            NbreMonomesParLigne := 8;
         2:
            NbreMonomesParLigne := 5;
         3, 4, 5:
            NbreMonomesParLigne := 4;
         OTHERWISE
            NbreMonomesParLigne := 3;
      END;
      IF Taille(p) = 0 THEN
         writeln('Polynome nul')
      ELSE BEGIN
         IF p[1, 0] < 0 THEN
            write('- ');
         EcrireMonome(n, p[1], indeterminee);
      END;
      FOR i := 2 TO Taille(p) DO BEGIN
         IF p[i, 0] > 0 THEN
            write(' + ')
         ELSE
            write(' - ');
         IF (i MOD NbreMonomesParLigne) = 1 THEN
            writeln;
         EcrireMonome(n, p[i], indeterminee);
      END;
      writeln;
   END;  { de "EcrirePolynome"}
```

L'addition de deux polynômes est maintenant facile à réaliser. Il suffit de mettre «bout à bout» les tableaux représentant les polynômes et de réduire le tableau ainsi obtenu.

```
PROCEDURE PolynomePlusPolynome (n: integer; p1, p2: polynome;
   VAR p: polynome);
```
$p = p_1 + p_2$.
```
   VAR
      i: integer;
```

```
    BEGIN
       p := p1;
       FOR i := 1 TO Taille(p2) DO
          MonomePlusPolynome(n, p2[i], p);
    END;  { de "PolynomePlusPolynome"}
```

Sur l'exemple numérique de l'énoncé, on trouve bien sûr

```
p1 = X_2^3X_3 - X_1^2X_3^2 + X_1^2X_2^2
p2 = X_2^2X_3^2 - X_2^3X_3 + 2X_1^2X_3^2
p1 + p2 = X_2^2X_3^2 + X_1^2X_3^2 + X_1^2X_2^2
```

La multiplication par un entier est encore plus simple.

```
PROCEDURE PolynomeParEntier (VAR p: polynome; entier: integer);
   VAR
      i: integer;
   BEGIN
      IF entier = 0 THEN
         p := PolynomeNul
      ELSE
         FOR i := 1 TO Taille(p) DO
            p[i, 0] := entier * p[i, 0];
   END;  { de "PolynomeParEntier"}
```

La différence de deux polynômes s'obtient en combinant les deux procédures précédentes.

```
PROCEDURE PolynomeMoinsPolynome (n: integer; p1, p2: polynome;
   VAR p: polynome);
   Calcule la différence p1 − p2 dans p.
   BEGIN
      PolynomeParEntier(p2, -1);
      PolynomePlusPolynome(n, p1, p2, p)
   END;  { de "PolynomeMoinsPolynome"}
```

Soit G un ensemble de générateurs du groupe des permutations de l'ensemble $\{1, \ldots, n\}$. Pour tester si un polynôme $f \in \mathbb{Z}[X_1, \ldots, X_n]$ est symétrique, il suffit de vérifier que

$$f(X_{\sigma(1)}, \ldots, X_{\sigma(n)}) = f(X_1, \ldots, X_n)$$

pour toute permutation $\sigma \in G$. Si $n > 1$, on peut prendre $G = \{\sigma, \tau\}$, où σ est la permutation circulaire $(1, 2, \ldots, n)$ et τ la transposition $(1, 2)$.

```
FUNCTION EstSymetrique (n: integer; p: polynome): boolean;
   Teste si p est symétrique.
   VAR
      q: polynome;
      m: monome;
      i, j: integer;
      b: boolean;
```

```
BEGIN
   IF n <= 1 THEN
      b := true
   ELSE BEGIN
      q := PolynomeNul;
      FOR i := 1 TO Taille(p) DO BEGIN   On permute les indéterminées X1 et X2.
         m := p[i];
         m[1] := p[i, 2];
         m[2] := p[i, 1];
         MonomePlusPolynome(n, m, q);
      END;
      b := SontEgauxPolynomes(n, p, q);
      IF b THEN BEGIN
         On fait une permutation circulaire des indéterminées X1, X2, ..., Xn.
         q := PolynomeNul;
         FOR i := 1 TO Taille(p) DO BEGIN
            FOR j := 1 TO n DO m[j] := p[i, (j MOD n) + 1];
            MonomePlusPolynome(n, m, q)
         END;
         b := SontEgauxPolynomes(n, p, q);
      END;
   END;
   EstSymetrique := b;
END;  { de "EstSymetrique"}
```

Sur les exemples de l'énoncé, seul le polynôme $p_1 + p_2$ est symétrique.

Dans le cas $n = 2$, le calcul du polynôme p_f associé à chaque polynôme symétrique f est simplifié et on a

$$p_f = \sum_{k \geq 0} X_2^k f_k^0$$

où les polynômes f_k sont définis par la récurrence

$$f_0 = f \text{ et } f_{k+1} = \frac{f_k - f_k^0(X_1 + X_2)}{X_1 X_2}$$

Une première procédure permet de passer du polynôme $f \in \mathbb{Z}[X_1, X_2]$ au polynôme $f^0 \in \mathbb{Z}[X_1]$ défini par $f^0(X_1) = p(X_1, 0)$.

```
PROCEDURE pZero (p: polynome; VAR p0: polynome);
   Passe de p(X1, X2) à p0(X1) = p(X1, 0).
   VAR
      i: integer;
      m: monome;
   BEGIN
      p0 := PolynomeNul;
      FOR i := 1 TO Taille(p) DO
         IF p[i, 2] = 0 THEN BEGIN
```

```
            m[0] := p[i, 0];
            m[1] := p[i, 1];
            m[2] := 0;
            MonomePlusPolynome(n, m, p0);
         END;
   END;  { de "pZero"}
```

La formule de récurrence peut être implémentée comme suit :

```
PROCEDURE XplusYpuissanceK (k: integer; VAR p: polynome);        p = (X + Y)^k
   VAR
      i: integer;
   BEGIN
      FixerTaille(p, k + 1);
      FOR i := 0 TO k DO BEGIN
         p[i + 1, 0] := Binomial(k, i);
         p[i + 1, 1] := i;
         p[i + 1, 2] := k - i;
      END
   END;  { de "XplusYpuissanceK"}
PROCEDURE Transforme (p: polynome; VAR q: polynome);
   Passe de p(X1, X2) à p(X1 + X2, 0).
   VAR
      i: integer;
      r: polynome;
   BEGIN
      q := PolynomeNul;
      FOR i := 1 TO Taille(p) DO
         IF p[i, 2] = 0 THEN BEGIN
            XplusYpuissanceK(p[i, 1], r);
            PolynomeParEntier(r, p[i, 0]);
            PolynomePlusPolynome(2, q, r, q);
         END;
    END;  { de "Transforme"}
PROCEDURE Recurrence (p: polynome; VAR q: polynome);
   Passe de p(X_1, X_2) à (p(X_1, X_2) - p(X_1 + X_2, 0))/X_1 X_2.
   VAR
      i: integer;
      r: polynome;
   BEGIN
      Transforme(p, r);
      PolynomeMoinsPolynome(2, p, r, q);
       FOR i := 1 TO Taille(q) DO BEGIN
         q[i, 1] := q[i, 1] - 1;
         q[i, 2] := q[i, 2] - 1;
      END;
   END;  { de "Recurrence"}
```

Le calcul de p_f s'obtient maintenant facilement.

```
PROCEDURE pParX2PuissanceK (K: integer; VAR p: polynome);
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO Taille(p) DO
         p[i, 2] := p[i, 2] + k;
   END;  { de "pParX2PuissanceK"}
PROCEDURE Calcul_pf (f: polynome; VAR pf: polynome);
   VAR
      i: integer;
      p, p0: polynome;
   BEGIN
      pf := PolynomeNul;
      p := f;
      i := 0;
      WHILE Taille(p) <> 0 DO BEGIN
         pZero(p, p0);
         pParX2PuissanceK(i, p0);
         PolynomePlusPolynome(2, pf, p0, pf);
         Recurrence(p, p);
         i := i + 1;
      END;
   END;  { de "Calcul_pf"}
```

Exemple numérique. On trouve

$$X_1^{10} + X_2^{10} = -2s_2^5 + 25s_1^2 s_2^4 - 50s_1^4 s_2^3 + 35s_1^6 s_2^2 - 10s_1^8 s_2 + s_1^{10}$$

Les formules de Newton reposent sur le calcul des coefficients

$$a_{r_1,r_2,\ldots,r_n} = (-1)^{r_2+r_4+\cdots+r_{2[n/2]}} \frac{(r_1 + 2r_2 + \cdots + nr_n)(r_1 + \cdots + r_n - 1)!}{r_1! \,\cdots\, r_n!}$$

On commence par calculer le coefficient multinomial

$$\binom{r_1 + r_2 + \cdots + r_n}{r_1, r_2, \,\ldots\,, r_n} = \frac{(r_1 + \cdots + r_n)!}{r_1! r_2! \,\cdots\, r_n!}$$

en observant que

$$\binom{r_1 + r_2 + \cdots + r_n}{r_1, r_2, \,\ldots\,, r_n} = \binom{r_1 + r_2}{r_2}\binom{r_1 + r_2 + r_3}{r_3} \cdots \binom{r_1 + r_2 + \cdots + r_n}{r_n}$$

La procédure correspondante fait appel à la procédure `Binomial` définie dans la bibliothèque générale.

```
FUNCTION Multinomial (r: monome): integer;
```
Donne la valeur du Multinomial $\binom{r_1+r_2+\cdots+r_n}{r_1,r_2,\,\ldots\,,r_n}$.

```
   VAR
      i, SommePartielle, M: integer;
   BEGIN
      M := 1;
      IF n > 1 THEN BEGIN
         SommePartielle := r[1];
         FOR i := 2 TO n DO BEGIN
            SommePartielle := SommePartielle + r[i];
            M := M * Binomial(SommePartielle, r[i]);
         END;
      END;
      Multinomial := M;
   END;  { de "Multinomial"}
```

Le calcul de $a_{r_1,r_2,\ldots,r_n}$ n'offre plus de difficulté.

```
FUNCTION Coefficient (r: monome): integer;
   VAR
      i, SommeRangPair, Somme, Poids, C: integer;
   BEGIN
      Somme := 0;
      Poids := 0;
      SommeRangPair := 0;
      FOR i := 1 TO n DO BEGIN
         Poids := Poids + i * r[i];
         Somme := Somme + r[i]
      END;
      FOR i := 1 TO n DIV 2 DO
         SommeRangPair := SommeRangPair + r[2 * i];
      C := Multinomial(r) * poids;
      C := C DIV Somme;
      IF odd(SommeRangPair) THEN C := -C;
      Coefficient := C;
   END;  { de "Coefficient"}
```

Il reste à engendrer, pour chaque couple d'entiers (k, n), la suite des n-uplets $(r_1, \ldots, r_n)$ tels que $r_1 + 2r_2 + \cdots + nr_n = k$. Ce type de problème est traité en détail au chapitre 8. Nous donnerons donc les procédures sans commentaires supplémentaires.

```
PROCEDURE PremierePartition (VAR a: monome; n, k: integer);
```
Calcule dans r la première suite $r_1, \ldots, r_n$ telle que $r_1 + 2r_2 + \cdots + nr_n = k$.
```
   VAR
      i, s: integer;
   BEGIN
      FOR i := 1 TO n DO
         a[i] := 0;
      a[n] := k DIV n;
      s := k MOD n;
      IF s > 0 THEN
```

```
      a[s] := 1;
   END;  { de "PremierePartition"}
PROCEDURE PartitionSuivante (VAR a: monome; k: integer;
   VAR derniere: boolean);
   Calcule dans r la suite suivante r_1, ... r_n telle que r_1 + 2r_2 + ··· + nr_n = k.
   VAR
      i, s: integer;
   BEGIN
      i := 2;
      WHILE a[i] = 0 DO i := i + 1;
      s := i + a[1];                          Reste à répartir.
      a[1] := 0;
      a[i] := a[i] - 1;
      i := i - 1;
      a[i] := s DIV i;
      IF i > 1 THEN BEGIN
         s := s MOD i;
         IF s > 0 THEN a[s] := 1
      END;
      derniere := a[1] = k;
   END;
```

Enfin, les formules de Newton sont calculées à l'aide de la procédure suivante :

```
PROCEDURE Newton (n, k: integer; VAR p: polynome);
   VAR
      i: integer;
      derniere: boolean;
      a: monome;
   BEGIN
      PremierePartition(a, n, k);
      p[1] := a;
      p[1, 0] := Coefficient(a);
      i := 1;
      IF k > 1 THEN
         REPEAT
            i := i + 1;
            PartitionSuivante(a, k, derniere);
            p[i] := a;
            p[i, 0] := Coefficient(a);
         UNTIL derniere;
      p[0, 0] := i;
   END;  { de "Newton"}
```

Voici quelques exemples d'exécution. On trouve, pour $k = 10$ et $n = 3$,

```
p = 10s_1s_3^3 + 15s_2^2s_3^2 - 60s_1^2s_2s_3^2 + 25s_1^4s_3^2 -
40s_1s_2^3s_3 + 100s_1^3s_2^2s_3 - 60s_1^5s_2s_3 + 10s_1^7s_3 -
2s_2^5 + 25s_1^2s_2^4 - 50s_1^4s_2^3 + 35s_1^6s_2^2 -
```

```
10s_1^8s_2 + s_1^10
```

Pour $n = 7$ et $k = 9$,

```
p = - 9s_2s_7 + 9s_1^2s_7 - 9s_3s_6 +
18s_1s_2s_6 - 9s_1^3s_6 - 9s_4s_5 +
18s_1s_3s_5 + 9s_2^2s_5 - 27s_1^2s_2s_5 +
9s_1^4s_5 + 9s_1s_4^2 + 18s_2s_3s_4 -
27s_1^2s_3s_4 - 27s_1s_2^2s_4 + 36s_1^3s_2s_4 -
9s_1^5s_4 + 3s_3^3 - 27s_1s_2s_3^2 +
18s_1^3s_3^2 - 9s_2^3s_3 + 54s_1^2s_2^2s_3 -
45s_1^4s_2s_3 + 9s_1^6s_3 + 9s_1s_2^4 -
30s_1^3s_2^3 + 27s_1^5s_2^2 - 9s_1^7s_2 +
s_1^9
```

Pour $n = 4$ et $k = 15$,

```
p = - 15s_3s_4^3 + 60s_1s_2s_4^3 - 50s_1^3s_4^3 + 90s_1s_3^2s_4^2 +
90s_2^2s_3s_4^2 - 450s_1^2s_2s_3s_4^2 + 225s_1^4s_3s_4^2 - 150s_1s_2^3s_4^2 +
450s_1^3s_2^2s_4^2 - 315s_1^5s_2s_4^2 + 60s_1^7s_4^2 + 60s_2s_3^3s_4 -
150s_1^2s_3^3s_4 - 450s_1s_2^2s_3^2s_4 + 900s_1^3s_2s_3^2s_4 - 315s_1^5s_3^2s_4 -
75s_2^4s_3s_4 + 900s_1^2s_2^3s_3s_4 - 1575s_1^4s_2^2s_3s_4 + 840s_1^6s_2s_3s_4 -
135s_1^8s_3s_4 + 90s_1s_2^5s_4 - 525s_1^3s_2^4s_4 + 840s_1^5s_2^3s_4 -
540s_1^7s_2^2s_4 + 150s_1^9s_2s_4 - 15s_1^11s_4 + 3s_3^5 -
75s_1s_2s_3^4 + 75s_1^3s_3^4 - 50s_2^3s_3^3 + 450s_1^2s_2^2s_3^3 -
525s_1^4s_2s_3^3 + 140s_1^6s_3^3 + 225s_1s_2^4s_3^2 - 1050s_1^3s_2^3s_3^2 +
1260s_1^5s_2^2s_3^2 - 540s_1^7s_2s_3^2 + 75s_1^9s_3^2 + 15s_2^6s_3 -
315s_1^2s_2^5s_3 + 1050s_1^4s_2^4s_3 - 1260s_1^6s_2^3s_3 + 675s_1^8s_2^2s_3 -
165s_1^10s_2s_3 + 15s_1^12s_3 - 15s_1s_2^7 + 140s_1^3s_2^6 -
378s_1^5s_2^5 + 450s_1^7s_2^4 - 275s_1^9s_2^3 + 90s_1^11s_2^2 -
15s_1^13s_2 + s_1^15
```

7.3 Factorisation de polynômes

7.3.1 Enoncé : factorisation de polynômes

Soit $K = \mathbb{Z}/2\mathbb{Z}$. Les polynômes considérés dans ce problème appartiennent à $K[X]$, l'anneau des polynômes à coefficients dans K. On note 0 (respectivement 1) le polynôme nul (respectivement le polynôme de degré 0 et de terme constant égal à 1). Le degré d'un polynôme est noté $\deg(f)$. Soient f et h deux polynômes. On dit que f *divise* (ou est un *diviseur* de) h s'il existe un polynôme g tel que $fg = h$. Un polynôme f est *irréductible* si ses seuls diviseurs sont 1 et f et il est *réductible* sinon. On appelle *factorisation* d'un polynôme f une suite finie de polynômes $(f_i)_{1 \leq i \leq n}$ telle que $f = f_1 f_2 \cdots f_n$. Une telle factorisation est *non triviale* si l'un des f_i est différent de f et de 1. On rappelle que tout polynôme non nul f se factorise comme produit de polynômes irréductibles et que cette décomposition est unique à l'ordre des facteurs près. Si f est un polynôme, on note f' son polynôme dérivé. Un polynôme f est un *carré* s'il existe un polynôme g tel que $f = g^2$. On dit que f est *sans carré* s'il n'est pas divisible par un carré de degré strictement positif. Enfin, si f, g et h sont des polynômes, on écrit

$$f \equiv g \bmod h$$

lorsque h divise $f - g$.

1.– a) Ecrire une procédure qui effectue la division euclidienne de deux polynômes.

b) Ecrire une procédure qui calcule le pgcd de deux polynômes, par l'algorithme d'Euclide.

Exemple numérique : calculer le pgcd des polynômes $X^{105} + 1$ et $X^{72} + X + 1$.

2.– a) Démontrer que f est un carré si et seulement si $f' = 0$.

b) Démontrer que tout polynôme non nul f se factorise de façon unique en $f = g^2h$, où h est sans carré.

c) Démontrer que cette unique factorisation est donnée par la formule $g^2 = \mathrm{pgcd}(f, f')$.

3.– Ecrire une procédure qui calcule g et h à partir de f.

Exemple numérique : $f = X^{17} + X^{14} + X^{13} + X^{12} + X^{11} + X^{10} + X^9 + X^8 + X^7 + X^5 + X^4 + X + 1$.

On suppose désormais que f est sans carré et que $f(0) = 1$. Soit $f = f_1 f_2 \cdots f_s$ la factorisation de f en produit de polynômes irréductibles.

4.– Démontrer que pour toute famille $(g_i)_{1 \le i \le s}$ de polynômes, il existe exactement un polynôme g tel que

$$\deg(g) < \deg(f) \quad \text{et, pour } 1 \le i \le s, \quad g \equiv g_i \bmod f_i$$

5.– On note $H(f)$ l'ensemble des polynômes h tels que

$$\deg(h) < \deg(f) \quad \text{et} \quad h^2 \equiv h \bmod f$$

a) Démontrer que $H(f)$ est un sous-espace vectoriel de dimension s de $K[X]$ et, que pour tout $h \in H(f)$, on a $f = \mathrm{pgcd}(f, h)\mathrm{pgcd}(f, h - 1)$.

b) On dit qu'un polynôme $h \in H(f)$ *sépare* deux facteurs f_i et f_j de f si ces deux facteurs ne divisent pas simultanément l'un des polynômes $\mathrm{pgcd}(f, h)$ ou $\mathrm{pgcd}(f, h-1)$. Démontrer que deux facteurs distincts de la décomposition de f en produit de facteurs irréductibles peuvent toujours être séparés par un élément de $H(f)$.

6.– Soit p_0 un polynôme. On considère la suite de polynômes p_n définie par $p_{n+1} = p_n^2$.

a) Démontrer qu'il existe un plus petit entier $n_0 > 0$ tel que f divise $p_{n_0} - p_0$. Démontrer que si f divise $p_n - p_0$, alors n_0 divise n.

b) En déduire qu'il existe un plus petit entier strictement positif $\alpha(f)$ tel que f divise $X^{\alpha(f)} - 1$. Démontrer que $\alpha(f)$ est impair.

7.– a) Démontrer que si f est irréductible de degré d, l'ensemble des polynômes de degré inférieur à d, muni de l'addition et de la multiplication modulo f, est un corps à 2^d éléments.

b) En déduire que f est irréductible si et seulement si $X^{2^d} \equiv X \bmod f$ et, pour tout diviseur premier p de d, $\mathrm{pgcd}(X^{2^{d/p}} - X, f) = 1$.

8.– Ecrire une procédure qui prend en argument un polynôme f et qui détermine si f est irréductible.

9.– a) Démontrer que, pour tout $i \geq 1$, il existe un plus petit entier naturel $\alpha_i(f)$ tel que

$$X^{i2^{\alpha_i(f)}} - X^i \equiv 0 \bmod f$$

On pose $r_0(f) = 1$ et, pour $i > 0$, on note $r_i(f)$ le reste de la division euclidienne du polynôme $\sum_{0 \leq j < \alpha_i(f)} X^{i2^j}$ par f. On pose également $g = X^{\alpha(f)} - 1$.

b) Démontrer que la famille $\big(r_i(g)\big)_{1 \leq i < \alpha(f)}$ engendre l'espace vectoriel $H(g)$.

10.– a) Démontrer que deux facteurs distincts de la décomposition de g en produit de facteurs irréductibles peuvent toujours être séparés par l'un des polynômes $r_i(g)$.

b) En déduire que deux facteurs distincts de la décomposition de f en produit de facteurs irréductibles peuvent toujours être séparés par l'un des polynômes $r_i(f)$.

11.– Ecrire une procédure qui prend en argument un polynôme f et qui affiche le message «polynôme irréductible» si f est irréductible, et qui affiche une factorisation non triviale de f sinon. Exemples numériques : $f = X^7 + X + 1$, $f = X^8 + X + 1$, $f = \sum_{0 \leq i \leq 16} X^i$.

7.3.2 Solution : Factorisation de polynômes

Le problème décrit une méthode de factorisation des polynômes à coefficients dans $\mathbb{Z}/2\mathbb{Z}$. Cette méthode se généralise en fait aux polynômes à coefficients dans $\mathbb{Z}/p\mathbb{Z}$ (pour p premier), ou même dans un corps fini. Dans le cas de $\mathbb{Z}/2\mathbb{Z}$, on peut toutefois tirer parti de l'arithmétique très particulière des anneaux commutatifs de caractéristique 2, résumée dans l'énoncé suivant.

PROPOSITION 7.3.1. *Dans un anneau commutatif K de caractéristique 2, tout élément est égal à son opposé et, pour toute famille $(a_i)_{1 \leq i \leq n}$ d'éléments de K, on a*

$$\Big(\sum_{1 \leq i \leq n} a_i\Big)^2 = \sum_{1 \leq i \leq n} a_i^2$$

Preuve. Pour tout $x \in K$, on a $2x = x + x = 0$, puisque K est de caractéristique 2, et donc $x = -x$. La seconde partie de l'énoncé résulte de la formule du binôme généralisée, puisque les termes de la forme $2a_ia_j$ sont nuls. ∎

Revenons à l'anneau $K = \mathbb{Z}/2\mathbb{Z}$. Pour factoriser les polynômes de $K[X]$, une première réduction consiste à se ramener à des polynômes sans carré. Elle est fondée sur les deux résultats suivants.

Proposition 7.3.2. *Un polynôme f est un carré si et seulement si $f' = 0$.*

Preuve. Si $f = g^2$, alors $f' = 2gg' = 0$. Réciproquement, soit $f = \sum_{0 \leq i \leq n} a_i X^i$ tel que $f' = 0$. On a alors

$$f' = \sum_{0 \leq i \leq n} i a_i X^{i-1} = \sum_{\substack{i \text{ impair} \\ 0 \leq i \leq n}} a_i X^{i-1} = 0$$

Par conséquent les coefficients a_i sont nuls pour i impair et donc, d'après la proposition 7.3.1,

$$f = \sum_{0 \leq i \leq n/2} a_{2i} X^{2i} = (\sum_{0 \leq i \leq n/2} a_{2i} X^i)^2 \qquad \blacksquare$$

Proposition 7.3.3. *Tout polynôme f non nul se factorise de façon unique sous la forme $f = g^2 h$, où h est sans carré. Cette factorisation unique est donnée par la formule $g^2 = \mathrm{pgcd}(f, f')$.*

Preuve. Puisque $K[X]$ est principal, tout polynôme non nul admet une décomposition unique comme produit de polynômes irréductibles. En regroupant entre eux les facteurs égaux, on obtient une factorisation de la forme suivante, où I désigne l'ensemble des polynômes irréductibles, et les n_p sont des entiers presque tous nuls (c'est-à-dire nuls sauf pour un nombre fini de p) :

$$f = \prod_{p \in I} p^{n_p}$$

On obtient alors une décomposition de la forme voulue en posant

$$g = \prod_{p \in I} p^{\lfloor n_p/2 \rfloor} \quad \text{et} \quad h = \prod_{p \in I} p^{r_p}$$

où $r_p = 1$ si n_p est impair et $r_p = 0$ sinon. L'unicité découle de ce qui suit : si $f = g^2 h$ avec h sans carré, on a $f' = g^2 h'$ et donc $\mathrm{pgcd}(f, f') = g^2 \mathrm{pgcd}(h, h')$. Soit p un facteur irréductible de $\mathrm{pgcd}(h, h')$ et posons $h = pq$. Il vient $h' = pq' + p'q$ et, puisque p divise h', p divise $p'q$. Si $p' = 0$, p est un carré d'après la proposition 7.3.2, ce qui contredit le fait que p est irréductible. Donc $p' \neq 0$ et, comme p' est de degré inférieur à p, p ne peut diviser p' et divise donc q d'après le lemme de Gauss. Il en résulte que p^2 divise h, ce qui contredit l'hypothèse que h est sans carré. Donc $\mathrm{pgcd}(h, h') = 1$ et $\mathrm{pgcd}(f, f') = g^2$. Comme f n'est pas nul, g est non nul et la factorisation est unique. $\blacksquare$

On considère désormais un polynôme f sans carré. On peut également supposer que $f(0) = 1$, sinon le polynôme X est un diviseur trivial de f. Soit $f = f_1 f_2 \cdots f_s$ la factorisation de f en produit de polynômes irréductibles. Comme f est sans carré, les f_i sont deux à deux distincts (et donc aussi premiers entre eux). L'algorithme de factorisation repose sur un résultat connu sous le nom de «théorème des restes chinois». Nous donnons ici la version relative aux polynômes. (L'énoncé général est le suivant : soient $I_1, \ldots, I_n$ des idéaux d'un anneau commutatif unitaire A tels que $I_i + I_j = A$ pour $i \neq j$. Quels que soient les éléments $a_1, \ldots, a_n$ de A, il existe $a \in A$ tel que $a \equiv a_i \bmod I_i$ pour tout i.)

Théorème 7.3.4 (Théorème des restes chinois). *Pour toute famille $(g_i)_{1\le i\le n}$ de polynômes, il existe un unique polynôme g tel que $\deg(g) < \deg(f)$ et $g \equiv g_i \bmod f_i$ pour $1 \le i \le n$.*

Preuve. Démontrons d'abord l'unicité. Si g et h sont deux solutions du problème, leur différence $g-h$ vérifie $g-h \equiv 0 \bmod f_i$ pour $1 \le i \le n$. Comme les f_i sont irréductibles et distincts, il en découle $g-h \equiv 0 \bmod f$ et, comme $\deg(g-h) < \deg(f)$, on a $g = h$.

Il reste à trouver une solution. Posons, pour $1 \le i \le n$,

$$\hat{f}_i = \prod_{j\neq i} f_j$$

Comme les polynômes f_j sont deux à deux distincts, les polynômes f_i et $\hat{f}_i$ sont premiers entre eux. D'après le théorème de Bezout, il existe des polynômes p_i et q_i tels que $p_i f_i + q_i \hat{f}_i = 1$. Posons $g = \sum_{1\le j\le n} g_j q_j \hat{f}_j$. Il vient

$$g \equiv \sum_{1\le j\le n} g_j q_j \hat{f}_j \equiv g_i q_i \hat{f}_i \equiv g_i(1 - p_i f_i) \equiv g_i \quad \bmod f_i$$

et donc le reste de la division euclidienne de g par f est solution du problème cherché. ∎

Notons $H(f)$ l'ensemble des polynômes h tels que

$$\deg(h) < \deg(f) \quad \text{et} \quad h^2 \equiv h \bmod f$$

L'algorithme est fondé sur la propriété suivante, qui fournit dans certains cas une factorisation de f.

Proposition 7.3.5. *L'ensemble $H(f)$ est un sous-espace vectoriel de $K[X]$ de dimension s et, pour tout $h \in H(f)$, on a $f = \mathrm{pgcd}(f,h)\mathrm{pgcd}(f,h-1)$.*

Preuve. Les conditions $h_1 \equiv h_1^2 \bmod f$ et $h_2 \equiv h_2^2 \bmod f$ entraînent, d'après la proposition 7.3.1, $h_1 + h_2 \equiv h_1^2 + h_2^2 = (h_1+h_2)^2 \bmod f$. En outre, $h_1 + h_2 = h_1 - h_2$ et $0 \in H(f)$. Donc $H(f)$ est un sous-espace vectoriel de $K[X]$. De plus, comme les f_i sont distincts, donc premiers entre eux, on a $h^2 \equiv h \bmod f$ si et seulement si $h^2 \equiv h \bmod f_i$ pour $1 \le i \le s$. Cette dernière condition équivaut à dire que f_i divise l'un des polynômes

h ou $(h-1)$ (puisque chaque f_i est irréductible et que $h^2 - h = h(h-1)$). En résumé, un polynôme h appartient à $H(f)$ si $\deg(h) < \deg(f)$ et s'il existe, pour $1 \leq i \leq s$, un élément a_i de K tel que $h \equiv a_i \bmod f_i$. Or le théorème chinois montre que, pour une famille $(a_i)_{1\leq i\leq s}$ donnée, les conditions précédentes sont vérifiées par un polynôme et un seul. Il en résulte que $H(f)$ contient 2^s éléments et sa dimension est donc s.

Soit $h \in H(f)$. Puisque f divise h^2-h, on a $f = \mathrm{pgcd}(f, h(h-1))$. Puisque h et $h-1$ sont premiers entre eux (d'après le théorème de Bezout), on a $f = \mathrm{pgcd}(f,h)\mathrm{pgcd}(f,h-1)$. ∎

On déduit de la proposition 7.3.5 un premier critère d'irréductibilité.

Corollaire 7.3.6. *Le polynôme f est irréductible si et seulement si $H(f) = \{0\}$.*

La proposition 7.3.5 ne fournit pas nécessairement une factorisation non triviale de f. Cependant, si le polynôme f est réductible, il est possible de retrouver tous ses facteurs irréductibles à l'aide de la proposition 7.3.5. On dit qu'un polynôme $h \in H(f)$ *sépare* deux facteurs f_i et f_j si ces deux facteurs ne divisent pas simultanément l'un des polynômes $\mathrm{pgcd}(f,h)$ ou $\mathrm{pgcd}(f,h-1)$.

Proposition 7.3.7. *Deux facteurs distincts de la décomposition de f en produit de facteurs irréductibles peuvent toujours être séparés par un élément de $H(f)$.*

Preuve. Soient f_i et f_j deux facteurs distincts de la décomposition de f en produit de facteurs irréductibles. D'après le théorème chinois, il existe un polynôme h (unique) tel que $\deg(h) < \deg(f)$, $h \equiv 1 \bmod f_i$ et $h \equiv 0 \bmod f_k$ pour $k \neq i$. Ce polynôme sépare f_i et f_j par construction, et appartient à $H(f)$, puisque $h^2 \equiv h \bmod f_k$ pour $1 \leq k \leq s$. ∎

La suite de l'algorithme consiste à rechercher un élément de $H(f)$. Elle repose sur le résultat suivant.

Proposition 7.3.8. *Soit p_0 un polynômeet soit $(p_n)_{n\geq 0}$ la suite de polynômes définie par $p_{n+1} = p_n^2$. Il existe alors un plus petit entier $n_0 > 0$ tel que f divise $p_{n_0} - p_0$. De plus, si f divise $p_n - p_0$, alors n_0 divise n.*

Preuve. Puisqu'il n'y a qu'un nombre fini de restes modulo f, il existe deux entiers $n > m$ tels que

$$p_n \equiv p_m \bmod f$$

On observe alors, en utilisant la proposition 7.3.1, que

$$p_n - p_m \equiv p_0^{2^n} - p_0^{2^m} \equiv (p_0^{2^{n-m}} - p_0)^{2^m} \equiv (p_{n-m} - p_0)^{2^m} \equiv 0 \bmod f$$

et par conséquent,

$$p_{n-m} - p_0 \equiv 0 \bmod f$$

puisque f est sans carré. Ceci démontre l'existence de n_0. Supposons $p_n \equiv p_0$ et soit $n = qn_0 + r$ la division euclidienne de n par n_0. Il vient, comme ci-dessus

$$p_n - p_{qn_0} \equiv p_0^{2^n} - p_0^{2^{qn_0}} \equiv (p_0^{2^r} - p_0)^{2^{qn_0}} \equiv 0 \bmod f$$

et par conséquent $p_r \equiv p_0 \bmod f$ puisque f est sans carré. Comme $r < n_0$, ceci entraîne $r = 0$ et donc n_0 divise n. ∎

Corollaire 7.3.9. *Il existe un plus petit entier strictement positif $\alpha(f)$ tel que f divise $X^{\alpha(f)} - 1$. Cet entier est impair.*

Preuve. En choisissant $p_0 = X$ dans la proposition 7.3.8, on constate que f divise un polynôme de la forme $X^{m+1} - X = X(X^m - 1)$. Comme f ne divise pas X (car $f(0) = 1$ et f n'est pas constant), f divise $X^m - 1$. Il existe donc un plus petit entier $\alpha(f)$ tel que f divise $X^{\alpha(f)} - 1$. Cet entier est impair, sinon $f = (X^{\alpha(f)/2} - 1)^2$ d'après la proposition 7.3.1. ∎

On déduit de ce qui précède un critère simple pour tester si f est irréductible. C'est ce test qui est réalisé dans le programme de la section suivante. Notons d le degré de f.

Proposition 7.3.10. *Le polynôme f est irréductible si et seulement si $X^{2^d} \equiv X \bmod f$ et $\mathrm{pgcd}(X^{2^{d/p}} - X, f) = 1$, pour tout diviseur premier p de d.*

Preuve. Supposons d'abord f irréductible. Soit (f) l'idéal de $K[X]$ engendré par f. L'anneau quotient $k = K[X]/(f)$ est un corps : en effet, si $g \notin (f)$, $\mathrm{pgcd}(f, g) = 1$ (puisque f est irréductible) et, d'après le théorème de Bezout, il existe des polynômes q et r tels que $qf + rg = 1$, ce qui montre que r est l'inverse de g modulo f. Puisqu'il y a exactement 2^d restes modulo f, le corps k contient 2^d éléments. En particulier, le groupe multiplicatif k^* est d'ordre $2^d - 1$. Il en résulte en particulier, puisque X est non nul modulo f,

$$X^{2^d - 1} \equiv 1 \bmod f \quad \text{et donc} \quad X^{2^d} \equiv X \bmod f$$

Ensuite, si $X^{2^m} \equiv X \bmod f$, on a aussi $X^{i2^m} \equiv X^i \bmod f$ pour tout $i > 0$ et, d'après la proposition 7.3.1, $g^{2^m} \equiv g \bmod f$ pour tout polynôme g. Il en découle que l'équation $Y^{2^m} - Y = 0$ possède au moins 2^d racines distinctes dans le corps k et, par conséquent, $m \geq d$. En particulier, si p est un diviseur premier de d, on a $X^{2^{d/p}} - X \not\equiv 0 \bmod f$ et donc $\mathrm{pgcd}(X^{2^{d/p}} - X, f) = 1$, puisque f est irréductible.

Supposons maintenant que $X^{2^d} \equiv X \bmod f$ et $\mathrm{pgcd}(X^{2^{d/p}} - X, f) = 1$, pour tout diviseur premier p de d. Supposons f réductible et soit f_1 un diviseur irréductible de f. Ce qui précède montre que

$$X^{2^{\deg(f_1)}} \equiv X \bmod f_1$$

Comme d'autre part, $X^{2^d} - X \equiv 0 \bmod f_1$, la proposition 7.3.8 montre que $\deg(f_1)$ divise d. En particulier, il existe un nombre premier p tel que $\deg(f_1)$ divise d/p. On a alors $X^{2^{d/p}} \equiv X \bmod f_1$ et donc $\mathrm{pgcd}(X^{2^{d/p}}, f) \neq 1$, ce qui contredit l'hypothèse. Donc f est irréductible. ∎

Si le test décrit dans la proposition 7.3.10 montre que f est réductible, il convient d'en chercher les facteurs irréductibles. Ceci sera fait au moyen d'une famille $r_i(f)$ de polynômes que nous introduisons maintenant. En appliquant la proposition 7.3.8 avec $p_0 = X^i$, on détermine un plus petit entier strictement positif $\alpha_i(f)$ tel que $X^{i2^{\alpha_i(f)}} \equiv X^i \bmod f$. On pose alors $r_0(f) = 1$ et, pour $i > 0$, on note $r_i(f)$ le reste de la division euclidienne du polynôme $\displaystyle\sum_{0\le j<\alpha_i(f)} X^{i2^j}$ par f.

Proposition 7.3.11. *Les polynômes $r_i(f)$ appartiennent à $H(f)$.*

Preuve. On a, d'après la définition de $\alpha_i(f)$ et la proposition 7.3.1 :

$$r_i^2(f) = X^{i2^1} + X^{i2^2} + \cdots + X^{i2^{\alpha_i(f)}} \equiv X^{i2^1} + X^{i2^2} + \cdots + X^{i2^{\alpha_i(f)-1}} + X^i \equiv r_i(f) \bmod f.$$

∎

On va démontrer que les $r_i(f)$ permettent de séparer deux à deux les diviseurs irréductibles de f. On commence par établir ce résultat pour le polynôme $g = X^{\alpha(f)} - 1$. Pour pouvoir appliquer les propositions 7.3.5, 7.3.7 et 7.3.11 à g, il faut d'abord s'assurer que g est sans carré. Or on a $g' = \alpha(f)X^{\alpha(f)-1}$ et ce polynôme n'est pas nul, puisque $\alpha(f)$ est impair. Il en résulte $\mathrm{pgcd}(g, g') = 1$ (puisque 0 n'est pas racine de g) et donc g est sans carré d'après la proposition 7.3.3.

Proposition 7.3.12. *Soit $g = X^{\alpha(f)} - 1$. La famille $\big(r_i(g)\big)_{0\le i<\alpha(f)}$ engendre $H(g)$.*

Preuve. Posons, pour simplifier, $\alpha = \alpha(f)$ et soit

$$h = \sum_{0\le k<\alpha} h_k X^k$$

un élément de $H(g)$. Il vient

$$h^2 = \sum_{0\le k<\alpha} h_k X^{2k} \equiv \sum_{0\le k\le \frac{\alpha-1}{2}} h_k X^{2k} + \sum_{\frac{\alpha-1}{2}<k<\alpha} h_k X^{2k-\alpha} \mod g$$

Comme α est impair et puisque $h^2 \equiv h \bmod g$, il vient, en notant $<i>$ le reste de la division de i par α,

$$h_k = h_{<2k>} \quad (0 \le k < \alpha)$$

On partitionne alors l'ensemble $\{0, \ldots, \alpha-1\}$ suivant la relation d'équivalence $\sim$ définie par

$$k \sim k' \text{ s'il existe } i \text{ tel que } k = <k'2^i>$$

Si S est un système de représentants de cette équivalence, on a

$$h \equiv \sum_{k\in S} h_k (X^k + X^{k2^1} + \cdots + X^{k2^{\alpha_k(g)-1}}) \equiv \sum_{k\in S} h_k r_k(g) \bmod g$$

Comme le degré des polynômes h et $r_k(g)$ est inférieur à $\deg(g)$, on a en fait $h = \displaystyle\sum_{k\in S} h_k r_k(g)$. ∎

Proposition 7.3.13. *Deux facteurs distincts d'une décomposition de g en produit de facteurs irréductibles peuvent toujours être séparés par l'un des polynômes $r_i(g)$ $(1 \leq i < \alpha(f))$.*

Preuve. Soit $g = g_1 g_2 \cdots g_s$ la décomposition de g en produit de polynômes irréductibles. Quitte à changer l'ordre des facteurs, il suffit d'établir que l'un des $r_i = r_i(g)$ sépare g_1 et g_2. Supposons que ce soit faux. Alors les polynômes g_1 et g_2 divisent simultanément l'un des polynômes r_i ou $r_i - 1$. Autrement dit, il existe $a_{r_i} \in K$ tel que

$$r_i \equiv a_{r_i} \bmod g_1 \quad \text{et} \quad r_i \equiv a_{r_i} \bmod g_2$$

Comme les r_i engendrent $H(g)$ d'après la proposition 7.3.12, la propriété précédente s'étend par linéarité : pour tout $h \in H(g)$, il existe $a_h \in K$ tel que

$$h \equiv a_h \bmod g_1 \quad \text{et} \quad h \equiv a_h \bmod g_2 \tag{$*$}$$

Or d'après la proposition 7.3.7, il existe un polynôme $h \in H(g)$ qui sépare g_1 et g_2, et ce polynôme ne peut vérifier les relations $(*)$ ci-dessus, ce qui conclut la démonstration par l'absurde. ∎

La proposition 7.3.13 permet de déterminer les facteurs irréductibles de g. Pour trouver ceux de f, il faut d'abord préciser le lien entre les polynômes $r_i(f)$ et $r_i(g)$.

Proposition 7.3.14. *Pour tout $i \geq 0$, il existe $c_i \in K$ tel que $r_i(g) \equiv c_i r_i(f) \bmod f$.*

Preuve. Reprenons les notations de la proposition 7.3.8 et prenons $p_0 = X^i$. Par définition, $\alpha_i(f)$ est le plus petit entier strictement positif tel que $p_{\alpha_i(f)} \equiv p_0 \bmod f$. Puisque $p_{\alpha_i(g)} \equiv p_0 \bmod g$, on a aussi $p_{\alpha_i(g)} \equiv p_0 \bmod f$. Or on a vu dans la preuve de la proposition 7.3.8 que si $p_n \equiv p_m \bmod f$, avec $n > m$, alors $p_{n-m} \equiv p_0 \bmod f$. Il en résulte que $\alpha_i(g)$ est multiple de $\alpha_i(f)$ (car sinon, le reste r de la division de $\alpha_i(g)$ par $\alpha_i(f)$ vérifierait $p_r \equiv p_0 \bmod f$, en contradiction avec la définition de $\alpha_i(f)$). Il en découle

$$r_i(g) \equiv p_0 + p_1 + \cdots + p_{\alpha_i(g)-1} \equiv \frac{\alpha_i(g)}{\alpha_i(f)}(p_0 + p_1 + \cdots + p_{\alpha_i(f)-1}) \equiv \frac{\alpha_i(g)}{\alpha_i(f)} r_i(f) \bmod f$$

∎

On en déduit le résultat suivant.

Proposition 7.3.15. *Deux facteurs distincts d'une décomposition de f en produit de facteurs irréductibles peuvent toujours être séparés par l'un des polynômes $r_i(f)$ $(1 \leq i < \alpha(f))$.*

Preuve. Soient f_1 et f_2 deux facteurs irréductibles distincts de f et donc de g. D'après la proposition 7.3.13, l'un des $r_i(g)$ sépare f_1 et f_2 dans la décomposition de g. Donc, quitte à échanger f_1 et f_2, on peut supposer que f_1 divise $\mathrm{pgcd}(g, r_i(g))$ mais pas $\mathrm{pgcd}(g, r_i(g) - 1)$ et que f_2 divise $\mathrm{pgcd}(g, r_i(g) - 1)$ mais pas $\mathrm{pgcd}(g, r_i(g))$. On en déduit, en réduisant modulo f et en utilisant la proposition 7.3.14, que f_1 divise $\mathrm{pgcd}(f, c_i r_i(f))$ mais pas pgcd$(f, c_i r_i(f) - 1)$ et que f_2 divise pgcd$(f, c_i r_i(f) - 1)$ mais pas pgcd$(f, c_i r_i(f))$. Ceci entraîne en particulier $c_i = 1$ (car sinon $c_i = 0$ et pgcd$(f, c_i r_i(f) - 1) = 1$) et donc $r_i(f)$ sépare f_1 et f_2. ∎

Les résultats qui précèdent fournissent un algorithme de factorisation d'un polynôme en trois étapes :

1. On se ramène à un polynôme sans carré à l'aide de la proposition 7.3.3.
2. Si f est sans carré, on utilise la proposition 7.3.10 pour déterminer si f est irréductible.
3. Si f est réductible, on calcule la suite des polynômes $r_i(f)$ pour séparer les polynômes irréductibles de f.

L'algorithme précédent est en fait une variante de l'algorithme de Berlekamp, qui repose sur les mêmes principes. Dans cet algorithme, on détermine l'espace vectoriel $H(f)$ de la manière suivante. On pose $d = \deg(f)$ et, pour tout $k \geq 0$, on note

$$c_{k,0} + c_{k,1}X + \cdots + c_{k,d-1}X^{d-1}$$

le reste de la division de X^{2^k} par f. On pose enfin $C = (c_{i,j})_{1 \leq i,j \leq d-1}$. Le polynôme

$$h = h_0 + h_1 X + \cdots + h_{d-1}X^{d-1}$$

appartient à $H(f)$ si et seulement si

$$(h_0, h_1, \ldots, h_{d-1})C = (h_0, h_1, \ldots, h_{d-1})$$

Autrement dit, $H(f)$ s'identifie à l'ensemble des vecteurs $h = (h_0, h_1, \ldots, h_{d-1})$ de K^d tels que

$$h(C - I_d) = 0$$

Pour résoudre ce problème d'algèbre linéaire, on peut par exemple trianguler la matrice $C - I_d$ par une des méthodes exposées par ailleurs dans ce livre. D'après la proposition 7.3.5, la dimension de $H(f)$ est égale au nombre de facteurs irréductibles de f.

7.3.3 Programme : factorisation de polynômes

Il est commode de représenter les polynômes à coefficients dans K par des tableaux d'entiers. Comme K possède deux éléments, on pourrait également songer à des tableaux de booléens, mais la mise au point serait plus délicate. Le degré du polynôme est mémorisé dans la composante d'indice -1 du tableau. Par convention, le degré du polynôme nul est -1.

```
CONST
   DegreMax = 127;                    Le degré maximal des polynômes.
   DegrePolNul = -1;                  Le degré du polynôme nul.
TYPE
   pol01 = ARRAY[-1..DegreMax] OF integer;
   Le degré du polynôme p est p[−1].
```

Les polynômes 0, 1 et X sont prédéfinis dans une procédure d'initialisation.

```
VAR
   Pol01Nul, Pol01Unite, X: pol01;
PROCEDURE InitPol01;
   Définition des polynômes 0, 1 et X.
   VAR
      n: integer;
   BEGIN
      FOR n := 0 TO DegreMax DO
         Pol01Nul[n] := 0;
      Pol01Unite := Pol01Nul;
      Pol01Unite[0] := 1;
      X := Pol01Nul;
      X[1] := 1;
      Pol01Nul[-1] := -1;
      Pol01Unite[-1] := 0;
      X[-1] := 1;
   END; { de "InitPol01" }
```

Pour connaître ou pour fixer le degré d'un polynôme, on utilise les procédures suivantes :

```
FUNCTION Degre (VAR p: pol01): integer;
   Donne le degré du polynôme p.
   VAR
      n: integer;
   BEGIN
      Degre := p[-1]
   END; { de "Degre" }
PROCEDURE FixerDegre (VAR p: pol01; d: integer);
   BEGIN
      p[-1] := d;
   END; { de "Degre" }
```

Nous utiliserons deux fonctions commodes pour les tests d'égalité. La première teste si un polynôme est nul et la seconde si deux polynômes sont égaux.

```
FUNCTION EstPol01Nul (VAR p: pol01): boolean;       Teste si p = 0.
   BEGIN
      EstPol01Nul := Degre(p) = DegreNul
   END; { de "EstPol01Nul" }
FUNCTION SontEgaux (f, g: pol01): boolean;
```

```
   VAR
      EncoreEgaux: boolean;
      deg, i: integer;
   BEGIN
      deg := Degre(f);
      EncoreEgaux := deg = Degre(g);
      i := -1;
      WHILE EncoreEgaux AND (i < deg) DO BEGIN
         i := i + 1;
         EncoreEgaux := f[i] = g[i];
      END;
      SontEgaux := EncoreEgaux;
   END; { de "SontEgaux" }
```

Pour définir un polynôme à coefficients dans K, il suffit de savoir quels sont les coefficients égaux à 1 : les autres coefficients sont nécessairement tous nuls. Les procédures d'entrée et d'affichage tirent parti de cette particularité.

```
PROCEDURE EntrerPol01 (VAR f: pol01; titre: texte);
   VAR
      i, degre: integer;
   BEGIN
      writeln(titre);
      f := Pol01Nul;
      write('Degré du polynôme (-1 pour le polynôme nul): ');
      readln(degre);
      FixerDegre(f, degre);
      IF degre > 0 THEN BEGIN
         writeln('Donner les coefficients égaux à 1, en terminant par
                  le degré');
         REPEAT
            readln(i);
            f[i] := 1;
         UNTIL i = degre;
      END
      ELSE
         f[0] := degre + 1;     f[0] = 0 si deg(f) = -1 et f[0] = 1 sinon.
   END; { de "EntrerPol01" }
PROCEDURE EcrireMonome (n: integer);
   BEGIN
      IF n > 1 THEN
         write('X^', n : 1)
      ELSE IF n = 1 THEN
         write('X')
      ELSE
         write('1')
   END; { de "EcrireMonome" }
PROCEDURE EcrirePol01 (f: pol01; titre: texte);
```

Affichage du polynôme f. Le commentaire est affiché avant le polynôme.
```
VAR
   i, deg, c: integer;
BEGIN
   write(titre);
   IF EstPol01Nul(f) THEN
      write('0')
   ELSE BEGIN
      deg := Degre(f);
      i := 0;
      c:= 0;
      WHILE (i <= deg) and (f[i] = 0) DO         { "and" séquentiel }
         i := i + 1;
      EcrireMonome(i);
      i := i + 1;
      WHILE (i <= deg) DO BEGIN
         IF f[i] <> 0 THEN BEGIN
            write(' + ');
            EcrireMonome(i);
            c := c + 1;
            IF c MOD 10 = 0 THEN writeln;
         END;
         i := i + 1
      END
   END
END; { de "EcrirePol01" }
```

Les deux procédures arithmétiques de base sont l'addition et la multiplication par le polynôme X^n. Notons qu'il est inutile de définir la soustraction, qui se confond avec l'addition.

```
  IF Degre(f) = Degre(g) THEN
   FixerDegre(h, deg)
  ELSE
   FixerDegre(h, Max(Degre(f), Degre(g)))
 END; { de "Pol01PlusPol01" }
PROCEDURE Pol01PlusPol01 (f, g: pol01; VAR h: pol01);              h = f + g
   VAR
      i, deg: integer;
   BEGIN
      IF Degre(f) >= Degre(g) THEN   On travaille sur le polynôme de plus grand degré.
         h := f
      ELSE
         h := g;
      deg := -1;
      FOR i := 0 TO Min(Degre(f), Degre(g)) DO BEGIN
         h[i] := (f[i] + g[i]) MOD 2;
         IF h[i] > 0 THEN deg := i;
```

```
      END;
      IF Degre(f) = Degre(g) THEN
         FixerDegre(h, deg)
      ELSE
         FixerDegre(h, Max(Degre(f), Degre(g)))
   END; { de "Pol01PlusPol01" }
```

```
PROCEDURE ProduitParXn (f: pol01; n: integer; VAR g: pol01);
```
$g = f.X^n$

```
   VAR
      i, deg: integer;
   BEGIN
      deg := Degre(f);
      IF deg >= 0 THEN BEGIN
         FixerDegre(g, deg + n);
         FOR i := 0 TO n - 1 DO
            g[i] := 0;
         FOR i := 0 TO deg DO
            g[i + n] := f[i]
      END
      ELSE
         g := Pol01Nul;
   END; { de "ProduitParXn" }
```

La procédure de division euclidienne peut être réalisée ainsi :

```
PROCEDURE Pol01DivEuclidPol01 (f, g: pol01; VAR q, r: pol01);
```
On suppose $g \neq 0$. On obtient $f = gq + r$ avec $d(r) < d(g)$.

```
   VAR
      h: pol01;
      n: integer;
   BEGIN
      q := Pol01Nul;
      r := f;
      FixerDegre(q, Max(Degre(f) - Degre(g), -1));     Calcul du degré de q.
      WHILE Degre(r) >= Degre(g) DO BEGIN
         n := Degre(r) - Degre(g);
         q[n] := 1;
         ProduitParXn(g, n, h);                        h = gX^n
         Pol01PlusPol01(r, h, r);                      r := r + gX^n
      END;
   END; { de "Pol01DivEuclidPol01" }
```

Par commodité, on introduit deux procédures très voisines, similaires aux procédures `mod` et `div` sur les entiers.

```
PROCEDURE Pol01ModPol01 (f, g: pol01; VAR r: pol01);
```
On suppose $g \neq 0$. Calcule le reste r de f mod g.

```
   VAR
      q: pol01;
   BEGIN
```

```
      Pol01DivEuclidPol01(f, g, q, r)
   END; { de "Pol01ModPol01" }
PROCEDURE Pol01DivPol01 (f, g: pol01; VAR q: pol01);
```
On suppose $g \neq 0$. Calcule le quotient q de f par g.
```
   VAR
      r: pol01;
   BEGIN
      Pol01DivEuclidPol01(f, g, q, r)
   END; { de "Pol01ModPol01" }
```

Le calcul du pgcd de deux polynômes est similaire au calcul du pgcd de deux entiers. Il suffit donc d'adapter la procédure du pgcd pour les entiers au cas des polynômes.

```
PROCEDURE PGCDPol01 (f, g: pol01; VAR pgcd: pol01);
   VAR
      r: pol01;
   BEGIN
      WHILE NOT EstPol01Nul(g) DO BEGIN
         Pol01ModPol01(f, g, r);
         f := g;
         g := r;
      END;
      pgcd := f;
   END; { de "PGCDPol01" }
```

Sur l'exemple numérique, on trouve

```
f = 1 + X^105
g = 1 + X + X^72
PGCD(f, g) = 1 + X^3 + X^4
```

Le calcul de la décomposition $f = g^2h$ nécessite trois procédures auxiliaires, permettant le calcul du polynôme dérivé, le calcul du carré d'un polynômeet le calcul de la racine carrée d'un polynôme carré. Le calcul du polynôme dérivé s'obtient facilement à partir des formules

$$(X^n)' = \begin{cases} X^{n-1} & \text{si } n \text{ est impair} \\ 0 & \text{si } n \text{ est pair} \end{cases}$$

Pour le calcul du carré et de la racine carrée, on utilise la formule

$$\Big(\sum_{1\le i\le n} f_i\Big)^2 = \sum_{1\le i\le n} f_i^2$$

```
PROCEDURE Derive (f: pol01; VAR Df: pol01);
   VAR
      i, deg: integer;
   BEGIN
      Df := Pol01Nul;
      deg := -1;
```

```
      FOR i := 1 TO Degre(f) DO
         IF odd(i) THEN BEGIN
            Df[i - 1] := f[i];
            IF f[i] <> 0 THEN deg := i - 1
         END
         ELSE Df[i - 1] := 0;
      FixerDegre(Df, deg)
   END; { de "Derive" }
PROCEDURE Carre (f: pol01; VAR g: pol01);
   VAR
      i: integer;
   BEGIN
      IF EstPolNul THEN
         g := Pol01Nul
      ELSE
         FixerDegre(g, 2 * Degre(f));
      FOR i := 0 TO Degre(f) DO BEGIN
         g[2 * i] := f[i];
         g[2 * i + 1] := 0
      END
   END; { de "Carre" }
PROCEDURE RacineCarree (f: pol01; VAR g: pol01);
   VAR
      i, deg: integer;
   BEGIN
      IF EstPol01Nul(f) THEN
         g := Pol01Nul
      ELSE BEGIN
         deg := Degre(f) DIV 2;
         FixerDegre(g, deg);
         FOR i := 0 TO deg DO
            g[i] := f[i * 2]
      END
   END; { de "RacineCarree" }
```

Le calcul de la décomposition $f = g^2h$ ne comporte plus de difficulté, puisque $g^2 = \mathrm{pgcd}(f, f')$.

```
PROCEDURE FacteursCarres (f: pol01; VAR g, h: pol01);
   VAR
      Df: pol01;
   BEGIN
      g := f;
      Derive(f, Df);
      IF EstPol01Nul(Df) THEN BEGIN
         RacineCarree(f, g);
         h := Pol01Unite;
      END
```

```
      ELSE BEGIN
         PGCDPol01(f, Df, g);
         Pol01DivPol01(f, g, h);
         RacineCarree(g, g);
      END;
   END; { "FacteursCarres" }
```

Pour l'exemple numérique, on trouve

```
f = 1 + X + X^4 + X^5 + X^7 + X^8 + X^9 + X^10 + X^11 + X^12 + X^13
 + X^14 + X^17
g = 1 + X^4 + X^5
h = 1 + X + X^4 + X^5 + X^7
```

Nous en arrivons au test d'irréductibilité, qui est fondé sur la proposition 7.3.10. Cette procédure utilise un test de primalité (`EstPremier`) : on pourra utiliser par exemple le test de primalité élémentaire décrit dans le chapitre 11 ou le remplacer par un test plus efficace.

```
FUNCTION EstIrreductible (f: pol01): boolean;
   On suppose f sans carré et f(0) = 1. Teste si f est irréductible.
   VAR
      p, q: pol01;
      d, i: integer;
      ToujoursIrreductible: boolean;
   BEGIN
      d := Degre(f);
      p := X;
      i := 0;
      ToujoursIrreductible := true;
      WHILE ToujoursIrreductible AND (i < d) DO BEGIN
         i := i + 1;
         Carre(p, p);
         Pol01ModPol01(p, f, p);
Si i est un diviseur de d et si d/i est premier, on teste si pgcd(X^(2^i) − X, f) = 1.
         IF (d MOD i = 0) AND EstPremier(d DIV i) THEN BEGIN
            Pol01PlusPol01(p, X, q);
            PGCDPol01(q, f, q);                 q = pgcd(X^(2^i) − X, f)
            ToujoursIrreductible := SontEgaux(q, Pol01Unite)
         END;
      END;
Il reste à vérifier que X^(2^d) − X ≡ 0 mod f.
      IF ToujoursIrreductible THEN BEGIN
         Pol01PlusPol01(p, X, q);
         Pol01ModPol01(q, f, q);
         ToujoursIrreductible := SontEgaux(q, Pol01Nul);
      END;
      EstIrreductible := ToujoursIrreductible;
   END;
```

Si f est réductible, la proposition 7.3.15 montre que l'on peut séparer ses facteurs irréductibles à l'aide des polynômes r_i. Le calcul de ces polynômes n'offre pas de difficulté particulière.

```
PROCEDURE CalculRi (i: integer; f: pol01; VAR r: pol01);
   VAR
      Xi, XiModf, p: pol01;
   BEGIN
      Xi := Pol01Nul;
      Xi[-1] := i;
      Xi[i] := 1;
      Pol01ModPol01(Xi, f, XiModf);
      p := XiModf;
      r := Pol01Nul;
      REPEAT
         Pol01PlusPol01(r, p, r);
         Pol01ModPol01(r, f, r);
         Carre(p, p);
         Pol01ModPol01(p, f, p);
      UNTIL SontEgaux(XiModf, p);
   END; { de "CalculRi" }
```

La procédure ci-dessous prend en argument un polynôme f, affiche le message «`polynome irreductible`» si f est irréductible et donne une décomposition non triviale de f sinon.

```
PROCEDURE Factorisation (f: pol01);
   VAR
      DecompositionTriviale: boolean;
      i: integer;
      g, h, r: pol01;
   BEGIN
```

Première étape : on calcule la décomposition $f = g^2h$ avec h sans carré. Si $g \neq 1$, on affiche la décomposition obtenue.

```
      FacteursCarres(f, g, h);
      IF NOT SontEgaux(g, Pol01Unite) THEN BEGIN
         writeln('f = g^2h  avec ');
         EcrirePol01(g, 'g = ');
         writeln;
         EcrirePol01(h, 'h = ');
         writeln;
      END
```

Deuxième étape : f est certainement sans carré; on effectue le test d'irréductibilité.

```
      ELSE IF EstIrreductible(f) THEN
         writeln('polynôme irréductible')
```

Troisième étape : f est réductible. On effectue le calcul des r_i jusqu'à ce que la factorisation $f = \mathrm{pgcd}(f, r_i)\mathrm{pgcd}(f, r_i - 1)$ soit non triviale.

```
      ELSE BEGIN
         i := 1;
         DecompositionTriviale := true;
```

```
      REPEAT
         CalculRi(i, f, r);
         DecompositionTriviale := Degre(r) <= 0;     Teste si r_i = 0 ou r_i = 1.
         IF NOT DecompositionTriviale THEN BEGIN
            PGCDPol01(f, r, h);
            DecompositionTriviale := (Degre(h) >= Degre(f)) OR
                                     (Degre(h) <= 0)
         END;
         i := i + 1;
      UNTIL NOT DecompositionTriviale;
```

Il reste à afficher le résultat obtenu.

```
      EcrirePol01(h, 'f = (');
      writeln(')');
      Pol01PlusPol01(r, Pol01Unite, r);
      PGCDPol01(f, r, h);
      EcrirePol01(h, '(');
      writeln(')');
   END
END; { de "Factorisation" }
```

Les diverses options du programme sont regroupées dans le menu suivant :

```
PROCEDURE Menu;
   VAR
      choix, i, k, n: integer;
      g, h, q, r, s: pol01;
   BEGIN
      writeln('Menu');
      writeln('(1) Diviser 2 polynômes');
      writeln('(2) PGCD de 2 polynômes');
      writeln('(3) Décomposition f = g^2.h');
      writeln('(4) Calcul des polynômes r_i associés à f');
      writeln('(5) Factorisation de 1 + X + X^2 + ... + X^(n -1) (pour n
            impair)');
      writeln('(6) Factorisation d''un polynôme ');
      write('Votre choix : ');
      readln(choix);
      CASE choix OF
         1:
            BEGIN
            EntrerPol01(f, 'Donner le premier polynôme : ');
            EcrirePol01(f, 'f = ');
            writeln;
            EntrerPol01(g, 'Donner le deuxième polynôme : ');
            EcrirePol01(g, 'g = ');
            writeln;
            Pol01DivEuclidPol01(f, g, q, r);
            writeln('f = gq + r avec');
            EcrirePol01(q, 'q = ');
```

```
      writeln;
      EcrirePol01(r, 'r = ');
      writeln;
   END;
   2:
      BEGIN
      EntrerPol01(f, 'Donner le premier polynôme : ');
      EcrirePol01(f, 'f = ');
      writeln;
      EntrerPol01(g, 'Donner le deuxième polynôme : ');
      EcrirePol01(g, 'g = ');
      writeln;
      PGCDPol01(f, g, h);
      EcrirePol01(h, 'PGCD(f, g) = ');
      writeln;
   END;
   3:
      BEGIN
      EntrerPol01(f, 'Donner le polynôme : ');
      EcrirePol01(f, 'f = ');
      writeln;
      FacteursCarres(f, g, h);
      EcrirePol01(g, 'g = ');
      writeln;
      EcrirePol01(h, 'h = ');
      writeln;
   END;
   4:
      BEGIN
      EntrerPol01(f, 'Donner le polynôme : ');
      EcrirePol01(f, 'f = ');
      writeln;
      write('Calcul des r_i de i = 1  i = ');
      readln(k);
      FOR i := 1 TO k DO BEGIN
         CalculRi(i, f, r);
         write(i : 1);
         EcrirePol01(r, '  r_i = (');
         writeln(')');
      END
   END;
   5:
      BEGIN
      write('n = ');
      readln(n);
      FixerDegre(f, n - 1);
      FOR i := 0 TO n - 1 DO
         f[i] := 1;
```

```
            EcrirePol01(f, 'f = ');
            writeln;
            Factorisation(f);
         END;
         6:
            BEGIN
            EntrerPol01(f, 'Donner le polynôme : ');
            EcrirePol01(f, 'f = ');
            writeln;
            Factorisation(f);
         END;
         OTHERWISE
      END;
   END; { de "Menu" }
```

Pour finir, voici quelques résultats numériques.

```
f = 1 + X + X^7
polynôme irréductible
f = 1 + X + X^8
f = (1 + X^2 + X^3 + X^5 + X^6)
(1 + X + X^2)
f = 1 + X + X^2 + X^3 + X^4 + X^5 + X^6 + X^7 + X^8 + X^9 + X^10
 + X^11 + X^12 + X^13 + X^14 + X^15 + X^16
f = (1 + X^3 + X^4 + X^5 + X^8)
(1 + X + X^2 + X^4 + X^6 + X^7 + X^8)
f = 1 + X + X^2 + X^3 + X^4 + X^5 + X^6 + X^7 + X^8 + X^9 + X^10
 + X^11 + X^12 + X^13 + X^14 + X^15 + X^16 + X^17 + X^18 + X^19 + X^20
 + X^21 + X^22 + X^23 + X^24 + X^25 + X^26 + X^27 + X^28 + X^29 + X^30
 + X^31 + X^32 + X^33 + X^34 + X^35 + X^36 + X^37 + X^38 + X^39 + X^40
f = (1 + X^2 + X^3 + X^4 + X^5 + X^6 + X^9 + X^10 + X^11 + X^14 + X^15
 + X^16 + X^17 + X^18 + X^20)
(1 + X + X^3 + X^4 + X^6 + X^9 + X^10 + X^11 + X^14 + X^16 + X^17
 + X^19 + X^20)
```

Notes bibliographiques

Sur les suites de Sturm et les majorations des zéros de polynômes, on pourra consulter :

M. Mignotte, *Mathématiques pour le calcul formel*, Paris, Presses Universitaires de France, 1989.

L'algorithme de localisation se trouve dans :

J. Davenport, Y. Siret, E. Tournier, *Calcul formel*, Paris, Masson, 1987.

Les polynômes symétriques sont traités dans la plupart des livres d'algèbre. On pourra consulter en particulier :

N. Bourbaki, *Eléments de mathématique, Algèbre*, Chapitre 5, Appendice 1, Paris, Masson, 1981. (Contrairement à son habitude, Bourbaki ne se place pas dans le cadre le plus général et ne traite que le cas des polynômes à coefficients dans un corps.)

R. Godement, *Cours d'algèbre*, Paris, Hermann, 1966. (Le problème sur les polynômes symétriques est en partie inspiré par l'exercice 13 du chapitre 33.)

B.L. Van der Waerden, *Moderne Algebra*, 2 volumes, Berlin, Springer, 1955 ou *Modern Algebra*, 2 volumes, New York, F. Ungar, 1950.

L'algorithme utilisé pour factoriser les polynômes est adapté de l'article suivant :

R.J. McEliece, Factorization of polynomials over finite fields, *Mathematics of Computation* **23** (1969), 861–867.

L'algorithme de Berlekamp est exposé en détail dans le livre de Knuth :

D.E. Knuth, *The Art of Computer Programming*, Reading, Addison Wesley, 1981, Vol. 2, *Seminumerical Algorithms*, 420–430.

On pourra également consulter l'article original de Berlekamp :

E.R. Berlekamp, Factoring polynomials over finite fields, *Bell System Technical J.* **46** (1967), 1853–1859.

Partie III

Combinatoire

Chapitre 8

Exemples combinatoires

8.1 Génération d'objets combinatoires

Les objets que nous considérons ici sont : les sous-ensembles d'un ensemble, les permutations, les partitions. Ils sont représentés comme suites finies d'entiers. Les suites d'entiers sont naturellement ordonnées par l'*ordre lexicographique* : si $s = (s_1, \ldots, s_n)$ et $t = (t_1, \ldots, t_m)$ sont deux suites d'entiers, alors $s < t$ dans l'ordre lexicographique si

- $n < m$ et $s_j = t_j$ pour $j = 1, \ldots, n$, ou
- il existe i tel que $s_i < t_i$ et $s_j = t_j$ pour $j = 1, \ldots, i-1$.

On cherche des algorithmes pour constituer des listes d'objets de taille donnée. Le *rang* d'un objet est le nombre d'objets qui le précèdent dans la liste. Le premier élément d'une liste a donc rang 0. Le rang sera calculé sans énumérer les objets qui le précèdent, et réciproquement, la construction de l'objet de rang donné sera faite directement.

Engendrer les objets consiste à en déterminer le *premier*, puis à construire le *suivant* jusqu'à l'obtention du *dernier*. L'objet considéré est le dernier si la tentative de calcul de l'objet suivant échoue.

8.1.1 Sous-ensembles

Soient à engendrer toutes les parties d'un ensemble E à n éléments. L'ensemble E est identifié à $\{1, \ldots, n\}$, un sous-ensemble X de E est représenté par le vecteur x associé à sa fonction caractéristique et défini par

$$x_i = \begin{cases} 1 & \text{si } i \in X \\ 0 & \text{sinon} \end{cases}$$

Les parties sont ordonnées par l'ordre lexicographique sur les vecteurs associés, et le rang d'un ensemble X, de vecteur x, est l'entier $\sum_{i=1}^{n} x_i 2^{i-1}$. La première partie est la partie vide.

Pour représenter les suites d'entiers, on utilise le type suivant :

```
TYPE
   suite = ARRAY[1..LongueurSuite] OF integer;
```

où `LongueurSuite` est une constante convenable (dans le cas des parties d'un ensemble, un tableau de booléens est bien entendu suffisant). Le calcul du premier sous-ensemble de E (la partie vide) se fait au moyen de la procédure que voici :

```
PROCEDURE PremierePartie (VAR x: suite; n: integer);
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO n DO
         x[i] := 0
   END; { de "PremierePartie" }
```

La partie qui suit une partie donnée s'obtient en changeant le dernier chiffre égal à 0 en 1 et en remplaçant par 0 tous les chiffres qui le suivent :

```
PROCEDURE PartieSuivante (VAR x: suite; n: integer; VAR derniere: boolean);
   VAR
      i: integer;
   BEGIN
      i := n;
      WHILE (i > 0) AND (x[i] = 1) DO BEGIN     { AND séquentiel }
         x[i] := 0;
         i := i - 1
      END;
      derniere := i = 0;
      IF NOT derniere THEN
         x[i] := 1
   END; { de "PartieSuivante" }
```

Finalement, pour lister les parties, on peut utiliser la procédure que voici :

```
PROCEDURE ListerParties (n: integer);
   VAR
      x: Suite;
      derniere: boolean;
   BEGIN
      PremierePartie(x, n);
      REPEAT
         EcrireSuite(x, n, '');
         PartieSuivante(x, n, derniere)
      UNTIL derniere
   END; { de "ListerParties" }
```

Bien entendu, `EcrireSuite` est une procédure d'affichage. Pour $n = 3$, on obtient :

```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

On reconnaît l'écriture en binaire des nombres de 0 à 7, ce qui suggère une autre procédure qui calcule directement l'écriture binaire des entiers de 0 à $2^n - 1$.

8.1.2 Sous-ensembles à k éléments

Soient à engendrer les parties à k éléments d'un ensemble E à n éléments, à nouveau identifié à $\{1, \ldots, n\}$. On utilise la même représentation, mais il apparaît plus simple d'employer une variante de l'ordre lexicographique : on pose $x < y$ s'il existe i tel que $x_i < y_i$ et $x_j = y_j$ pour $j = i + 1, \ldots, n$. Ainsi, les parties à 3 éléments d'un ensemble à 5 éléments s'écrivent, dans cet ordre, comme suit :

```
0   1 1 1 0 0
1   1 1 0 1 0
2   1 0 1 1 0
3   0 1 1 1 0
4   1 1 0 0 1
5   1 0 1 0 1
6   0 1 1 0 1
7   1 0 0 1 1
8   0 1 0 1 1
9   0 0 1 1 1
```

La recherche de la partie suivante se décompose en deux phases : on cherche, en progressant par indices croissants, le premier intervalle constitué de chiffres 1 (disons de longueur p). Le chiffre qui suit (s'il existe) est transformé en 1 ; tous les chiffres précédents sont remplacés par un groupe de $p - 1$ chiffres 1, suivis de zéros. Voici les procédures :

```
PROCEDURE PremierePartieRestreinte (VAR x: suite; n, k: integer);
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO k DO x[i] := 1;
      FOR i := k + 1 TO n DO x[i] := 0
   END; { de "PremierePartieRestreinte" }

PROCEDURE PartieSuivanteRestreinte (VAR x: suite; n, k: integer;
      VAR derniere: boolean);
```

```
VAR
   i, j, m: integer;
BEGIN
   j := 1;
   WHILE (j <= n) AND (x[j] = 0) DO
      j := j + 1;               Recherche du premier coefficient non nul.
   i := j;                      Calcul de l'intervalle des coefficients non nuls.
   WHILE (i <= n) AND (x[i] = 1) DO BEGIN
      x[i] := 0;
      i := i + 1
   END;
   derniere := i = n + 1;
   IF NOT derniere THEN BEGIN
      x[i] := 1;                Modification du coefficient.
      FOR m := 1 TO i - j - 1 DO
         x[m] := 1              Répartition des chiffres restants.
   END
END; { de "PartieSuivanteRestreinte" }
```

Le *rang* d'une partie X à k éléments de E est le nombre

$$r(X) = \binom{n_1}{1} + \cdots + \binom{n_k}{k}$$

où les entiers $0 \leq n_1 < \ldots n_k < n$ sont définis par

$$X = \{n_1 + 1, \ldots, n_k + 1\}$$

La proposition suivante montre que la fonction r est une bijection des parties à k éléments sur les premiers entiers.

Proposition 8.1.1. *Soient $n, k \geq 0$ des entiers. Tout entier m, avec $0 \leq m < \binom{n}{k}$ s'écrit de manière unique sous la forme*

$$m = \binom{n_1}{1} + \cdots + \binom{n_k}{k} \qquad 0 \leq n_1 < \cdots < n_k < n$$

Preuve. L'existence se prouve par récurrence sur m, en considérant le plus petit entier n' tel que $m \geq \binom{n'}{k}$. Alors $m < \binom{n'+1}{k} = \binom{n'}{k} + \binom{n'}{k-1}$, donc $m' = m - \binom{n'}{k} < \binom{n'}{k-1}$. L'entier m' s'écrit par conséquent sous la forme

$$m' = \binom{n_1}{1} + \cdots + \binom{n_{k-1}}{k-1}$$

avec $0 \leq n_1 < \cdots < n_k < n'$, d'où l'on tire l'écriture pour m.

Pour montrer l'unicité, on observe l'inégalité

$$\binom{n-k}{1} + \cdots + \binom{n-1}{k} < \binom{n}{k}$$

qui se prouve par récurrence sur n en utilisant l'égalité $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$. Il en résulte que si

$$\binom{n_1}{1} + \cdots + \binom{n_k}{k} = \binom{n'_1}{1} + \cdots + \binom{n'_k}{k}$$

avec par exemple $n'_k > n_k$, alors

$$\binom{n'_k}{k} \le \binom{n_1}{1} + \cdots + \binom{n_k}{k} \le \binom{n_k - k - 1}{1} + \cdots + \binom{n_k - 1}{k-1} + \binom{n_k}{k} < \binom{n_k + 1}{k}$$

donc $n'_k \le n_k$, d'où une contradiction. ∎

Il reste à vérifier que si la partie X précède Y dans l'ordre défini ci-dessus, alors $r(X) < r(Y)$. Pour cela, posons

$$r(X) = \binom{n_1}{1} + \cdots + \binom{n_k}{k} \quad r(Y) = \binom{n'_1}{1} + \cdots + \binom{n'_k}{k}$$

Comme X précède Y, il existe $i \le k$ tel que $n_i < n'_i$, et $n_j = n'_j$ pour $j = i+1, \ldots, k$. Il en résulte qu'avec $s = \binom{n_{i+1}}{i+1} + \cdots + \binom{n_k}{k}$, on a

$$r(X) - s = \binom{n_1}{1} + \cdots + \binom{n_i}{i} < \binom{n'_i}{i} \le r(Y) - s$$

d'où la propriété.

On obtient facilement la procédure de calcul du rang :

```
FUNCTION RangRestreint (VAR x: Suite; n, k: integer): integer;
   VAR
      r: integer;
   BEGIN
      r := 0;
      WHILE k > 0 DO BEGIN
         IF x[n] = 1 THEN BEGIN
            r := r + binomial(n - 1, k); k := k - 1
         END;
         n := n - 1
      END;
      RangRestreint := r
   END; { de "RangRestreint" }
```

et de la même manière, la procédure inverse qui, pour un entier donné, construit le sous-ensemble correspondant.

8.1.3 Permutations

Une permutation σ de l'ensemble $\{1, \ldots, n\}$ est représentée par la suite $(\sigma(1), \ldots, \sigma(n))$. On considère à nouveau la génération des permutations dans l'ordre lexicographique.

La première permutation est la permutation identique, et la dernière la suite $(n, \dots, 1)$. Soit σ une permutation. Pour calculer la permutation suivante, on considère la dernière *montée* de σ, c'est-à-dire le plus grand entier $i < n$ tel que $\sigma(i) < \sigma(i+1)$. Si σ n'a pas de montée, c'est la dernière permutation, sinon, soit τ la suivante; elle vérifie

$$\begin{aligned} \tau(j) &= \sigma(j) \qquad j < i \\ \tau(i) &= \min_{i+1 \leq j \leq n} \{\sigma(j) \mid \sigma(j) > \sigma(i)\} \end{aligned}$$

et $(\tau(i+1), \dots, \tau(n))$ est la suite croissante des nombres $\{\sigma(i), \dots, \sigma(n)\} - \{\tau(i)\}$. Par exemple, si $\sigma = (8, 6, 3, 4, 7, 5, 2, 1)$, la dernière montée est 4, et la permutation qui suit σ est $(8, 6, 3, 5, 1, 2, 4, 7)$. Comme la suite $(\sigma(i+1), \dots, \sigma(n))$ est décroissante, on obtient la suite $(\tau(i+1), \dots, \tau(n))$ en renversant $(\sigma(i+1), \dots, \sigma(n))$ après y avoir substitué $\sigma(i)$. Les procédures suivantes réalisent l'énumération des permutations :

```
PROCEDURE PremierePermutation (VAR s: suite; n: integer);
   VAR
      i: integer;
   BEGIN
      FOR i := 1 TO n DO s[i] := i
   END; { de "PremierePermutation" }
PROCEDURE PermutationSuivante (VAR s: suite; n: integer;
      VAR derniere: boolean);
   VAR
      i, j, k: integer;
   BEGIN
      i := n - 1;
      WHILE (i > 0) AND (s[i] > s[i + 1]) DO
         i := i - 1;                        i est la dernière montée.
      derniere := i = 0;
      IF NOT derniere THEN BEGIN
         j := n;
         WHILE s[i] > s[j] DO
            j := j - 1;
         EchangerE(s[j], s[i]);             Nouvelle valeur de σ(i).
         k := 1 + ((n + i + 1) DIV 2);
         FOR j := k TO n DO
            EchangerE(s[j], s[n + i + 1 - j])
      END
   END; { de "PermutationSuivante" }
```

Pour obtenir une liste des permutations, on utilise, comme précédemment, une procédure de la forme :

```
PROCEDURE ListerPermutations (n: integer);
   VAR
      s: Suite;
      derniere: boolean;
```

```
BEGIN
   PremierePermutation(s, n);
   REPEAT
      EcrireSuite(s, n, '');
      PermutationSuivante(s, n, derniere)
   UNTIL derniere
END; { de "ListerPermutations" }
```

Une *suite factorielle* d'ordre n est une suite finie $(a_1, \ldots, a_{n-1})$ d'entiers tels que $0 \leq a_i \leq i$ pour $i = 1, \ldots, n-1$. Il y a $n!$ suites factorielles d'ordre n. L'application

$$\nu : (a_1, \ldots, a_{n-1}) \mapsto \sum_{i=1}^{n-1} a_i i!$$

est une bijection des suites factorielles d'ordre n sur les entiers $0, \ldots, n!-1$. De plus, si a et b sont deux suites factorielles, et $a < b$ dans l'ordre lexicographique, alors $\nu(a) < \nu(b)$.

Il nous reste à associer une suite factorielle à une permutation. Pour cela, nous considérons l'application δ qui à une permutation sur $\{1, \ldots, n\}$ associe une permutation sur $\{1, \ldots, n-1\}$ par

$$\delta(\sigma)(i) = \begin{cases} \sigma(i+1) & \text{si } \sigma(i+1) < \sigma(1) \\ \sigma(i+1) - 1 & \text{sinon} \end{cases}$$

Clairement, l'application $\sigma \mapsto (\sigma(1), \delta(\sigma))$ est une bijection. La suite factorielle associée à σ est la suite $(\delta^{n-2}(\sigma)(1) - 1, \delta^{n-3}(\sigma)(1) - 1, \ldots, \delta(\sigma)(1) - 1, \sigma(1) - 1)$. Par exemple, voici les images successives par δ de la permutation $(2, 8, 6, 4, 5, 1, 3, 7)$:

2	8	6	4	5	1	3	7
	7	5	3	4	1	2	6
		5	3	4	1	2	6
			3	4	1	2	5
				3	1	2	4
					1	2	3
						1	1
							1

La suite factorielle associée est $(0, 0, 2, 2, 4, 6, 1)$, et le rang est 9900. Le rang d'une permutation s'obtient aussi directement par la formule

$$r(\sigma) = \sum_{i=1}^{n-1} (n-i)! \left(\delta^{i-1}(\sigma)(1) - 1\right)$$

C'est cette formule qui est évaluée, en utilisant le schéma de Horner, dans la procédure ci-dessous. La fonction δ est mise en place par :

```
PROCEDURE DiminuerPermutation (VAR s: Suite; i, n: integer);
   VAR
      j, k: integer;
   BEGIN
      k := s[i];
      FOR j := i TO n DO IF s[j] >= k THEN s[j] := s[j] - 1
   END; { de "DiminuerPermutation" }
```

La permutation $\delta(\sigma)$ est calculée dans le tableau contenant σ.

```
FUNCTION RangPermutation (s: Suite; n: integer): integer;
   VAR
      rang, i: integer;
   BEGIN
      rang := 0;
      FOR i := 1 TO n - 1 DO BEGIN
         DiminuerPermutation(s, i, n);
         rang := (rang + s[i]) * (n - i)
      END;
      RangPermutation := rang
   END; { de "RangPermutation" }
```

Pour utiliser cette fonction sur des exemples de taille raisonnable, on aura intérêt à la déclarer de type `longint` (il faut alors faire le même changement pour la variable `rang`). Les opérations inverses, donnant la permutation à partir du rang, sont les suivantes :

```
PROCEDURE AugmenterPermutation (VAR s: Suite; i, n: integer);
   VAR
      j, k: integer;
   BEGIN
      k := s[i];
      FOR j := i + 1 TO n DO IF s[j] >= k THEN s[j] := s[j] + 1
   END; { de "AugmenterPermutation" }

PROCEDURE PermutationDeRang (VAR s: Suite; n, r: integer);
   VAR
      i: integer;
   BEGIN
      s[n] := 1;
      FOR i := n - 1 DOWNTO 1 DO BEGIN
         s[i] := 1 + (r MOD (n + 1 - i));
         AugmenterPermutation(s, i, n);
         r := r DIV (n + 1 - i)
      END;
   END; { de "PermutationDeRang" }
```

8.2 Nombres de Bernoulli

8.2.1 Enoncé : nombres de Bernoulli

Les *polynômes de Bernoulli* $B_n(x)$ sont définis par leur série génératrice

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \frac{t^n}{n!} \qquad (x \in \mathbb{R},\ t \in \mathbb{C},\ |t| < 2\pi)$$

et les *nombres de Bernoulli* sont par définition les nombres $B_n = B_n(0)$, pour $n \geq 0$.

1.– Démontrer que l'on a $B_0 = 1$, $B_1 = -1/2$ et que $B_{2n+1} = 0$ pour $n \geq 1$. Démontrer que

$$B_n = \sum_{k=0}^{n} \binom{n}{k} B_k \qquad (n \geq 2)$$

(On note $\binom{p}{m} = \frac{p!}{m!(p-m)!}$.)

2.– Démontrer que, pour $n \geq 0$,

$$B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k}$$

Les nombres de Bernoulli sont rationnels et seront représentés comme quotients d'entiers premiers entre eux, le dénominateur étant positif.

3.– Ecrire des procédures d'addition, soustraction, multiplication, division de rationnels ainsi représentés. Ecrire une procédure qui calcule et imprime les nombres de Bernoulli et les coefficients des polynômes de Bernoulli pour $n \leq 12$.

4.– Démontrer que $B_n(1 + x) = nx^{n-1} + B_n(x)$ et en déduire que, pour $n \geq 1$,

$$\sum_{k=1}^{m} k^n = \frac{B_{n+1}(m + 1) - B_{n+1}}{n + 1}$$

5.– Ecrire une procédure qui affiche un «formulaire» pour les sommes des puissances n-ièmes d'entiers, pour $2 \leq n \leq 6$.

8.2.2 Solution : nombres de Bernoulli

Les *polynômes de Bernoulli* $B_n(x)$ sont définis par leur série génératrice

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \frac{t^n}{n!} \qquad (x \in \mathbb{R},\ t \in \mathbb{C},\ |t| < 2\pi)$$

et les *nombres de Bernoulli* sont, par définition, les nombres $B_n = B_n(0)$, pour $n \geq 0$. Bien entendu,

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!} \qquad (|t| < 2\pi)$$

Voici les premières propriétés des nombres de Bernoulli.

Proposition 8.2.1. *On a*

$$B_n = \sum_{k=0}^{n} \binom{n}{k} B_k \qquad (n \geq 2) \tag{2.1}$$

$$B_0 = 1 \qquad B_1 = -1/2$$

$$B_{2n+1} = 0 \qquad (n \geq 1) \tag{2.2}$$

Preuve. On a

$$t = (e^t - 1)\left(\sum_{n=0}^{\infty} B_n \frac{t^n}{n!}\right) = \sum_{n=0}^{\infty}\left(\sum_{k=0}^{n}\binom{n}{k}B_k - B_n\right)\frac{t^n}{n!}$$

Par identification, on obtient $B_0 = 1$, $0 = 2B_1 + B_0$ et les relations (2.1). Par ailleurs, la fonction $t/(e^t - 1) - B_0 - B_1 t$ est paire, ce qui prouve (2.2). ∎

Proposition 8.2.2. *On a, pour $n \geq 0$,*

$$B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k} \tag{2.3}$$

$$B_n(1 - x) = (-1)^n B_n(x)$$

et pour $n \geq 1$,

$$B_n'(x) = nB_{n-1}(x)$$

$$B_n(1 + x) - B_n(x) = nx^{n-1} \tag{2.4}$$

La formule (2.3) montre que les polynômes de Bernoulli sont bien des polynômes.

Preuve. Posons

$$\Phi(x,t) = \frac{te^{xt}}{e^t - 1}$$

On a

$$\sum_{n=0}^{\infty} B_n(x)\frac{t^n}{n!} = \Phi(x,t) = e^{xt}\Phi(0,t) = e^{xt}\sum_{n=0}^{\infty} B_n \frac{t^n}{n!}$$

$$= \sum_{n=0}^{\infty}\left(\sum_{k=0}^{n}\binom{n}{k}B_k x^{n-k}\right)\frac{t^n}{n!}$$

ce qui prouve (2.3). Ensuite, on a $\Phi(1-x,t) = \Phi(x,-t)$, d'où la deuxième relation. La troisième découle par identification de $\partial\Phi/\partial x = t\Phi$. Enfin, on a

$$\Phi(1+x,t) - \Phi(x,t) = te^{xt} = \sum_{n=1}^{\infty} nx^{n-1}\frac{t^n}{n!}$$

ce qui prouve (2.4). ∎

Corollaire 8.2.3. *Pour $m \geq 1$ et $n \geq 1$, on a*

$$\sum_{k=1}^{m} k^n = \frac{B_{n+1}(m+1) - B_{n+1}}{n+1}$$

Preuve. Sommons la relation (2.4), pour $x = 0,\ldots,m$. Les termes des membres gauches successifs se compensent, et on obtient

$$B_n(1+m) - B_n = n\sum_{k=0}^{m} k^{n-1}$$

ce qui donne la formule cherchée. ∎

8.2.3 Programme : nombres de Bernoulli

Les programmes de cette section utilisent la bibliothèque de manipulation des nombres rationnels donnée en annexe.

Pour le calcul des nombres de Bernoulli, on utilise la formule

$$B_n = \sum_{k=0}^{n} \binom{n}{k} B_k \qquad (n \geq 2)$$

sous la forme

$$B_{n-1} = -\frac{1}{n}\sum_{k=0}^{n-2} \binom{n}{k} B_k$$

Il suffit d'évaluer cette expression, pour $n \geq 3$ et $n-1$ pair, puisque les nombres de Bernoulli d'indice impair sont nuls.

On range la suite des nombres de Bernoulli ainsi calculée dans un tableau de type `SuiteRat` déclaré comme suit :

```
TYPE
   SuiteRat = ARRAY[0..LongueurMax] OF rat;
```

où `LongueurMax` est une constante de taille appropriée. Voici la procédure cherchée :

```
PROCEDURE NombresDeBernoulli (nn: integer; VAR B: SuiteRat);
   Calcule les nombres de Bernoulli B[n] = B_n pour 0 ≤ n ≤ nn.
   VAR
      k, n: integer;
      q, s: rat;
   BEGIN
      FOR n := 2 TO nn DO                  Initialisation :
         B[n] := RatZero;                  B_n = 0 pour n ≥ 2;
      B[0] := RatUn;                       B_0 = 1;
      FaireRat(-1, 2, B[1]);               B_1 = -1/2.
      FOR n := 3 TO nn + 1 DO
         IF odd(n) THEN BEGIN              Calcul de B_{n-1} pour n - 1 pair :
            s := RatZero;                  s := 0
            FOR k := 0 TO n - 2 DO BEGIN
               RatParEntier(B[k], binomial(n, k), q);
               RatPlusRat(s, q, s)         s := s + (n k) B_k
            END;
            RatSurEntier(s, n, s);
            RatOppose(s, B[n - 1])         B_{n-1} := -s/n
         END;
   END; { "NombresDeBernoulli" }
```

Le calcul donne les résultats suivants :

```
Nombres de Bernoulli :
B_0 = 1
B_1 = -1/2
B_2 = 1/6
B_3 = 0
B_4 = -1/30
B_5 = 0
B_6 = 1/42
B_7 = 0
B_8 = -1/30
B_9 = 0
B_10 = 5/66
B_11 = 0
B_12 = -691/2730
B_13 = 0
```

Les polynômes de Bernoulli se calculent en utilisant la formule qui les définit. On a besoin de polynômes à coefficients rationnels. Leur type est déclaré par :

```
TYPE
   PolRat = ARRAY[0..DegreMax] OF rat;
   SuitePolRat = ARRAY[0..LongueurMax] OF PolRat;
```

où `DegreMax` est une constante de taille appropriée. La procédure cherchée s'écrit alors comme suit :

```
PROCEDURE PolynomesDeBernoulli (nn: integer; VAR B: SuiteRat;
   VAR PolB: SuitePolRat);
   Calcule dans PolB une table des polynômes de Bernoulli.
   VAR
      k, n: integer;
   BEGIN
      PolB[0][0]:=B[0];                    B_0 = 1
      PolB[1][0]:=B[1];
      PolB[1][1]:=B[0];                    B_1 = -1/2 + X
      FOR n := 2 TO nn DO BEGIN
         FOR k := 0 TO n DO                (n k) B_k
            RatParEntier(B[k], binomial(n, k), PolB[n][n - k]);
      END
   END; { de "PolynomesDeBernoulli" }
```

On obtient, en adaptant la procédure d'impression des polynômes à coefficients réels donnée dans l'annexe A, les résultats suivants :

```
B_0(X) = 1
B_1(X) =  X - 1/2
B_2(X) =  X^2 - X + 1/6
B_3(X) =  X^3 - 3/2 X^2 + 1/2 X
B_4(X) =  X^4 - 2 X^3 + X^2 - 1/30
B_5(X) =  X^5 - 5/2 X^4 + 5/3 X^3 - 1/6 X
B_6(X) =  X^6 - 3 X^5 + 5/2 X^4 - 1/2 X^2 + 1/42
B_7(X) =  X^7 - 7/2 X^6 + 7/2 X^5 - 7/6 X^3 + 1/6 X
B_8(X) =  X^8 - 4 X^7 + 14/3 X^6 - 7/3 X^4 + 2/3 X^2 - 1/30
B_9(X) =  X^9 - 9/2 X^8 + 6 X^7 - 21/5 X^5 + 2 X^3 - 3/10 X
B_10(X) =  X^10 - 5 X^9 + 15/2 X^8 - 7 X^6 + 5 X^4 - 3/2 X^2 + 5/66
B_11(X) =  X^11 - 11/2 X^10 + 55/6 X^9 - 11 X^7 + 11 X^5 - 11/2 X^3 + 5/6 X
B_12(X) =  X^12 - 6 X^11 + 11 X^10 - 33/2 X^8 + 22 X^6 - 33/2 X^4 + 5 X^2
            - 691/2730
B_13(X) =  X^13 - 13/2 X^12 + 13 X^11 - 143/6 X^9 + 286/7 X^7 - 429/10 X^5
            + 65/3 X^3 - 691/210 X
```

Pour établir le «formulaire», on utilise l'expression $B_{n+1}(1+x) = (n+1)x^n + B_{n+1}(x)$ pour écrire

$$\sum_{k=1}^{m} k^n = m^n + \frac{B_{n+1}(m) - B_{n+1}}{n+1}$$

Il suffit donc d'annuler le terme constant de $B_{n+1}(X)$, de diviser les autres coefficients par $n+1$ et d'incrémenter de 1 le coefficient de X^n. Ceci est réalisé dans la procédure suivante :

```
PROCEDURE SommePuissancesNiemes (n: integer; VAR p, q: PolRat);
   VAR
      k: integer;
   BEGIN
```

```
      q := p;                                 Partant de p, de degré n + 1,
      q[0] := RatZero;                        on annule le terme constant,
      FOR k := 1 TO n + 1 DO BEGIN
         RatSurEntier(q[k], n + 1, q[k]);     on divise par n + 1,
      END;
      RatPlusRat(RatUn, q[n], q[n])           et on ajoute 1 au coefficient de X^n.
   END; { de "SommePuissancesNiemes" }
```

Cette procédure est mise en œuvre par une procédure qui calcule le tableau cherché :

```
PROCEDURE SommeDesPuissances (nn: integer;
      VAR PolB, SommePuissances: SuitePolRat);
   VAR
      n: integer;
   BEGIN
      FOR n := 1 TO nn DO
         SommePuissancesNiemes(n, PolB[n + 1], SommePuissances[n]);
   END; { de "SommeDesPuissances" }
```

Voici le résultat. Chaque formule donne l'expression de la somme des puissances n-ièmes jusqu'à X.

```
Somme des puissances
Ordre 1 : 1/2 X^2 + 1/2 X
Ordre 2 : 1/3 X^3 + 1/2 X^2 + 1/6 X
Ordre 3 : 1/4 X^4 + 1/2 X^3 + 1/4 X^2
Ordre 4 : 1/5 X^5 + 1/2 X^4 + 1/3 X^3 - 1/30 X
Ordre 5 : 1/6 X^6 + 1/2 X^5 + 5/12 X^4 - 1/12 X^2
Ordre 6 : 1/7 X^7 + 1/2 X^6 + 1/2 X^5 - 1/6 X^3 + 1/42 X
Ordre 7 : 1/8 X^8 + 1/2 X^7 + 7/12 X^6 - 7/24 X^4 + 1/12 X^2
Ordre 8 : 1/9 X^9 + 1/2 X^8 + 2/3 X^7 - 7/15 X^5 + 2/9 X^3 - 1/30 X
Ordre 9 : 1/10 X^10 + 1/2 X^9 + 3/4 X^8 - 7/10 X^6 + 1/2 X^4 - 3/20
X^2
```

8.3 Partitions d'entiers

8.3.1 Enoncé : partitions d'entiers

Une *partition* d'un entier positif n est une suite $a = (n_1, \ldots, n_s)$ d'entiers tels que $n_1 \geq \ldots \geq n_s > 0$ et $n_1 + \cdots + n_s = n$. Chaque n_i $(1 \leq i \leq s)$ est une *part* de a, et s est le *nombre de parts* de la partition.

Les partitions d'un entier n sont ordonnées comme suit : si $a = (n_1, \ldots, n_s)$ et $a' = (n'_1, \ldots, n'_{s'})$ sont deux partitions de n, on pose $a < a'$ si et seulement s'il existe un entier $i \leq s, s'$ tel que $n_j = n'_j$ pour $j = 1 \ldots i-1$ et $n_i < n'_i$.

1.– a) Ecrire une procédure qui prend en argument deux entiers positifs n et m et qui affiche en ordre décroissant toutes les partitions de n dont toutes les parts sont inférieures ou égales à m (on pourra supposer que $n \leq 20$).

Exemple numérique : $n = 9$, $m = 4$.

b) Ecrire une procédure qui prend en argument un entier positif n et qui affiche toutes les partitions de n.

Exemple numérique : $n = 9$.

On note $p(n)$ le nombre de partitions de $n > 0$ et on pose $p(0) = 1$. Pour $m > 0$ et $n > 0$, on note $p_m(n)$ le nombre de partitions de n en parts toutes inférieures ou égales à m, et on pose $p_m(0) = 1$.

2.– a) Démontrer que

$$p_m(n) = \begin{cases} p_{m-1}(n) & \text{si } m > n \\ p_{m-1}(n) + p_m(n-m) & \text{si } n \geq m > 1 \end{cases}$$

b) Ecrire une procédure qui prend en argument n et qui calcule $p(n)$. Ecrire cette procédure sans utiliser de tableau à double indice. Comme les entiers $p(n)$ croissent assez vite avec n, on représentera un entier $p(n)$ sous la forme d'un couple $(u(n), v(n))$ avec

$$p(n) = u(n)10^4 + v(n)$$

où $u(n)$ et $v(n)$ sont des entiers naturels inférieurs à 10^4 (ce qui permet de calculer $p(n)$ pour tout entier n tel que $p(n) < 10^8$).

Exemple numérique : $n = 60$.

3.– Démontrer que pour $0 \leq x < 1$ et pour $m > 0$, on a

$$\sum_{n=0}^{\infty} p_m(n)x^n = \prod_{k=1}^{m} \frac{1}{1-x^k}$$

4.– a) Démontrer que pour tout $0 \leq x < 1$, la suite

$$F_m(x) = \prod_{k=1}^{m} \frac{1}{1-x^k}$$

est convergente. On note $\prod_{k=1}^{\infty} \frac{1}{1-x^k}$ sa limite.

b) Démontrer que

$$\sum_{n=0}^{\infty} p(n)x^n = \prod_{k=1}^{\infty} \frac{1}{1-x^k}$$

5.– Ecrire une procédure qui affiche toutes les partitions de n en parts distinctes.

Exemple numérique : $n = 11$.

6.– Ecrire une procédure qui affiche toutes les partitions de n en parts distinctes et toutes impaires.

Exemple numérique : $n = 30$.

7.– Démontrer que $p_m(n)$ est égal au nombre de partitions de n en au plus m parts.

8.– Démontrer que le nombre de partitions de n en parts toutes distinctes est égal au nombre de partitions de n en parts toutes impaires.

9.– On définit, pour $0 \leq x < 1$ et pour $y \neq 0$ réel,

$$P(x,y) = \prod_{n=1}^{\infty}(1-x^{2n})(1+x^{2n-1}y)(1+x^{2n-1}y^{-1})$$

On admet que

$$P(x,y) = 1 + \sum_{n=1}^{\infty} x^{n^2}(y^n + y^{-n})$$

a) Démontrer que

$$\prod_{n=1}^{\infty}(1-x^n) = 1 + \sum_{n=1}^{\infty}(-1)^n \left(x^{\frac{1}{2}n(3n+1)} + x^{\frac{1}{2}n(3n-1)}\right)$$

b) On note $E(n)$ le nombre de partitions de n en un nombre pair de parts toutes distinctes, et $U(n)$ le nombre de partitions de n en un nombre impair de parts toutes distinctes. Démontrer que $E(n) = U(n)$ sauf si n est un entier de la forme $n = \frac{1}{2}k(3k\pm1)$, auquel cas $E(n) - U(n) = (-1)^k$.

8.3.2 Solution : partitions d'entiers

Une *partition* d'un entier positif n est une suite $a = (n_1, \ldots, n_s)$ d'entiers tels que $n_1 \geq \ldots \geq n_s > 0$ et $n_1 + \cdots + n_s = n$. Chaque n_i $(1 \leq i \leq s)$ est une *part* de a, et s est le *nombre de parts* de la partition.

On note $p(n)$ le nombre de partitions de $n > 0$ et on pose $p(0) = 1$. Pour $m > 0$ et $n > 0$, on note $p_m(n)$ le nombre de partitions de n en parts toutes inférieures ou égales à m, et on pose $p_m(0) = 1$.

On pose

$$F_m(x) = \prod_{k=1}^{m} \frac{1}{1-x^k} \qquad F(x) = \prod_{k=1}^{\infty} \frac{1}{1-x^k} = \lim_{m\to\infty} F_m(x)$$

L'existence de cette limite sera justifiée ultérieurement.

Proposition 8.3.1. *Pour $n \geq 0$ et $m \geq 1$, on a*

$$p_m(n) = \begin{cases} p_{m-1}(n) & \text{si } m > n \\ p_{m-1}(n) + p_m(n-m) & \text{si } n \geq m > 1 \end{cases}$$

Preuve. Si $m > n$, alors une partition dont les parts sont $\leq m$, est une partition dont les parts sont $\leq m-1$, ce qui donne la première alternative ; si $m \leq n$, une partition peut soit commencer avec une part égale à m, puis continuer avec une partition de ce qui reste, à savoir de $n-m$, soit n'être constituée que de parts toutes $\leq m-1$. ∎

PROPOSITION 8.3.2. *Pour $0 \leq x < 1$ et pour $m > 0$, on a*

$$\sum_{n=0}^{\infty} p_m(n)x^n = \prod_{k=1}^{m} \frac{1}{1-x^k}$$

Preuve. Le membre de droite est le produit d'un nombre fini de séries absolument convergentes, donc converge absolument. Il s'écrit

$$\prod_{k=1}^{m} \sum_{n=0}^{\infty} x^{kn} = \sum_{n_1,\ldots,n_m \geq 0} x^{n_1+2n_2+\cdots+mn_m}$$

En regroupant les termes, le coefficient de x^n est

$$\mathrm{Card}\{(n_1,\ldots,n_m) \mid n_1 + 2n_2 + \cdots + mn_m = n\}$$

et ce cardinal est $p_m(n)$. ∎

Notons que la proposition 8.3.1 en est une conséquence immédiate, puisque

$$F_m(x) = F_{m-1}(x) + x^m F_m(x)$$

PROPOSITION 8.3.3. *Pour $0 \leq x < 1$, on a*

$$\sum_{n=0}^{\infty} p(n)x^n = \prod_{k=1}^{\infty} \frac{1}{1-x^k}$$

Preuve. Le produit infini converge absolument car, en passant au logarithme, on a la série de terme général $-\log(1-x^k)$ qui est équivalente à x^k. De plus, pour chaque x, $0 \leq x < 1$, la suite $(F_m(x))_{m\geq 1}$ est croissante, et $F_m(x) \leq F(x)$ pour tout x fixé et pour tout m.

Comme $p_m(n) = p(n)$ pour $n \leq m$, on a, pour $0 \leq x < 1$,

$$\sum_{n=0}^{m} p(n)x^n < F_m(x) < F(x)$$

donc la série $\sum p(n)x^n$ converge. De plus, comme $p_m(n) \leq p(n)$, on a

$$\sum_{n=0}^{\infty} p_m(n)x^n \leq \sum_{n=0}^{\infty} p(n)x^n \leq F(x)$$

fonction génératrice	nombre de partitions de n dont les parts sont
$\displaystyle\prod_{m=1}^{\infty} \frac{1}{1-x^{2m-1}}$	impaires
$\displaystyle\prod_{m=1}^{\infty} \frac{1}{1-x^{2m}}$	paires
$\displaystyle\prod_{m=1}^{\infty} \frac{1}{1-x^{m^2}}$	des carrés
$\displaystyle\prod_{p} \frac{1}{1-x^{p}}$	des nombres premiers
$\displaystyle\prod_{m=1}^{\infty} 1+x^{m}$	distinctes
$\displaystyle\prod_{m=1}^{\infty} 1+x^{2m-1}$	distinctes et impaires
$\displaystyle\prod_{m=1}^{\infty} 1+x^{2m}$	distinctes et paires
$\displaystyle\prod_{m=1}^{\infty} 1+x^{m^2}$	des carrés distincts
$\displaystyle\prod_{p} 1+x^{p}$	des nombres premiers distincts

montrant que, pour chaque x fixé, la série $\sum p_m(x)x^n$ converge uniformément en m. Lorsque $m \to \infty$, on obtient

$$F(x) = \lim_{m\to\infty} F_m(x) = \lim_{m\to\infty} \sum_{n=0}^{\infty} p_m(n)x^n = \sum_{n=0}^{\infty} \lim_{m\to\infty} p_m(n)x^n = \sum_{n=0}^{\infty} p(n)x^n$$

ce qui prouve l'identité. ∎

Par des méthodes en tout point similaires, on peut obtenir les séries génératrices de nombreuses familles de partitions. Quelques exemples sont regroupés dans le tableau ci-dessus (où l'indice de sommation p signifie une sommation sur les nombres premiers).

PROPOSITION 8.3.4. *Le nombre de partitions de n en exactement m parts est égal au nombre de partitions de n dont la plus grande part est m.*

Cette proposition s'illustre par les diagrammes de Ferrer. Le *diagramme de Ferrer* d'une partition $a = (n_1, \ldots, n_s)$ est l'ensemble des points

$$\{(i, j) \mid 1 \leq i \leq s,\ 1 \leq j \leq n_i\}$$

Ce diagramme se prête à une représentation graphique. Ainsi, la partition $(6, 3, 3, 2, 1)$ de 15 est représentée par le diagramme de Ferrer composé des cinq lignes :

$$\begin{array}{llllll}
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & & & \\
\bullet & \bullet & \bullet & & & \\
\bullet & \bullet & & & & \\
\bullet & & & & &
\end{array}$$

Si on lit ce diagramme verticalement, on obtient une autre partition de 15, (appelée la partition *conjuguée*) à savoir $(5, 4, 3, 1, 1, 1)$ dont la plus grande part est 5. Cette observation conduit à la preuve formelle.

Preuve. Soit $a = (n_1, \ldots, n_m)$ une partition de n en m parts. Posons

$$m_j = \mathrm{Card}\{i \mid j \leq n_i\}$$

Alors $m_j \geq 1$ pour $j = 1, \ldots, n_1$. La suite $a^* = (m_1, \ldots, m_{n_1})$ est une partition de n car $m_1 + \cdots + m_{n_1} = n$. De plus, on a $i \leq m_j$ si et seulement si $j \leq n_j$, ce qui montre que l'application $a \mapsto a^*$ est bijective. La plus grande part de a^* est $m_1 = m$. ∎

Nous prouvons maintenant le résultat suivant, connu sous le nom de théorème des nombres pentagonaux d'Euler.

Théorème 8.3.5 (Euler). *Pour $0 \leq x < 1$, on a*

$$\prod_{n=1}^{\infty}(1 - x^n) = 1 + \sum_{n=1}^{\infty}(-1)^n \left(x^{\frac{1}{2}n(3n+1)} + x^{\frac{1}{2}n(3n-1)}\right)$$

Nous donnons d'abord une preuve directe du théorème; nous le dériverons plus loin d'un résultat plus général. Posons $P_0 = G_0 = 1$ et, pour $n \geq 1$,

$$P_n = \prod_{r=1}^{n}(1 - x^r) \qquad G_n = \sum_{r=0}^{n}(-1)^r \frac{P_n}{P_r} x^{rn+g(r)}$$

où $g(r) = r(r+1)/2$. Posons aussi, pour plus de commodité,

$$\omega(n) = n(3n-1)/2$$

Lemme 8.3.6. *Pour $n \geq 1$, on a $G_n - G_{n-1} = (-1)^n(x^{\omega(n)} + x^{\omega(-n)})$.*

Preuve. Par récurrence sur n. On a $G_1 - G_0 = -x - x^2$, ce qui amorce la récurrence. Ensuite,

$$\begin{aligned} G_n - G_{n-1} &= (-1)^n x^{n^2+g(n)} + (1-x^n) \sum_{r=0}^{n-1} (-1)^r \frac{P_{n-1}}{P_r} x^{rn+g(r)} \\ &\quad - \sum_{r=0}^{n-1} (-1)^r \frac{P_{n-1}}{P_r} x^{r(n-1)+g(r)} \\ &= (-1)^n x^{n^2+g(n)} + \sum_{r=0}^{n-1} (-1)^r \frac{P_{n-1}}{P_r} (x^r - 1) x^{r(n-1)+g(r)} \\ &\quad -x^n \sum_{r=0}^{n-1} (-1)^r \frac{P_{n-1}}{P_r} x^{rn+g(r)} \end{aligned}$$

Le deuxième terme de cette somme s'écrit

$$-\sum_{r=1}^{n-1} (-1)^r \frac{P_{n-1}}{P_{r-1}} x^{r(n-1)+g(r)} = \sum_{r=1}^{n-1} (-1)^{r-1} \frac{P_{n-1}}{P_{r-1}} x^{rn+g(r-1)}$$

parce que $r(n-1) + g(r) = rn + g(r-1)$. Le troisième terme s'écrit

$$\sum_{r=0}^{n-1} (-1)^r \frac{P_{n-1}}{P_r} x^{n(r+1)+g(r)} = \sum_{r=1}^{n} (-1)^{r-1} \frac{P_{n-1}}{P_{r-1}} x^{rn+g(r-1)}$$

Ces deux sommes se compensent donc au terme extrême près, et par conséquent

$$G_n - G_{n-1} = (-1)^n x^{n^2+g(n)} + (-1)^n x^{n^2+g(n-1)}$$

Or $n^2 + g(n) = \omega(-n)$ et $n^2 + g(n-1) = \omega(n)$, d'où le lemme. ∎

Preuve du théorème 8.3.5. Le produit infini $\prod_{m=1}^{\infty}(1-x^m)$ converge pour $0 \leq x < 1$, donc $\lim_{n\to\infty} P_n = \prod_{m=1}^{\infty}(1-x^m)$. Posons $S_0 = 1$ et

$$S_n = 1 + \sum_{r=1}^{n} (-1)^r (x^{\omega(r)} + x^{\omega(-r)})$$

On a $S_0 = G_0$ et, en vertu du lemme,

$$G_n - G_{n-1} = S_n - S_{n-1}$$

donc $S_n = G_n$ pour tout $n \geq 0$. On a

$$G_n = P_n + \sum_{r=1}^{n} (-1)^r \frac{P_n}{P_r} x^{rn+g(r)}$$

Par ailleurs, $0 < P_n/P_r \leq 1$ parce que $0 \leq x < 1$. Comme $x^{rn+g(r)} \leq x^{n+1}$ pour $1 \leq r \leq n$, on a $G_n \leq P_n + nx^{n+1}$. Il en résulte

$$|P_n - S_n| = |P_n - G_n| \leq nx^{n+1}$$

ce qui prouve l'identité d'Euler. ■

Corollaire 8.3.7. *Soient $E(n)$ le nombre de partitions de n en un nombre pair de parts toutes distinctes et $U(n)$ le nombre de partitions de n en un nombre impair de parts toutes distinctes. Alors $E(n) = U(n)$ sauf si n est un entier de la forme $n = \frac{1}{2}k(3k \pm 1)$, auquel cas $E(n) - U(n) = (-1)^k$.*

Preuve. Le coefficient de x^n dans la série

$$\prod_{n=1}^{\infty}(1 - x^n)$$

est $\sum(-1)^\nu$, où la somme porte sur les partitions en parts distinctes et où ν est le nombre de parts de la partition. Mais

$$\sum(-1)^\nu = E(n) - U(n)$$

d'où le résultat par l'identité d'Euler. ■

Proposition 8.3.8. *Le nombre de partitions de n en parts toutes distinctes est égal au nombre de partitions de n en parts toutes impaires.*

Preuve. Il s'agit de prouver que pour $0 \leq x < 1$,

$$\prod_{n=1}^{\infty}(1 + x^n) = \prod_{n=0}^{\infty} \frac{1}{1 - x^{2n+1}}$$

Les produits sont absolument convergents. Posons

$$P(x) = \prod_{n=1}^{\infty}(1 + x^n) \qquad Q(x) = \prod_{n=0}^{\infty}(1 - x^{2n+1})$$

Alors

$$\begin{aligned} P(x)Q(x) &= \prod_{n=1}^{\infty}(1 + x^{2n}) \prod_{n=0}^{\infty}(1 + x^{2n+1}) \prod_{n=0}^{\infty}(1 - x^{2n+1}) \\ &= \prod_{n=1}^{\infty}(1 + (x^2)^n) \prod_{n=0}^{\infty}(1 - (x^2)^{2n+1}) \\ &= P(x^2)Q(x^2) \end{aligned}$$

Par conséquent, $P(x)Q(x) = P(x^{2^k})Q(x^{2^k})$ pour tout $k \geq 0$. Lorsque k tend vers l'infini, ce produit tend vers 1, d'où le résultat. ∎

L'identité d'Euler est une conséquence d'un théorème remarquable connu sous le nom d'*identité du triple produit de Jacobi* :

THÉORÈME 8.3.9. *Soient x et z des nombres complexes avec $|x| < 1$ et $z \neq 0$; alors on a*

$$\prod_{n=1}^{\infty}(1-x^{2n})(1+x^{2n-1}z)(1+x^{2n-1}z^{-1}) = 1 + \sum_{n=1}^{\infty} x^{n^2}(z^n + z^{-n}) \tag{3.1}$$

Preuve. La condition $|x| < 1$ assure la convergence absolue de chacun des trois produits $\prod(1-x^{2n})$, $\prod(1+x^{2n-1}z)$, $\prod(1+x^{2n-1}z^{-1})$ et de la série du membre droit de (3.1). De plus, pour chaque x fixé avec $|x| < 1$, la série et les produits convergent uniformément sur des parties compactes de $\mathbb{C}$ ne contenant pas 0, de sorte que chaque membre de l'équation (3.1) est une fonction analytique de z pour $z \neq 0$. Pour un $z \neq 0$ fixé, la série et les produits convergent aussi uniformément pour $|x| \leq r < 1$ et représentent donc des fonctions analytiques de x dans le disque $|x| < 1$.

Pour démontrer (3.1), nous fixons x et définissons $F(z)$ pour $z \neq 0$ par

$$F(z) = \prod_{n=1}^{\infty}(1+x^{2n-1}z)(1+x^{2n-1}z^{-1}) \tag{3.2}$$

Nous vérifions d'abord l'équation fonctionnelle

$$xzF(x^2z) = F(z)$$

En effet, de (3.2), on a

$$\begin{aligned} F(x^2z) &= \prod_{n=1}^{\infty}(1+x^{2n+1}z)(1+x^{2n-3}z^{-1}) \\ &= \prod_{m=2}^{\infty}(1+x^{2m-1}z)\prod_{r=0}^{\infty}(1+x^{2r-1}z^{-1}) \end{aligned}$$

Comme $xz = (1+xz)/(1+x^{-1}z^{-1})$, la multiplication de la dernière équation par xz donne l'équation fonctionnelle.

Soit maintenant $G(z)$ le membre gauche de (3.1), de sorte que

$$G(z) = F(z)\prod_{n=1}^{\infty}(1-x^{2n})$$

Alors $G(z)$ vérifie la même équation fonctionnelle. De plus, $G(z)$ est analytique pour $z \neq 0$ et admet le développement en série de Laurent

$$G(z) = \sum_{m=-\infty}^{\infty} a_m z^m \tag{3.3}$$

où $a_{-m} = a_m$ parce que $G(z) = G(z^{-1})$. Bien entendu, les coefficients a_m dépendent de x. En reportant l'équation (3.3) dans l'équation fonctionnelle, on obtient pour les coefficients a_m la formule de récurrence

$$a_{m+1} = x^{2m+1} a_m$$

qui, en itérant, donne

$$a_m = a_0 x^{m^2} \qquad (m \geq 0)$$

parce que $1 + 3 + \cdots + (m-1) = m^2$. Notons que cette égalité vaut aussi pour $m < 0$. Ainsi, l'équation (3.3) devient

$$G_x(z) = a_0(x) \sum_{m=-\infty}^{\infty} x^{m^2} z^m \tag{3.4}$$

où nous avons indiqué explicitement la dépendance de x. Il résulte de cette équation que $a_0(x)$ tend vers 1 lorsque x tend vers 0, et il reste à prouver que $a_0(x) = 1$ pour tout x. Nous allons montrer que

$$a_0(x) = a_0(x^4) \tag{3.5}$$

Le résultat s'ensuit, parce que l'on obtient $a_0(x) = a_0(x^{4^k})$ pour tout $k \geq 0$ et que x^{4^k} tend vers 0 lorsque k tend vers l'infini.

Pour $z = i$, l'équation (3.4) donne

$$\frac{G_x(i)}{a_0(x)} = \sum_{m=-\infty}^{\infty} x^{m^2} i^m = \sum_{m=-\infty}^{\infty} (-1)^m x^{(2m)^2}$$

car $i^m = -i^{-m}$ pour m impair, et par conséquent

$$\frac{G_x(i)}{a_0(x)} = \frac{G_{x^4}(-1)}{a_0(x^4)}$$

Nous vérifions que $G_x(i) = G_{x^4}(-1)$. Pour cela, observons d'abord que

$$F(i) = \prod_{n=1}^{\infty} (1 + x^{4n-2})$$

Comme tout nombre pair est de la forme $4n$ ou $4n-2$, on a

$$\prod_{n=1}^{\infty} (1 - x^{2n}) = \prod_{n=1}^{\infty} (1 - x^{4n})(1 - x^{4n-2})$$

de sorte que

$$\begin{aligned} G_x(i) &= \prod_{n=1}^{\infty} (1 - x^{4n})(1 - x^{8n-4}) \\ &= \prod_{n=1}^{\infty} (1 - x^{8n})(1 - x^{8n-4})(1 - x^{8n-4}) = G_{x^4}(-1) \end{aligned}$$

ce qui achève la démonstration. ∎

Comme annoncé, l'identité d'Euler (Théorème 8.3.5) s'obtient comme un simple corollaire de l'identité de Jacobi, en faisant la substitution de x par $x^{3/2}$ et de z par $-x^{-1/2}$ dans l'équation (3.1). Il vient :

$$\begin{aligned}\prod_{n=1}^{\infty}&(1-x^{3n})(1-x^{\frac{3}{2}(2n-1)-\frac{1}{2}})(1-x^{\frac{3}{2}(2n-1)+\frac{1}{2}})\\ &=\prod_{n=1}^{\infty}(1-x^{3n})(1-x^{3n-2})(1-x^{3n-1})=\prod_{n=1}^{\infty}(1-x^n)\end{aligned}$$

et par ailleurs, le terme général du membre droit de l'identité d'Euler s'écrit

$$\begin{aligned}(x^{3/2})^{n^2}\Big((-x^{-1/2})^n+(-x^{-1/2})^{-n}\Big)&=(-1)^n x^{\frac{3}{2}n^2}(x^{n/2}+x^{-n/2})\\ &=(-1)(x^{\frac{1}{2}n(3n+1)}+x^{\frac{1}{2}n(3n-1)})\end{aligned}$$

L'identité de Jacobi a de nombreuses autres conséquences intéressantes que nous n'évoquerons pas ici.

8.3.3 Programme : partitions d'entiers

Une partition est une suite finie d'entiers. On utilisera, pour la représenter, le type `mot` défini dans le chapitre suivant. Ici, nous n'avons besoin que de la définition et de quelques procédures :

```
TYPE
   mot = ARRAY[0..LongueurMax] OF integer;
PROCEDURE EntrerMot (VAR u: mot; titre: texte);
PROCEDURE EcrireMot (VAR u: mot; titre: texte);
FUNCTION Longueur (VAR u: mot): integer;
PROCEDURE FixerLongueur (VAR u: mot; n: integer);
```

La constante `LongueurMax` est fixée à une taille appropriée. La rédaction des procédures est expliquée au chapitre suivant.

Pour engendrer les partitions de n (ou des partitions restreintes de n), il faut connaître la première partition, reconnaître la dernière et savoir calculer la partition suivante dans l'ordre lexicographique inverse. La première partition de n est bien sûr la partition (n), et la dernière est $(1, 1, \dots, 1)$. C'est la seule partition dont la longueur est égale à n, donc elle est facile à reconnaître. Pour calculer la partition suivante d'une partition $a = (a_1, \dots, a_s)$, on cherche la suite $b = (a_j, \dots, a_s)$, avec $a_j > a_{j+1} = \cdots = a_s$, et en posant $t = a_j + \cdots + a_s = a_j + s - j + 1$, on détermine la première partition de t dont la plus grande part est $a_j - 1$. Cette partition vient se substituer à b.

Dans le cas où l'on cherche les partitions de n en parts majorées par m, seule la première partition change. Voici une réalisation :

```
PROCEDURE PremierePartition (VAR a: Mot; m, n: integer);
   Calcule dans a la première partition de n dont toutes les parts sont majorées par m.
   VAR
      k, s: integer;
   BEGIN
      s := n DIV m;
      FOR k := 1 TO s DO a[k] := m;            Des parts de taille m.
      IF (n MOD m) > 0 THEN BEGIN              Une dernière part, plus petite.
         s := s + 1; a[s] := n MOD m
      END;
      FixerLongueur(a, s)                      La longueur de la première partition.
   END; { de "PremierePartition" }
```

Voici la procédure de calcul de la partition suivante :

```
PROCEDURE PartitionSuivante (VAR a: Mot; n: integer; VAR derniere: boolean);
   Calcule dans le tableau a la partition suivante de n, si elle existe. Dans la négative, la
   variable booléenne devient vraie.
   VAR
      m, t, j: integer;
   BEGIN
      derniere := Longueur(a) = n;             Caractérisation simple.
       IF NOT derniere THEN BEGIN
         j := Longueur(a);                     Calcul du nombre de parts égales à 1.
         WHILE a[j] = 1 DO j := j - 1;
         t := Longueur(a) - j + a[j];          C'est le total à redistribuer.
         m := a[j] - 1;                        En parts majorées par m.
         WHILE t >= m DO BEGIN                 Tant que possible, des parts de taille m.
            a[j] := m;
            t := t - m;
            j := j + 1
         END;
         IF t > 0 THEN                         Une dernière part plus petite.
            a[j] := t
         ELSE
            j := j - 1;
      END;
      FixerLongueur(a, j)                      La longueur de la nouvelle partition.
   END; { de "PartitionSuivante" }
```

Ces deux procédures sont employées pour lister toutes les partitions, ou les partitions en parts majorées par m, dans les deux procédures que voici :

```
PROCEDURE ListerPartitions (n: integer);
   Les partitions de n.
   VAR
      a: Mot;
      derniere: boolean;
   BEGIN
```

```
      PremierePartition(a, n, n);
      REPEAT
         EcrireMot(a, '');
         PartitionSuivante(a, n, derniere)
      UNTIL derniere
   END; { de "ListerPartitions" }
PROCEDURE ListerPartitionsPartsMajorees (n, m: integer);
   Les partitions de n en parts majorées par m.
   VAR
      a: Mot;
      derniere: boolean;
   BEGIN
      PremierePartition(a, m, n);
      REPEAT
         EcrireMot(a, '');
         PartitionSuivante(a, n, derniere)
      UNTIL derniere
   END; { de "ListerPartitionsPartsMajorees" }
```

Les partitions de 9 en parts majorées par 4 sont :

```
4 4 1
4 3 2
4 3 1 1
4 2 2 1
4 2 1 1 1
4 1 1 1 1 1
3 3 3
3 3 2 1
3 3 1 1 1
3 2 2 2
3 2 2 1 1
3 2 1 1 1 1
3 1 1 1 1 1 1
2 2 2 2 1
2 2 2 1 1 1
2 2 1 1 1 1 1
2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
```

Le calcul des partitions de n en part distinctes se fait sur le même schéma. La première partition est (n). Lors du calcul de la partition suivante, on cherche à déterminer la première partition en parts distinctes commençant par un entier donné. Une telle partition n'existe pas toujours. Plus précisément, on a le lemme suivant.

Lemme 8.3.10. *Il existe une partition de n en parts distinctes et de plus grande part m si et seulement si $m \leq n \leq m(m+1)/2$.*

Preuve. La condition est nécessaire, car si $a = (a_1, \ldots, a_s)$, avec $a_1 = m$, est une partition de n, alors $n = a_1 + \cdots + a_s \leq m + (m-1) + (m-s+1) \leq m(m+1)/2$. La réciproque se montre de la même manière. ∎

Nous traduisons ce lemme en une petite procédure qui teste la deuxième condition :

```
FUNCTION CompatiblePartsDistinctes (n, m: integer): boolean;
   Vraie s'il existe une partition de n en parts distinctes dont la plus grande part est m.
   BEGIN
      CompatiblePartsDistinctes := m * (1 + m) DIV 2 >= n
   END; { de "CompatiblePartsDistinctes" }
```

Cette procédure est ensuite utilisée pour calculer la partition suivante :

```
PROCEDURE PartitionSuivantePartsDistinctes (VAR a: Mot; n: integer;
   VAR derniere: boolean);
   Calcule dans le tableau a la partition suivante de n, si elle existe. Dans la négative, la
   variable booléenne devient vraie.
   VAR
      m, t, j: integer;
      possible: boolean;
   BEGIN
      j := Longueur(a);                      Le nombre de parts de a.
      t := 0;                                Calcul du total à redistribuer.
      possible := false;                     Pour l'instant ...
      WHILE (j >= 1) AND (NOT possible) DO BEGIN
         t := t + a[j];                      Total à redistribuer
                                             en parts majorées par a[j] − 1.
         possible := CompatiblePartsDistinctes(t, a[j] - 1);   Est-ce possible ?
         IF NOT possible THEN j := j - 1
      END;{ "while" possible }
      derniere := NOT possible;
      IF possible THEN BEGIN
         m := a[j] - 1;                      Part maximale.
         WHILE (t >= m) AND (m > 0) DO BEGIN
            a[j] := m;                       On fait des parts.
            t := t - m;                      Ce qui reste.
            m := m - 1;                      La taille des parts décroît.
            j := j + 1                       Le nombre de parts croît.
         END;
         IF t > 0 THEN                       Une dernière part, plus petite.
            a[j] := t
         else
            j := j - 1;
         FixerLongueur(a, j)                 Longueur de la nouvelle partition.
      END { c'est possible }
   END; { de "PartitionSuivantePartsDistinctes" }
```

A nouveau, une procédure d'énumération gère ces procédures :

```
PROCEDURE ListerPartitionsPartsDistinctes (n: integer);
```

```
VAR
   a: Mot;
   derniere: boolean;
BEGIN
   PremierePartition(a, n, n);
   REPEAT
      EcrireMot(a, '');
      PartitionSuivantePartsDistinctes(a, n, derniere)
   UNTIL derniere
END; { de "ListerPartitionsPartsDistinctes" }
```

Voici les résultats pour $n = 11$:

```
11
10 1
 9 2
 8 3
 8 2 1
 7 4
 7 3 1
 6 5
 6 4 1
 6 3 2
 5 4 2
 5 3 2 1
```

La génération des partitions distinctes et impaires, sur le même modèle, est plus compliquée, essentiellement parce que l'entier 2 n'a pas de telle partition. Par conséquent, certains débuts de partition ne peuvent pas être complétés. Par exemple, parmi les partitions de 30, la suite $(15, 13)$ ne peut pas être complétée. De même, un entier impair n n'admet pas de partition dont la plus grande part est $n - 2$. La situation est décrite par le lemme suivant.

LEMME 8.3.11. *Soient $n \geq m \geq 1$ des entiers, avec m impair. Il existe une partition de n en parts distinctes et impaires de plus grande part m si et seulement si*

$$m \leq n \leq \left(\frac{m+1}{2}\right)^2 \quad \text{et} \quad \left(\frac{m+1}{2}\right)^2 \neq n + 2 \quad \text{et} \quad m + 2 \neq n$$

Preuve. Posons $m = 2k - 1$. Soit $(a_1, \ldots, a_s)$ une partition de n en parts distinctes et impaires et $a_1 = m$. Posons $m = 2k-1$ et $b_i = a_{s+1-i}$ pour $1 \leq i \leq s$. On a évidemment $n \neq m + 2$. Par ailleurs, pour $i = 1, \ldots, s$,

$$2i - 1 \leq b_i \leq 2(k - s) + 2i - 1$$

d'où, en sommant sur i,

$$\sum_{i=1}^{s} 2i - 1 = s^2 \leq \sum_{i=1}^{s} b_i = n \leq 2s(k - s) + s^2 = 2sk - s^2 \leq k^2$$

Supposons que $k^2 = n + 2$. Alors $k^2 - 2 \leq 2sk - s^2$, soit $(k - s)^2 \leq 2$, ou encore $k = s$ ou $k = s + 1$. Or $s^2 \leq n = k^2 - 2$, donc $k \neq s$. Comme chaque b_i est impair, on a

$$n = \sum_{i=1}^{s} b_i \equiv s \pmod 2$$

et d'autre part $n = k^2 - 2 \equiv k \equiv s + 1 \bmod 2$, une contradiction.

Réciproquement, supposons les conditions vérifiées et posons $n' = n - m$, $m' = m - 2$. Si $m' > n'$, alors (m, n') ou $(m, n' - 1, 1)$ est une partition de n en parts impaires, selon que n' est impair ou pair. Les parts sont distinctes parce que $n' \neq 2$.

On peut donc supposer $m' \leq n'$. On a alors $n' \leq ((m' + 1)/2)^2$, $n' + 2 \neq ((m' + 1)/2)^2$, et on a également $m' + 2 \neq n'$ sauf si $n = 2m$. On peut donc conclure par récurrence pour $n \neq 2m$.

Si $n = 2m$, il existe une partition de m en parts impaires distinctes commençant par un entier strictement plus petit que m, à savoir $(m-4, 3, 1)$, d'où la partition $(m, m-4, 3, 1)$ de n, sauf si $m-4 \leq 3$, c'est-à-dire pour $m = 3, 5, 7$. Or ces cas ne peuvent pas se produire car, pour $m = 3, 5$, on a $n > \left(\frac{m+1}{2}\right)^2$ et, pour $m = 7$, on a $n + 2 = \left(\frac{m+1}{2}\right)^2$. ■

Voici une procédure qui teste une partie des conditions du lemme :

```
FUNCTION CompatiblePartsDistinctesImpaires (n, m: integer): boolean;
   Partie de la condition d'existence d'une partition de n en parts distinctes impaires dont
   la plus grande part est m. On suppose m ≤ n.
   VAR
      c: integer;
   BEGIN
      c := sqr((1 + m) DIV 2);
      CompatiblePartsDistinctesImpaires := (c >= n) AND (c <> n + 2)
   END; { de "CompatiblePartsDistinctesImpaires" }
```

Elle est utilisée dans la procédure suivante; le calcul de la partition suivante, lorsque l'on sait qu'elle existe, doit être mené avec soin.

```
PROCEDURE PartitionSuivantePartsDistinctesImpaires (VAR a: Mot; n: integer;
   VAR derniere: boolean);
   Calcule dans le tableau a la partition suivante de n, si elle existe. Dans la négative, la
   variable booléenne devient vraie.
   VAR
      m, t, j: integer;
      possible: boolean;
   BEGIN
      j := Longueur(a);                        Le nombre de parts de a.
      t := 0;                                  Calcul du total à redistribuer.
      possible := false;                       Pour l'instant ...
      WHILE (j >= 1) AND (NOT possible) DO BEGIN
         t := t + a[j];                        Total à redistribuer en parts
                                               majorées par a[j] − 2.
```

```
         possible := CompatiblePartsDistinctesImpaires(t, a[j] - 2);
         IF NOT possible THEN j := j - 1
      END;{ "while" possible }
      derniere := NOT possible;
      IF possible THEN BEGIN
         m := a[j] - 2;                        Part maximale.
         WHILE t > 0 DO BEGIN                  Répartition.
            WHILE t < m DO m := m - 2;         Ajustement.
            IF (t = m + 2) AND (m > 1) THEN    Pour qu'il ne reste pas la part 2.
               m := m - 2;
            a[j] := m;                         On fait des parts.
            t := t - m;                        Ce qui reste.
            m := m - 2;                        La taille des parts décroît.
            j := j + 1                         Le nombre de parts croît.
          END; { "while" }                     Fin de la répartition.
         FixerLongueur(a, j - 1)               Longueur de la nouvelle partition.
      END { c'est possible }
   END; { de "PartitionSuivantePartsDistinctesImpaires" }
```

Voici le résultat de cette procédure, utilisée pour lister les partitions de 30 en parts impaires et distinctes.

```
29  1
27  3
25  5
23  7
21  9
21  5  3  1
19 11
19  7  3  1
17 13
17  9  3  1
17  7  5  1
15 11  3  1
15  9  5  1
15  7  5  3
13 11  5  1
13  9  7  1
13  9  5  3
11  9  7  3
```

Evidemment, on peut s'épargner la peine de programmer tout cela : il suffit d'engendrer toutes les partitions de n et de ne conserver que celles qui conviennent. Mais ceci n'est pas très efficace : pour $n = 30$, on a $p(30) = 5604$, alors qu'il n'y a que 18 partitions en parts distinctes et impaires. Pour des valeurs de n plus grandes, les calculs deviennent prohibitifs.

L'emploi des procédures récursives est très avantageux, puisque les partitions se définissent de manière récurrente.

```
PROCEDURE LesPartitionsDistinctesImpaires (n: integer);
   Procédure calculant ces partitions de manière récursive.
   VAR
      a: mot;
   PROCEDURE Parts (p, n, m: integer);
   p est la première position libre dans le tableau a ;
   n est l'entier dont on calcule les partitions;
   m est la plus grande part d'une partition de n.
      BEGIN
         IF n = 0 THEN BEGIN                  Construction terminée.
            FixerLongueur(a, p - 1);
            EcrireMot(a, '')
         END
         ELSE IF (n = 2) OR (m < 0) THEN      Pas de partition!
         ELSE BEGIN
            IF n >= m THEN                    On prolonge la partition courante.
               BEGIN
               a[p] := m;
               Parts(p + 1, n - m, m - 2);    Tentative pour la terminer.
            END;
            IF sqr((m - 1) DIV 2) >= n THEN   Mais dans certains cas,
               Parts(p, n, m - 2)             il y en a peut-être d'autres.
         END
      END; { de "Parts" }
   BEGIN
      IF odd(n) THEN
         Parts(1, n, n)
      ELSE
         Parts(1, n, n + 1)
   END; { de "LesPartitionsDistinctesImpaires" }
```

Pour terminer, considérons le calcul du nombre de partitions de n. On utilise pour cela les formules $p_1(k) = 1$ $(1 \le k \le n)$ et

$$p_m(k) = \begin{cases} p_{m-1}(k) & \text{si } k > n \\ p_{m-1}(k) + p_m(k-m) & \text{si } k \ge m > 1 \end{cases}$$

et on évalue $p_m(k)$ pour $1 \le m \le n$ et $m \le k \le n$. Le résultat s'obtient parce que $p(n) = p_n(n)$. Voici une implémentation avec un seul tableau (dédoublé) contenant, pour chaque $m = 1, \ldots, n$, les nombres $p_m(k)$ pour $1 \le k \le n$:

```
PROCEDURE NombreDePartitions (n: integer; VAR alpha, beta: integer);
   Calcul du nombre p(n) de partitions par la formule p(n) = pn(n). Résultat sous la forme
   p(n) = α10^4 + β.
   CONST
      nmax = 100;                             Pour les p(k) avec k ≤ nmax.
      base = 10000;                           Base pour l'écriture des entiers.
```

```
VAR
   a, b: ARRAY[0..nmax] OF integer;
   k, m, s: integer;
BEGIN
   FOR k := 0 TO n DO BEGIN                Initialisation : p(k) = p_1(k) = 1.
      a[k] := 0;
      b[k] := 1
   END;
   FOR m := 2 TO n DO                      Calcul des p_m(k) par la formule
      FOR k := m TO n DO BEGIN             p_m(k) = p_{m-1}(k) + p_m(k - m).
         s := b[k] + b[k - m];
         b[k] := s MOD base;               Décomposition sur la base.
         a[k] := a[k] + a[k - m] + s DIV base;
      END;
   alpha := a[n];
   beta := b[n]
END; { de "NombreDePartitions" }
```

Quelques résultats numériques :

```
p(30) = 5604
p(60) = 966467
p(90) = 56634173
p(100) = 190569292
```

Notes bibliographiques

Pour la génération d'objets combinatoires, on peut consulter :

D. Stanton, D. White, *Constructive Combinatorics*, New York, Springer-Verlag, 1986. (Ce livre contient de nombreux programmes.)

Les livres traitant des partitions sont très nombreux. Le livre de Stanton et White contient de nombreuses preuves «bijectives». Nous nous sommes inspirés de :

G.H. Hardy, E.M. Wright, *An Introduction to the Theory of Numbers*, London, Oxford University Press, 1965.

T.M. Apostol, *Introduction to Analytic Number Theory*, New York, Springer-Verlag, 1976.

Les nombres et polynômes de Bernoulli interviennent souvent, par exemple dans la formule d'Euler-MacLaurin. Pour d'autres propriétés, on peut consulter :

L. Comtet, *Analyse combinatoire*, Paris, Presses Universitaires de France, 1970.

D.E. Knuth, *The Art of Computer Programming*, Vol. I, Reading, Addison-Wesley, 1968.

Des tables numériques, ainsi qu'un résumé des propriétés principales, se trouvent dans :

M. Abramowitz, I. Stegun, *Handbook of Mathematical Functions*, New York, Dover, 1964.

Chapitre 9

Combinatoire des mots

9.1 Terminologie

Soit A un ensemble. On appelle *mot* sur A toute suite finie

$$u = (u_1, \ldots, u_n)$$

où $n \geq 0$ et $u_i \in A$ pour $i = 1, \ldots, n$. L'entier n est la *longueur* de u, souvent notée $|u|$. Si $n = 0$, le mot est appelé le *mot vide*, et est noté 1 ou ε. Si

$$v = (v_1, \ldots, v_m)$$

est un autre mot, le *produit de concaténation* de u et v est le mot

$$uv = (u_1, \ldots, u_n, v_1, \ldots, v_m)$$

Tout mot étant le produit de concaténation de mots de longueur 1, on identifie les mots de longueur 1 et les éléments de A. On appelle alors A l'*alphabet*, et les éléments de A des *lettres*.

Si u, v et w sont des mots et $w = uv$, alors u est un *préfixe* et v est un *suffixe* de w. Si de plus $u \neq w$ (resp. $v \neq w$), alors u est un *préfixe propre* (resp. un *suffixe propre*) de w. Si u, v, v' et w sont des mots tels que $w = vuv'$, alors u est un *facteur* de w. Pour raccourcir l'écriture, nous dirons qu'un préfixe propre non vide d'un mot est un *début*, et qu'un suffixe propre non vide est une *fin*. Un *conjugué* d'un mot $u = (u_1, \ldots, u_n)$ est un mot de la forme $v = (u_i, \ldots, u_n, u_1, \ldots, u_{i-1})$ avec $2 \leq i \leq n$.

Dans les énoncés qui suivent, les lettres sont des entiers. Un mot de longueur n est donc représenté par un tableau d'entiers à $n + 1$ éléments, l'élément d'indice 0 contenant la longueur du mot. Voici un début de bibliothèque de manipulation de mots.

```
CONST
   LongueurMot = 30;
TYPE
   mot = ARRAY[0..LongueurMot] OF integer;
FUNCTION Longueur (VAR u: mot): integer;
PROCEDURE FixerLongueur (VAR u: mot; n: integer);
PROCEDURE EntrerMot (VAR u: mot; titre: texte);
PROCEDURE EcrireMot (VAR u: mot; titre: texte);
FUNCTION EstMotVide (VAR u: mot): boolean;
PROCEDURE Concatener (u, v: mot; VAR w: mot);
PROCEDURE LettreEnMot (VAR u: mot; x: integer);
```

Les procédures de gestion de la longueur du mot sont :

```
FUNCTION Longueur (VAR u: mot): integer;
   BEGIN
      Longueur := u[0]
   END; { de "Longueur" }

PROCEDURE FixerLongueur (VAR u: mot; n: integer);
   BEGIN
      u[0] := n
   END; { de "FixerLongueur" }
```

Les deux procédures suivantes sont écrites sur le modèle déjà rencontré plusieurs fois :

```
PROCEDURE EntrerMot (VAR u: mot; titre: texte);
   VAR
      i: integer;
   BEGIN
      write(titre);
      i := 0;
      WHILE NOT eoln DO BEGIN
         i := i + 1; read(u[i])
      END;
      FixerLongueur(u,i); readln
   END; { de "EntrerMot" }

PROCEDURE EcrireMot (VAR u: mot; titre: texte);
   VAR
      i: integer;
   BEGIN
      writeln; write(titre);
      IF EstMotVide(u) THEN write('-')              par exemple...
      ELSE
         FOR i := 1 TO Longueur(u) DO write(u[i] : precision);
   END; { de "EcrireMot" }
```

Rappelons que `precision` est une variable globale de notre environnement (voir l'annexe A). La procédure de concaténation de deux mots s'écrit :

```
PROCEDURE Concatener (u, v: mot; VAR w: mot);
   VAR
      i, n: integer;
   BEGIN
      w := u; n := Longueur(u);
      FOR i := 1 TO Longueur(v) DO w[i + n] := v[i];
      FixerLongueur(w, n + Longueur(v))
   END; { de "Concatener" }
```

Les deux procédures suivantes sont parfois utiles :

```
FUNCTION EstMotVide (VAR u: mot): boolean;
   BEGIN
      EstMotVide := Longueur(u) = 0
   END; { de "EstMotVide" }
PROCEDURE LettreEnMot (VAR u: mot; x: integer);
   BEGIN
      u[1] := x; FixerLongueur(u, 1);
   END; { de "LettreEnMot" }
```

Nous allons utiliser des procédures qui permettent d'extraire un préfixe, un suffixe, ou un facteur d'un mot donné. Ces procédures sont :

```
PROCEDURE LePrefixe (u: mot; i: integer; VAR v: mot);
```
Détermine le préfixe $v = u_1 \cdots u_i$ de u.
```
   BEGIN
      v := u;
      FixerLongueur(v, i)
   END; { de "LePrefixe" }

PROCEDURE LeSuffixe (u: mot; i: integer; VAR v: mot);
```
Détermine le suffixe $v = u_i \cdots u_n$ de u.
```
   VAR
      j: integer;
   BEGIN
      FOR j := i TO Longueur(u) DO v[j - i + 1] := u[j];
      FixerLongueur(v, Longueur(u) - i + 1)
   END; { de "LeSuffixe" }

PROCEDURE LeFacteur (u: mot; i, j: integer; VAR v: mot);
```
Détermine le facteur $v = u_i \cdots u_j$ de u.
```
   VAR
      k: integer;
   BEGIN
      FOR k := i TO j DO v[k - i + 1] := u[k];
      FixerLongueur(v, j - i + 1)
   END; { de "LeFacteur" }
```

Bien entendu, on peut aussi bien définir l'extraction d'un préfixe ou d'un suffixe au moyen de l'extraction d'un facteur. Enfin, voici une procédure qui calcule le conjugué d'un mot :

```
PROCEDURE LeConjugue (u: mot; i: integer; VAR v: mot);
   Calcule le mot v = u_{i+1} ··· u_n u_1 ··· u_i conjugué de u.
   VAR
      j, n: integer;
   BEGIN
      n := Longueur(u);
      FixerLongueur(v, n);
      FOR j := 1 TO n - i DO v[j] := u[i + j];
      FOR j := n - i + 1 TO n DO v[j] := u[i + j - n]
   END; { de "LeConjugue" }
```

9.2 Mots de Lukasiewicz

9.2.1 Enoncé : mots de Lukasiewicz

Dans cet énoncé, on appelle *mot* toute suite finie d'entiers $u = (u_1, \ldots, u_n)$, où $n \geq 1$ et où $u_i \in \{-1, 1\}$ pour $i = 1, \ldots, n$. L'entier n est la *longueur* de u.

Un mot de longueur n est représenté par un tableau d'entiers à $n+1$ éléments, l'élément d'indice 0 contenant la longueur du mot.

Un mot $u = (u_1, \ldots, u_n)$ est un *mot de Lukasiewicz* si

$$\sum_{i=1}^{n} u_i = -1 \quad \text{et} \quad \sum_{i=1}^{k} u_i \geq 0 \quad \text{pour} \quad 1 \leq k \leq n-1$$

1.– Ecrire une procédure qui teste si un mot est un mot de Lukasiewicz.

Exemples numériques : $(1, -1, 1, -1, -1, 1, -1, -1)$ et $(1, -1, 1, 1, -1, -1, 1, -1, -1)$.

Un *pic* du mot $u = (u_1, \ldots, u_n)$ est un entier i $(1 \leq i \leq n-2)$ tel que $u_i = 1$ et $u_{i+1} = u_{i+2} = -1$. On définit une fonction ρ qui à un mot $u = (u_1, \ldots, u_n)$ associe un mot $\rho(u)$ de la manière suivante :

$\rho(u) = u$, si u ne possède pas de pic ;

$\rho(u) = (u_1, \ldots, u_{i-1}, u_{i+2}, \ldots, u_n)$, si u possède un pic et si i est le plus petit pic de u.

Soit ρ^* la fonction définie pour un mot u par $\rho^*(u) = \rho^m(u)$, où m est le plus petit entier ≥ 0 tel que $\rho^m(u) = \rho^{m+1}(u)$. (On pose $\rho^{m+1} = \rho \circ \rho^m$.)

2.– Démontrer que u est un mot de Lukasiewicz si et seulement si $\rho^*(u) = (-1)$.

3.– Ecrire des procédures qui calculent $\rho(u)$ et $\rho^*(u)$ pour un mot u.

4.– Démontrer que si $u = (u_1, \dots, u_n)$ et $v = (v_1, \dots, v_m)$ sont des mots de Lukasiewicz, alors $(1, u_1, \dots, u_n, v_1, \dots, v_m)$ est un mot de Lukasiewicz.

5.– Démontrer que réciproquement, si $w = (w_1, \dots, w_n)$ est un mot de Lukasiewicz et si $n > 1$, alors $w_1 = 1$ et il existe un unique couple $u = (u_1, \dots, u_m)$ et $v = (v_1, \dots, v_k)$ de mots de Lukasiewicz tels que

$$w = (1, u_1, \dots, u_m, v_1, \dots, v_k)$$

Les mots u et v sont respectivement appelés le *fils gauche* et le *fils droit* de w.

6.– Ecrire une procédure qui calcule et affiche tous les mots de Lukasiewicz de longueur $\leq n$, en utilisant la caractérisation des deux questions précédentes. On pourra prendre $n = 9$.

7.– Soit $u = (u_1, \dots, u_n)$ un mot tel que $\sum_{i=1}^{n} u_i = -1$. Démontrer qu'il existe un unique i, avec $1 \leq i \leq n$ tel que $(u_i, u_{i+1}, \dots, u_n, u_1, \dots, u_{i-1})$ soit un mot de Lukasiewicz. En déduire que le nombre c_k de mots de Lukasiewicz de longueur $2k+1$ est

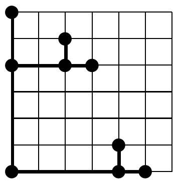
$$\frac{1}{k+1}\binom{2k}{k}$$

(On note $\binom{p}{q} = \frac{p!}{q!(p-q)!}$.)

8.– Ecrire une procédure qui, pour un mot $u = (u_1, \dots, u_n)$, calcule l'entier i de la question précédente.

9.– La *profondeur* $p(w)$ d'un mot de Lukasiewicz est définie par récurrence sur la longueur de w comme suit : si w est de longueur 1, alors $p(w) = -1$; sinon, $p(w) = 1 + \max(p(u), p(v))$, où u et v sont les fils gauche et droit de w. Ecrire une procédure qui calcule la profondeur d'un mot.

10.– On associe à chaque mot de Lukasiewicz w une figure $F(w)$ dans un plan affine euclidien muni d'un repère orthonormé $(O, \vec{i}, \vec{j})$ de la manière suivante, par récurrence sur la longueur de w : si w est de longueur 1, la figure $F(w)$ consiste en un point unique placé en O. Si w est de longueur > 1, soient u et v les fils gauche et droit de w ; la figure $F(w)$ est la réunion des segments de droite joignant l'origine aux points de coordonnées $(2^{p(w)}, 0)$ et $(0, 2^{p(w)})$ et des figures $F(u)$ et $F(v)$, translatées respectivement de $(2^{p(w)}, 0)$ et de $(0, 2^{p(w)})$. Ci-dessous est représentée la figure associée à $w = (1, 1, -1, -1, 1, 1, -1, -1, -1)$. Ecrire une procédure qui détermine et trace la figure associée à un mot de Lukasiewicz.



9.2.2 Solution : mots de Lukasiewicz

On appelle *mot* toute suite finie d'entiers $u = (u_1, \ldots, u_n)$, où $n \geq 1$ et où $u_i \in \{-1, 1\}$ pour $i = 1, \ldots, n$.

Etant donné un mot $u = (u_1, \ldots, u_n)$, on pose

$$s(u,k) = \sum_{i=1}^{k} u_i \qquad k = 1, \ldots, n$$

de sorte que u est un mot de Lukasiewicz si et seulement si

$$s(u,n) = -1 \mathrm{et} s(u,k) \geq 0 \mathrm{pour} k = 1, \ldots, n-1$$

En particulier, on a $u_n = -1$, et si $n > 1$, alors $u_1 = 1$ et $u_{n-1} = -1$.

Un *pic* du mot $u = (u_1, \ldots, u_n)$ est un entier i $(1 \leq i \leq n-2)$ tel que $u_i = 1$ et $u_{i+1} = u_{i+2} = -1$. On définit une fonction ρ qui à un mot $u = (u_1, \ldots, u_n)$ associe un mot $\rho(u)$ de la manière suivante :

$\rho(u) = u$, si u ne possède pas de pic;
$\rho(u) = (u_1, \ldots, u_{i-1}, u_{i+2}, \ldots, u_n)$, si u possède un pic et si i est le plus petit pic de u.

Lemme 9.2.1. *Un mot u est un mot de Lukasiewicz si et seulement si $\rho(u)$ est un mot de Lukasiewicz.*

Preuve. Le lemme est évident si u n'a pas de pic. Supposons donc que u possède un pic et soit i le plus petit pic de u. Soit $v = \rho(u) = (v_1, \ldots, v_{n-2})$, avec

$$v_j = \begin{cases} u_j & \text{si } 1 \leq j \leq i-1 \\ u_{j+2} & \text{si } i \leq j \leq n-2 \end{cases}$$

On a

$$s(v,k) = \begin{cases} s(u,k) & \text{si } k \leq i-1 \\ s(u,k+2) & \text{si } k \geq i \end{cases} \qquad k = 1, \ldots, n-2$$

ce qui montre que v est un mot de Lukasiewicz si u l'est. Réciproquement, on a

$$s(u,i) = 1 + s(u,i-1) \quad s(u,i+1) = s(u,i-1)$$

ce qui montre que u est de Lukasiewicz si v l'est. ∎

Soit ρ^* la fonction définie pour un mot u par $\rho^*(u) = \rho^m(u)$, où m est le plus petit entier ≥ 0 tel que $\rho^m(u) = \rho^{m+1}(u)$. (On pose $\rho^{m+1} = \rho \circ \rho^m$.)

Proposition 9.2.2. *Un mot u est un mot de Lukasiewicz si et seulement si $\rho^*(u) = (-1)$.*

Preuve. Comme (-1) est un mot de Lukasiewicz, la condition est suffisante par le lemme ci-dessus. Réciproquement, soit u un mot de Lukasiewicz ; alors par le lemme, $v = \rho^*(u) = (v_1, \ldots, v_n)$ est un mot de Lukasiewicz. Si $n = 1$, on a $v = (-1)$; supposons donc $n > 1$.

On a $v_n = v_{n-1} = -1$ et $v_1 = 1$. Soit k le plus grand entier tel que $v_k = 1$. Alors $k \leq n-2$ et k est un pic, donc $v \neq \rho(v)$. ∎

Proposition 9.2.3. *Si u et v sont des mots de Lukasiewicz, alors le mot $w = 1uv$ est un mot de Lukasiewicz.*

Preuve. Soient $n = |u|$ et $m = |v|$. On a

$$s(w,k) = \begin{cases} 1 & \text{si } k = 1 \\ 1 + s(u, k-1) & \text{si } 2 \leq k \leq n+1 \\ s(v, k-n-1) & \text{si } n+2 \leq k \leq n+m+1 \end{cases}$$

Ceci montre que w est un mot de Lukasiewicz. ∎

Proposition 9.2.4. *Soit $w = (w_1, \ldots, w_n)$ un mot de Lukasiewicz de longueur $n > 1$; alors $w_1 = 1$ et il existe un unique couple $u = (u_1, \ldots, u_m)$ et $v = (v_1, \ldots, v_k)$ de mots de Lukasiewicz tels que*

$$w = (1, u_1, \ldots, u_m, v_1, \ldots, v_k)$$

Preuve. On ne peut avoir $w_1 = -1$ que si $n = 1$. Donc $w_1 = 1$. Soit m le plus petit entier tel que $s(w, m) = 0$. Le mot $u = (w_2, \ldots, w_m)$ vérifie alors $s(u, i) = s(w, i+1) - 1$ pour $i = 1, \ldots, m-1$, donc u est un mot de Lukasiewicz. Soit $v = (w_{m+1}, \ldots, w_n)$. Alors $w = 1uv$. Par ailleurs, on a $s(v, i) = s(w, m+i)$ pour $i = 1, \ldots, n-m$, ce qui prouve que v est un mot de Lukasiewicz.

Prouvons l'unicité. Supposons que $w = 1uv = 1u'v'$, avec par exemple u' plus court que u. Posons $n = |u'|$. Alors $s(u', n) = -1 = s(u, n) \leq 0$, d'où la contradiction. ∎

Proposition 9.2.5. *Soit $u = (u_1, \ldots, u_n)$ un mot tel que $s(u, n) = -1$. Il existe un unique i, avec $1 \leq i \leq n$ tel que $(u_i, u_{i+1}, \ldots, u_n, u_1, \ldots, u_{i-1})$ soit un mot de Lukasiewicz.*

Preuve. Soit $p = \min_{1 \leq k \leq n} s(u, k)$ et soit i le plus petit entier tel que $s(u, i) = p$. On pose

$$v = (u_{i+1}, \ldots, u_n, u_1, \ldots, u_i)$$

Vérifions que v est un mot de Lukasiewicz. On a

$$s(v,k) = \begin{cases} s(u, i+k) - p & \text{pour } 1 \leq k \leq n-i \\ s(u,n) - p + s(u, k+i-n) & \text{pour } n-i+1 \leq k \leq n \end{cases}$$

Or $s(u, i+k) \geq p$ par définition de p et, pour $n-i+1 \leq k \leq n-1$, on a $1 \leq k+i-n \leq i-1$, donc $s(u, k+i-n) > p$ par le choix de i, ce qui montre que $s(v, k) > -1$ dans ces cas. Enfin, $s(v, n) = -1$. ∎

En vue de dénombrer les mots de Lukasiewicz, nous avons besoin d'un résultat supplémentaire qui est intéressant en lui-même.

Proposition 9.2.6. *Soient u et v deux mots non vides. Les conditions suivantes sont équivalentes :*

(1) $uv = vu$;
(2) *il existe deux entiers $n, m \geq 1$ tels que $u^n = v^m$;*
(3) *il existe un mot w non vide et deux entiers $k, \ell \geq 1$ tels que $u = w^k$, $v = w^\ell$.*

Preuve. (1) $\Rightarrow$ (3). Si $|u| = |v|$, alors $u = v$ et l'implication est évidente. En raisonnant par récurrence sur $|uv|$, supposons $|u| > |v|$. Soit alors w tel que $u = vw$. En reportant dans l'équation $uv = vu$, on obtient $vwv = vvw$, d'où en simplifiant $wv = vw$. Par récurrence, il existe un mot x et des entiers $k, \ell \geq 1$ tels que $v = w^k$, $w = x^\ell$, d'où $u = x^{k+\ell}$.

(3) $\Rightarrow$ (2). Si $u = w^k$ et $v = w^\ell$, alors $u^\ell = v^k$.

(2) $\Rightarrow$ (1). La conclusion est évidente si $u = v$. Supposons donc $|u| > |v|$, et soit w tel que $u = vw$. Alors

$$u^n v = (vw)^n v = v(wv)^n = v^{m+1}$$

et en simplifiant la dernière égalité, $(wv)^n = v^m$. Comme $v^m = u^n = (vw)^n$, on a $(wv)^n = (vw)^n$, donc $wv = vw$, ou encore $uv = vu$. ■

Corollaire 9.2.7. *Le nombre de mots de Lukasiewicz de longueur $2n + 1$ est*

$$\frac{1}{n+1}\binom{2n}{n}$$

Preuve. Le nombre de mots u de longueur $2n + 1$ tels que $s(u, 2n + 1) = -1$ est égal à $\binom{2n+1}{n+1}$, puisqu'il s'agit de choisir $n + 1$ positions pour les lettres égales à -1. D'après la proposition 9.2.5, les $2n + 1$ mots qui résultent d'un tel mot par permutation circulaire donnent un unique mot de Lukasiewicz. Il reste à prouver que ces $2n+1$ mots sont deux à deux distincts. Supposons au contraire que

$$x = (u_1, \ldots, u_{2n+1}) = (u_i, \ldots, u_{2n+1}, u_1, \ldots, u_{i-1})$$

pour un entier $i > 1$; alors on a $uv = vu$ en posant $u = (u_1, \ldots, u_{i-1})$ et $v = (u_i, \ldots, u_{2n+1})$. Par la proposition précédente, on a $u = w^k$ et $v = w^\ell$ pour un mot w de longueur disons h. Posons $m = s(w, h)$. Alors $-1 = s(x, 2n+1) = (k+\ell)m$, ce qui est impossible parce que $k + \ell \geq 2$. Le nombre de mots de Lukasiewicz est donc

$$\frac{1}{2n+1}\binom{2n+1}{n+1} = \frac{1}{n+1}\binom{2n}{n}$$ ■

Les entiers du corollaire 9.2.7 sont les *nombres de Catalan.*

9.2.3 Programme : mots de Lukasiewicz

Pour tester qu'un mot est un mot de Lukasiewicz, il suffit de calculer la fonction s et de vérifier les conditions. Ceci conduit à la procédure suivante :

```
FUNCTION EstLukasiewicz (VAR u: mot): boolean;
   VAR
      s, k, n: integer;
   BEGIN
      s := 0; k := 0; n := Longueur(u);
      WHILE (s >= 0) AND (k < n) DO BEGIN       L'entier s doit être positif ou nul,
         k := k + 1; s := s + u[k]
      END;
      EstLukasiewicz := (k = n) AND (s = -1)    et s = −1 à la fin du mot.
   END; { de "EstLukasiewicz" }
```

On peut aussi tester qu'un mot est de Lukasiewicz en calculant ρ^* et en vérifiant que le résultat est le mot (-1). Nous nous contentons des procédures réalisant ρ et ρ^*.

```
FUNCTION EstPic (VAR u: mot; i: integer): boolean;
   Teste si un entier i ≤ Longueur(u) − 2 est un pic de u.
   BEGIN
      EstPic := (u[i] = 1) AND (u[i + 1] = -1) AND (u[i + 2] = -1)
   END; { de "EstPic" }

PROCEDURE Rho (VAR u: mot);
   VAR
      i, j, n: integer;
   BEGIN
      i := 1;
      n := Longueur(u);
      WHILE (i <= n - 2) AND NOT EstPic(u, i) DO    { AND séquentiel }
         i := i + 1;
      IF i <= n - 2 THEN BEGIN                      Elimination du pic.
         FOR j := i TO n - 2 DO u[j] := u[j + 2];
         FixerLongueur(u, n - 2)
      END;
   END; { de "Rho" }

PROCEDURE RhoEtoile (VAR u: mot);
   VAR
      n: integer;
   BEGIN
      REPEAT
         n := Longueur(u);                   Sauvegarde de l'ancienne longueur.
         Rho(u)                              Calcul de ρ.
      UNTIL n = Longueur(u)                  Comparaison à l'ancienne longueur.
   END; { de "RhoEtoile" }
```

La décomposition d'un mot de Lukasiewicz en ses deux fils gauche et droit ainsi que la recomposition se font par les procédures que voici :

```
PROCEDURE FilsLukasiewicz (w: mot; VAR u, v: mot);
   Le mot w commence par 1. Les mots u et v reçoivent respectivement le fils gauche et le
   fils droit de w.
   VAR
      i, j, s: integer;
   BEGIN
      s := 0;
      i := 1;
      REPEAT                                        Calcul du fils gauche.
         i := i + 1;
         s := s + w[i]
      UNTIL s = -1;
      LeFacteur(w, 2, i, u);
      LeSuffixe(w, i + 1, v)                        Le fils droit.
   END; { de "FilsLukasiewicz" }
PROCEDURE PereLukasiewicz (u, v: mot; VAR w: mot);
   BEGIN
      LettreEnMot(w, 1);
      Concatener(w, u, w);
      Concatener(w, v, w)
   END; { de "PereLukasiewicz" }
```

Pour engendrer tous les mots de Lukasiewicz, on range, dans une suite de mots, les mots de Lukasiewicz de longueur donnée. Les mots de longueur $2n + 1$ s'obtiennent en concaténant des mots de longueur $2k + 1$ et de longueur $2\ell + 1$, pour $k + \ell = n - 1$.

Une suite de mots se définit par :

```
CONST
   LongueurSuiteMot = 42;                           42 = 1/6 (10 choose 5)
TYPE
   SuiteMot = ARRAY[1..LongueurSuiteMot] OF mot;
```

et une table de suite de mots par :

```
CONST
   LongueurTable = 5;
TYPE
   TableSuite = ARRAY[0..LongueurTable] OF SuiteMot;
   TailleSuite = ARRAY[0..LongueurTable] OF integer;
```

où l'entier d'indice n du tableau de type `TailleSuite` contient le nombre d'éléments dans la n-ième suite de mots.

Ceci conduit aux procédures suivantes :

```
PROCEDURE EngendrerLukasiewicz (nn: integer; VAR luk: TableSuite;
            VAR l: TailleSuite);
   VAR
      n, k: integer;
   BEGIN
      LettreEnMot(luk[0][1], -1);              L'unique mot de longueur 1.
      l[0] := 1;
      FOR n := 1 TO nn DO BEGIN
         l[n] := 0;
         FOR k := 0 TO n - 1 DO
            ComposerLukasiewicz(luk[k],luk[n-k-1],luk[n],l[k],l[n-k-1],l[n])
      END;
   END; { de "EngendrerLukasiewicz" }
```

avec :

```
PROCEDURE ComposerLukasiewicz (VAR a, b, c: SuiteMot;
   VAR la, lb, m: integer);
   Compose les mots de la suite a avec les mots de la suite b.
   VAR
      i, j: integer;
   BEGIN
      FOR i := 1 TO la DO
         FOR j := 1 TO lb DO BEGIN
            m := m + 1;
            PereLukasiewicz(a[i], b[j], c[m])
         END
   END; { de "ComposerLukasiewicz" }
```

Voici le résultat obtenu :

```
n = 0  l[0] = 1
 -1
n = 1  l[1] = 1
  1 -1 -1
n = 2  l[2] = 2
  1 -1  1 -1 -1
  1  1 -1 -1 -1
n = 3  l[3] = 5
  1 -1  1 -1  1 -1 -1
  1 -1  1  1 -1 -1 -1
  1  1 -1 -1  1 -1 -1
  1  1 -1  1 -1 -1 -1
  1  1  1 -1 -1 -1 -1
n = 4  l[4] = 14
  1 -1  1 -1  1 -1  1 -1 -1
  1 -1  1 -1  1  1 -1 -1 -1
  1 -1  1  1 -1 -1  1 -1 -1
  1 -1  1  1 -1  1 -1 -1 -1
```

```
   1 -1  1  1  1 -1 -1 -1 -1
   1  1 -1 -1  1 -1  1 -1 -1
   1  1 -1 -1  1  1 -1 -1 -1
   1  1 -1  1 -1 -1  1 -1 -1
   1  1  1 -1 -1 -1  1 -1 -1
   1  1 -1  1 -1  1 -1 -1 -1
   1  1 -1  1  1 -1 -1 -1 -1
   1  1  1 -1 -1  1 -1 -1 -1
   1  1  1 -1  1 -1 -1 -1 -1
   1  1  1  1 -1 -1 -1 -1 -1
 n = 5  l[5] = 42
   1 -1  1 -1  1 -1  1 -1  1 -1 -1
   1 -1  1 -1  1 -1  1  1 -1 -1 -1
   . . . . .
```

Soit maintenant $u = (u_1, \ldots, u_n)$ un mot tel que $s(u, n) = -1$. La preuve de la proposition 9.2.5 donne en fait l'algorithme pour calculer le conjugué de u qui est un mot de Lukasiewicz : on calcule le plus petit entier i tel que $s(u, i) = \min_{1 \leq k \leq n} s(u, k)$, puis on décale le mot u à cette position. On obtient donc la procédure :

```
PROCEDURE ConjuguerEnLukasiewicz (u: mot; VAR i: integer; VAR v: mot);
   VAR
      j, n, s, smin: integer;
   BEGIN
      n := Longueur(u);
      smin := 0;                          Conserve min_{1≤k≤n} s(u, k).
      s := 0;
      i := 0;                             Conserve l'indice du minimum.
      FOR j := 1 TO n DO BEGIN
         s := s + u[j];
         IF s < smin THEN BEGIN           Ajustement du minimum.
            smin := s; i := j
         END;
      END;
      LeConjugue(u, i, v);                Calcul du conjugué.
   END; { de "ConjuguerEnLukasiewicz" }
```

Le calcul de la *profondeur* d'un mot de Lukasiewicz ainsi que le tracé de la figure associée se font plus simplement en utilisant des procédures récursives. Ainsi, la définition de la profondeur conduit directement à l'écriture récursive suivante :

```
FUNCTION Profondeur (VAR w: mot): integer;
   VAR
      u, v: mot;
   BEGIN
      IF Longueur(w) = 1 THEN
         Profondeur := -1
      ELSE BEGIN
         FilsLukasiewicz(w, u, v);
```

```
      Profondeur := 1 + max(Profondeur(u), Profondeur(v))
    END
  END; { de "Profondeur" }
```

Pour le tracé lui-même, il convient de fixer la distance, en nombre de pixels par exemple, qui doit séparer deux points voisins à coordonnées entières. Moyennant quoi, on peut écrire des primitives de tracé d'un point et d'un segment de droite; sur le Macintosh par exemple, une réalisation est la suivante (pour une interface graphique un peu plus conséquente, voir le chapitre 10) :

```
CONST
  ecart = 20;                    Nombre de pixels séparant deux points voisins.
PROCEDURE TracerPoint (x, y: integer);
  BEGIN
    Paintcircle(ecart * x, ecart * y, 5);
  END; { de "TracerPoint" }
PROCEDURE TracerSegment (x, y, xx, yy: integer);
  BEGIN
    Drawline(ecart * x, ecart * y, ecart * xx, ecart * yy)
  END; { de "TracerSegment" }
```

La définition de la figure associée à un mot de Lukasiewicz se traduit alors directement en :

```
PROCEDURE FigureLukasiewicz (w: mot; x, y: integer);
  Trace la figure associée à w, en débutant au point de coordonnées (x, y).
  VAR
    u, v: mot;
    n: integer;
  BEGIN
    IF Longueur(w) = 1 THEN
      TracerPoint(x, y)
    ELSE BEGIN
      FilsLukasiewicz(w, u, v);
      n := puissanceE(2, Profondeur(w));              n = 2^p(w)
      TracerSegment(x, y, x + n, y);
      FigureLukasiewicz(u, x + n, y);
      TracerSegment(x, y, x, y + n);
      FigureLukasiewicz(v, x, y + n);
    END
  END; { de "FigureLukasiewicz" }
```

On appelle cette procédure par :

```
FigureLukasiewicz( w, 1, 1);
```

par exemple. L'exemple de l'énoncé est obtenu de cette manière, à une symétrie près.

9.3 Mots de Lyndon

9.3.1 Enoncé : mots de Lyndon

Dans cet énoncé, on appelle *mot* toute suite finie d'entiers $u = (u_1, \ldots, u_n)$, où $n \geq 1$ et où $u_i \in \{0, 1\}$ pour $i = 1, \ldots, n$. L'entier n est la *longueur* de u. Si $u = (u_1, \ldots, u_n)$ et $v = (v_1, \ldots, v_m)$ sont des mots, on note uv le mot $(u_1, \ldots, u_n, v_1, \ldots, v_m)$.

Un mot de longueur n est représenté par un tableau d'entiers à $n+1$ éléments, l'élément d'indice 0 contenant la longueur du mot.

1.– Ecrire une procédure qui, à partir de deux mots u et v, construit le mot uv.

Soient $u = (u_1, \ldots, u_n)$ et $v = (v_1, \ldots, v_m)$ deux mots. Alors u est un *début* de v si $n < m$ et $u_i = v_i$ pour $i = 1, \ldots, n$; le mot u est une *fin* de v si $n < m$ et si $u_i = v_{m-n+i}$ pour $i = 1, \ldots, n$.

On pose $u \prec v$ si u est un début de v ou s'il existe un entier k tel que $u_i = v_i$ pour $i = 1, \ldots, k-1$ et $u_k < v_k$.

2.– Démontrer que pour deux mots $u \neq v$, on a soit $u \prec v$, soit $v \prec u$.

La relation $\preceq$ définie par $u \preceq v$ si et seulement si $u = v$ ou $u \prec v$ s'appelle l'*ordre lexicographique*.

3.– Ecrire une procédure qui, pour deux mots u et v, détermine si $u \prec v$.

Un *conjugué* d'un mot $u = (u_1, \ldots, u_n)$ est un mot de la forme $v = (u_i, \ldots, u_n, u_1, \ldots, u_{i-1})$ avec $2 \leq i \leq n$. Un mot u est un *mot de Lyndon* si $u \prec v$ pour tout conjugué v de u. Clairement, un mot de longueur 1 est un mot de Lyndon.

4.– Ecrire une procédure qui teste si un mot est un mot de Lyndon.

5.– Soit u un mot. Démontrer que s'il existe un mot qui est à la fois un début et une fin de u, alors u n'est pas un mot de Lyndon.

6.– Démontrer qu'un mot u est un mot de Lyndon si et seulement si pour toute fin h de u, on a $u \prec h$. Ecrire une procédure qui teste si u est un mot de Lyndon en utilisant cette caractérisation.

7.– Démontrer que si un mot u de longueur > 1 est un mot de Lyndon, il existe deux mots de Lyndon f et g tels que $u = fg$ et $f \prec g$ et, réciproquement, si f et g sont des mots de Lyndon tels que $f \prec g$, alors fg est un mot de Lyndon. Utiliser cette caractérisation pour écrire une procédure qui détermine tous les mots de Lyndon de longueur au plus n. On les affichera dans l'ordre lexicographique. On testera pour $n = 5$.

Soit $u = (u_1, \ldots, u_n)$ un mot. Une *factorisation de Lyndon* du mot u est une séquence $u^{(1)}, u^{(2)}, \ldots, u^{(p)}$ de mots de Lyndon telle que

$$u = u^{(1)} u^{(2)} \cdots u^{(p)} \quad \text{et} \quad u^{(1)} \succeq u^{(2)} \succeq \cdots \succeq u^{(p)}$$

Par exemple, pour $u = (0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0)$, une factorisation de Lyndon est

$$(0, 1), (0, 1), (0, 0, 1, 1), (0, 0, 1), (0), (0)$$

On construit une factorisation de Lyndon d'un mot $u = (u_1, \ldots, u_n)$ comme suit. On part de la séquence $u^{(1)}, u^{(2)}, \ldots, u^{(n)}$ où $u^{(i)} = (u_i)$ pour $i = 1, \ldots, n$.

Si $u^{(1)}, u^{(2)}, \ldots, u^{(p)}$ est une séquence construite et s'il existe un indice k $(1 \leq k < p)$ tel que $u^{(k)} \prec u^{(k+1)}$, on construit une nouvelle séquence $v^{(1)}, v^{(2)}, \ldots, v^{(p-1)}$ par $v^{(j)} = u^{(j)}$ pour $j = 1, \ldots, k-1$, $v^{(k)} = u^{(k)}u^{(k+1)}$, et $v^{(j)} = u^{(j+1)}$ pour $j = k+1, \ldots, p-1$. On itère ce processus tant qu'il est possible de raccourcir la séquence. La séquence finale est une factorisation de Lyndon.

8.– Ecrire une procédure qui calcule une factorisation de Lyndon d'un mot par ce procédé.

Exemple numérique : $(1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1)$.

9.– Démontrer que tout mot possède une unique factorisation de Lyndon.

9.3.2 Solution : mots de Lyndon

La définition de l'*ordre lexicographique* s'étend sans difficulté à tout alphabet A totalement ordonné. L'ordre lexicographique n'est rien d'autre que l'ordre dans lequel on rencontre les mots dans un dictionnaire. On l'appelle aussi l'ordre *alphabétique*.

Il est commode, pour les démonstrations, d'introduire la définition suivante : un mot u est *fortement inférieur* à un mot v, relation notée $u \ll v$, s'il existe un entier k tel que $u_i = v_i$ pour $i = 1, \ldots, k-1$ et $u_k < v_k$. On a donc $u \prec v$ si u est un début de v ou si $u \ll v$. Notons que si $u \ll v$, alors quels que soient les mots x, y, on a $ux \ll vy$. Si $u \preceq v$, alors $wu \preceq wv$ pour tout mot w.

PROPOSITION 9.3.1. *L'ordre lexicographique est un ordre total.*

Preuve. Soient u et v deux mots distincts et montrons que $u \prec v$ ou $v \prec u$. Ceci est clair si u est un début de v ou si v est un début de u. Sinon, il existe un indice k tel que $u_k \neq v_k$ et $u_i = v_i$ pour $i = 1, \ldots, k-1$. Mais alors $u \ll v$ ou $v \ll u$, selon que $u_k < v_k$ ou $v_k < u_k$. ∎

Pour l'étude des mots de Lyndon, le lemme suivant est très utile.

LEMME 9.3.2. *Soit u un mot. S'il existe un mot qui est à la fois un début et une fin de u, alors u n'est pas un mot de Lyndon.*

Preuve. Supposons que v soit à la fois un début et une fin de u. Alors il existe deux mots w et w', de même longueur, tels que

$$u = vw = w'v$$

Si $w' \preceq w$, alors $vw' \preceq vw = u$ et, si $w \prec w'$, on a $w \ll w'$, donc $wv \prec w'v = u$. Dans les deux cas, on a trouvé un conjugué de u qui est inférieur ou égal à u. Donc u n'est pas de Lyndon. ∎

On obtient alors la caractérisation suivante des mots de Lyndon.

Proposition 9.3.3. *Un mot u est un mot de Lyndon si et seulement s'il est strictement plus petit que toutes ses fins dans l'ordre lexicographique.*

Preuve. Soit h une fin de u. Alors $u = vh$ pour un mot v.

Supposons d'abord que u soit un mot de Lyndon. Alors $u \prec hv$ et, par le lemme précédent, h n'est pas un début de u. Donc $u \prec h$.

Réciproquement, si $u \prec h$, alors comme $h \prec hv$, on a $u \prec hv$ par transitivité, montrant que u est plus petit que ses conjugués. ∎

Proposition 9.3.4. *Soit u un mot de Lyndon qui n'est pas une lettre; il existe alors deux mots de Lyndon f et g, avec $f \prec g$, tels que $u = fg$.*

Notons que de manière générale, si f et g sont deux mots de Lyndon tels que fg est un mot de Lyndon, alors $f \prec fg \prec g$ en vertu de la proposition précédente.

Preuve. Soit g la fin de u de longueur maximale qui est un mot de Lyndon. Un tel mot existe puisque la dernière lettre de u est un mot de Lyndon. Posons $u = fg$. On a $f \prec fg \prec g$ parce que fg est un mot de Lyndon. Il reste à démontrer que f est un mot de Lyndon. Soit r une fin de f. Puisque rg n'est pas un mot de Lyndon, il existe une fin s de rg telle que $s \prec rg$. Le mot r n'est pas un début de s. Sinon, en posant $s = rt$, on aurait $t \prec g$. Or t est une fin de g et, comme g est un mot de Lyndon, $g \prec t$. Ceci montre que $s \preceq r$.

Comme fg est un mot de Lyndon et que s est une fin de rg et donc de fg, on a $fg \prec s$, donc

$$f \prec fg \prec s \preceq r$$

ce qui montre que f est un mot de Lyndon. ∎

Observons que la factorisation de u en deux mots de Lyndon f et g n'est pas unique. Par exemple, le mot $u = (0, 0, 1, 0, 1, 0, 1, 1)$ se factorise de quatre façons : la première est $(0, 0, 1, 0, 1, 0, 1)$, (1), la deuxième $(0, 0, 1, 0, 1)$, $(0, 1, 1)$, la troisième $(0, 0, 1)$, $(0, 1, 0, 1, 1)$ et enfin la quatrième (0), $(0, 1, 0, 1, 0, 1, 1)$. La factorisation particulière donnée dans la preuve de la proposition, qui s'obtient en choisissant comme deuxième facteur le plus long mot de Lyndon qui est une fin, s'appelle la *factorisation standard*.

Proposition 9.3.5. *Soient f et g deux mots de Lyndon tels que $f \prec g$. Alors fg est un mot de Lyndon.*

Preuve. Soit $u = fg$. Nous allons démontrer que u est plus petit que toutes ses fins.

Notons d'abord que $fg \prec g$. Ceci est clair si f n'est pas un début de g, parce que $f \prec g$, donc $f \ll g$, d'où $fg \prec g$. Si f est un début de g, alors $g = fw$ pour un mot w et $g \prec w$ parce que g est un mot de Lyndon. Mais alors $fg \prec fw = g$.

Soit maintenant h une fin de u. Si h est une fin de g, alors $u \prec g \prec h$. Le cas $h = g$ a été traité. Si $h = h'g$, où h' est une fin de f, alors $f \prec h'$ et même $f \ll h'$ parce que h' est plus court que f, donc $fg \prec h' \prec h$. ∎

Soit $u = (u_1, \ldots, u_n)$ un mot. Une *factorisation de Lyndon* du mot u est une séquence $u^{(1)}, u^{(2)}, \ldots, u^{(p)}$ de mots de Lyndon telle que

$$u = u^{(1)}u^{(2)} \cdots u^{(p)} \quad \text{et} \quad u^{(1)} \succeq u^{(2)} \succeq \cdots \succeq u^{(p)}$$

Par exemple, pour $u = (0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0)$, une factorisation de Lyndon est

$$(0, 1), (0, 1), (0, 0, 1, 1), (0, 0, 1), (0), (0)$$

On construit une factorisation de Lyndon d'un mot $u = (u_1, \ldots, u_n)$ comme suit. On part de la séquence $u^{(1)}, u^{(2)}, \ldots, u^{(n)}$ où $u^{(i)} = (u_i)$ pour $i = 1, \ldots, n$.

Si $u^{(1)}, u^{(2)}, \ldots, u^{(p)}$ est une séquence construite et s'il existe un indice k $(1 \leq k < p)$ tel que $u^{(k)} \prec u^{(k+1)}$, on construit une nouvelle séquence $v^{(1)}, v^{(2)}, \ldots, v^{(p-1)}$ par $v^{(j)} = u^{(j)}$ pour $j = 1, \ldots, k-1$, $v^{(k)} = u^{(k)}u^{(k+1)}$, et $v^{(j)} = u^{(j+1)}$ pour $j = k+1, \ldots, p-1$. On itère ce processus tant qu'il est possible de raccourcir la séquence. La séquence finale est une factorisation de Lyndon.

Proposition 9.3.6. *Tout mot possède une unique factorisation de Lyndon.*

Preuve. L'existence de la factorisation de Lyndon est évidente. L'unicité se prouve par l'absurde. Soit en effet u un mot de longueur minimale ayant deux factorisations de Lyndon distinctes

$$(u_1, \ldots, u_n) \quad \text{et} \quad (v_1, \ldots, v_m)$$

Si $n = 1$ et $m \geq 2$, alors $v_1 \prec u_1 \prec v_m$ et $v_1 \succeq v_m$, une contradiction. Si $n, m \geq 2$, supposons que u_1 est plus long que v_1, sinon $u_2 \cdots u_n = v_2 \cdots v_m$ est un mot plus court que u ayant deux factorisations distinctes. Alors

$$u_1 = v_1 \cdots v_k \quad \text{ou} \quad u_1 = v_1 \cdots v_k w$$

pour un entier k et pour un début w de v_{k+1}. Dans le premier cas, on a comme ci-dessus $v_1 \prec u_1 \prec v_k$ et, dans le deuxième cas, $v_1 \prec u_1 \prec w \prec v_{k+1}$. ∎

9.3.3 Programme : mots de Lyndon

Nous commençons par une procédure qui compare deux mots pour l'ordre lexicographique, en utilisant la définition. Cette procédure s'écrit :

```
FUNCTION CompareMot (VAR u, v: mot): integer;
   Compare u et v dans l'ordre lexicographique. Le résultat est −1, 0, 1, selon que l'on a
   u ≺ v, u = v ou u ≻ v.
   VAR
      i, n: integer;
   BEGIN
      n := min(Longueur(u), Longueur(v));
      i := 1;
      WHILE (i <= n) AND (u[i] = v[i]) DO
         i := i + 1;
      IF i <= n THEN                                    Fortement inférieur.
         CompareMot := signe(u[i] - v[i])
      ELSE                                              Préfixe.
         CompareMot := signe(Longueur(u) - Longueur(v))
   END; { de "CompareMot" }
```

On en déduit immédiatement une procédure qui teste si un mot est inférieur à un deuxième :

```
FUNCTION EstInferieurMot (VAR u, v: mot): boolean;
   Vrai si et seulement si u ≺ v.
   BEGIN
      EstInferieurMot := CompareMot(u, v) = -1
   END; { de "EstInferieurMot" }
```

Il y a deux façons simples de tester si un mot u est un mot de Lyndon (on verra plus loin un algorithme plus compliqué, mais linéaire en fonction de la longueur du mot, pour calculer la factorisation de Lyndon, donc aussi pour tester si un mot est un mot de Lyndon). On peut calculer tous les conjugués de u et vérifier que u est plus petit que ces mots. Ce test se réalise comme suit :

```
FUNCTION EstLyndonParConjugue (u: mot): boolean;
   Teste, en calculant les conjugués de u, si u est un mot de Lyndon.
   VAR
      i, n: integer;
      v: mot;
      inferieur: boolean;
   BEGIN
      n := Longueur(u); i := 1; inferieur := true;
      WHILE (i < n) AND inferieur DO BEGIN
         LeConjugue(u, i, v);
         inferieur := EstInferieurMot(u, v);
         i := i + 1
      END;
      EstLyndonParConjugue := inferieur
   END; { de "EstLyndonParConjugue" }
```

Dans cette procédure, `inferieur` reste vrai tant que l'on n'a pas trouvé de conjugué inférieur au mot donné. Une deuxième manière de réaliser le test est de comparer le mot donné à toutes ses fins. On obtient alors la procédure suivante :

```
FUNCTION EstLyndonParSuffixe (u: mot): boolean;
   Teste si u est un mot de Lyndon en le comparant à toutes ses fins.
   VAR
      i, n: integer;
      v: mot;
      inferieur: boolean;
   BEGIN
      n := Longueur(u); i := 2; inferieur := true;
      WHILE (i <= n) AND inferieur DO BEGIN
         LeSuffixe(u, i, v);
         inferieur := EstInferieurMot(u, v);
         i := i + 1
      END;
      EstLyndonParSuffixe := inferieur
   END; { de "EstLyndonParSuffixe" }
```

L'écriture d'une procédure qui engendre tous les mots de Lyndon est un peu plus compliquée. L'idée de départ est que tout mot de Lyndon de longueur au moins 2 s'obtient comme produit de mots plus courts. Mais plusieurs produits peuvent donner le même mot de Lyndon et il faut donc vérifier qu'un mot n'est pas déjà construit avant de l'insérer dans la liste. La procédure suivante calcule la table des mots de Lyndon :

```
PROCEDURE EngendrerLyndon (nn: integer; VAR lyn: TableSuite;
   VAR l: TailleSuite);
   Calcul des mots de Lyndon de longueur au plus nn.
   VAR
      n, k: integer;
   BEGIN
      LettreEnMot(lyn[1][1], 0);                      Les deux mots de Lyndon de
      LettreEnMot(lyn[1][2], 1);                      longueur 1 : 0 et 1.
      l[1] := 2;
      FOR n := 2 TO nn DO BEGIN
         l[n] := 0;
         FOR k := 1 TO n - 1 DO
            ComposerLyndon(lyn[k], lyn[n - k], lyn[n], l[k], l[n - k], l[n])
      END;
   END; { de "EngendrerLyndon" }
```

Rappelons d'abord les définitions de types, qui sont les mêmes que pour les mots de Lukasiewicz :

```
CONST
   LongueurSuiteMot = 10;
   LongueurTable = 6;
TYPE
   SuiteMot = ARRAY[1..LongueurSuiteMot] OF mot;
   TableSuite = ARRAY[0..LongueurTable] OF SuiteMot;
   TailleSuite = ARRAY[0..LongueurTable] OF integer;
```

où l'entier d'indice n du tableau de type `TailleSuite` contient le nombre d'éléments dans la n-ième suite de mots. Les mots de Lyndon de longueur n s'obtiennent par concaténation de mots de longueur k et de mots de longueur $n-k$, avec $1 \leq k \leq n-1$. Voici la procédure de composition :

```
PROCEDURE ComposerLyndon (VAR a, b, c: SuiteMot; VAR la, lb, m: integer);
   VAR
      i, j: integer;
      u: mot;
   BEGIN
      FOR i := 1 TO la DO
         FOR j := 1 TO lb DO
            IF EstInferieurMot(a[i], b[j]) THEN BEGIN
               Concatener(a[i], b[j], u);             u est un mot de Lyndon.
               InsererMot(u, c, m)                    On l'ajoute à la liste.
            END
   END; { de "ComposerLyndon" }
```

Notons que l'on compose deux mots a_i et b_j seulement si $a_i \prec b_j$. Détaillons l'insertion du mot u obtenu dans la liste c des mots déjà construits. Les mots de la liste c sont rangés en ordre croissant. Pour insérer u, on parcourt la liste à la recherche du premier élément de c qui est supérieur ou égal à u. Si un tel mot n'existe pas, u est ajouté en fin de liste. Dans le cas contraire, on vérifie s'il n'est pas égal à u, car dans ce cas, le mot u ne doit pas être ajouté une deuxième fois à la liste. Sinon, u est inséré à sa place, en décalant tous les éléments qui lui sont plus grands. Voici comment cet algorithme d'insertion s'écrit :

```
PROCEDURE InsererMot (VAR u: mot; VAR c: SuiteMot; VAR m: integer);
   Insertion d'un mot u dans une suite triée c.
   VAR
      j, k: integer;
   BEGIN
      k := 1;
      WHILE (k <= m) AND EstInferieurMot(c[k], u) DO      { AND séquentiel }
         k := k + 1;                                 On avance tant que c_k ≺ u.
      IF k = m + 1 THEN BEGIN                        Si la fin de c est atteinte,
         m := m + 1;                                 on insère u à la fin;
         c[m] := u
      END                                            si c_k = u, on ne fait rien;
      ELSE IF EstInferieurMot(u, c[k]) THEN BEGIN
         FOR j := m DOWNTO k DO                      sinon, on décale la fin de c,
            c[j + 1] := c[j];
         c[k] := u;                                  puis on insère u.
         m := m + 1
      END
   END; { de "InsererMot" }
```

En utilisant ces procédures, on obtient les résultats numériques suivants :

```
Mots de Lyndon :
n = 1  l[1] = 2
  0
  1
n = 2  l[2] = 1
  01
n = 3  l[3] = 2
  001
  011
n = 4  l[4] = 3
  0001
  0011
  0111
n = 5  l[5] = 6
  00001
  00011
  00101
  00111
  01011
  01111
n = 6  l[6] = 9
  000001
  000011
  000101
  000111
  001011
  001101
  001111
  010111
  011111
```

Le calcul d'une factorisation de Lyndon par l'algorithme exposé dans la section précédente se réalise très simplement. La procédure est :

```
PROCEDURE FactorisationLyndon (u: mot; VAR f: Suitemot; VAR m: integer);
   VAR
      n, i: integer;
   BEGIN
      m := Longueur(u);
      FOR i := 1 TO m DO                          Factorisation initiale.
         LettreEnMot(f[i], u[i]);
      REPEAT
         n := m;                                  n = ancienne longueur.
         CompacterFactorisationLyndon(f, m);      m = nouvelle longueur.
      UNTIL m = n;                                Test d'arrêt.
   END; { de "FactorisationLyndon" }
```

A chaque étape, on tente de compacter la factorisation obtenue. Lorsqu'on ne peut plus la raccourcir, la longueur de la factorisation reste inchangée. Ceci fournit le test d'arrêt.

On compacte une factorisation comme suit :

```
PROCEDURE CompacterFactorisationLyndon (VAR f: Suitemot; VAR m: integer);
   Partant d'une factorisation (f1, ..., fm), on cherche i tel que fi ≺ fi+1 et, s'il existe, on
   rend (f1, ..., fifi+1, ..., fm).
   VAR
      i, j: integer;
   BEGIN
      i := 1;
      WHILE (i <= m - 1) AND NOT EstInferieurMot(f[i], f[i + 1]) DO
         i := i + 1;
      IF i < m THEN BEGIN                    Maintenant fi ≺ fi+1.
         Concatener(f[i], f[i + 1], f[i]);
         FOR j := i + 1 TO m - 1 DO          La suite est raccourcie.
            f[j] := f[j + 1];
         m := m - 1;
      END;
   END; { de "CompacterFactorisationLyndon" }
```

Voici un exemple détaillé d'exécution :

```
Mot donné :  0101001100100
Factorisation de longueur 12
     01  0  1  0  0  1  1  0  0  1  0  0
Factorisation de longueur 11
     01  01  0  0  1  1  0  0  1  0  0
Factorisation de longueur 10
     01  01  0  01  1  0  0  1  0  0
Factorisation de longueur 9
     01  01  001  1  0  0  1  0  0
Factorisation de longueur 8
     01  01  0011  0  0  1  0  0
Factorisation de longueur 7
     01  01  0011  0  01  0  0
Factorisation de longueur 6
     01  01  0011  001  0  0
Factorisation de Lyndon
     01  01  0011  001  0  0
```

Pour terminer cette section, nous présentons un algorithme, dû à J.-P. Duval, qui calcule la factorisation de Lyndon en temps linéaire en fonction de la longueur du mot donné.

On appelle *factorisation de Duval* d'un mot x non vide une factorisation

$$\Phi(x) = (\ell_1, \ldots, \ell_q, v)$$

et un entier $p = p(x)$ tels que $x = \ell_1 \cdots \ell_q v$ et $0 \le p < q$, et vérifiant les deux conditions :

(1) $\ell_1, \ldots, \ell_q$ sont des mots de Lyndon et v est un préfixe propre (éventuellement vide) de ℓ_q ;

(2) si $p \geq 1$, alors $\ell_1 \succeq \cdots \succeq \ell_p \succ \ell_{p+1} = \ldots = \ell_q$ et $\ell_p \gg \ell_{p+1} \cdots \ell_q v$.

Exemple. Si x est réduit à une seule lettre, $\Phi(x) = (x, \varepsilon)$. Pour $x = 0100100100$, on a $\Phi(x) = (01, 001, 001, 00)$ et $p = 1$.

Lemme 9.3.7. *Tout mot non vide admet au plus une factorisation de Duval.*

Preuve. Soit $\Phi(x) = (\ell_1, \ldots, \ell_p, \ell_{p+1}, \ldots, \ell_q, v)$ et $p = p(x)$. Si v est le mot vide, la suite

$$(\ell_1, \ldots, \ell_p, \ell_{p+1}, \ldots, \ell_q)$$

est la factorisation de Lyndon de x, donc unique. Sinon, soit $(v_1, \ldots, v_n)$ la factorisation de Lyndon de v. Alors

$$(\ell_1, \ldots, \ell_p, \ell_{p+1}, \ldots, \ell_q, v_1, \ldots, v_n)$$

est la factorisation de Lyndon de x, parce que $v_1 \prec v \prec \ell_q$. D'où l'unicité. ∎

Proposition 9.3.8. *Tout mot non vide possède une factorisation de Duval.*

Preuve. La preuve que voici est en fait un algorithme pour construire la factorisation de Duval d'un mot. Soit x un mot non vide et a une lettre, et soit

$$\Phi(x) = (\ell_1, \ldots, \ell_p, \ell_{p+1}, \ldots, \ell_q, v)$$

la factorisation de Duval de x, et $p = p(x)$. Quatre cas se présentent :
Si va est préfixe propre de ℓ_q, alors

$$\Phi(xa) = (\ell_1, \ldots, \ell_p, \ell_{p+1}, \ldots, \ell_q, va)$$

et $p(xa) = p$. En effet, cette factorisation vérifie les deux conditions ci-dessus.
Si $\ell_q = va$, alors $p(xa) = p$ et

$$\Phi(xa) = (\ell_1, \ldots, \ell_p, \ell_{p+1}, \ldots, \ell_q, va, \varepsilon)$$

Si $\ell_q \ll va$, alors $p(xa) = p$, et

$$\Phi(xa) = (\ell_1, \ldots, \ell_p, \ell_{p+1} \cdots \ell_q va, \varepsilon)$$

est une factorisation de Duval. En effet, d'après le lemme démontré plus loin, le mot $\ell_q va$ est un mot de Lyndon, d'où l'on déduit que

$$\ell_{p+1} \cdots \ell_q va = \ell_q^{q-p} va$$

est lui aussi un mot de Lyndon. Donc la condition (1) est vérifiée. Ensuite, on a $\ell_p \gg \ell_q^{q-p} va$ parce que $\ell_p \gg \ell_q^{q-p} v$, donc (2) est également vérifiée.

Reste enfin le cas où $\ell_q \gg va$. Soit

$$\Phi(va) = (\ell'_1, \ldots, \ell'_{p'}, \ell'_{p'+1}, \ldots, \ell'_{q'}, v')$$

la factorisation de Duval de va qui existe par récurrence, avec $p' = p(va)$. Alors $\Phi(xa)$ est la concaténation des deux factorisations :

$$\Phi(xa) = (\ell_1, \ldots, \ell_p, \ell_{p+1}, \ldots, \ell_q, \ell'_1, \ldots, \ell'_{p'}, \ell'_{p'+1}, \ldots, \ell'_{q'}, v')$$

et $p(xa) = p'$. En effet, la condition (1) est vérifiée pour cette factorisation parce qu'elle l'est pour la factorisation de va. Quant à la condition (2), le seul cas à examiner est celui où $p' = 0$. On doit avoir $\ell_q \gg \ell'_1 \cdots \ell'_{q'} v' = va$, ce qui est vrai par hypothèse. ∎

Pour achever la preuve, il reste à démontrer le lemme suivant.

LEMME 9.3.9. *Soit u un mot de Lyndon, soit v un préfixe propre de u et soit a une lettre. Si $u \prec va$, alors va est un mot de Lyndon.*

Preuve. Le mot u se factorise en $u = vbw$, avec b une lettre et $b \prec a$. Soit ha une fin de va et posons $v = gh$. Montrons que $va \prec ha$.

Si h est un préfixe de v, alors v se factorise en $v = hch'$ pour une lettre c et un mot h'. Le mot $hbwg$ est un conjugué de $u = hch'bw$ et, comme u est un mot de Lyndon, on a $c \preceq b$. Mais alors $c \prec a$, donc $va = hch'a \ll ha$.

Si h n'est pas préfixe de v, il y a deux possibilités : soit $v \ll h$, alors évidemment $va \ll ha$; soit $h \ll v$. Or, ce cas ne peut pas se présenter, car en effet $h \ll v$ entraîne $hwg \ll vw = u$, ce qui est impossible parce que hwg est un conjugué de u. ∎

COROLLAIRE 9.3.10. *Soit $\Phi(x) = (\ell_1, \ldots, \ell_p, \ell_{p+1}, \ldots, \ell_q, v)$ la factorisation de Duval d'un mot non vide x. Si v est le mot vide, la factorisation de Lyndon de x est $(\ell_1, \ldots, \ell_p, \ell_{p+1}, \ldots, \ell_q)$, sinon elle est la concaténation de cette factorisation et de la factorisation de Lyndon de v.* ∎

Cette observation conduit donc naturellement à l'algorithme suivant : calculer d'abord une factorisation de Duval du mot donné. Si le dernier facteur est le mot vide, la factorisation, privée du dernier facteur, est une factorisation de Lyndon. Sinon, tous les facteurs sauf le dernier font partie de la factorisation de Lyndon et on itère le procédé sur le dernier facteur. Cet algorithme s'écrit comme suit :

```
PROCEDURE FactorisationLyndonParDuval (w: mot; VAR f: Suitemot;
      VAR l: integer);
   BEGIN
      f[1] := w;
      l := 1;
      REPEAT
         FactorisationDuval(f[l], f, l)
      UNTIL EstMotVide(f[l]);
```

```
    l := l - 1                    Car le dernier facteur est le mot vide.
  END; { de "FactorisationLyndonParDuval" }
```

Les facteurs de la factorisation sont ajoutés au fur et à mesure au tableau f. La longueur de la factorisation est l. L'écriture de la procédure de factorisation de Duval est plus délicate, car il y a plusieurs indices à gérer.

```
PROCEDURE FactorisationDuval (w: mot; VAR f: Suitemot; VAR l: integer);
  Calcule, dans (f_ℓ, f_ℓ+1, ...), la factorisation de Duval du mot w. Avec les notations du
  texte, les variables i, j, k et d ont la signification suivante :
    d est la longueur de ℓ_q,
    w_i est la première lettre de ℓ_p+1,
    w_j est la première lettre de v,
    k est l'indice de la première lettre à examiner.
  VAR
    i, j, k, d, m: integer;
  PROCEDURE CopierFacteurs;                  Copie les facteurs ℓ_p+1, ..., ℓ_q dans f.
    VAR
      r: integer;
    BEGIN
      r := i;
      WHILE r + d <= j DO BEGIN
        LeFacteur(w, r, r + d - 1, f[l]);
        r := r + d; l := l + 1
      END;
    END; { de "CopierFacteurs" }
  BEGIN { de "FactorisationDuval" }
    m := Longueur(w);
    i := 1; j := 2; k := j; d := 1;          Factorisation de la première lettre.
    WHILE (k <= m) DO BEGIN
      CASE signe(w[i + k - j] - w[k]) OF
       -1: BEGIN                             Nouveau facteur par concaténation.
             k := k + 1; j := k; d := k - i
           END;
        0: BEGIN                             Prolongation.
             k := k + 1;
             IF k - j = d THEN j := k
           END;
        1: BEGIN                             Factorisation de v.
             CopierFacteurs;
             i := j; j := j + 1; k := j; d := 1
           END
      END; { case }
    END; { while }
    CopierFacteurs;                          Copie des facteurs en attente.
    LeFacteur(w, j, m, f[l]);
  END; { de "FactorisationDuval" }
```

Voici un exemple détaillé d'exécution de la procédure :

```
w = 0 0 1 0 2 0 0 1 0 2 0 0 0 1 0 0 1 0 0 0 0
|0:
|0 0:
|0 0 1:
|0 0 1:0
|0 0 1 0 2:
|0 0 1 0 2:0
|0 0 1 0 2:0 0
|0 0 1 0 2:0 0 1
|0 0 1 0 2:0 0 1 0
|0 0 1 0 2 0 0 1 0 2:
|0 0 1 0 2 0 0 1 0 2:0
|0 0 1 0 2 0 0 1 0 2:0 0
 0 0 1 0 2;0 0 1 0 2|0:
 0 0 1 0 2;0 0 1 0 2|0 0:
 0 0 1 0 2;0 0 1 0 2|0 0 0:
 0 0 1 0 2;0 0 1 0 2|0 0 0 1:
 0 0 1 0 2;0 0 1 0 2|0 0 0 1:0
 0 0 1 0 2;0 0 1 0 2|0 0 0 1:0 0
 0 0 1 0 2;0 0 1 0 2|0 0 0 1 0 0 1:
 0 0 1 0 2;0 0 1 0 2|0 0 0 1 0 0 1:0
 0 0 1 0 2;0 0 1 0 2|0 0 0 1 0 0 1:0 0
 0 0 1 0 2;0 0 1 0 2|0 0 0 1 0 0 1:0 0 0
 0 0 1 0 2;0 0 1 0 2;0 0 0 1 0 0 1|0:
 0 0 1 0 2;0 0 1 0 2;0 0 0 1 0 0 1|0 0:
 0 0 1 0 2;0 0 1 0 2;0 0 0 1 0 0 1|0 0 0:
 0 0 1 0 2;0 0 1 0 2;0 0 0 1 0 0 1|0 0 0 0:
```

Les barres verticales indiquent la position de la variable i ; les facteurs à gauche de la barre ($\ell_1, \ldots, \ell_p$ dans nos notations) sont séparés par des points-virgules ; les autres facteurs ($\ell_{p+1} \ldots, \ell_q$) sont à droite de la barre verticale et s'étendent jusqu'à la position de j, repérée par le signe deux-points. La factorisation de Duval est :

```
0  0  1  0  2
0  0  1  0  2
0  0  0  1  0  0  1
0
0
0
0
```

Elle coïncide ici avec la factorisation de Lyndon.

Il nous reste à estimer le temps d'exécution de l'algorithme, en fonction de la longueur m du mot w à factoriser. Commençons par la factorisation de Duval. A tout moment, le triplet d'entiers (i, j, k) de la procédure vérifie les inégalités $i < j \leq k$. Chaque tour dans la boucle `WHILE` remplace le triplet (i, j, k) par un nouveau triplet (i', j', k'), à savoir

$$(i', j', k') = \begin{cases} (i, k, k+1) & \text{si } w_{i+k-j} \prec w_k \\ (i, j, k+1) & \text{si } w_{i+k-j} = w_k \text{ et } k+1-j \neq d \\ (i, k+1, k+1) & \text{si } w_{i+k-j} = w_k \text{ et } k+1-j = d \\ (j, j+1, j+1) & \text{si } w_k \prec w_{i+k-j} \end{cases}$$

De plus, lorsque la dernière situation se produit, on a $k - j < d$. Il en résulte que $i' + k' > i + k$. En effet, cette inégalité est évidente dans les trois premiers cas et, dans le dernier, on a $i+k < i+d+j \leq i+j < i'+k'$. Comme $k \leq m$, la valeur de $i+k$ n'excède jamais $2m$, et le nombre de tours dans l'algorithme est donc majoré par $2m$. Mise à part la recopie de facteurs, le temps d'exécution de chaque tour de boucle est constant. Le total des recopies ne dépasse pas la longueur du mot, puisque tout facteur n'est recopié qu'une seule fois. Le temps d'exécution d'un appel de la procédure de factorisation de Duval est donc linéaire en fonction de la longueur m du mot. Notons ce temps $\delta(m)$. Si la factorisation de Lyndon n'est pas achevée, la factorisation de Duval retourne un suffixe non vide v du mot w restant à factoriser. Or ce suffixe est un préfixe du dernier facteur de la factorisation déjà obtenue, donc en particulier sa longueur est au plus $m/2$. Il en résulte que le coût total de la factorisation de Lyndon par l'algorithme de Duval est majoré par $\delta(m) + \delta(m/2) + \delta(m/4) + \cdots$. Si $\delta(m) \leq cm$, ce nombre est donc majoré par $2cm$.

9.4 Suite de Thue-Morse

9.4.1 Enoncé : suite de Thue-Morse

On considère la suite $s = (s_n)_{n \geq 0}$ définie par

$$s_n = (-1)^{b(n)}$$

où, pour tout entier $n \geq 0$,

$$b(n) = \sum_{i=0}^{\infty} b_i(n)$$

les $b_i(n)$ étant définis par le développement binaire de n :

$$n = \sum_{i=0}^{\infty} b_i(n) 2^i \qquad b_i(n) \in \{0, 1\}$$

Les premiers termes de s sont : $1, -1, -1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, \ldots$

1.– Démontrer que $s_{2n} = s_n$, $s_{2n+1} = -s_n$ et écrire une procédure qui calcule s_n pour $0 \leq n \leq N$ (on pourra prendre $N = 100$).

Une suite $(t_0, \ldots, t_{k-1})$ est un *facteur* de longueur k de s s'il existe n tel que

$$t_i = s_{n+i} \quad \text{pour} 0 \leq i \leq k-1 \tag{$*$}$$

On se propose de calculer les facteurs de s. Le plus petit entier $n \geq 0$ tel que $(*)$ soit vérifiée est noté $R(t_0, \ldots, t_{k-1})$. On pose

$$r(k) = \max R(t_0, \ldots, t_{k-1})$$

où le maximum porte sur tous les facteurs $(t_0, \ldots, t_{k-1})$ de longueur k de s et $F(k) = k + r(k)$. Par exemple, la suite s possède les 6 facteurs

$$(1, 1, -1),\ (1, -1, 1),\ (1, -1, -1),\ (-1, 1, 1),\ (-1, 1, -1),\ (-1, -1, 1)$$

de longueur 3 et on a $F(3) = 8$. On admettra dans les questions 2 et 3 que $F(1) = 2$, $F(2) = 7$ et que

$$F(k) \leq 2F(1 + [k/2]) \qquad k \geq 3 \tag{$**$}$$

(On note $[x]$ la partie entière de x). Soit $P(k)$ le nombre de facteurs de longueur k de la suite s. On a $P(1) = 2$, $P(2) = 4$, $P(3) = 6$. On admettra dans les questions 2 et 3 que

$$\begin{aligned} P(2k) \quad &= P(k+1) + P(k) \\ P(2k+1) &= 2P(k+1) \end{aligned} \tag{$***$}$$

2.– Démontrer que $F(k) \leq 8(k-2)$ pour $k \geq 3$ et que $P(k) \leq 4(k-1)$ pour $k \geq 2$.

3.– Le *poids* d'une suite $(t_0, \ldots, t_{k-1})$ est par définition l'entier $\sum_{i=0}^{k-1} t_i 3^i$. Utiliser les résultats de la question précédente pour écrire un programme qui prend en argument un entier k, qui calcule, une fois et une seule, tous les facteurs de s de longueur k, et qui les affiche par ordre de poids croissant. On testera avec $k = 3, 4, 5$.

4.– Démontrer $(**)$ et $(***)$.

On se fixe un entier $N > 0$ et on considère $S_N = (s_0, \ldots, s_N)$. Une *occurrence* d'une suite finie $(t_0, \ldots, t_{k-1})$ comme sous-mot dans S_N est une suite d'indices $(n_0, \ldots, n_{k-1})$, avec $0 \leq n_0 < n_1 < \cdots < n_{k-1} \leq N$ telle que $t_i = s_{n_i}$ pour $i = 0, \ldots, k-1$.

5.– Ecrire une procédure qui prend en donnée une suite $(t_0, \ldots, t_{k-1})$ et qui calcule le nombre d'occurrences de cette suite comme sous-mot dans S_N. On testera avec $N = 12$, $k = 4$ et $(t_0, \ldots, t_{k-1}) = (1, -1, -1, 1)$.

6.– (Question indépendante des précédentes.) Prouver que le produit

$$\prod_{n=0}^{\infty} \left(\frac{2n+1}{2n+2}\right)^{s_n}$$

converge (sa limite est $1/\sqrt{2}$).

Version 15 janvier 2005

9.4.2 Solution : suite de Thue-Morse

On considère la *suite de Thue-Morse* $s = (s_n)_{n\geq 0}$ définie par

$$s_n = (-1)^{b(n)} \quad \text{avec} \quad b(n) = \sum_{i=0}^{\infty} b_i(n)$$

les $b_i(n)$ étant définis par le développement binaire de n :

$$n = \sum_{i=0}^{\infty} b_i(n) 2^i \qquad b_i(n) \in \{0, 1\}$$

Une suite finie $t = (t_0, \ldots, t_{k-1})$ est un *facteur* de longueur k de s s'il existe n tel que

$$t_i = s_{n+i} \quad \text{pour} 0 \leq i \leq k-1 \tag{4.1}$$

Tout entier n pour lequel (4.1) est vérifié est un *début d'occurrence* de t. Nous commençons par quelques lemmes simples.

Lemme 9.4.1. *Pour tout $n \geq 0$, on a $s_{2n} = s_n$ et $s_{2n+1} = -s_n$.*

Preuve. Clairement, $b(2n) = b(n)$ et $b(2n+1) = 1 + b(n)$, d'où les formules. ∎

Il en résulte en particulier que $(s_{2n}, s_{2n+2}, \ldots, s_{2n+2k}) = (s_n, s_{n+1}, \ldots, s_{n+k})$ et que l'application

$$(s_n, \ldots, s_{n+k-1}) \mapsto (s_{2n}, \ldots, s_{2(n+k)-1})$$

est une bijection de l'ensemble des facteurs de longueur k sur l'ensemble des facteurs de longueur $2k$ qui ont un début d'occurrence pair.

Lemme 9.4.2. *Aucun des quatre mots*

$$(1, 1, 1),\ (-1, -1, -1),\ (1, -1, 1, -1, 1),\ (-1, 1, -1, 1, -1)$$

n'est facteur de s.

Preuve. Supposons, en raisonnant par l'absurde, que $s_n = s_{n+1} = s_{n+2}$ pour un entier n. Si n est pair, alors par le lemme précédent $s_n = -s_{n+1}$. De même, si n est impair, alors $s_{n+1} = -s_{n+2}$. Ceci exclut les deux premiers facteurs. Considérons ensuite, en raisonnant par l'absurde, un début d'occurrence n de $(1, -1, 1, -1, 1)$ dans la suite de Thue-Morse. Si n est pair, $n = 2m$, alors $s_m = s_{m+1} = s_{m+2} = 1$ par le lemme précédent ; si n est impair, $n = 2m + 1$, alors, à nouveau par le lemme, $s_m = s_{m+1} = s_{m+2} = -1$. Les deux cas sont impossibles. ∎

Soit $t = (t_0, \ldots, t_{k-1})$ un facteur de s. Le plus petit début d'occurrence de t dans s est noté $R(t)$. On pose

$$r(k) = \max R(t)$$

où le maximum porte sur tous les facteurs t de longueur k de s et on pose $F(k) = k + r(k)$. La connaissance, ou au moins une majoration de $F(k)$ permet de calculer tous les facteurs de longueur k de s, puisque chaque facteur apparaît dans la suite finie $(s_0, \ldots, s_{F(k)-1})$. Le lemme suivant est utile.

LEMME 9.4.3. *Soit t un facteur de la suite de Thue-Morse, de longueur au moins 4. Si n et m sont deux débuts d'occurrence de t, alors $n \equiv m \pmod 2$.*

Preuve. Supposons le contraire. Alors il existe n pair et m impair tels que $s_{n+i} = s_{m+i}$ pour $i = 0, 1, 2, 3$. On a $s_n = -s_{n+1}$ et $s_{n+2} = -s_{n+3}$, parce que n est pair. De même, $s_m = -s_{m-1}$ et $s_{m+2} = -s_{m-1}$, parce que m est impair. En combinant ces égalités, on obtient

$$-s_{m-1} = s_m = -s_{m+1} = s_{m+2} = -s_{m+3}$$

ce qui contredit le lemme précédent. ∎

PROPOSITION 9.4.4. *Pour $k \geq 2$, on a $r(2k) = r(2k+1) = 1 + 2r(k+1)$.*

Preuve. Clairement, la fonction r est croissante au sens large. Soit $k \geq 2$ et soit $u = (s_m, \ldots, s_{m+k})$ un facteur de longueur $k+1$, avec $m = R(u) = r(k+1)$. Le mot

$$t = (s_{2m+1}, \ldots, s_{2m+2k})$$

est un facteur de longueur $2k$. On a $R(t) = 2m + 1$. En effet, sinon on aurait $R(t) = 2n + 1$ pour un entier $n < m$, car t n'a que des débuts d'occurrence impairs. Or, la suite $(s_n, \ldots, s_{n+k})$ est égale à u, et donc $R(u) \leq n$ contrairement à l'hypothèse. Par conséquent, $r(2k+1) \geq r(2k) \geq R(t) = 1 + 2r(k+1)$.

Réciproquement, soit $t = (s_n, \ldots, s_{n+2k})$ un facteur de longueur $2k+1$ tel que $n = R(t) = r(2k+1)$. Si n est pair et $n = 2m$, alors $u = (s_n, s_{n+2}, \ldots, s_{n+2k})$ est un facteur de s de longueur $k+1$, et $R(u) = m$. Si $n = 2m+1$ est impair, alors $s_{2m} = -s_n$ et $u = (s_{n-1}, s_{n+1}, \ldots, s_{n+2k-1})$ est facteur de longueur $k+1$, avec $R(u) = m$. Dans les deux cas, $n = r(2k+1) \leq 2m + 1 \leq 2r(k+1) + 1$. ∎

COROLLAIRE 9.4.5. *On a $F(1) = 2$, $F(2) = 7$ et $F(k) \leq 2F(1 + [k/2])$ pour $k \geq 3$. En particulier, $F(k) \leq 8(k-2)$ pour $k \geq 3$.*

Preuve. Les deux premières valeurs s'obtiennent par inspection. Comme $F(3) = 8$, la formule est vraie pour $k = 3$. Ensuite, on a $r(k) = 1 + 2r(1 + [k/2])$ pour $k \geq 4$, donc $F(k) = 1 + k + 2r(1 + [k/2]) \leq 2(1 + [k/2] + r(1 + [k/2])) = F(1 + [k/2])$. Enfin, la dernière inégalité est vraie pour $k = 3$ et, par récurrence, on obtient

$$F(k) \leq 2F(1 + [k/2]) \leq 2 \cdot 8([k/2] - 1) = 8(2[k/2] - 2) \leq 8(k-2)$$

∎

On peut en fait obtenir une expression close pour $r(k)$. On vérifie en effet facilement à partir de la relation de récurrence que, pour $n \geq 0$,

$$r(k) = 3 \cdot 2^{n+1} - 1 \qquad (2^n + 2 \leq k \leq 2^{n+1} + 1)$$

Soit $P(k)$ le nombre de facteurs de longueur k de la suite s. On a $P(1) = 2$, $P(2) = 4$, et comme les mots $(1, 1, 1)$ et $(-1, -1, -1)$ ne sont pas facteurs, on a $P(3) = 6$.

PROPOSITION 9.4.6. *Pour tout $k \geq 2$, on a*

$$\begin{aligned} P(2k) &= P(k+1) + P(k) \\ P(2k+1) &= 2P(k+1) \end{aligned}$$

Preuve. L'application

$$(s_n, \ldots, s_{n+k}) \mapsto (s_{2n}, \ldots, s_{2(n+k)-1})$$

est une bijection de l'ensemble des facteurs de longueur $k+1$ sur l'ensemble des facteurs de longueur $2k$ dont le début d'occurrence est pair, et l'application

$$(s_n, \ldots, s_{n+k}) \mapsto (s_{2n+1}, \ldots, s_{2(n+k)})$$

est une bijection de l'ensemble des facteurs de longueur $k+1$ sur l'ensemble des facteurs de longueur $2k$ dont le début d'occurrence est impair. Ceci prouve la première formule. De même, l'application

$$(s_n, \ldots, s_{n+k}) \mapsto (s_{2n}, \ldots, s_{2(n+k)})$$

est une bijection des facteurs de longueur $k+1$ sur les facteurs de longueur $2k+1$ dont le début d'occurrence est pair, et

$$(s_n, \ldots, s_{n+k}) \mapsto (s_{2n+1}, \ldots, s_{2(n+k)-1})$$

est une bijection des facteurs de longueur $k+1$ sur les facteurs de longueur $2k+1$ dont le début d'occurrence est impair. D'où la deuxième formule. ∎

Les formules de récurrence de la proposition donnent également une formule close pour $P(k)$. Nous nous contentons de la majoration qui suit.

COROLLAIRE 9.4.7. *On a $P(1) = 2$, et $P(k) \leq 4P(k-1)$ pour $k \geq 2$.*

Preuve. L'inégalité est immédiate pour $k = 2, 3$. Pour $k \geq 4$, l'inégalité résulte de la proposition. ∎

PROPOSITION 9.4.8. *On a*

$$\prod_{n=0}^{\infty} \left(\frac{2n+1}{2n+2} \right)^{s_n} = \frac{1}{\sqrt{2}}$$

Preuve. Posons

$$P = \prod_{n=0}^{\infty} \left(\frac{2n+1}{2n+2} \right)^{s_n} \qquad Q = \prod_{n=1}^{\infty} \left(\frac{2n}{2n+1} \right)^{s_n} \qquad R = \prod_{n=1}^{\infty} \left(\frac{n}{n+1} \right)^{s_n}$$

Prouvons que le premier produit converge (pour les autres, le raisonnement est le même) en appliquant la règle d'Abel à la série des logarithmes : la suite de terme général $|\sum_{m=0}^{n} s_m|$ ne prend que les deux valeurs 0 ou 1, et la suite des $\ln \frac{2n+1}{2n+2}$ tend vers 0 de façon monotone. Donc la série

$$\sum_{n=0}^{\infty} s_n \ln \frac{2n+1}{2n+2}$$

converge. Ensuite, on a

$$PQ = \frac{1}{2} \prod_{n=1}^{\infty} \left(\frac{2n+1}{2n+2} \frac{2n}{2n+1} \right)^{s_n} = \frac{1}{2} \prod_{n=1}^{\infty} \left(\frac{n}{n+1} \right)^{s_n} = R/2$$

Par ailleurs,

$$\begin{aligned} R &= \prod_{n=1}^{\infty} \left(\frac{2n}{2n+1} \right)^{s_{2n}} \prod_{n=0}^{\infty} \left(\frac{2n+1}{2n+2} \right)^{s_{2n+1}} \\ &= \prod_{n=1}^{\infty} \left(\frac{2n}{2n+1} \right)^{s_n} \left(\prod_{n=0}^{\infty} \left(\frac{2n+1}{2n+2} \right)^{s_n} \right)^{-1} = \frac{Q}{P} \end{aligned}$$

Il en résulte que $P^2 = 1/2$. ∎

9.4.3 Programme : suite de Thue-Morse

On calcule les premiers termes de la suite de Thue-Morse par les formules du lemme 9.4.1. Dans les procédures qui suivent, on utilise la bibliothèque de manipulation des mots introduits au début de ce chapitre. Pour disposer d'un début assez long de la suite de Thue-Morse, il est utile d'introduire un type spécifique qui permet de ranger des mots longs. On écrira donc :

```
CONST
   GrandeLongueur = 255;
TYPE
   motlong = ARRAY[0..GrandeLongueur] OF integer;
VAR
   s: motlong;                    Le début du mot de Thue-Morse.
```

La procédure de calcul est :

```
PROCEDURE ThueMorse (VAR s: motlong);
   VAR
      n: integer;
   BEGIN
      s[0] := 1;
      FOR n := 0 TO GrandeLongueur DIV 2 DO BEGIN
         s[2 * n] := s[n];
```

```
      s[2 * n + 1] := -s[n]
    END
  END; { de "ThueMorse" }
```

Voici le résultat. Pour plus de lisibilité, on écrit le signe plus à la place de 1 et un point à la place de -1 :

```
Suite de Thue-Morse :
+..+.++..++.+..+.++.+..++..+.++..++.+..++..+.++.+..+.++..++.+..+
.++.+..++..+.++.+..+.++..++.+..++..+.++..++.+..+.++.+..++..+.++.
.++.+..++..+.++.+..+.++..++.+..++..+.++..++.+..+.++.+..++..+.++.
+..+.++..++.+..+.++.+..++..+.++..++.+..++..+.++.+..+.++..++.+..+
```

Désormais, le début de la suite de Thue-Morse est supposé calculé et disponible dans le tableau `s`. Extraire un facteur de la suite se fait très simplement :

```
PROCEDURE LeFacteurTM (VAR t: mot; k, n: integer);
  Détermine t = (s_n, ..., s_{n+k-1}).
  VAR
    i: integer;
  BEGIN
    FOR i := 1 TO k DO
      t[i] := s[n + i - 1];
    FixerLongueur(t, k)
  END; { de "LeFacteurTM" }
```

Pour établir une liste des facteurs de longueur k de la suite de Thue-Morse, on extrait successivement les facteurs dont le début d'occurrence varie entre 0 et $r(k)$. Puis, chaque facteur est inséré, s'il n'y figure pas déjà, dans une liste de mots, rangés par poids croissant. Il n'est pas difficile de voir que cet ordre est l'opposé de l'ordre lexicographique sur les images miroirs (les mots lus de la droite vers la gauche). On pourrait donc utiliser la procédure d'insertion élaborée pour les mots de Lyndon. Nous préférons écrire une procédure spécifique qui utilise le poids comme critère de comparaison. Commençons par le calcul du poids :

```
FUNCTION Poids (VAR t: mot): integer;
  Calcul du poids d'un mot t par le schéma de Horner.
  VAR
    i, p: integer;
  BEGIN
    p := 0;
    FOR i := Longueur(t) DOWNTO 1 DO
      p := 3 * p + t[i];
    Poids := p
  END; { de "Poids" }
```

On peut ne pas calculer $r(k)$ explicitement, mais utiliser une majoration, telle que nous l'avons établie précédemment. Le calcul explicite de $r(k)$ se fait de manière naturelle en traduisant les relations de récurrence de la proposition 9.4.4 en une procédure récursive :

```
FUNCTION r (k: integer): integer;
   BEGIN
      IF k = 1 THEN r := 1
      ELSE IF k <= 3 THEN r := 5
      ELSE
         r := 1 + 2 * r(1 + k DIV 2)
   END; { de "r" }
```

Cette procédure peut bien entendu s'écrire aussi de manière itérative, mais nous préférons plutôt donner une évaluation de la formule close :

```
FUNCTION r (k: integer): integer;
   VAR
      DeuxPuissanceN: integer;
   BEGIN
      IF k = 1 THEN r := 1
      ELSE IF k <= 3 THEN r := 5
      ELSE BEGIN
         DeuxPuissanceN := 2;
         WHILE DeuxPuissanceN + 2 <= k DO
            DeuxPuissanceN := 2 * DeuxPuissanceN;
         r := 3 * DeuxPuissanceN - 1
      END
   END; { de "r" }
```

Voici le calcul des facteurs de longueur k :

```
PROCEDURE LesFacteursTM (k: integer; VAR a: SuiteMot; VAR l: integer);
   Calcule, dans a, les facteurs de longueur k de la suite de Thue-Morse.
   VAR
      n: integer;
      t: mot;
      pa: suite;
   BEGIN
      l := 0;
      FOR n := 0 TO r(k) DO BEGIN
         LeFacteurTM(t, k, n);
         InsererMotParPoids(t, poids(t), a, pa, l)
      END;
   END; { de "LesFacteursTM" }
```

L'insertion est réalisée par la procédure ci-dessous, inspirée de la procédure `InsererMot`. Le poids de chaque mot n'est calculé qu'une seule fois, pour rendre les comparaisons plus rapides. Les poids sont rangés dans un tableau de type `Suite` défini au chapitre précédent (nous avons ici augmenté la longueur maximale d'une suite) :

```
CONST
        LongueurSuite = 48;
TYPE
```

```
        suite = ARRAY[0..LongueurSuite] OF integer;
```

La majoration du nombre de facteurs démontrée plus haut dit quelle valeur choisir en fonction de la longueur des facteurs que l'on veut calculer (ici 8).

```
PROCEDURE InsererMotParPoids (VAR t: mot; p: integer; VAR c: SuiteMot;
   VAR pc: suite; VAR m: integer);
   Insertion d'un mot t de poids p dans une suite c de mots de poids croissants. L'entier
   pc_k contient le poids du mot c_k.
   VAR
      j, k: integer;
   BEGIN
      k := 1;
      WHILE (k <= m) AND (p < pc[k]) DO
         k := k + 1;                          On avance tant que poids(c_k) < p.
      IF k = m + 1 THEN BEGIN                 Si la fin de c est atteinte,
         m := m + 1;                          on insère t à la fin;
         c[m] := t; pc[m] := p
      END
      ELSE IF p <> pc[k] THEN BEGIN           si c_k = t, on ne fait rien;
         FOR j := m DOWNTO k DO               sinon, on décale la fin de c,
            BEGIN
            c[j + 1] := c[j]; pc[j + 1] := pc[j]
         END;
         c[k] := t; pc[k] := p;               puis on insère u.
         m := m + 1
      END
   END; { de "InsererMotParPoids" }
```

On obtient les résultats numériques suivants :

```
Facteurs de longueur  1
  1
 -1
Facteurs de longueur  2
  1  1
 -1  1
  1 -1
 -1 -1
Facteurs de longueur  3
 -1  1  1
  1 -1  1
 -1 -1  1
  1  1 -1
 -1  1 -1
  1 -1 -1
Facteurs de longueur  4
  1 -1  1  1
```

```
 -1 -1  1  1
  1  1 -1  1
 -1  1 -1  1
  1 -1 -1  1
 -1  1  1 -1
  1 -1  1 -1
 -1 -1  1 -1
  1  1 -1 -1
 -1  1 -1 -1
     ...
Facteurs de longueur  8
  1  1 -1 -1  1 -1  1  1
 -1  1 -1 -1  1 -1  1  1
  1 -1  1  1 -1 -1  1  1
  1  1 -1  1 -1 -1  1  1
 -1 -1  1 -1  1  1 -1  1
  1  1 -1 -1  1  1 -1  1
 -1  1  1 -1 -1  1 -1  1
  1 -1  1 -1 -1  1 -1  1
 -1  1 -1  1  1 -1 -1  1
  1 -1 -1  1  1 -1 -1  1
 -1  1  1 -1  1 -1 -1  1
  1 -1 -1  1 -1  1  1 -1
 -1  1  1 -1 -1  1  1 -1
  1 -1  1 -1 -1  1  1 -1
 -1  1 -1  1  1 -1  1 -1
  1 -1 -1  1  1 -1  1 -1
 -1 -1  1  1 -1 -1  1 -1
  1  1 -1  1 -1 -1  1 -1
 -1 -1  1 -1  1  1 -1 -1
 -1  1 -1 -1  1  1 -1 -1
  1 -1  1  1 -1  1 -1 -1
 -1 -1  1  1 -1  1 -1 -1
```

Nous terminons par le calcul du nombre d'occurrences d'un mot t donné comme sous-mot dans le préfixe $S_N = (s_0, \ldots, s_{N-1})$ de la suite de Thue-Morse. Si t est de longueur k, il s'agit d'engendrer les parties de $\{0, \ldots, N-1\}$ à k éléments et de vérifier si les valeurs de la suite S_N pour ces indices coïncident avec les éléments du mot t. Nous utilisons donc les procédures `PremierePartieRestreinte` et `PartieSuivanteRestreinte` du chapitre précédent pour engendrer les sous-ensembles à k éléments. Dans la procédure suivante, `x` représente une partie à k éléments de $\{1, \ldots, n\}$.

```
FUNCTION EstSousMotTM (t: mot; n, k: integer; x: suite): boolean;
   Teste si t est sous-mot de (s_0, ..., s_{n-1}) aux positions données par le sous-ensemble x
   de {1, ..., n}.
   VAR
      p, j: integer;
      coincide: boolean;
```

```
   BEGIN
      p := 0; j := 1;
      coincide := true;
      WHILE (j <= k) AND coincide DO BEGIN
         REPEAT p := p + 1
         UNTIL x[p] = 1;
         coincide := (s[p - 1] = t[j]);
         j := j + 1
      END;
      EstSousMotTM := coincide
   END; { de "EstSousMotTM" }
```

On utilise cette procédure pour compter le nombre d'occurrences comme suit :

```
PROCEDURE OccurrencesSousMotTM (t: mot; n: integer;
      VAR NombreOccurrences: integer);
   VAR
      k: integer;
      x: Suite;
      derniere: boolean;
   BEGIN
      NombreOccurrences := 0;
      k := Longueur(t);
      PremierePartieRestreinte(x, n, k);
      REPEAT
         IF EstSousMotTM(t, n, k, x) THEN
            NombreOccurrences := NombreOccurrences + 1;
         PartieSuivanteRestreinte(x, n, k, derniere)
      UNTIL derniere
   END; { de "OccurrencesSousMotTM" }
```

Voici un exemple de résultat :

```
Mot t = 1 -1 -1 1
N = 7
Occurrences :
1111000
1110010
1100110
1010110
1110001
1100101
1010101

Le mot a 7 occurrences
```

Le même mot a 37 occurrences comme sous-mot dans S_{12}.

Notes bibliographiques

Une référence générale pour la combinatoire des mots est :

M. Lothaire, *Combinatorics on Words*, Reading, Addison-Wesley, 1983.

Les mots de Lukasiewicz sont l'archétype des notations préfixées ou «polonaises». Elles permettent d'écrire les expressions (et notamment les expressions arithmétiques) sans parenthèses, tout en respectant la priorité des opérateurs. Les mots de Lyndon ont été introduits par Lyndon sous le nom de *standard lexicographic sequences.* Les mots de Lyndon sur l'ensemble A donnent en fait une base de l'algèbre de Lie libre sur A. L'algorithme de Duval est dans :

J.-P. Duval, Factorizing words over an ordered alphabet, *J. Algorithms* **4** (1983), p. 363–381.

J.-P. Duval a aussi donné un algorithme efficace pour engendrer les mots de Lyndon de longueur bornée en ordre lexicographique.

La suite de Thue-Morse a été étudiée en détail par A. Thue et par M. Morse qui ont notamment établi l'essentiel des résultats présentés ici. Cette suite a de nombreuses autres propriétés ; le fait qu'elle est *sans cube*, c'est-à-dire ne contient pas trois occurrences consécutives du même facteur, a été exploité dans la preuve, par Adjan et Novikov, de l'existence de groupes contredisant la conjecture de Burnside. Elle est l'un des premiers exemples cités en théorie ergodique. Parmi ses définitions équivalentes, mentionnons la possibilité de l'engendrer par itération d'un morphisme : on remplace 1 par $(1, -1)$ et -1 par $(-1, 1)$, et on recommence sur chaque terme de la suite. La suite de Thue-Morse est invariante pour cette substitution. On peut également la reconnaître par un automate fini. Enfin, elle peut être définie comme série algébrique sur le corps à deux éléments. Cette suite et de nombreuses autres de nature semblable sont regroupées sous le terme de *suites automatiques.* Voir l'article de synthèse de :

J.-P. Allouche, Automates finis en théorie des nombres, *Expo. Math.* **5** (1987), p. 239–266.

Partie IV

Géométrie

Chapitre 10

Géométrie algorithmique

La géométrie algorithmique est une branche récente de l'informatique théorique qui a pour objet la résolution efficace des problèmes géométriques rencontrés, par exemple, en CAO (conception assistée par ordinateur), en robotique ou encore en synthèse d'image. Ces problèmes se situent, en général, en basse dimension et portent sur les objets élémentaires de la géométrie : points, polygones, cercles, etc.

Citons, à titre d'exemple, le problème du déménageur de piano : comment déplacer un objet (le «piano») dans un environnement formé d'obstacles, d'une position à une autre en évitant toute collision avec les obstacles ?

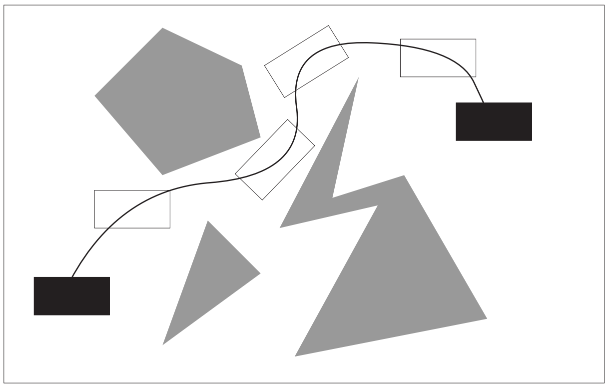


Figure 10.0.1: Le problème du déménageur de piano.

La résolution de ce type de problèmes fait appel à tout un arsenal d'algorithmes variés, pour effectuer des opérations de recherche et de tri, pour coder des arrangements d'objets géométriques, pour calculer des objets géométriques associés à d'autres objets géométriques, et ceci avec la contrainte d'effectuer les opérations de manière efficace. Qu'entend-on par efficacité ?

Considérons le problème suivant. Etant donnés 10 points du plan, calculer les points extrêmes, c'est-à-dire les points qui se situent sur leur enveloppe convexe. Une solution est de trier angulairement autour de chaque point P les autres points ; si l'un des angles entre deux directions consécutives est supérieur à $180°$ alors P est un point extrême. Rien de plus simple ! Mais que se passe-t-il s'il y a 10 000 points au lieu de 10 ? Le tri de n nombres nécessitant de l'ordre de $n \log_2 n$ opérations élémentaires (i.e. comparaisons de deux nombres), et ce tri devant être effectué autour de chaque point, l'ensemble du calcul nécessite plus de 100 millions d'opérations pour $n = 10\,000$ points. En fait, une analyse plus fine de la géométrie du problème permet de trouver un algorithme, pratiquement aussi simple, ne nécessitant que de l'ordre de $n \log_2 n$ opérations. L'ensemble du calcul peut alors être effectué en quelques secondes sur un ordinateur. De manière générale nous dirons qu'un algorithme a une *complexité* en $O(f(n))$ s'il ne requiert pas plus de $cf(n)$ opérations élémentaires (comparaison, addition, multiplication) pour traiter des données de taille n, pour une certaine constante c. On peut montrer que le tri de n nombres a pour complexité $O(n \log_2 n)$.

Dans ce chapitre, nous étudions trois structures de base de la géométrie algorithmique : l'enveloppe convexe d'une famille finie de points, la triangulation de Delaunay d'une famille finie de points et la triangulation d'un polygone simple. L'accent est mis sur les aspects géométriques et combinatoires : équivalence entre polyèdres convexes bornés et enveloppes convexes, dualité entre triangulation de Delaunay et diagramme de Voronoï, coloriage d'une triangulation d'un polygone simple. Les aspects algorithmiques, deuxième acte de la «pièce» géométrie algorithmique, sont également abordés ; en l'absence, au niveau où nous nous plaçons, d'outils algorithmiques adéquats nous nous bornons aux algorithmes les plus simples ; ce ne sont pas toujours les plus rapides.

10.1 Polyèdres et enveloppes convexes

Soit E un espace affine réel euclidien de dimension d.

Une partie C de E est dite *convexe* si pour tout $x, y \in C$, le segment $[x, y] = \{\lambda x + (1-\lambda)y \mid \lambda \in [0,1]\}$ est inclus dans C. La *dimension* d'un convexe non vide est, par définition, la dimension du sous-espace affine engendré par ce convexe. Les convexes de dimension 0 sont les points ; ceux de dimension 1 sont les droites, les demi-droites et les segments. La dimension d'un convexe caractérise la vacuité de son intérieur : un convexe $C \subset E$ est d'intérieur non vide si et seulement si sa dimension est celle de l'espace E. Rappelons enfin que l'intérieur relatif d'une partie X de E est, par définition, l'intérieur de X dans l'espace affine engendré par X.

Comme exemples de convexes citons : les sous-espaces affines (en particulier les points, les droites, les hyperplans) ; les boules ouvertes ou fermées ; les demi-espaces, c'est-à-dire, les composantes connexes du complémentaire d'un hyperplan ; enfin on vérifie aisément que la famille des convexes est stable par intersection (finie ou non). Cette dernière propriété permet de définir deux classes importantes et naturelles de convexes : les *polyèdres* et les *enveloppes*.

On appelle *polyèdre convexe* toute intersection d'un nombre fini de demi-espaces fermés. Les plus célèbres d'entre eux, en dimension trois, sont les cinq polyèdres réguliers de Platon : le cube, le tétraèdre régulier, l'octaèdre, le dodécaèdre et l'icosaèdre.

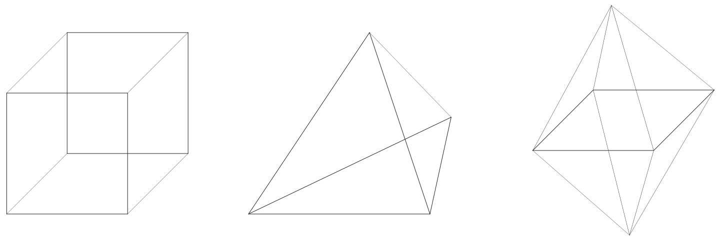


Figure 10.1.1: Le cube, le tétraèdre et l'octaèdre.

On appelle *enveloppe convexe* d'une partie $X \subset E$, et on note $\mathcal{E}(X)$, le plus petit convexe (pour la relation d'inclusion) de E contenant X ; on vérifie aisément que $\mathcal{E}(X)$ est l'ensemble des barycentres des parties finies de points de X affectés de masses positives :

$$\mathcal{E}(X) = \left\{ \sum_{x \in X'} \lambda_x x \mid \lambda_x \geq 0, \sum_{x \in X'} \lambda_x = 1, X' \text{ partie finie de } X \right\} \tag{1.1}$$

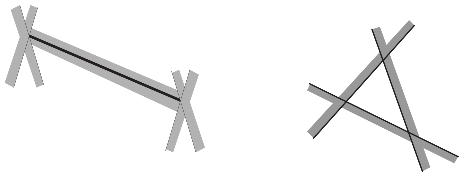
L'enveloppe convexe de $k+1$ points affinement indépendants est appelée un *simplexe* de dimension k. Les points, les segments, les triangles, les tétraèdres sont les simplexes de dimensions respectives $0, 1, 2$ et 3.

En fait il y a identité entre les polyèdres convexes *bornés* et les enveloppes convexes des parties *finies*. La démonstration de ce résultat, qui peut paraître évident pour les dimensions 2 et 3, n'est pas triviale. Elle nécessite l'étude de la structure des polyèdres convexes (qu'est-ce qu'une face ?) et l'introduction de la notion de polaire d'un convexe. Nous commençons par la structure des polyèdres.

De toute écriture d'un polyèdre convexe P comme intersection finie de demi-espaces fermés R on peut extraire, en supprimant tour à tour les demi-espaces superflus, une écriture *minimale*

$$P = \bigcap_{i=1}^{n} R_i \tag{1.2}$$

c'est-à-dire telle que P est un sous-ensemble strict de $\bigcap_{j \neq i} R_j$ pour tout $i = 1, \ldots, n$. L'exemple du segment dans le plan (voir la figure 10.1.2) montre que cette écriture n'est en général pas unique ; cependant c'est le cas si le polyèdre P est d'intérieur non vide.



(a) (b)

Figure 10.1.2: L'écriture minimale n'est pas toujours unique.

THÉORÈME 10.1.1. *Soit P un polyèdre convexe de E d'intérieur non vide et soit $\mathcal{F}$ l'ensemble des demi-espaces R, contenant P, dont la frontière intersecte P selon un convexe de codimension* 1*. Alors*

(*i*) *$\mathcal{F}$ est fini;*
(*ii*) *si $R \in \mathcal{F}$ alors $\operatorname{Fr}(R) \cap P$ est un polyèdre convexe d'intérieur non vide dans $\operatorname{Fr}(P)$, noté $\operatorname{Face}_R P$;*
(*iii*) *$P = \bigcap_{\mathcal{F}} R$ est l'unique écriture minimale de P;*
(*iv*) *$\operatorname{Fr}(P) = \bigcup_{\mathcal{F}} \operatorname{Face}_R P$.*

Preuve. Soit $P = \bigcap_{i=1}^n R_i$ une écriture minimale de P; on note H_i l'hyperplan frontière de R_i. Nous montrons que $R_i \in \mathcal{F}$. La seule difficulté est de prouver que $H_i \cap P$ est d'intérieur non vide dans H_i. Soit a un point de l'intérieur de P et posons $P' = \bigcap_{j \neq i} R_j$. L'écriture étant minimale, $P' \setminus P$ est non vide; soit x un point de $P' \setminus P$ et soit $\{y\} =]a, x[\cap H_i$; le segment $]a, x[$ étant inclus dans l'intérieur de P', il en est de même de y; donc $H_i \cap P = H_i \cap P'$ est d'intérieur non vide dans H_i.

Inversement soit $R \in \mathcal{F}$ de frontière H; par hypothèse, H intersecte la frontière de P selon un convexe de codimension 1; cette frontière étant incluse dans l'union finie des H_i, H coïncide avec l'un des H_i et $R = R_i$.

Pour démontrer le dernier point on écrit

$$\begin{aligned}
\operatorname{Fr}(P) &= P \cap \overline{E \setminus P} = P \cap \left(\bigcup_{R \in \mathcal{F}} \overline{E \setminus R} \right) \\
&= \bigcup_{R \in \mathcal{F}} \left(\overline{E \setminus R} \cap P \right) = \bigcup_{R \in \mathcal{F}} (\operatorname{Fr}(R) \cap P) \\
&= \bigcup_{R \in \mathcal{F}} Face_R P
\end{aligned}$$

Ceci termine la démonstration du théorème. ∎

Les ensembles de la forme $\operatorname{Face}_R(P)$ sont appelés les *facettes* de P (*côtés* en dimension deux). Comme ces ensembles sont eux-mêmes des polyèdres convexes, de dimension

$(d-1)$, on peut parler de leurs facettes, qui sont alors des polyèdres de dimension $(d-2)$, et ainsi de suite. Plus généralement, pour $k = 0, \ldots, d-1$, les *k-faces* de P sont les facettes d'une $k+1$-face de P où, au départ de la récurrence, les $(d-1)$-faces sont les facettes de P. Les k-faces sont des polyèdres convexes de dimension k ; en particulier les 0-faces sont appelées les *sommets* de P et les 1-faces sont appelées les *arêtes* de P. Il est utile d'introduire l'ensemble vide comme étant la (-1)-face de P. Dans la suite le terme de *face*, d'un polyèdre convexe P, désigne l'une quelconque de ses k-faces. Il est clair que les faces d'un polyèdre sont en nombre fini.

Soit $P = \bigcap_{i=1}^{n} R_i$ l'écriture minimale de P et soit H_i l'hyperplan frontière de R_i. D'après le théorème 10.1.1, les $(d-k)$-faces de P sont des intersections de la forme

$$H_{i_1} \cap \cdots \cap H_{i_k} \cap P \tag{1.3}$$

de codimension k (plus précisemment $H_{i_1} \cap \cdots \cap H_{i_j} \cap P$ est de codimension j pour $j = 1, \ldots, k$). Il faut prendre garde qu'une telle écriture n'est en général pas unique. Le théorème suivant précise ce point.

Théorème 10.1.2. *Soient P un polyèdre convexe de dimension d et $\bigcap_{\mathcal{F}} R$ son écriture minimale. Soit F une face de P et soit $\mathcal{F}_1 \subset \mathcal{F}$ l'ensemble des demi-espaces de $\mathcal{F}$ dont les frontières sont les hyperplans supports des $(d-1)$-faces de P qui contiennent F. Posons $F_1 = \bigcap_{\mathcal{F}_1} \mathrm{Fr}(R)$. Alors*

(*i*) *$F = F_1 \cap P$ et F_1 est l'espace affine engendré par F ;*

(*ii*) *$\mathring{F} = F_1 \cap \left(\bigcap_{\mathcal{F} \setminus \mathcal{F}_1} \mathring{R} \right)$, où $\mathring{F}$ désigne l'intérieur relatif de F ;*

(*iii*) *pour tout point x de P, si l'enveloppe convexe de $\{x\} \cup F$ et F ont même dimension, alors x appartient à F.*

Preuve. L'assertion (*i*) est conséquence de l'écriture (1.3) pour toute face de P. Soit R un demi-espace de $\mathcal{F} \setminus \mathcal{F}_1$; R ne contenant pas F_1, l'intérieur relatif de $F_1 \cap R$ coïncide avec la trace sur F_1 de son intérieur :

$$\overset{\circ}{\widehat{F_1 \cap R}} = F_1 \cap \mathring{R}$$

Il vient alors

$$\mathring{F} = \bigcap_{R \in \mathcal{F} \setminus \mathcal{F}_1} \overset{\circ}{\widehat{F_1 \cap R}} = \bigcap_{R \in \mathcal{F} \setminus \mathcal{F}_1} F_1 \cap \mathring{R} = F_1 \cap \left(\bigcap_{R \in \mathcal{F} \setminus \mathcal{F}_1} \mathring{R} \right)$$

De $F = F_1 \cap P$, on déduit que l'appartenance de x à F est équivalente à son appartenance à F_1 ; or les dimensions de F_1 et $\mathcal{E}(F_1 \cup \{x\}) = \mathcal{E}(F \cup \{x\})$ coïncident si et seulement si x appartient à F_1. ∎

Proposition 10.1.3. *Soit P un polyèdre convexe et soient F, G deux faces de P. Alors soit F est une face de G (ou G est une face de F), soit $F \cap G$ est une face de F et de G.*

PROPOSITION 10.1.4. *Soit P un polyèdre convexe et soit R un demi-espace contenant P. Alors $\mathrm{Fr}(R) \cap P$ est une face de P.*

Preuve des propositions. Nous écartons le cas trivial où $F \cap G = \emptyset$. Du théorème 10.1.2 assertion (*ii*) on déduit que

(1) soit $F \subset G$, et dans ce cas $F = F \cap G \subset \mathrm{Fr}(G)$;
(2) soit $G \subset F$, et dans ce cas $G = F \cap G \subset \mathrm{Fr}(F)$;
(3) soit $F \cap G \subset \mathrm{Fr}(F) \cap \mathrm{Fr}(G)$.

Supposons, sans perte de généralité, que $F \cap G \subset \mathrm{Fr}\, F$; pour $x \in F \cap G$ soit H_x la face de F de dimension maximale contenant x dans son intérieur relatif; alors $H_x \subset F \cap G$ (sinon H_x serait sécant à l'un des hyperplans support des $(d-1)$-faces contenant $F \cap G$) et

$$F \cap G = \bigcup_x H_x$$

Le nombre de faces de F étant fini, la réunion ci-dessus est finie. Par suite, l'un des H_x a la même dimension que $F \cap G$ (un convexe de dimension k ne peut être réunion finie de convexes de dimension strictement inférieure) ; mais en vertu de l'assertion (*iii*) du théorème 10.1.2 tout point de $F \cap G$ est un point de H_x donc $F \cap G = H_x$.

La démonstration de la seconde proposition utilise le même argument. Soit $C = \mathrm{Fr}(R) \cap P$ et pour $x \in C$ soit H_x la face de dimension maximale contenant x dans son intérieur relatif; de $P \subset R$ on déduit que $H_x \subset C$ et $C = \bigcup_x H_x$; la démonstration se complète alors comme précédemment. ∎

Deux faces sont dites *incidentes* si l'une est incluse dans l'autre et si leurs dimensions diffèrent d'une unité. Deux $(k+1)$-faces sont dites *adjacentes* si leur intersection est une k-face.

PROPOSITION 10.1.5. *Une $(d-2)$-face est incidente à exactement deux $(d-1)$-faces de P.*

Preuve. Si une $(d-2)$-face est incidente aux $(d-1)$-faces définies par trois hyperplans H_{i_1}, H_{i_2} et H_{i_3} alors ces hyperplans ont pour intersection commune un espace affine de dimension $d-2$; soit o un point de l'intérieur du polyèdre et soient les vecteurs $\vec{u}_j$ pour $j = 1, 2, 3$ tels que

$$R_{i_j} = \{x \in E \mid \vec{ox}.\vec{u}_j \leq 1\}$$

De $H_{i_1} \cap H_{i_2} = H_{i_1} \cap H_{i_3}$ on déduit l'existence de deux réels α, β de somme 1 tels que $\vec{u}_1 = \alpha\vec{u}_2 + \beta\vec{u}_3$; quitte à permuter les hyperplans on peut supposer que ces deux réels sont positifs (en effet si β est négatif alors $\vec{u}_2 = \frac{1}{\alpha}\vec{u}_1 + \frac{-\beta}{\alpha}\vec{u}_3$ avec $1/\alpha$ et $-\beta/\alpha$ positifs), par suite $R_{i_2} \cap R_{i_3} \subset R_{i_1}$ ce qui condredit la minimalité de l'écriture de P. ∎

En particulier un côté d'un polyèdre convexe de dimension deux n'ayant qu'un ou deux sommets incidents, selon qu'il est ou non borné, admet un ou deux côtés adjacents. La frontière d'un convexe étant connexe on obtient la caractérisation complète de la relation d'incidence entre côtés d'un polyèdre convexe en dimension 2 sous la forme suivante.

Proposition 10.1.6. *Les côtés $E_1, \ldots, E_n$ d'un polyèdre convexe de dimension deux forment une chaîne de côtés adjacents c'est-à-dire que, à une permutation près, E_i et E_{i+1} sont adjacents pour $i = 1, \ldots, n-1$ avec E_n et E_1 adjacents si le polyèdre est borné, et E_n et E_1 sont les deux côtés infinis si le polyèdre est non borné.*

La caractérisation de la relation d'incidence entre faces pour les polyèdres de dimension ≥ 3 est un problème nettement plus difficile. Seul le cas de la dimension trois est entièrement résolu.

Nous introduisons maintenant la notion d'ensemble polaire. Soient $o \in E$ et $A \subset E$; le sous-ensemble *polaire* de A (par rapport à o) est

$$A^* = \{y \in E \mid \overrightarrow{oy}.\overrightarrow{ox} \leq 1 \ \text{ pour tout } x \in A\} \tag{1.4}$$

On vérifie facilement que A^* est un fermé convexe (c'est une intersection de demi-espaces fermés), que si $A \subset B$ alors $B^* \subset A^*$ et enfin que $A \subset A^{**}$. L'opération * est en fait une bonne dualité sur l'ensemble des convexes compacts d'intérieur contenant le point o.

Proposition 10.1.7. *Soit C un convexe fermé borné contenant le point o dans son intérieur. Alors*

(*i*) *C^* est un convexe fermé borné dont l'intérieur contient o;*
(*ii*) *$C^{**} = C$.*

Preuve. La boule ouverte de centre o et de rayon r est notée $B(o, r)$. De $B(o, r_1) \subset C \subset B(o, r_2)$ et de $B(o, r)^* = B(o, 1/r)$ on déduit que

$$B(o, 1/r_2) \subset C^* \subset B(o, 1/r_1)$$

Il reste à montrer que $C^{**} \subset C$; soit $a \notin C$ et soit H un hyperplan qui sépare a et C (par exemple l'hyperplan médiateur de a et du point de C le plus proche de a). Le point o n'appartenant pas à H, il existe $h \in E$ tel que

$$H = \{x \in E \mid \overrightarrow{ox}.\overrightarrow{oh} = 1\}$$

De plus $\overrightarrow{oa}.\overrightarrow{oh} > 1$ tandis que, pour tout $x \in C$, on a $\overrightarrow{ox}.\overrightarrow{oh} < 1$; par suite $h \in C^*$ et $a \notin C^{**}$. ∎

Nous pouvons maintenant démontrer l'identité des polyèdres convexes bornés et des enveloppes convexes de parties finies de E.

Théorème 10.1.8. *Un polyèdre convexe borné est l'enveloppe convexe de ses sommets. Réciproquement l'enveloppe convexe d'un ensemble fini de points est un polyèdre convexe borné dont les sommets sont des points de cet ensemble.*

Preuve. Pour démontrer le premier point on raisonne par récurrence sur la dimension du polyèdre P ; si $x \in \mathrm{Fr}(P)$ alors x appartient à l'une des faces de P qui coïncide, par hypothèse de récurrence, avec l'enveloppe convexe de ses sommets ; si $x \in \overset{\circ}{P}$, soit δ

une droite passant par x; δ intersecte P selon un segment $[x_1, x_2]$ où les x_i, étant des points de la frontière de P, appartiennent à l'enveloppe convexe des sommets de P; de $x \in [x_1, x_2]$ on déduit alors l'appartenance de x à l'enveloppe convexe des sommets de P.

Inversement soit $C = \mathcal{E}(\{x_1, \ldots, x_n\})$ l'enveloppe convexe d'une partie finie. Il est clair que C est un convexe fermé borné que nous supposons, sans perte de généralité, de dimension d. D'après la proposition 10.1.7, C^* est un convexe fermé borné d'intérieur non vide; or par définition C^* est l'intersection des n demi-espaces $\{x \in E \mid \overrightarrow{ox}.\overrightarrow{ox}_i \leq 1\}$; c'est donc un polyèdre convexe borné de E. D'après la première partie du théorème, C^* est l'enveloppe convexe d'un ensemble fini de points d'où, en appliquant le raisonnement précédent à C^* et en utilisant encore la proposition 10.1.7, $C^{**} = C$ est un polyèdre convexe borné. ■

En particulier un simplexe de dimension d est un polyèdre convexe. On vérifie sans peine que le nombre de ses k-faces est $\binom{d+1}{k+1}$.

Comment calculer effectivement l'enveloppe convexe d'un ensemble fini S de n points? Et quelle est la complexité en nombre d'opérations élémentaires d'un tel calcul? Supposons, pour simplifier la discussion, que les points sont *en position générale* c'est-à-dire que $d + 1$ points quelconques de S sont affinement indépendants. Dans ce cas le simplexe engendré par d points quelconques $s_1, \ldots, s_d$ de S est une $(d-1)$-face de $\mathcal{E}(S)$ si et seulement si les autres points de S se situent tous dans le même demi-espace dont la frontière est l'hyperplan engendré par ce simplexe :

$$\text{dét}(\overrightarrow{ss}_1, \ldots, \overrightarrow{ss}_d) \text{ est de signe constant.} \tag{1.5}$$

Une première solution au calcul de l'enveloppe convexe est de tester les $\binom{n}{d}$ simplexes possibles. Une mesure évidente de la complexité d'un tel calcul est le nombre de déterminants calculés : chaque test nécessite le calcul d'au plus n déterminants; le nombre total de calcul est alors $\leq \binom{n}{d}n = O(n^{d+1})$, pour d constant. Cette méthode est-elle efficace? La réponse est non, car on peut montrer que le nombre de $(d-1)$-faces de l'enveloppe convexe de n points est, sous l'hypothèse d constant, dans $O(n^{\lfloor d/2 \rfloor})$.

Une seconde solution est de calculer les faces par ordre d'adjacence. Pour cela nous avons besoin de définir les angles dièdres d'un polyèdre.

Par définition *l'angle* (non orienté) de deux demi-droites vectorielles d, d' est le réel dans $[0, \pi]$, noté $\widehat{d, d'}$, défini comme la valeur commune de

$$\text{Arccos}\left(\frac{\vec{u}}{\|\vec{u}\|} \cdot \frac{\vec{v}}{\|\vec{v}\|}\right)$$

où $\vec{u}$ et $\vec{v}$ sont des vecteurs non nuls quelconques de d et d'. On note aussi $\widehat{\vec{u}, \vec{v}} = \widehat{d, d'}$.

Etant données trois demi-droites d, d' et d'', on dit que d'' est *entre* d et d' si $d = d' = d''$ ou si $d = -d'$, ou si $d \neq \pm d'$ et d'' se situe dans l'intersection des demi-plans de frontière respectives $d \cup -d$ et $d' \cup -d'$ contenant respectivement d' et d. On a alors la relation d'additivité :

$$\widehat{d, d'} = \widehat{d, d''} + \widehat{d'', d'} \tag{1.6}$$

Soient F et F' deux $(d-1)$-faces adjacentes d'un polyèdre P de dimension d et soit A leur intersection. Les hyperplans de F et F' déterminent deux vecteurs unitaires $\vec{u}, \vec{v}$ bien définis par la condition que $\vec{u}$ soit orthogonal à F et du même côté de F que F', et de même pour $\vec{v}$. Par définition l'*angle dièdre* de P en A est l'angle $\widehat{\vec{u}, \vec{v}}$.

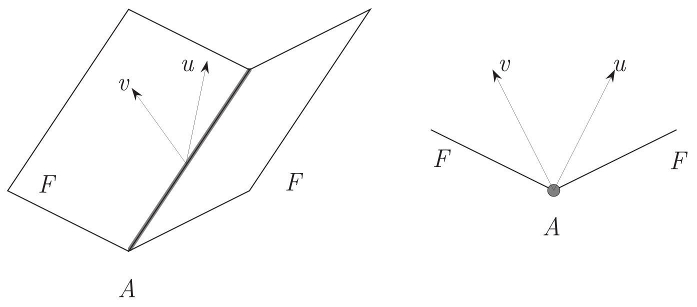


Figure 10.1.3: Angle dièdre de deux faces.

PROPOSITION 10.1.9. *Soient F et F' deux $(d-1)$-faces adjacentes d'un polyèdre convexe P de dimension d et soit A leur intersection. Soit x un point de $P \setminus F$ et $F_x = \mathcal{E}(F \cup \{x\})$ le cône de sommet x et de base F. Alors l'angle dièdre de P en A est inférieur à l'angle dièdre de F_x en A.*

Preuve. Soit F'' la face de F_x adjacente à F en A et soit $\vec{w}$ le vecteur unitaire orthogonal à F'' et du même côté de F'' que F. Le vecteur $\vec{v}$ se situe entre $\vec{u}$ et $\vec{w}$ d'où le résultat. ∎

D'après cette proposition le calcul d'une face adjacente à une face donnée F en la $(d-2)$-face A peut se faire en cherchant le point $x \in S$ qui minimise l'angle dièdre du cône F_x en A. La complexité en nombre d'angles dièdres calculés est alors dans $O(nh)$ où h est le nombre de $(d-1)$-faces de $\mathcal{E}(S)$. Le nombre de $(d-1)$-faces étant dans $O(n^{\lfloor d/2 \rfloor})$, la complexité du calcul est dans $O(n^{\lfloor d/2 \rfloor + 1})$, sous réserve d'avoir déterminé une première face pour amorcer le calcul. Comment calculer une première face ? Rapportons l'espace E à un repère $(O, \vec{\imath}_1, \ldots, \vec{\imath}_d)$ et appelons H l'hyperplan $(O, \vec{\imath}_1, \ldots, \vec{\imath}_{d-1})$. En vertu de la proposition 10.1.4, le point s_1 dont la coordonnée en $\vec{\imath}_d$ est minimale, est un sommet de $\mathcal{E}(S)$; puis l'arête $[s_1, s_2]$ qui minimise l'angle du vecteur $\vec{s_1 s}$ avec sa projection sur l'hyperplan H est une arête de $\mathcal{E}(S)$; quitte à effectuer une rotation, nous pouvons maintenant supposer que l'arête $[s_1, s_2]$ se situe dans l'hyperplan H ; le triangle $s_1 s_2 s_3$ qui minimise l'angle du plan $s_1 s_2 s$ avec sa projection sur l'hyperplan H est alors une 2-face de $\mathcal{E}(S)$; etc. Le nombre d'angles calculés dans cette procédure est de l'ordre de dn. La complexité du calcul reste donc dans $O(n^{\lfloor d/2 \rfloor + 1})$. Par exemple en dimension 2 et 3 cet algorithme a une complexité en $O(n^2)$ alors que la première solution proposée conduit à des complexités en $O(n^3)$ et $O(n^4)$. Nous sommes cependant encore loin d'avoir obtenu les algorithmes les plus efficaces. En fait, en dimension deux et trois, il existe des algorithmes de complexité $O(n \log_2 n)$ pour calculer l'enveloppe convexe de n points. Tel est le cas, en dimension deux, de l'algorithme de Graham que nous décrivons maintenant.

L'enveloppe convexe de $S = \{s_1, \ldots, s_n\}$, ensemble fini de n points, est un polygone convexe qu'il est utile d'écrire sous la forme

$$\mathcal{E}(S) = [u_1, \ldots, u_l] \tag{1.7}$$

où les u_i sont les sommets de l'enveloppe et où les $[u_i, u_{i+1}]$ sont les côtés de l'enveloppe ; sans perte de généralité nous supposons que le polygone est parcouru dans le sens positif c'est-à-dire que pour tout i (les indices sont calculés modulo l)

$$\sigma(u_i, u_{i+1}, u_{i+2}) = +1$$

où $\sigma(p, q, r)$ est la fonction à valeur dans $\{-1, 0, +1\}$ définie comme le signe du déterminant des vecteurs $\overrightarrow{pq}, \overrightarrow{pr}$ où $p, q, r \in E$:

$$\sigma(p, q, r) = signe\,(\text{dét}(\overrightarrow{pq}, \overrightarrow{pr})) \tag{1.8}$$

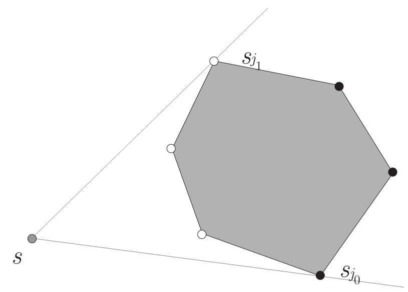


Figure 10.1.4: Les deux tangentes issues de s.

La proposition suivante fait le lien entre l'enveloppe convexe de S et celle de l'ensemble obtenu en adjoignant à S un point s.

Proposition 10.1.10. *Soit $[u_1, \ldots, u_l]$ l'enveloppe convexe d'un ensemble fini S et soit s un point n'appartenant pas à $\mathcal{E}(S)$. Soit $\omega(j)$ le signe du déterminant des vecteurs $\overrightarrow{su}_j$ et $\overrightarrow{su}_{j+1}$:*

$$\omega(j) = \sigma(s, u_j, u_{j+1})$$

Alors il existe j_0 et j_1 tels que

(i) $\omega(j_0) = \omega(j_0 + 1) = \cdots = \omega(j_1 - 1) = +1$;
(ii) $\omega(j_1) = \omega(j_1 + 1) = \cdots = \omega(j_0 - 1) = -1$.

De plus on a

$$\mathcal{E}(S \cup \{s\}) = [u_{j_0}, \ldots, u_{j_1}, s]$$

Preuve. Les points u_{j_0} et u_{j_1} sont les points d'appui des deux tangentes au convexe $\mathcal{E}(S)$ issues du point s (voir la figure 10.1.4) ∎

L'idée générale de l'algorithme de Graham est de calculer successivement les enveloppes convexes des ensembles $S_1, S_2, S_3, \ldots, S_n$ où $S_1 = \{s_1\}$ et où S_{i+1} est obtenu à partir de S_i en lui ajoutant le point s_{i+1}. Pour déterminer de manière efficace les valeurs des

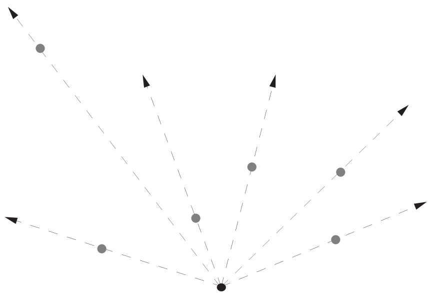


Figure 10.1.5: La première phase.

indices j_0 et j_1 de la proposition précédente, dans une première phase de l'algorithme, nous trions les points s_i de telle sorte que le point s_1 soit le point d'abscisse minimale et que les autres points soient triés angulairement autour du point s_1 :

$$\sigma(s_1, s_i, s_j) = +1 \tag{1.9}$$

pour tout $2 \leq i < j \leq n$. La deuxième phase de l'algorithme consiste à calculer $\mathcal{E}(S_{i+1})$ à partir de $\mathcal{E}(S_i)$ pour i variant de 3 à n.

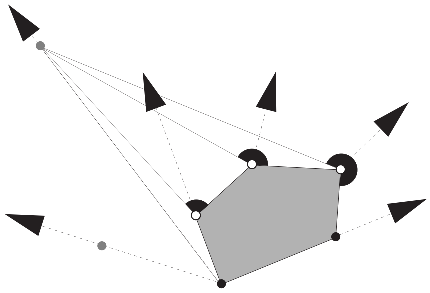


Figure 10.1.6: La deuxième phase.

Les observations suivantes permettent d'effectuer l'ensemble de ce calcul avec une complexité en $O(n)$. Posons

$$\mathcal{E}(S_i) = [u_1, \ldots, u_l]$$

où u_1 est le point d'abscisse minimale de S_i. Dans ce cas il est clair que :
(1) $\mathcal{E}(S_3) = [s_1, s_2, s_3]$;
(2) $u_1 = s_1$, $u_2 = s_2$ et $u_l = s_i$.
D'autre part de $\sigma(s_1, s_2, s_{i+1}) = +1$ et $\sigma(s_i, s_1, s_{i+1}) = -\sigma(s_1, s_i, s_{i+1}) = -1$ on déduit, en vertu de la proposition précédente, que

$$\mathcal{E}(S_{i+1}) = [u_1, \ldots, u_k, s_{i+1}] \tag{1.10}$$

où k est défini comme le plus grand indice j tel que $\sigma(s_{i+1}, u_{j-1}, u_j) = +1$.

Enfin, et c'est là le point clé de l'analyse, le calcul de k peut se faire en évaluant, successivement, les $\sigma(s_{i+1}, u_{j-1}, u_j)$ pour $j = l, l-1, \ldots, k$, soit en effectuant $l-k+1 = \tau(S_i) - \tau(S_{i+1}) + 2$ calculs de la fonction σ, où $\tau(X)$ est le nombre de sommets de l'enveloppe convexe de X. La complexité du calcul de la seconde phase est donc

$$\sum_{i=3}^{n-1} (\tau(S_i) - \tau(S_{i+1}) + 2) = 2n - 3 - \tau(S_n) \leq 2n$$

L'algorithme de Graham montre que, modulo un tri, la complexité du calcul de l'enveloppe convexe de n points dans le plan est linéaire c'est-à-dire en $O(n)$.

10.2 Quelques primitives géométriques

Une application géométrique ne se programme pas tout à fait comme une application numérique, et ceci pour deux raisons. La première réside dans le fait que le raisonnement géométrique se place dans un espace affine réel, et donc que les coordonnées des points sont des nombres réels, alors que l'image, sur l'écran, est formée de pixels, qui sont représentés par des entiers. La deuxième raison provient de la spécificité des opérations de lecture et écriture, qui sont ici des opérations de saisie de points et de tracé de points ou de lignes; ces opérations dépendent considérablement du matériel (compatible PC ou Macintosh).

Il est donc naturel d'organiser un programme géométrique en trois couches. La couche la plus basse, dépendante du matériel employé, réalise les opérations de saisie de points, de suivi d'affichage et de tracé de lignes. Dans les réalisations exposées plus bas, nous les avons appelées `Graphik`. La deuxième couche, indépendante du matériel, se charge de la traduction des objets géométriques réels en objets sur l'écran, typiquement la traduction entre points et pixels. Le nom de cette unité est `Geometrie`.

La couche la plus externe est le programme géométrique à proprement parler qui ne connaît plus que des points, des segments, droites et autres objets géométriques usuels, et se décharge sur le module `Geometrie` de toutes les opérations de communication avec l'écran.

Nos réalisations sont très rudimentaires et ne sont en aucun cas suffisantes pour une programmation plus poussée. Pour les compatibles PC, on pourra utiliser avec profit d'autres interfaces, comme Modulog.

L'unité `Geometrie` met à la disposition des programmes géométriques les points d'un plan euclidien affine, muni d'un repère orthonormé direct, et quelques opérations sur ces objets. On déclare un point comme un couple de nombres réels :

```
TYPE
   Point = ARRAY[0..1] OF real;
```

Les procédures nécessaires sont :

```
PROCEDURE InitGeometrie;
FUNCTION Abscisse (q: Point): real;
FUNCTION Ordonnee (q: Point): real;
PROCEDURE FairePoint (x, y: real; VAR q: Point);
PROCEDURE SaisirPoint (VAR q: Point);
PROCEDURE TracerSegment (p, q: Point);
PROCEDURE TracerDemiDroite (p, d: Point);
PROCEDURE MarquerPointCarre (p: Point);
PROCEDURE MarquerPointRond (p: Point);
PROCEDURE NumeroterPoint (p: Point; n: integer);
```

Les fonctions d'accès et de construction s'écrivent bien sûr :

```
FUNCTION Abscisse (q: Point): real;
   BEGIN
      Abscisse := q[0]
   END; { de "Abscisse" }
FUNCTION Ordonnee (q: Point): real;
   BEGIN
      Ordonnee := q[1]
   END; { de "Ordonnee" }
PROCEDURE FairePoint (x, y: real; VAR q: Point);
   BEGIN
      q[0] := x;
      q[1] := y
   END; { de "FairePoint" }
```

La lecture d'un point est remplacée par une saisie, et l'écriture par un dessin. Les procédures que voici provoquent l'affichage d'un point :

`PROCEDURE MarquerPointCarre (p: Point);`	*Marque le point d'un* □.
`PROCEDURE MarquerPointRond (p: Point);`	*Marque le point d'un* •.
`PROCEDURE NumeroterPoint (p: Point; n: integer);`	*Ajoute n à la marque du point.*

On trace des segments de droites ou des demi-droites qui sont données par une origine et une direction. On peut voir, dans les sections suivantes, le genre de dessins que l'on obtient. Ces procédures seront employées pour la saisie de points ou de polygones, et pour l'affichage des résultats.

La réalisation des procédures de saisie et d'affichage fait appel à des primitives qui dépendent du matériel utilisé (et disponibles à travers la bibliothèque appelée `Graphik`). Nous supposons que les objets et les primitives suivantes sont disponibles :

```
TYPE
   Pixel = ARRAY[0..1] OF integer;
VAR
   HauteurEcran: integer;   La hauteur disponible de l'écran, mesurée en pixels.
   LargeurEcran: integer;   La largeur disponible de l'écran, mesurée en pixels.
PROCEDURE InitGraphik;
PROCEDURE TracerCarre (a: Pixel; r: integer);
   Trace un petit carré de côté 2r centré sur le pixel a.
PROCEDURE TracerDisque (a: Pixel; r: integer);
   Trace un disque plein de rayon r centré sur le pixel a.
PROCEDURE LireCurseur (VAR curseur: Pixel);
   Donne dans la variable les coordonnées du point saisi, en pixels.
PROCEDURE RelierPixels (a, b: Pixel);
   Trace un trait qui relie les pixels a et b.
PROCEDURE EcrireGraphique (a: Pixel; n: integer);
   Ajoute, au dessin, l'entier n au pixel a.
```

Avec ces primitives, les procédures de la bibliothèque `Geometrie` s'écrivent facilement. Commençons par :

```
PROCEDURE InitGeometrie;
   BEGIN
      InitGraphik
   END; { de "InitGeometrie" }
```

Puis, les procédures de conversion entre points et pixels :

```
PROCEDURE PointEnPixel (p: Point; VAR a: Pixel);
   BEGIN
      a[0] := round(Abscisse(p));
      a[1] := HauteurEcran - round(Ordonnee(p))
   END; { de "PointEnPixel" }
PROCEDURE PixelEnPoint (c: Pixel; VAR q: Point);
   VAR
      x, y: real;
   BEGIN
      x := c[0];
      y := HauteurEcran - c[1];
      FairePoint(x, y, q)
   END; { de "PixelEnPoint" }
```

Les «lectures» se font par :

```
PROCEDURE SaisirPoint (VAR q: Point);
   VAR
      curseur: Pixel;
   BEGIN
      LireCurseur(curseur);
      PixelEnPoint(curseur, q)
```

```
   END; { de "SaisirPoint" }
```

On marque un point par l'une des deux procédures que voici :

```
PROCEDURE MarquerPointCarre (p: Point);
   VAR
      a: Pixel;
   BEGIN
      PointEnPixel(p, a);
      TracerCarre(a, 4);         Le choix du nombre 4 est bien sûr affaire de goût.
   END; { de "MarquerPointCarre" }
PROCEDURE MarquerPointRond (p: Point);
   VAR
      a: Pixel;
   BEGIN
      PointEnPixel(p, a);
      TracerDisque(a, 4);        Idem.
   END; { de "MarquerPointRond" }
PROCEDURE NumeroterPoint (p: Point; n: integer);
   VAR
      a: Pixel;
   BEGIN
      PointEnPixel(p, a);
      a[0] := a[0] + 5;          Petit décalage, pour que n n'écrase pas le point.
      EcrireGraphique(a, n)
   END; { de "NumeroterPoint" }
```

Venons-en au tracé de droites. Pour les segments, on utilise :

```
PROCEDURE TracerSegment (p, q: Point);
   VAR
      a, b: Pixel;
   BEGIN
      PointEnPixel(p, a);
      PointEnPixel(q, b);
      RelierPixels(a, b);
   END; { de "TracerSegment" }
```

Pour tracer une demi-droite d'origine et de direction données, on détermine le point d'intersection de la demi-droite avec le rectangle délimitant la partie utilisée de l'écran, puis on trace le segment obtenu. Ainsi :

```
PROCEDURE TracerDemiDroite (p, d: Point);
   p est l'origine, et d est la direction.
   VAR
      q: Point;
   BEGIN
      Clip(p, d, q);          q est l'intersection de la demi-droite avec le «cadre».
      TracerSegment(p, q)
```

```
    END; { de "TracerDemiDroite" }
```

La procédure `Clip` fait le découpage. Voici une écriture :

```
PROCEDURE Clip (p, d: Point; VAR q: Point);
```
Calcule le point d'intersection q de la demi-droite Δ d'origine p et de direction d avec le rectangle délimité par les verticales d'équations $x = 0$ et $x = HauteurEcran$, et les horizontales d'ordonnées $y = 0$ et $y = LargeurEcran$.
```
     VAR
       dx, dy, px, py, m: real;
    BEGIN
       dx := Abscisse(d);
       dy := Ordonnee(d);
       px := Abscisse(p);
       py := Ordonnee(p);
       IF EstNul(dx) THEN     { Δ est verticale. }
          FairePoint((1 + signe(dy)) * HauteurEcran / 2, px, q)
       ELSE BEGIN             { m est l'ordonnée de l'intersection de Δ avec une verticale. }
          m := py + ((1 + signe(dx)) * LargeurEcran / 2 - px) / dx * dy;
          IF m > HauteurEcran THEN
             FairePoint(px + (HauteurEcran - py) / dy * dx, HauteurEcran, q)
          ELSE IF m < 0 THEN
             FairePoint(px - py / dy * dx, 0, q)
          ELSE
             FairePoint((1 + signe(dx)) * LargeurEcran / 2, m, q)
       END
    END; { de "Clip" }
```

Ceci définit la bibliothèque `Geometrie`. Bien entendu, il reste à réaliser les primitives graphiques. Commençons par le Macintosh :

```
PROCEDURE RelierPixels (a, b: Pixel);
```
Pour tracer un trait du pixel a au pixel b.
```
    BEGIN
       MoveTo(a[0], a[1]);
       LineTo(b[0], b[1])
    END; { de "RelierPixels" }
PROCEDURE LireCurseur (VAR curseur: Pixel);
    BEGIN
       REPEAT
       UNTIL button;
       GetMouse(curseur[0], curseur[1]);
       WHILE button DO;
    END; { de "LireCurseur" }
PROCEDURE TracerCarre (a: Pixel; r: integer);
    VAR
       rectangle: rect;       { Pour les coordonnées du carré. }
    BEGIN
       SetRect(rectangle, a[0] - r, a[1] - r, a[0] + r, a[1] + r);
```

```
      FrameRect(rectangle);      Trace les côtés du carré.
   END; { de "TracerCarre" }
PROCEDURE TracerDisque (a: Pixel; r: integer);
   VAR
      rectangle: rect;           Pour le carré circonscrit au cercle.
   BEGIN
      SetRect(rectangle, a[0] - r, a[1] - r, a[0] + r, a[1] + r);
      PaintOval(rectangle);      Trace le disque.
   END; { de "TracerDisque" }
PROCEDURE EcrireGraphique (a: Pixel; n: integer);
   BEGIN
      moveto(a[0], a[1]);
      WriteDraw(n : 1);          Ecriture dans la fenêtre graphique.
   END; { de "EcrireGraphique" }
PROCEDURE InitGraphik;
   VAR
      cadre: rect;
   BEGIN
      showdrawing;               Montre la fenêtre graphique.
      GetDrawingRect(cadre);     Récupère les coordonnées de cette fenêtre.
      HauteurEcran := cadre.bottom - cadre.top;
      LargeurEcran := cadre.right - cadre.left
   END; { de "InitGraphik" }
```

Pour un compatible PC, certaines des procédures sont un peu plus longues à écrire, si l'on ne dispose pas par ailleurs de fonctions sophistiquées. Nous faisons le choix de nous placer en mode graphique et d'y rester. Le «curseur» est une petite croix (appelée «réticule») qui est activée par les flèches. Le point vu sera saisi lorsque l'on enfonce la touche retour. Voici les procédures, avec quelques commentaires :

```
CONST
   hRet = 4;  vRet = 4;                   Hauteur et largeur du réticule.
VAR
   Reticule: pointer;                     Un pointeur sans type.
PROCEDURE InitReticule;
   BEGIN
      MoveTo(0, 0); LineTo(2 * hRet, 2 * vRet);               Tracé.
      MoveTo(2 * hRet, 0); LineTo(0, 2 * vRet);
      GetMem(Reticule, ImageSize(0, 0, 2 * hRet, 2 * vRet));  Réserve place.
      GetImage(0, 0, 2 * hRet, 2 * vRet, Reticule^);          Copie.
      ClearViewPort                       Et on efface tout.
   END; { de "InitReticule" }
```

Le réticule doit simuler une souris. Pour cela, il se déplace en fonction des touches du clavier enfoncées. Mais pour déplacer le réticule, il faut l'effacer et le redessiner. La procédure suivante réalise l'opération :

```
PROCEDURE BasculerReticule;
   Efface le réticule, s'il est dessiné, et le dessine, s'il est absent.
   BEGIN
      PutImage(GetX - hRet, GetY - vRet, Reticule^, XOrPut)
   END; { de "BasculerReticule" }
```

Le réticule suit les flèches du clavier; plus précisément, si une des quatre flèches est enfoncée, le réticule se déplace, ainsi que le «pixel courant»; si la touche retour est enfoncée, la variable `bouton` ci-dessous est mise à vrai; dans les autres cas, il ne se passe rien :

```
PROCEDURE SuivreReticule (VAR bouton: boolean);
   Quand on arrive, le réticule est affiché; il le restera en sortant.
   CONST
      FlecheGauche = 'K'; FlecheDroite = 'M';       Les numéros des flèches.
      FlecheHaut = 'H'; FlecheBas = 'P';
      pas = 5;
   BEGIN
      CASE Readkey OF
         #13: bouton := true;                        Touche «retour-chariot».
         #0:
            BEGIN
            BasculerReticule;
            CASE Readkey OF
               FlecheGauche:
                  IF GetX >= pas THEN MoveRel(-pas, 0);
               FlecheDroite:
                  IF GetX <= LargeurEcran - pas THEN MoveRel(pas, 0);
               FlecheHaut:
                  IF GetY >= pas THEN MoveRel(0, -pas);
               FlecheBas:
                  BEGIN
                     IF GetY <= HauteurEcran - pas THEN MoveRel(0, pas)
                  END
               ELSE
            END;
            BasculerReticule
         END
         ELSE
      END { du CASE }
   END; { de "SuivreReticule" }
```

Voici la procédure de saisie d'un pixel; elle suit le réticule jusqu'à ce que la touche retour soit enfoncée :

```
PROCEDURE LireCurseur (a: Pixel);
   Quand on arrive, on est en mode graphique et le réticule n'est pas affiché; il ne le sera
   pas en sortant.
   VAR
```

```
      bouton: boolean;
   BEGIN
      BasculerReticule;
      bouton := false;
      REPEAT
         SuivreReticule(bouton)
      UNTIL bouton;
      a[0] := GetX; a[1] := GetY
      BasculerReticule
   END; { de "LireCurseur" }
```

Les autres procédures se réalisent facilement :

```
PROCEDURE TracerDisque (a: pixel; r: integer);
   BEGIN
      PieSlice(a[0], a[1], 0, 350, r)
   END; { de "TracerDisque" }
PROCEDURE TracerCarre (a: pixel; r: integer);
   BEGIN
      Rectangle(a[0] - r, a[1] + r, a[0] + r, a[1] - r)
   END; { de "TracerCarre" }
PROCEDURE RelierPixels (a, b: Pixel);
   BEGIN
      MoveTo(a[0], a[1]); LineTo(b[0], b[1])
   END; { de "RelierPixels" }
PROCEDURE EcrireGraphique (a: Pixel; n: integer);
   VAR
      c: String[6];
   BEGIN
      Str(n, c);
      OutTextXY(a[0], a[1] - TextHeight(c), c)
   END; { de "EcrireGraphique" }
PROCEDURE InitGraphik;
   VAR
      Pilote, ModeGraphique: integer;
   BEGIN
      Pilote := Detect;                        Incantation usuelle.
      InitGraph(Pilote, ModeGraphique, '');
      IF GraphResult <> GrOk THEN BEGIN        Un problème ?
         writeln(GraphErrorMsg(GraphResult)); halt
      END;
      LargeurEcran := GetMaxX; HauteurEcran := GetMaxY;
      InitReticule;                            Au milieu de l'écran.
      MoveTo(LargeurEcran DIV 2, HauteurEcran DIV 2)
   END; { de "InitGraphik" }
```

Le choix de rester en mode graphique demande bien entendu de revoir le style des menus, en réservant par exemple une ligne de l'écran au texte.

10.3 Triangulation de Delaunay

10.3.1 Enoncé : triangulation de Delaunay

Soit S un ensemble fini de points, appelés *sites*, d'un plan affine euclidien E muni d'un repère orthonormé $(O, \vec{i}, \vec{j})$. On suppose les points de S en position générale, c'est-à-dire que trois quelconques d'entre eux ne sont pas alignés et qu'aucun cercle ne passe par quatre sites quelconques de S.

On note pq la droite passant par les points p et q, et $[p, q]$ le segment de droite joignant p et q. Les sites de S sont donnés par leurs coordonnées.

On appelle *enveloppe convexe* de S, et on note $\mathcal{E}(S)$, la plus petite région convexe, au sens de l'inclusion, contenant S. On admettra que $\mathcal{E}(S)$ est un polygone (sur l'exemple de la figure, la frontière de l'enveloppe convexe est en traits épais).

1.– Ecrire une procédure qui détermine et trace les arêtes du polygone $\mathcal{E}(S)$ (on pourra utiliser le fait que pour $s, t \in S$, le segment $[s, t]$ est une arête de $\mathcal{E}(S)$ si et seulement si tous les autres sites se trouvent d'un même côté de la droite st).

Exemple numérique : $S = \{s_1, \ldots, s_7\}$, $s_1(0,0)$, $s_2(0,6)$, $s_3(6,6)$, $s_4(10,12)$, $s_5(8,3)$, $s_6(10,5)$, $s_7(16,0)$.

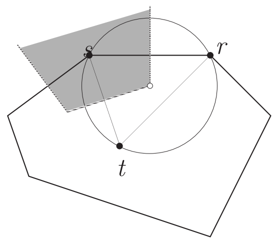


Figure 10.3.1: Un exemple.

Un cercle $\mathcal{C}$ est un *cercle de Delaunay* (de S) si son intérieur ne contient aucun point de S. Un segment $[s, t]$, où s, t sont deux sites distincts, est une *arête de Delaunay* s'il existe un cercle de Delaunay passant par s et t. Un triangle rst (où r, s, et t sont trois éléments distincts de S) est un *triangle de Delaunay* si le cercle circonscrit au triangle est un cercle de Delaunay. Sur l'exemple de la figure, rst est un triangle de Delaunay.

2.– Ecrire une procédure qui détermine et trace les arêtes de tous les triangles de Delaunay de S.

3.– a) Démontrer que tout site de S est extrémité d'au moins une arête de Delaunay. Démontrer que toute arête de Delaunay est un côté d'un triangle de Delaunay.

b) Démontrer que si une arête de Delaunay est une arête de $\mathcal{E}(S)$, elle n'appartient qu'à un seul triangle de Delaunay ; en revanche, si l'arête n'est pas une arête de $\mathcal{E}(S)$, elle appartient à deux triangles de Delaunay.

4.– Démontrer que deux arêtes distinctes de Delaunay de S sont disjointes, sauf éventuellement aux extrémités. Démontrer que les intérieurs de deux triangles de Delaunay distincts sont disjoints.

5.– Démontrer que la réunion des triangles de Delaunay de S est égale à $\mathcal{E}(S)$.

On note $d(p,q)$ la distance (euclidienne) entre deux points p et q, et on pose $d(p,S) = \min_{q \in S} d(p,q)$. Pour tout $s \in S$, on pose

$$V(s) = \{q \in E \mid d(s,q) = d(s,S)\}$$

Les régions $V(s)$ s'appellent les *régions de Voronoï* de S. Sur l'exemple ci-dessus, la région hachurée représente la région de Voronoï de s.

6.– Démontrer que pour tout $s \in S$, la région $V(s)$ est un convexe dont la frontière est une ligne polygonale.

a) Démontrer que les sommets de cette ligne polygonale sont les centres des cercles circonscrits aux triangles de Delaunay dont s est un sommet.

b) Démontrer que les arêtes de cette ligne polygonale sont portées par les médiatrices des arêtes de Delaunay de sommets s.

c) Démontrer que deux sommets de cette ligne polygonale sont les extrémités de l'une de ces arêtes si et seulement si les triangles de Delaunay associés sont adjacents.

7.– Ecrire une procédure qui trace la frontière de $V(s)$, pour tout $s \in S$.

8.– Soient f le nombre de triangles de Delaunay et a le nombre d'arêtes de Delaunay de S, et soit l le nombre de sommets de $\mathcal{E}(S)$. Trouver une relation linéaire entre f, a et l.

10.3.2 Solution : triangulation de Delaunay

Soit S un ensemble fini de points, appelés *sites*, d'un plan affine euclidien E.

Une *triangulation* de S est un ensemble fini de triangles tels que :

(1) l'ensemble des sommets des triangles coïncide avec S ;

(2) les intérieurs des triangles sont disjoints deux à deux ;

(3) la réunion des triangles coïncide avec l'enveloppe convexe de S.

Parmi toutes les triangulations possibles d'un ensemble de points, la *triangulation de Delaunay* (que nous définirons dans un instant) est la plus étudiée et la plus utilisée car elle possède de nombreuses propriétés de maximalité, utiles dans les applications pratiques, et plus prosaïquement il existe des algorithmes performants pour la calculer.

Un cercle $\mathcal{C}$ est un *cercle de Delaunay* (de S) si son intérieur (i.e. le disque ouvert dont $\mathcal{C}$ est la frontière) ne contient aucun point de S. Un segment $[s,t]$, où s,t sont deux sites distincts, est une *arête de Delaunay* s'il existe un cercle de Delaunay passant par s et t. Un triangle rst (où r,s,t sont trois sites distincts de S) est un *triangle de Delaunay* si son cercle circonscrit est un cercle de Delaunay.

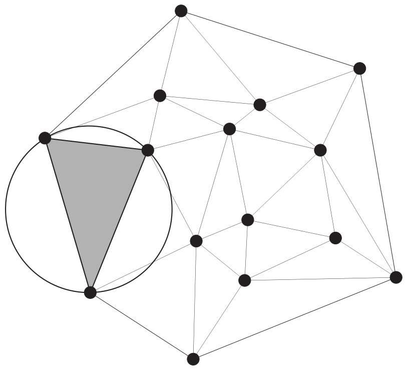


Figure 10.3.2: Une triangulation de Delaunay.

Pour simplifier la discussion nous ferons l'hypothèse restrictive que les sites sont *en position générale* c'est-à-dire que trois d'entre eux ne sont jamais alignés et que quatre d'entre eux ne sont jamais cocycliques. Sous cette hypothèse nous avons le résultat suivant.

THÉORÈME 10.3.1. *L'ensemble des triangles de Delaunay de S forment une triangulation de S, appelée la* triangulation de Delaunay *de S.*

Avant de donner la preuve de ce théorème nous rappelons la caractérisation des points d'un cercle connue sous le nom de la proposition de l'arc capable. Soit $\mathcal{C}$ un cercle de centre ω passant par les points p et q ; soit H_1 le demi-plan ouvert, de frontière la droite pq, contenant (éventuellement dans son adhérence si pq est un diamètre de $\mathcal{C}$) le centre ω du cercle $\mathcal{C}$ et soit H_2 l'autre demi-plan ouvert de frontière la droite pq. Etant donnés trois points distincts p, q, r l'angle des demi-droites définies par les vecteurs $\overrightarrow{pq}$ et $\overrightarrow{pr}$ est noté $\widehat{qpr}$.

PROPOSITION 10.3.2. *Soit $2\alpha = \widehat{p\omega q}$. Le cercle $\mathcal{C}$ est caractérisé par*

$$\mathcal{C} \cap H_1 = \{m \in H_1 \mid \widehat{pmq} = \alpha\} \tag{3.1}$$

$$\mathcal{C} \cap H_2 = \{m \in H_2 \mid \widehat{pmq} = \pi - \alpha\} \tag{3.2}$$

PROPOSITION 10.3.3. *Le disque $\mathcal{D}$ de frontière $\mathcal{C}$ est caractérisé par*

$$\mathcal{D} \cap H_1 = \{m \in H_1 \mid \widehat{pmq} > \alpha\} \tag{3.3}$$

$$\mathcal{D} \cap H_2 = \{m \in H_2 \mid \widehat{pmq} > \pi - \alpha\} \tag{3.4}$$

Preuve du théorème 10.3.1. Nous commençons par démontrer que tout point de S est l'extrémité d'une arête de Delaunay et que toute arête de l'enveloppe convexe de S est un côté d'un unique triangle de Delaunay.

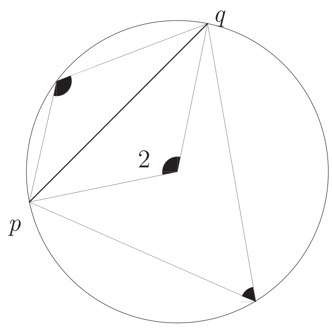


Figure 10.3.3: La caractérisation de l'arc capable.

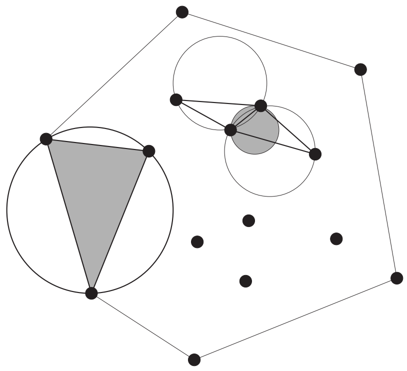


Figure 10.3.4: Preuve de l'existence des triangles de Delaunay.

Soit $p \in S$ et soit q le point de S le plus proche de p. Alors $[p, q]$ est une arête de Delaunay ; en effet le disque de diamètre $[p, q]$ est inclus dans le disque de centre p et de rayon $[p, q]$ qui, par définition de q, ne contient aucun autre point de S.

Soit $[p, q]$ une arête de $\mathcal{E}(S)$; tous les points de S étant situés d'un même côté de la droite pq et les points S étant en position générale, il existe un unique point r de S qui réalise le maximum de

$$\{\widehat{prq} \mid r \in S \setminus \{p, q\}\} \tag{3.5}$$

D'après (3.3) il est clair que pqr est un triangle de Delaunay et que c'est le seul triangle de Delaunay d'arête $[p, q]$.

Nous montrons maintenant que toute arête de Delaunay, autre qu'un côté de l'enveloppe convexe de S, est le côté de deux triangles de Delaunay. Soit $[p, q]$ une arête de Delaunay, $\mathcal{C}$ un cercle de Delaunay passant par p et q, $\mathcal{D}$ son intérieur et H_1, H_2 les deux demi-plans ouverts dont la droite pq est la frontière. Les points m de $\mathcal{D} \cap H_1$ sont caractérisés par

$$\widehat{pmq} > c$$

pour une certaine constante $c \in [0, \pi]$. On a $\neq \emptyset$ car $[p, q]$ n'est pas une arête de l'enveloppe convexe de S. Soit r l'unique point de $S \cap H_1$ qui réalise le maximum de

$$\{\widehat{prq} \mid r \in S \cap H_1\}$$

Pour tout point $s \in S \setminus \{p, q\}$ on a :

(1) soit $s \in H_1$ et alors $\widehat{psq} \leq \widehat{prq}$ par définition de r ;

(2) soit $s \in H_2$ et alors de $s, r \notin \mathcal{D}$ on déduit que $\widehat{psq} \leq \pi - c$ et $\widehat{prq} \leq c$ d'où $\widehat{psq} \leq \pi - \widehat{prq}$.

En d'autres termes l'intérieur du cercle passant par p, q et r ne contient aucun point de S ce qui prouve que pqr est un (et unique!) triangle de Delaunay inclus dans H_1 dont $[p, q]$ est un côté. En permutant les rôles de H_1 et H_2 on obtient l'existence d'un unique triangle de Delaunay inclus dans H_2 dont $[p, q]$ est un côté.

Nous montrons maintenant que les intérieurs de deux triangles de Delaunay sont disjoints. Un triangle étant inclus dans son cercle circonscrit, il est clair qu'un triangle de Delaunay ne peut en contenir un autre. Il suffit donc de démontrer que deux arêtes de Delaunay ne peuvent être sécantes. Raisonnons par contradiction. Soient $[p, q]$ et $[r, s]$ deux arêtes de Delaunay sécantes. Il existe un cercle passant par p, q et ne contenant ni r ni s ; soit a un point de ce cercle distinct de p et q ; les points r et s étant de part et d'autre de la droite pq on a, à une permutation près de r et s,

$$\widehat{paq} > \widehat{prq} \text{ et } \pi - \widehat{paq} > \widehat{psq}$$

d'où

$$\widehat{prq} + \widehat{psq} < \pi$$

En permutant les rôles de $[p, q]$ et $[r, s]$ on obtient l'inégalité $\widehat{rps} + \widehat{rqs} < \pi$, qui ajoutée à la précédente donne

$$\widehat{prq} + \widehat{psq} + \widehat{rps} + \widehat{rqs} < 2\pi$$

Or ceci est impossible car la somme des angles d'un quadrilatère convexe vaut 2π.

Nous terminons la démonstration du théorème en montrant que la réunion des triangles de Delaunay est exactement l'enveloppe convexe de S. Soit x un point intérieur à l'enveloppe convexe de S et soit a l'arête de Delaunay la plus proche de x dans une direction donnée. Le triangle de Delaunay de côté a inclus dans le demi-plan, de frontière la droite pq, contenant x contient x. ∎

La démonstration précédente est constructive et suggère de calculer la triangulation de Delaunay par ordre d'adjacence de ses triangles. Le triangle adjacent, d'un côté donné, à une arête de Delaunay est obtenu en cherchant le point de S qui voit cette arête sous un angle maximal. Cette procédure de calcul n'est pas sans rappeler celle que nous avons décrite pour calculer l'enveloppe convexe d'un ensemble fini de points. Cette apparente similarité n'est pas fortuite et s'explique aisément via la transformation géométrique que nous explicitons maintenant.

Considérons le plan E comme le plan $(O, \vec{\imath}, \vec{\jmath})$ d'un espace euclidien $F = (O, \vec{\imath}, \vec{\jmath}, \vec{k})$ de dimension trois. A tout point p de coordonnées (x, y) du plan E, on associe le point de coordonnées $(x, y, x^2 + y^2)$, noté $\phi(p)$, du paraboloïde d'équation $z = x^2 + y^2$. On appelle *enveloppe convexe inférieure* d'une partie X de F l'ensemble $\mathcal{E}(X) + \mathbb{R}^+\vec{k}$.

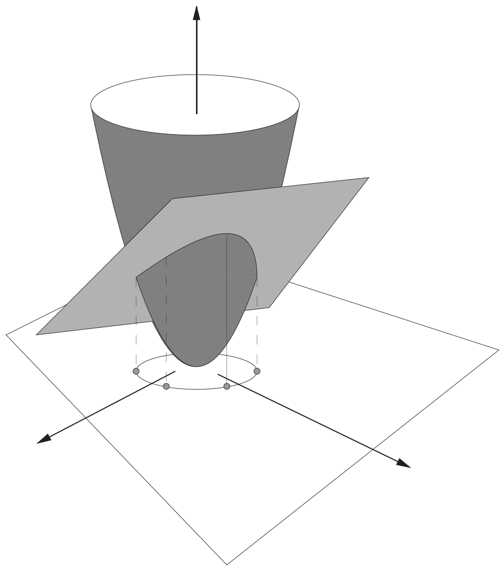


Figure 10.3.5: Le paraboloïde $z = x^2 + y^2$.

THÉORÈME 10.3.4. *Soit S un ensemble fini de points de E. Alors pqr est un triangle de Delaunay de S si et seulement si $\phi(p)\phi(q)\phi(r)$ est une face de l'enveloppe convexe inférieure de $\phi(S)$.*

Preuve. En effet soit $\mathcal{C}$ un cercle passant par les points a, b, c et d'équation $x^2 + y^2 + \alpha x + \beta y + \gamma = 0$ et soit $\mathcal{D}$ son intérieur. Alors

$$\begin{aligned} p = (x, y) \notin \mathcal{D} &\iff x^2 + y^2 + \alpha x + \beta y + \gamma \geq 0 \\ &\iff z + \alpha x + \beta y + \gamma \geq 0 \quad \text{et} \quad z = x^2 + y^2 \\ &\iff \phi(p) \in R[\phi(a), \phi(b), \phi(c)] \end{aligned}$$

où $R[u, v, w]$ est l'enveloppe convexe inférieure du plan passant par les points u, v, w. Par suite pqr est un triangle de Delaunay de S si et seulement si $\phi(S)$ est inclus dans le demi-espace $R[\phi(p), \phi(q), \phi(r)]$ ou encore si et seulement si le plan $\phi(p)\phi(q)\phi(r)$ est le support d'une face de l'enveloppe convexe inférieure de $\phi(S)$. Il reste à montrer que cette face coïncide avec le triangle $\phi(p)\phi(q)\phi(r)$; or quatre points de S n'étant, par hypothèse, jamais cocycliques, leurs images par ϕ ne sont jamais coplanaires et toute face de l'enveloppe convexe de $\phi(S)$ est un triangle. ∎

La triangulation de Delaunay est étroitement liée à une autre structure de base de la géométrie algorithmique. Cette structure apparaît naturellement dans le problème suivant : étant donné un point de E, quel est le point de S le plus proche ? Pour tout point s de S on appelle *région de Voronoï* de s l'ensemble, noté $V(s)$, des points du plan dont la distance à l'ensemble S est réalisée par s

$$V(s) = \{x \in E \mid d(x,s) = d(x,S)\} \tag{3.6}$$

L'ensemble $V(s)$ est un polyèdre convexe car il s'écrit comme l'intersection finie des demi-espaces fermés

$$R_{s'} = \{x \in E \mid d(x,s) \leq d(x,s')\}$$

de frontières les médiatrices des segments $[s,s']$ pour $s' \in S \setminus \{s\}$; de plus $V(s)$ est d'intérieur non vide car, la fonction distance étant continue, $V(s)$ contient un voisinage de s. L'ensemble des régions de Voronoï de S, noté $DV(S)$, est appelé le *diagramme de Voronoï* de S. Les *arêtes* et les *sommets* de $DV(S)$ sont par définition les sommets et les côtés des régions de Voronoï. La triangulation de Delaunay et le diagramme de Voronoï de S sont étroitement liés. Le théorème qui suit précise ce lien.

THÉORÈME 10.3.5. *Les droites support des arêtes de $V(s)$ sont les médiatrices des arêtes de Delaunay de sommet s.*

Les sommets de $V(s)$ sont les centres des cercles circonscrits aux triangles de Delaunay dont l'un des sommets est s.

Deux sommets de $V(s)$ sont les extrémités d'une arête de $V(s)$ si et seulement si les triangles de Delaunay associés sont adjacents.

Preuve. Pour $s_i \in S \setminus \{s\}$, soit $R_i = \{x \in E \mid d(x,s) \leq d(x,s_i)\}$ le demi-plan, contenant s, de frontière la médiatrice H_i du segment $[s,s_i]$.

Les arêtes de $V(S)$ sont les $H_i \cap V(s)$ d'intérieur relatif non vide ; c'est-à-dire qu'il existe un ouvert de H_i dans lequel on a

$$d(x,s) = d(x,s_i) = d(x,S)$$

en particulier $[s,s_i]$ est une arête de Delaunay. Inversement si $[s,s_i]$ est une arête de Delaunay alors il existe un $x \in H_i$ tel que $d(x,s) = d(x,s_i) = d(x,S)$; mais les sites étant en position générale, cette relation reste vraie dans un voisinage à droite ou à gauche (ou les deux) de x dans H_i et $H_i \cap V(s)$ est bien une arête de $V(s)$.

Les sommets de $V(s)$ sont les $H_i \cap H_j \cap V(s)$ qui se réduisent à un point ; c'est-à-dire les points ω tels que

$$d(\omega,s) = d(\omega,s_i) = d(\omega,s_j) = d(\omega,S)$$

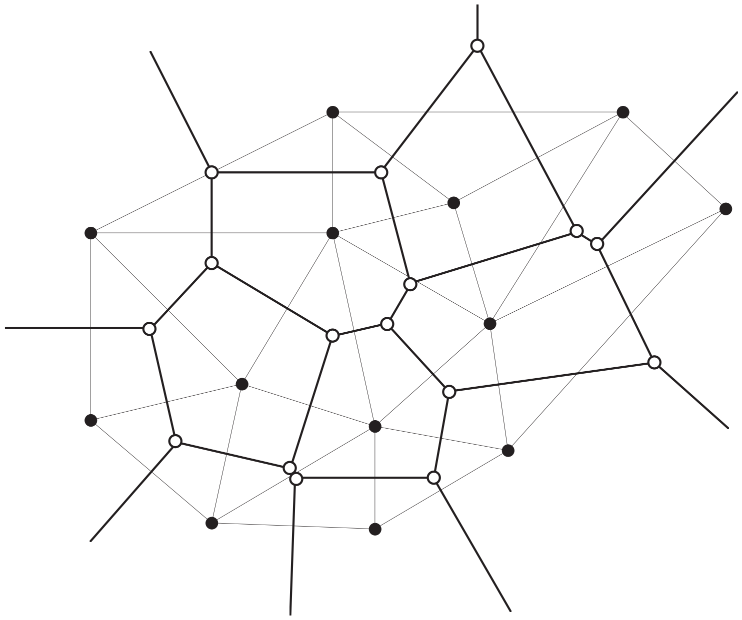


Figure 10.3.6: Le diagramme de Voronoï est le dual de la triangulation de Delaunay.

pour un certain i et un certain j. En d'autres termes ce sont les centres des cercles circonscrits aux triangles de Delaunay de sommet s.

Enfin, les sites étant supposés en position générale, un sommet ω de $V(s)$ détermine de manière unique les droites H_i et H_j dont il est l'intersection; par suite $V(s) \cap H_i$ et $V(s) \cap H_j$ sont bien des arêtes de $V(s)$ dont une extrémité est ω. ∎

Nous terminons en dénombrant les sommets, arêtes et régions de Voronoï.

PROPOSITION 10.3.6. *Soient n le nombre de sites de S et l le nombre de sites de S situés sur la frontière de son enveloppe convexe. Alors les nombres f, a, s de faces, d'arêtes et de sommets du diagramme de Voronoï sont liés par les relations*

(1) $f = n$
(2) $3s = 2a - l$
(3) $s = 2f - l - 2$

Preuve. La première relation est évidente car une région de Voronoï est associée bijectivement à un site.

La seconde relation est obtenue en remarquant que chaque sommet du diagramme de Voronoï est incident à trois arêtes et que chaque arête du diagramme de Voronoï est incidente à deux sommets pour les arêtes bornées et un seul sommet pour les l arêtes non bornées.

Pour démontrer la troisième relation nous choisissons un repère $(O, \vec{\imath}, \vec{\jmath})$ et supposons sans perte de généralité qu'aucune arête du $DV(S)$ n'est verticale (c'est-à dire de direction $\vec{\jmath}$). Un sommet de $DV(S)$ est dit *gauche* (respectivement *droit*) s'il est le point d'abscisse minimale (respectivement maximale) d'une région de $DV(S)$. On observe alors que tout sommet de $DV(S)$ est le sommet gauche ou droit d'une seule région de $DV(S)$, qu'une région de $DV(S)$ bornée admet un sommet droit et un sommet gauche tandis qu'une région non bornée admet un unique point gauche ou droit sauf deux d'entre elles qui n'en admettent aucun. ■

En vertu du théorème 10.3.5, les nombres s', f' et a' de sommets, triangles et arêtes de la triangulation de Delaunay sont respectivement égaux aux nombres f, s et a de faces, sommets et arêtes du diagramme de Voronoï. Les relations de la proposition précédente sont ainsi également vérifiées par les nombres s', f' et a' en lieu et place des nombres f, s et a.

10.3.3 Programme : triangulation de Delaunay

Pour calculer la triangulation de Delaunay d'un ensemble de sites, il faut d'abord saisir les sites, dans un tableau dont le type est défini par :

```
TYPE
   SuitePoints = ARRAY[1..LongueurSuite] OF Point;
```

La constante `LongueurSuite` est choisie de façon convenable. La lecture des sites se fait par saisie successive des points. Il faut fixer une convention pour arrêter la saisie : nous convenons de l'arrêter lorsque deux points consécutifs sont égaux (ceci correspond à un «double-clic» sur un Macintosh). On est amené aux procédures :

```
PROCEDURE SaisirSites (VAR n: integer; VAR S: SuitePoints);
   VAR
      q: Point;
      SaisieTerminee: boolean;
   BEGIN
      n := 1;
      SaisirPoint(S[1]);                          Saisie du premier point.
      MarquerPointRond(S[1]);                     Le point est marqué
      NumeroterPoint(S[1], 1);                    et numéroté.
      REPEAT
         SaisirPoint(q);                          Saisie d'un point.
         SaisieTerminee := SontEgauxPoints(q, S[n]);  Egal au précédent ?
         IF NOT SaisieTerminee THEN BEGIN
            n := n + 1; S[n] := q;                Le point est enregistré,
            MarquerPointRond(S[n]);               marqué et numéroté.
            NumeroterPoint(S[n], n)
         END
      UNTIL SaisieTerminee
   END; { de "SaisirSites" }
```

avec, bien entendu :

```
FUNCTION SontEgauxPoints (VAR p, q: Point): boolean;   p = q ?
   BEGIN
      SontEgauxPoints := (EstNul(Abscisse(p) - Abscisse(q)))
                         AND (EstNul(Ordonnee(p) - Ordonnee(q)))
   END; { de "SontEgauxPoints" }
```

Avant de procéder à la description de l'algorithme de calcul de la triangulation, nous présentons quelques procédures auxiliaires; pour la plupart, le nom décrit entièrement la fonction :

```
FUNCTION det (VAR p, q, r: Point): real;   Calcule dét(pq→, pr→).
        VAR
           pqx, pqy, qrx, qry: real;
   BEGIN
      pqx := Abscisse(q) - Abscisse(p);   pqy := Ordonnee(q) - Ordonnee(p);
      qrx := Abscisse(r) - Abscisse(q);   qry := Ordonnee(r) - Ordonnee(q);
      det := pqx * qry - pqy * qrx
   END; { de "det" }
FUNCTION EstSitue (VAR p, q, r: Point): integer;
   Prend la valeur 1, 0, −1, selon que p est situé à gauche, sur, ou à droite de la droite
   orientée qr.
   BEGIN
      EstSitue := signe(det(p, q, r))
   END; { de "EstSitue" }
FUNCTION EstAGauche (VAR p, q, r: Point): boolean;
   BEGIN
      EstAGauche := EstSitue(p, q, r) = 1
   END; { de "EstAGauche" }
FUNCTION EstADroite (VAR p, q, r: Point): boolean;
   BEGIN
      EstADroite := EstSitue(p, q, r) = -1
   END; { de "EstADroite" }
FUNCTION CarreNorme (VAR p: Point): real;
   BEGIN
      CarreNorme := sqr(Abscisse(p)) + sqr(Ordonnee(p))
   END; { de "CarreNorme" }
FUNCTION CarreDistance (VAR p, q: Point): real;
   BEGIN
      CarreDistance := sqr(Abscisse(q) - Abscisse(p))
         + sqr(Ordonnee(q) - Ordonnee(p))
   END; { de "CarreDistance" }
FUNCTION CosAngle (VAR p, q, r: Point): real;
   Calcule pq→ · pr→/‖pq→‖ ‖pr→‖.
   VAR
      pqx, pqy, prx, pry: real;
```

```
BEGIN
   pqx := Abscisse(q) - Abscisse(p);   pqy := Ordonnee(q) - Ordonnee(p);
   prx := Abscisse(r) - Abscisse(p);   pry := Ordonnee(r) - Ordonnee(p);
   CosAngle := (pqx * prx + pqy * pry) /
      sqrt((sqr(pqx) + sqr(pqy)) * (sqr(prx) + sqr(pry)))
END; { de "CosAngle" }
```

La triangulation de Delaunay de l'ensemble S est calculée par l'algorithme décrit dans la section précédente : pour chaque site s_i, on détermine d'abord son voisin le plus proche s_j puis, en tournant dans le sens des aiguilles d'une montre autour de s_i, on détermine successivement la suite des triangles de Delaunay adjacents dont s_i est un sommet. Deux cas peuvent alors se produire : si s_i n'est pas sur l'enveloppe convexe, on retombe, après un tour complet, sur le sommet s_j. En revanche, si la recherche du triangle de Delaunay suivant s'arrête à une étape donnée, faute de site à explorer, alors le sommet s_i est sur l'enveloppe convexe de S et l'exploration continue en partant de l'arête $[s_i, s_j]$ et en tournant, cette fois-ci, autour de s_i dans le sens contraire.

Considérons par exemple la figure 10.3.7. Le sommet 2 a le sommet 6 comme voisin le plus proche, donc en tournant autour de 2, on découvre successivement les sommets 1, 3, 4, 5, avant de retomber sur 6 qui indique que 2 n'est pas sur l'enveloppe convexe. En revanche, pour le sommet 5, le plus proche voisin est 2. La recherche à droite donne uniquement le sommet 4, ce qui indique que 5 est sur l'enveloppe convexe; puis la recherche à gauche donne le site 6.

Cette recherche conduit à associer, à chaque site s_i, les informations suivantes : la suite finie des sommets qui, avec s_i, forment les triangles de Delaunay dont s_i est un des sommets, puis une marque indiquant que s_i est sur l'enveloppe convexe de S ou non. Nous utilisons, pour conserver ces informations, des tableaux d'entiers qui contiennent les indices des sites concernés. Les types utiles sont :

```
TYPE
   SuiteIndices = ARRAY[1..LongueurSuite] OF integer;
   TableIndices = ARRAY[1..LongueurSuite] OF SuiteIndices;
```

Une table d'indices contient, pour chaque i, la suite des indices des sites trouvés en tournant autour de s_i. La longueur de cette suite sera rangée dans une table d'entiers. L'indication de présence sur l'enveloppe convexe pourra aussi être rangée dans une table de cette nature. Ceci conduit donc à l'en-tête de procédure suivant :

```
PROCEDURE Delaunay (n: integer; S: SuitePoints;
   VAR long: Suiteindices;       long[i] est la longueur de la suite des voisins de i.
   VAR Voisins: TableIndices;    Voisins[i] contient la suite des voisins de i.
   VAR EnvConv: Suiteindices);
```

Le tableau `EnvConv` indique les sommets qui sont sur l'enveloppe convexe. Plus précisément, `EnvConv`$[i] = 1$ si s_i est sur l'enveloppe convexe, 0 sinon. La procédure débute par deux constantes qui servent à orienter la recherche

```
CONST
```

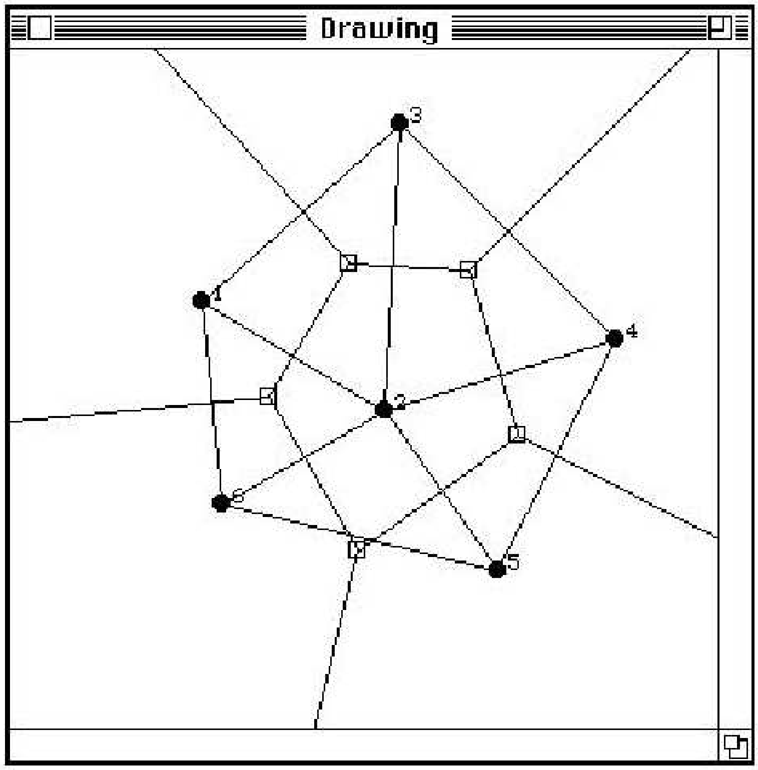


Figure 10.3.7: Une triangulation.

```
      AGauche = 1;
      ADroite = -1;
```

La fonction que voici calcule le site s_j le plus proche de s_i, en comparant les carrés des distances de s_i aux autres sites :

```
   FUNCTION LePlusProcheVoisin (i: integer): integer;
      VAR
         j, k: integer;
         d, dk: real;
      BEGIN
         IF i = 1 THEN j := 2 ELSE j := 1;       Premier site examiné.
         d := CarreDistance(S[i], S[j]);          Le carré de la distance.
         FOR k := 1 TO n DO
            IF k <> i THEN BEGIN                  Pour les sites autres que s_i,
               dk := CarreDistance(S[i], S[k]);   calcul du carré de la distance,
```

```
            IF dk < d THEN BEGIN              et si le site est plus proche, on
               d := dk; j := k                mémorise son indice.
            END
         END;
      LePlusProcheVoisin := j
   END; { de "LePlusProcheVoisin" }
```

On détermine un triangle de Delaunay en considérant une arête de Delaunay $[s_i, s_j]$ et en cherchant le site s_k qui forme, avec les deux sites, un angle maximal. La fonction suivante calcule l'indice k d'un tel site situé à droite ou à gauche de $\vec{s_i s_j}$, selon la valeur du paramètre `cote`; s'il n'y en a pas, la fonction retourne 0.

```
FUNCTION VoisinSuivant (i, j, cote: integer): integer;
   VAR
      a, k: integer;
      cosmin, cosinus: real;
   BEGIN
      a := 0;
      cosmin := 1;            Initialisation : au début, le cosinus minimal vaut 1.
      FOR k := 1 TO n DO      Recherche parmi les sites du côté donné.
         IF (k <> i) AND (k <> j)
         AND (EstSitue(S[k], S[i], S[j]) = cote) THEN BEGIN
            cosinus := CosAngle(S[k], S[i], S[j]);     Calcul du cosinus,
            IF cosinus < cosmin THEN BEGIN             on garde le plus petit.
               cosmin := cosinus; a := k
            END;
         END;
      VoisinSuivant := a
   END; { de "VoisinSuivant" }
```

Cette procédure est utilisée dans :

```
FUNCTION VoisinDroit (i, j: integer): integer;
   BEGIN
     VoisinDroit := VoisinSuivant(i, j, ADroite)
   END; { de "VoisinDroit" }
FUNCTION VoisinGauche (i, j: integer): integer;
   BEGIN
      VoisinGauche := VoisinSuivant(i, j, AGauche)
   END; { de "VoisinGauche" }
```

Lorsque ces fonctions rendent un indice non nul, il est inséré dans la table des voisins soit à gauche, soit à droite. Les deux procédures suivantes se chargent de l'insertion :

```
PROCEDURE InsererADroite (k, i: integer);    Insertion de k à la fin.
   BEGIN
      long[i] := long[i] + 1;
      Voisins[i][long[i]] := k
   END; { de "InsererADroite" }
```

```
PROCEDURE InsererAGauche (k, i: integer);   Insertion de k au début.
   VAR
      j: integer;
   BEGIN
      FOR j := long[i] DOWNTO 1 DO          On décale d'abord les autres voisins,
         Voisins[i][j + 1] := Voisins[i][j];
      long[i] := long[i] + 1;               puis on insère en tête.
      Voisins[i][1] := k
   END; { de "InsererAGauche" }
```

Après les déclarations :

```
VAR
   i, j, k, m: integer;
```

Voici le corps de la procédure :

```
BEGIN { de "Delaunay" }
   FOR i := 1 TO n DO BEGIN                 Pour chaque site s_i, on cherche
      m := LePlusProcheVoisin(i);           d'abord le plus proche voisin.
      long[i] := 1; Voisins[i][1] := m;     On initialise la liste des voisins.
      j := m;
      k := VoisinDroit(i, j);               Premier site adjacent à droite.
      WHILE (k <> 0) AND (k <> m) DO BEGIN  Progression à droite.
         InsererADroite(k, i);
         j := k; k := VoisinDroit(i, j)     Recherche du suivant.
      END;
      IF k = m THEN                         Si la boucle est bouclée : s_i n'est pas
         EnvConv[i] := 0                    sur l'enveloppe convexe.
      ELSE BEGIN                            Sinon, s_i est
         EnvConv[i] := 1;                   sur l'enveloppe convexe.
         j := m;
         k := VoisinGauche(i, j);           Premier site adjacent à gauche.
         WHILE k <> 0 DO BEGIN              Progression à gauche.
            InsererAGauche(k, i);
            j := k; k := VoisinGauche(i, j)
         END
      END
   END { boucle sur i }
END; { de "Delaunay" }
```

Voici les calculs faits pour les sites de la figure 10.3.7 :

```
Le plus proche voisin de 1 est 6
Le voisin droit  de (1,6) est 0
Le voisin gauche de (1,6) est 2
Le voisin gauche de (1,2) est 3
Le voisin gauche de (1,3) est 0
Le plus proche voisin de 2 est 6
Le voisin droit  de (2,6) est 1
```

```
Le voisin droit  de (2,1) est 3
Le voisin droit  de (2,3) est 4
Le voisin droit  de (2,4) est 5
Le voisin droit  de (2,5) est 6
Le plus proche voisin de 3 est 2
Le voisin droit  de (3,2) est 1
Le voisin droit  de (3,1) est 0
Le voisin gauche de (3,2) est 4
Le voisin gauche de (3,4) est 0
Le plus proche voisin de 4 est 5
Le voisin droit  de (4,5) est 2
Le voisin droit  de (4,2) est 3
Le voisin droit  de (4,3) est 0
Le voisin gauche de (4,5) est 0
Le plus proche voisin de 5 est 2
Le voisin droit  de (5,2) est 4
Le voisin droit  de (5,4) est 0
Le voisin gauche de (5,2) est 6
Le voisin gauche de (5,6) est 0
Le plus proche voisin de 6 est 2
Le voisin droit  de (6,2) est 5
Le voisin droit  de (6,5) est 0
Le voisin gauche de (6,2) est 1
Le voisin gauche de (6,1) est 0
```

Venons-en au tracé du diagramme de Voronoï (le tracé des triangles de Delaunay est immédiat). Pour cela, on parcourt les triangles de Delaunay, on calcule le centre du cercle circonscrit et on relie des centres adjacents. Voici une réalisation :

```
PROCEDURE TracerVoronoi (n: integer; S: SuitePoints;
   long, conv: Suiteindices; Voisin: TableIndices);
   Trace les sommets et arêtes du diagramme de Voronoï associé à la triangulation de
   Delaunay.
   VAR
      a, b, c, dir: Point;
      i, j: integer;
   PROCEDURE CentreCercleCirconscrit (i, j, k: integer; VAR m: Point);
      Calcule le centre m du cercle circonscrit au triangle s_i, s_j, s_k,
      en résolvant directement le système linéaire.
      VAR
         abx, aby, acx, acy, ab, ac, delta: real;
         a, b, c: Point;
      BEGIN
         a := S[i]; b := S[j]; c := S[k];
         abx := Abscisse(b) - Abscisse(a); aby := Ordonnee(b) - Ordonnee(a);
         acx := Abscisse(c) - Abscisse(a); acy := Ordonnee(c) - Ordonnee(a);
         delta := abx * acy - aby * acx;
         ab := CarreNorme(b) - CarreNorme(a);
```

```
        ac := CarreNorme(c) - CarreNorme(a);
        FairePoint((ab * acy - ac * aby) / (2 * delta),
                   (ac * abx - ab * acx) / (2 * delta), m);
    END; { de "CentreCercleCirconscrit" }

  PROCEDURE FaireDirectionPerpendiculaire (p, q: point; VAR d: point);
    BEGIN
      FairePoint(Ordonnee(p) - Ordonnee(q), Abscisse(q) - Abscisse(p), d);
    END; { de "FaireDirectionPerpendiculaire" }

  BEGIN { de "TracerVoronoi" }
    FOR i := 1 TO n DO
      IF EnvConv[i] = 1 THEN BEGIN              s_i est sur l'enveloppe convexe.
        CentreCercleCirconscrit(i, Voisin[i][1], Voisin[i][2], c);
        FaireDirectionPerpendiculaire(S[i], S[Voisin[i][1]], dir);
        MarquerPointCarre(c, 2);
        TracerDemiDroite(c, dir);
        FOR j := 2 TO long[i] - 1 DO BEGIN
          b := c;
          CentreCercleCirconscrit(i, Voisin[i][j], Voisin[i][j + 1], c);
          MarquerPointCarre(c, 2);
          TracerSegment(b, c)
        END;
      END
      ELSE BEGIN
        CentreCercleCirconscrit(i, Voisin[i][long[i]], Voisin[i][1], c);
        a := c;
        FOR j := 1 TO long[i] - 1 DO BEGIN
          b := c;
          CentreCercleCirconscrit(i, Voisin[i][j], Voisin[i][j + 1], c);
          MarquerPointCarre(c);
          TracerSegment(b, c)                   Segment du centre au précédent.
        END;
        TracerSegment(c, a)                     Segment du premier au dernier.
      END
  END; { de "TracerVoronoi" }
```

10.4 Galerie d'art

10.4.1 Enoncé : galerie d'art

Le problème dit de la *galerie d'art* (ou des gardiens de prisons) est le suivant : « Quel est le nombre de gardiens nécessaires pour surveiller une galerie d'art et où faut-il les placer ? » On le modélise comme suit.

Soit E un plan euclidien affine orienté muni d'un repère orthonormé $(O, \vec{\imath}, \vec{\jmath})$. On note $[p, q]$ le segment de droite fermé joignant les points p et q.

Soit $n \geq 3$ et soit $S = (p_1, \ldots, p_n)$ une suite de points distincts (appelés *sommets*) dont trois quelconques ne sont pas alignés. Le contour polygonal fermé $C = [p_1, p_2] \cup \ldots \cup [p_n, p_1]$ est *simple* si $[p_i, p_{i+1}] \cap [p_j, p_{j+1}] = \emptyset$ pour $1 \leq i, j \leq n$ et $i \neq j-1, j, j+1$ (les indices sont pris modulo n). Dans ce cas, le contour C sépare le plan en deux régions ; le *polygone simple* P défini par S est la réunion de C et de la région bornée, appelée l'*intérieur* de P. De plus, on suppose que pour un observateur qui parcourt le contour dans le sens croissant des indices, l'intérieur est à gauche (voir figure à la page suivante).

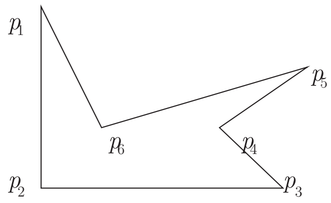


Figure 10.4.1: Un polygone simple.

Les sommets de P sont donnés par leurs coordonnées. Soit P un polygone simple. Un sommet p_i de P est une *oreille* si le triangle $p_{i-1}p_ip_{i+1}$ est contenu dans P (les indices sont pris modulo n). Sur la figure, p_1, p_3, p_5 sont les oreilles. On admet que tout polygone simple P possède au moins deux oreilles.

1.– Ecrire une procédure qui teste si un sommet p_i est une oreille. (On pourra se servir du fait que p_i est une oreille si et seulement si p_{i+1} est à gauche de la droite orientée $\overrightarrow{p_{i-1}p_i}$ et le triangle $p_{i-1}p_ip_{i+1}$ ne contient aucun autre sommet de P.)

Une *triangulation* de P est un ensemble de triangles vérifiant les trois conditions suivantes :

(1) les sommets des triangles sont des sommets de P ;
(2) la réunion des triangles est P ;
(3) les intérieurs des triangles sont deux à deux disjoints.

2.– Ecrire une procédure qui calcule une triangulation du polygone P. Quel est le nombre de triangles dans une triangulation ?

3.– Ecrire un programme qui prend en argument la suite des coordonnées des sommets d'un polygone P que l'on suppose simple, et qui dessine P ainsi que la triangulation calculée.

Exemple numérique : $n = 8$, $p_1(0,8)$, $p_2(0,0)$, $p_3(4,0)$, $p_4(2,6)$, $p_5(11,6)$, $p_6(10,1)$, $p_7(16,1)$, $p_8(16,8)$.

Un *coloriage* (à 3 couleurs) d'une triangulation de P est une application de l'ensemble des sommets de P dans un ensemble à 3 éléments (les « couleurs ») telle que les 3 sommets d'un triangle aient des couleurs différentes.

4.– Prouver qu'il existe un et un seul coloriage à une permutation près des couleurs. Ecrire une procédure qui en calcule un.

Revenons au problème de la galerie d'art. On dit qu'un point $g \in P$ *couvre* le point $x \in P$ si $[g, x] \subset P$. Chercher un ensemble de gardiens revient donc à chercher une partie finie $G \subset P$ telle que tout point de P est couvert par un point de G. Un tel ensemble est une *couverture* de P.

5.– Soit c l'une des couleurs d'un coloriage et soit G l'ensemble des sommets de P de couleur c. Démontrer que G est une couverture de P. En déduire que pour surveiller une galerie d'art à n sommets, $\lfloor n/3 \rfloor$ gardiens suffisent. (On note $\lfloor x \rfloor$ la partie entière de x.)

6.– Compléter votre programme pour qu'il marque d'un signe distinctif les sommets d'une couverture du polygone.

7.– Donner un exemple d'un polygone simple à n sommets pour lequel toute couverture a au moins $\lfloor n/3 \rfloor$ sommets.

8.– Démontrer que tout polygone simple possède au moins deux oreilles.

9.– Donner un exemple d'un polygone simple à n sommets possédant exactement deux oreilles.

10.4.2 Solution : galerie d'art

Soit P un polygone simple d'un plan affine euclidien.

Une *triangulation* de P est un ensemble $\mathcal{T}$ de triangles vérifiant les trois conditions suivantes :

(1) les sommets des triangles sont des sommets de P ;
(2) la réunion des triangles est P ;
(3) les intérieurs des triangles sont deux à deux disjoints.

Observons que toute arête du polygone P est un côté d'un unique triangle de la triangulation de P.

Il sera utile de noter $[p_1, p_2, \ldots, p_n]$ le polygone défini par le contour polygonal simple $[p_1, p_2] \cup [p_2, p_3] \cup \cdots \cup [p_n, p_1]$.

On appelle *diagonale* de P tout segment $[p_i, p_j]$ joignant deux sommets non consécutifs de P tel que le segment ouvert $]p_i, p_j[$ soit inclus dans l'intérieur du polygone P. Dans ce cas en supposant $i < j$ le polygone P est la réunion des deux polygones d'intérieurs disjoints

$$[p_1, \ldots, p_i, p_j, \ldots, p_n] \text{ et } [p_i, \ldots, p_j] \tag{4.1}$$

Dans le cas où $j = i + 2$, le point p_{i+1} est une oreille.

Théorème 10.4.1. *Tout polygone simple à $n \geq 4$ sommets admet une diagonale.*

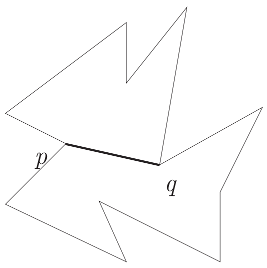


Figure 10.4.2: Tout polygone ayant au moins 4 sommets admet une diagonale.

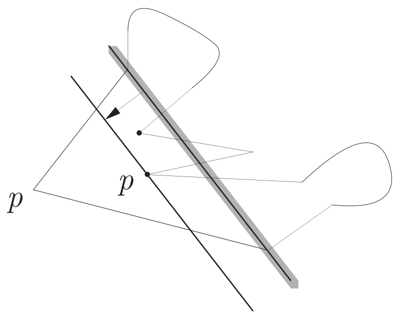


Figure 10.4.3: Preuve du théorème de l'existence d'une diagonale.

Preuve. Soit p_i un sommet convexe du polygone P, c'est-à-dire tel que l'angle du polygone P en p_i soit de mesure $\leq \pi$ (par exemple tout sommet de l'enveloppe convexe du polygone P), et soit C l'ensemble des sommets de P intérieur au triangle $p_{i-1}p_ip_{i+1}$. Si C est vide alors $[p_{i-1}, p_{i+1}]$ est une diagonale et le théorème est démontré ; dans le cas contraire soit p_j un point de C qui réalise le maximum des distances des points de C à la droite $p_{i-1}p_{i+1}$; alors $[p_i, p_j]$ est une diagonale de P ; ce qui termine la démonstration du théorème. ∎

Corollaire 10.4.2. *Tout polygone simple admet une triangulation.*

Corollaire 10.4.3. *Tout polygone simple à $n \geq 4$ sommets possède au moins deux oreilles non consécutives.*

Preuve des corollaires. Nous démontrons ces résultats par récurrence sur la taille du polygone. Si le polygone $P = [p_1, p_2, p_3]$ est un triangle alors $\{P\}$ est une triangulation de P et P admet trois oreilles. Si $P = [p_1, p_2, p_3, p_4]$ est un quadrilatère alors, en supposant

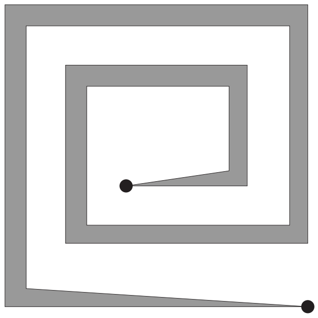


Figure 10.4.4: Un polygone ayant exactement deux oreilles.

sans perte de généralité que $[p_1, p_3]$ est une diagonale, p_2 et p_4 sont deux oreilles non consécutives et les triangles $p_1p_2p_3$ et $p_1p_4p_3$ forment une triangulation de P.

Supposons maintenant que le polygone P possède au moins cinq sommets et soit, en vertu du théorème précédent, $[p_i, p_j]$ une diagonale de P qui scinde P en deux polygones simples P_1 et P_2. La réunion d'une triangulation de P_1 et d'une triangulation de P_2 forme alors une triangulation de P. Enfin les polygones P_1 et P_2 admettent chacun une oreille distincte de p_i et p_j, soit par hypothèse de récurrence s'ils ont plus de 4 sommets soit parce qu'un triangle admet trois oreilles. Ces deux oreilles sont alors des oreilles non consécutives de P. ∎

Corollaire 10.4.4. *Toute triangulation d'un polygone simple à n sommets est formée de $n - 2$ triangles.*

Preuve. Raisonnons par récurrence sur la taille n du polygone simple P. La seule triangulation d'un triangle P étant le singleton $\{P\}$ le résultat est acquis pour $n = 3$. Supposons maintenant $n \geq 4$ et soit $\mathcal{T}$ une triangulation de P. Les arêtes de la triangulation, autres que celles de P, sont des diagonales de P. Une telle arête partage le polygone P en deux polygones P_1 et P_2 de tailles $n_1, n_2 < n$ et la triangulation $\mathcal{T}$ est la réunion disjointe d'une triangulation de P_1 et d'une triangulation de P_2. Par hypothèse de récurrence, ces deux triangulations contiennent respectivement $n_1 - 2$ et $n_2 - 2$ triangles ; comme $n_1+n_2 = n+2$, la triangulation $\mathcal{T}$ a pour taille $(n_1-2)+(n_2-2) = n-2$. ∎

Proposition 10.4.5. *Toute triangulation d'un polygone simple admet un et un seul coloriage, à une permutation près des couleurs.*

Preuve. Soit $\mathcal{T}$ une triangulation de P ; on observe que si les couleurs des sommets d'un triangle T de $\mathcal{T}$ sont fixés, la couleur des sommets des triangles adjacents à T est également fixée et par suite la couleur de tout sommet d'un triangle obtenu par la fermeture transitive $\mathcal{R}$ de la relation d'adjacence. Pour montrer l'unicité et l'existence

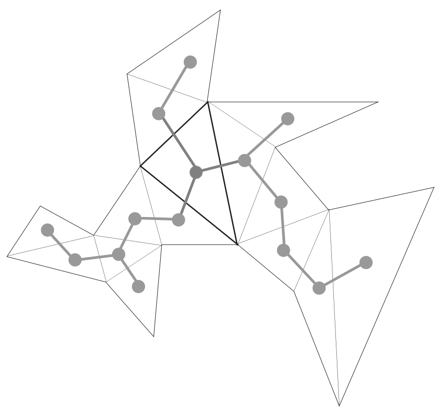


Figure 10.4.5: La relation d'adjacence est connexe et sans cycle.

d'un tel coloriage, il suffit de montrer que la relation d'adjacence est connexe et sans cycle, c'est-à-dire que pour tout triangle $T' \in \mathcal{T}$, il existe une suite

$$T = T_1, \ldots, T_h = T'$$

de triangles adjacents et qu'il n'existe pas de suite circulaire de triangles adjacents de longueur supérieure à trois. Soit

$$\mathcal{U}_T = \bigcup_{T'} T'$$

la réunion des triangles T' tels que $T'\mathcal{R}T$. L'ensemble $\mathcal{U}_T$ est non seulement un fermé, comme réunion finie des fermés que sont les triangles T', mais est aussi un ouvert de P car tout point x de $\mathcal{U}_T$ appartient à l'intérieur, dans P, de la réunion des triangles T' qui contiennent x. Le polygone P étant connexe, il coïncide avec $\mathcal{U}_T$; ce qui prouve que la relation d'adjacence est connexe.

Pour démontrer que la relation est sans cycle on observe que si $T = [p_i, p_j, p_k]$ avec $i < j < k$ alors les triangles obtenus par adjacence avec respectivement les côtés $[p_i, p_j]$, $[p_j, p_k]$ et $[p_k, p_i]$ se situent respectivement dans les polygones

$$P_1 = [p_1, \ldots, p_i, p_k, \ldots, p_n] \quad P_2 = [p_i, \ldots, p_j] \quad P_3 = [p_k, \ldots, p_i]$$

qui sont disjoints deux à deux sauf en p_i, p_j et p_k. ∎

Proposition 10.4.6. *Pour couvrir une galerie d'art à n sommets, il suffit de $\lfloor n/3 \rfloor$ gardiens.*

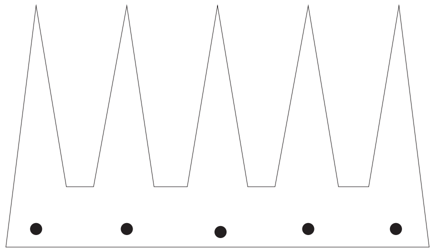


Figure 10.4.6: $\lfloor n/3 \rfloor$ gardiens sont parfois nécessaires.

Preuve. En effet il est clair que le sommet de couleur c d'un triangle couvre ce triangle et que par suite l'ensemble des sommets de couleur c couvrent tout le polygone. Soient g_1, g_2 et g_3 le nombre de sommets de couleur c_1, c_2 et c_3. Quitte à permuter les indices nous pouvons supposer que $g_1 \leq g_2 \leq g_3$; de

$$g_1 + g_2 + g_3 = n$$

on déduit

$$g_1 \leq \lfloor n/3 \rfloor$$

En plaçant les gardiens aux sommets de la couleur la moins fréquente, la galerie est couverte par au plus $\lfloor n/3 \rfloor$ gardiens. ∎

L'exemple de la figure 10.4.6 se généralise aisément pour montrer l'existence de galerie de taille n nécessitant $\lfloor n/3 \rfloor$ gardiens pour la surveiller.

10.4.3 Programme : galerie d'art

Nous allons utiliser, pour la programmation du problème des gardiens d'une galerie d'art, les types et certaines des procédures déjà employés pour le calcul de la triangulation de Delaunay. En particulier, un polygone est une suite finie de points, donc rangé dans un tableau défini par

```
TYPE
   SuitePoints = ARRAY[1..LongueurSuite] OF Point;
```

La lecture des points pourrait se faire par la procédure de saisie déjà développée. Il est bien plus agréable, à l'usage, de pouvoir suivre la construction du polygone, en observant le contour polygonal, au fur et à mesure de sa saisie. Nous traçons donc les côté du polygone, dès que nous les connaissons :

```
PROCEDURE SaisirPolygone (VAR n: integer; VAR P: SuitePoints);
   VAR
      q: Point;
```

```
      SaisieTerminee: boolean;
   BEGIN
      n := 1;
      SaisirPoint(P[1]);                          Saisie du premier sommet.
      NumeroterPoint(P[1], 1);                    Le sommet est numéroté.
      REPEAT
         SaisirPoint(q);
         SaisieTerminee := SontEgauxPoints(q, P[n]);
         IF NOT SaisieTerminee THEN BEGIN
            n := n + 1; P[n] := q;                Le sommet est enregistré,
            NumeroterPoint(P[n], n);              numéroté et
            TracerSegment(P[n - 1], P[n])         relié au sommet précédent.
         END
      UNTIL SaisieTerminee;
      TracerSegment(P[n], P[1]);                  Le tracé du polygone est fermé.
   END; { de "SaisirPolygone" }
```

La première tâche est de vérifier si un sommet p_i d'un polygone P est une oreille. Pour cela, on considère les sommets p_j et p_k qui le précèdent et le suivent sur le polygone. Le triangle $p_jp_ip_k$ doit être direct et ne doit contenir aucun autre sommet du polygone. Cette condition s'écrit comme suit :

```
FUNCTION EstOreille (i, n: integer; VAR P: SuitePoints): boolean;
   Répond affirmativement, si p_i est une oreille.
   VAR
      oreille: boolean;
      j, k, m: integer;
   BEGIN
      j := prec(i, n);                            p_j précède p_i.
      k := suiv(i, n);                            p_k suit p_i.
      oreille := EstAGauche(P[j], P[i], P[k]);    p_j p_i p_k est un triangle direct.
      m := 1;
      WHILE (m <= n) AND oreille DO BEGIN
         IF (m <> j) AND (m <> i) AND (m <> k) THEN
            oreille := NOT (
               EstAGauche(P[j], P[i], P[m]) AND   Le triangle p_j p_i p_k ne
               EstAGauche(P[i], P[k], P[m]) AND   contient aucun autre sommet.
               EstAGauche(P[k], P[j], P[m]));
         m := m + 1
      END;
      EstOreille := oreille;
   END; { de "EstOreille" }
```

Pour calculer l'indice précédent et suivant modulo n, on fait appel aux deux fonctions :

```
FUNCTION suiv (i, n: integer): integer;
   BEGIN
      IF i = n THEN suiv := 1 ELSE suiv := i + 1;
   END; { de "suiv" }
```

```
FUNCTION prec (i, n: integer): integer;
   BEGIN
      IF i = 1 THEN prec := n ELSE prec := i - 1
   END; { de "prec" }
```

La première peut bien sûr être programmée aussi bien par `1 + i MOD n`, pour la deuxième, il n'y a pas de formule aussi simple.

Pour trianguler P, nous allons dresser la liste des triangles. Un triangle est une suite de trois indices de sommets, d'où les déclarations de type :

```
TYPE
   Triangle = ARRAY[1..3] OF integer;
   Triangulation = ARRAY[1..LongueurSuite] OF Triangle;
   SuiteIndices = ARRAY[1..LongueurSuite] OF integer;
```

Le dernier type servira, comme on le verra, pour le calcul des couleurs. Revenons à la triangulation. L'algorithme procède comme suit : on cherche une oreille, ce qui nous donne un triangle, formé de l'oreille et des deux sommets voisins. On enlève l'oreille du polygone et on recommence, tant qu'il reste au moins trois sommets. Pour que les triangles contiennent les bons numéros, ils doivent porter les numéros des sommets dans le polygone de départ. Il convient donc de les conserver et de les mettre à jour, dans un tableau que nous appelons `num`. Voici la procédure :

```
PROCEDURE Trianguler (n: integer; P: SuitePoints; VAR X: triangulation);
   Calcule dans X une triangulation du polygone P.
   VAR
      i, m: integer;
      num: SuiteIndices;
   PROCEDURE ChercherOreille (VAR i: integer);      Voir ci-dessous.
   PROCEDURE CreerTriangle (i, m: integer);         Voir ci-dessous.
   PROCEDURE CouperOreille (i: integer);            Voir ci-dessous.
   BEGIN  { de "Trianguler" }
      FOR i := 1 TO n DO num[i] := i;               Initialisation des numéros.
      m := 0;                                       Compte le nombre de triangles.
      REPEAT
         ChercherOreille(i);                        On cherche une oreille p_i.
         m := m + 1; CreerTriangle(i, m);           On crée un m-ième triangle.
         CouperOreille(i);                          On enlève le sommet p_i.
         n := n - 1                                 On met à jour la taille de P.
      UNTIL n = 2;
   END; { de "Trianguler" }
```

Voici le détail des trois procédures employées. La recherche d'une oreille se fait par un parcours du polygone :

```
   PROCEDURE ChercherOreille (VAR i: integer);
      BEGIN
         i := 1;
```

```
      WHILE (i <= n) AND NOT EstOreille(i, n, P) DO
         i := i + 1;
   END; { de "ChercherOreille" }
```

Dans la procédure de création du triangle, on relève les numéros d'origine des sommets :

```
PROCEDURE CreerTriangle (i, m: integer);
   BEGIN
      X[m][1] := num[prec(i, n)];
      X[m][2] := num[i];
      X[m][3] := num[suiv(i, n)]
   END; { de "CreerTriangle" }
```

Pour supprimer le i-ième sommet, on décale les autres; mais il ne faut pas oublier leurs numéros :

```
PROCEDURE CouperOreille (i: integer);
   VAR
      j: integer;
   BEGIN
      FOR j := i TO n - 1 DO BEGIN
         P[j] := P[j + 1];
         num[j] := num[j + 1]
      END
   END; { de "CouperOreille" }
```

Pour visualiser la triangulation, on trace les triangles à l'aide de la procédure :

```
PROCEDURE TracerTriangle (VAR P: SuitePoints; VAR t: Triangle);
   BEGIN
      TracerSegment(P[t[1]], P[t[2]]);
      TracerSegment(P[t[2]], P[t[3]]);
      TracerSegment(P[t[3]], P[t[1]])
   END; { de "TracerTriangle" }
```

Pour colorer les sommets, on examine successivement les triangles de la triangulation. On colore arbitrairement le premier triangle, puis les triangles adjacents. On recommence jusqu'à avoir coloré tous les triangles. Au lieu d'une structure compliquée, nous testons l'adjacence d'un triangle aux triangles déjà colorés par le fait que deux de ses sommets sont déjà colorés. Voici une réalisation :

```
PROCEDURE Colorer (n: integer; VAR Couleur: SuiteIndices; X: Triangulation);
   VAR
      i, k, m, SommetsNonColores: integer;
   FUNCTION Acolorer (t: triangle): integer;
      Le triangle t est à colorer s'il possède exactement un sommet incolore.
      VAR
         k, j, incolores: integer;
      BEGIN
         k := 0; incolores := 0;
```

```
      FOR j := 1 TO 3 DO                    On parcourt les sommets et on
         IF Couleur[t[j]] = 0 THEN BEGIN    compte les sommets incolores.
            incolores := incolores + 1;
            k := t[j];
         END;
      IF incolores = 1 THEN                 Si un seul sommet est à colorer,
         Acolorer := k                      son numéro est k ;
      ELSE
         Acolorer := 0                      sinon, ne rien faire.
   END; { de "Acolorer" }
BEGIN { de "Colorer" }
   FOR i := 1 TO n DO                       Initialisation.
      Couleur[i] := 0;
   Couleur[X[1][1]] := 1;                   Coloration du premier triangle.
   Couleur[X[1][2]] := 2;
   Couleur[X[1][3]] := 3;
   SommetsNonColores := n - 3;              Restent à colorer.
   WHILE SommetsNonColores > 0 DO
      FOR m := 2 TO n - 2 DO BEGIN
         k := Acolorer(X[m]);               Triangle à colorer ?
         IF k > 0 THEN BEGIN                La couleur complémentaire :
            Couleur[k] := 6 -
               (Couleur[X[m][1]] + Couleur[X[m][2]] + Couleur[X[m][3]]);
            SommetsNonColores := SommetsNonColores - 1
         END
      END
END; { de "Colorer" }
```

Enfin, pour choisir le meilleur emplacement des gardiens, on choisi la couleur la moins fréquente :

```
FUNCTION LaMeilleureCouleur (n: integer; Couleur: SuiteIndices): integer;
   Calcule une couleur qui apparaît le moins souvent dans la coloration des sommets.
   VAR
      Card: ARRAY[1..3] OF integer;
      Cardmin, i, c: integer;
   BEGIN
      FOR c := 1 TO 3 DO Card[c] := 0;
      FOR i := 1 TO n DO BEGIN                    Calcul des fréquences.
         c := Couleur[i];
         Card[c] := Card[c] + 1
      END;                                        La plus petite fréquence.
      Cardmin := min(Card[1], min(Card[2], Card[3]));
      IF Card[1] = Cardmin THEN
         LaMeilleureCouleur := 1
      ELSE IF Card[2] = Cardmin THEN
         LaMeilleureCouleur := 2
      ELSE
         LaMeilleureCouleur := 3
   END; { de "LaMeilleureCouleur" }
```

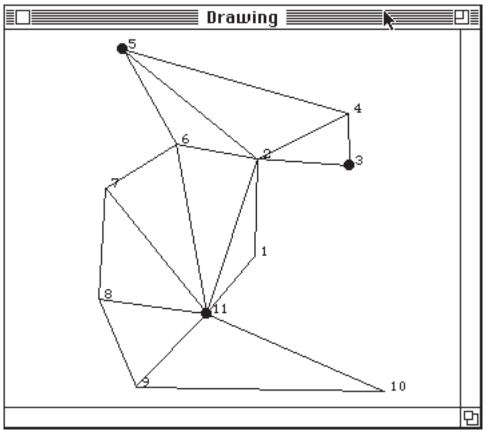


Figure 10.4.7: Une triangulation et l'emplacement des gardiens.

Notes bibliographiques

Les algorithmes géométriques efficaces sont en général très sophistiqués. Ils font appel à des structures de données élaborées; de plus, l'analyse de la performance ainsi que la preuve d'un tel algorithme sont souvent fort complexes. Ils dépassent donc le cadre de ce livre. Un exposé de synthèse est :

R. Graham, F. Yao, A whirlwind tour of computational geometry, *American Math. Monthly* **97**, Octobre 1990, 687–701.

Pour les fondements géométriques, on pourra consulter :

M. Berger, *Géométrie : Convexes et Polytopes, Polyèdres Réguliers, Aires et Volumes*, Paris, Cedic/Fernand Nathan, 1978, Vol.3.

B. Grünbaum, *Convex Polytopes*, London, John Wiley & Sons, London, 1967.

Les algorithmes de calcul de l'enveloppe convexe et des triangulations sont décrits dans les ouvrages de base suivants :

K. Melhorn, *Data Structures and Algorithms 3, Multi-dimensional Searching and Computational Geometry*, Berlin, Springer-Verlag, 1984.

F.P. Preparata et M.I. Shamos, *Computational Geometry : an Introduction*, New York, Springer-Verlag 1985.

H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Berlin, Springer-Verlag, 1987.

Le problème de la galerie d'art est tiré de la monographie :

J. O'Rourke, *Art Gallery Theorems and Algorithms*, London, Oxford University Press, 1987.

La géométrie algorithmique est en pleine expansion. Le résultat le plus spectaculaire de l'année 1990 fut l'annonce, par B. Chazelle de l'université de Princeton, d'un algorithme de complexité linéaire pour trianguler un polygone simple :

B. Chazelle, Triangulating a simple polygon in linear time, Actes du colloque *Foundations of Computer Science*, 1990.

Partie V

Arithmétique

Chapitre 11

Problèmes arithmétiques

11.1 Entiers de Gauss

11.1.1 Enoncé : entiers de Gauss

On note K l'ensemble des nombres complexes de la forme $a + ib$ avec $a, b \in \mathbb{Z}$. On note P l'ensemble des éléments de K de la forme $a + ib$ avec $a > 0$ et $b \geq 0$.

1.– a) Démontrer que pour tout $u \in K, v \in K - \{0\}$, il existe $q \in K$ et $r \in K$ tels que

$$u = qv + r \text{ et } |r| < |v| \tag{1}$$

b) En déduire que K est un anneau principal.

c) Ecrire une procédure qui prend en argument un élément u de K et un élément v de $K - \{0\}$, et calcule un couple (q, r) vérifiant les conditions (1).

2.– On dit qu'un élément z de K est *irréductible* s'il appartient à $P\backslash\{1\}$ et si tous ses diviseurs dans K sont de module 1 ou $|z|$. On note I l'ensemble des éléments irréductibles.

a) Démontrer que si z est irréductible et divise z_1z_2, alors z divise z_1 ou z divise z_2.

b) Démontrer que tout élément z de $K - \{0\}$ admet une décomposition unique de la forme

$$z = i^{c(z)} \prod_{x \in I} x^{c_x(z)} \tag{2}$$

avec $c(z) \in \{0, 1, 2, 3\}, c_x(z) \in \mathbb{N}$ et où l'ensemble des $x \in I$ tels que $c_x(z) \neq 0$ est fini.

c) Démontrer que 2 n'est pas irréductible.

3.– Soit p un nombre premier.

a) Démontrer que $(p - 1)! \equiv -1 \bmod p$.

b) Démontrer que si $p \not\equiv 3 \bmod 4$, il existe un unique entier naturel $d_p \leq p/2$ tel que

$$d_p^2 \equiv -1 \bmod p \tag{3}$$

c) Ecrire une procédure qui prend en argument un nombre premier p et qui calcule d_p. (On pourra supposer p inférieur à 10^4.)

Exemple numérique : calculer d_{7001} (on admettra que 7001 est premier).

4.– a) Démontrer que si un entier naturel irréductible p divise $a + ib$, alors p divise a et b.

b) Démontrer que si un élément z, irréductible mais n'appartenant pas à $\mathbb{N}$, divise un entier naturel n, alors $|z|^2$ divise n.

5.– a) Démontrer qu'un entier naturel p est irréductible si et seulement si p est un nombre premier congru à 3 modulo 4.

b) Soit z un élément de $P \setminus \mathbb{N}$. Démontrer que les conditions suivantes sont équivalentes :
- (i) z est irréductible,
- (ii) $|z|^2$ est un nombre premier,
- (iii) $|z|^2$ est un nombre premier non congru à 3 modulo 4.

c) Ecrire une procédure qui teste si un élément z de P est irréductible. (On pourra supposer que $|z| < 150$.)

Exemple numérique : $z = 6 + 11i$, $z = 17$.

6.– a) Soit p un nombre premier non congru à 3 modulo 4. Démontrer qu'il existe une unique paire d'entiers positifs $\{a, b\}$ telle que $a^2 + b^2 = p$.

b) En déduire la décomposition de p donnée par (2).

c) Ecrire une procédure qui prend en argument un entier naturel $n < 150$ et qui calcule la décomposition de n donnée par (2).

Exemple numérique : $n = 100$.

7.– Soit $z = u + iv$ un élément de $K - \{0\}$ et soit p un diviseur premier de $|z|^2$ non congru à 3 modulo 4. D'après la question précédente, il existe une unique paire d'entiers positifs $\{a, b\}$ tels que $a^2 + b^2 = p$.

a) Démontrer que si $va - ub$ est un multiple de p, alors $a + ib$ divise z, et que si $va - ub$ n'est pas un multiple de p, alors $b + ia$ divise z.

b) Ecrire une procédure qui prend en argument un élément z de P et qui calcule la décomposition de z donnée par (2) (On pourra supposer que $|z| < 150$).

Exemple numérique : Décomposer les nombres $56 + 98i$ et $19 + 61i$.

11.1.2 Solution : entiers de Gauss

On appelle *entier de Gauss* un nombre complexe de la forme $a + ib$ avec $a, b \in \mathbb{Z}$. Notre propos est d'établir que l'anneau K des entiers de Gauss est principal et de fournir un algorithme de décomposition des entiers de Gauss en produit d'éléments irréductibles.

On dit qu'un anneau intègre K est *euclidien* s'il existe un ensemble bien ordonné $(E, \leq)$ (c'est-à-dire un ensemble E muni d'une relation d'ordre total $\leq$ telle que toute partie non vide admet un plus petit élément) et une application $\varphi : K \to E$ telle que, pour tout $v \in K \setminus \{0\}$ et pour tout $u \in K$, il existe $q, r \in K$ tels que $u = vq + r$ et $\varphi(r) < \varphi(v)$. En pratique, on prend le plus souvent $E = \mathbb{N} \cup \{-\infty\}$, muni de l'ordre naturel sur les entiers.

PROPOSITION 11.1.1. *Tout anneau euclidien est principal.*

Notons que la réciproque est fausse : on peut prouver que l'anneau $\mathbb{Z}[x]$, où $x = \frac{1+i\sqrt{19}}{2}$, est principal mais pas euclidien.

Preuve. Soit K un anneau euclidien. Montrons d'abord que, quel que soit $x \in K \setminus \{0\}$, $\varphi(0) < \varphi(x)$. En effet, soit $v \in K$ tel que $\varphi(v)$ soit minimal. Si $v \neq 0$, on a $0 = vq + r$ avec $\varphi(r) < \varphi(v)$, d'où une contradiction.

Pour démontrer que K est principal, soit I un idéal non nul de K et soit v un élément non nul de I tel que $\varphi(v)$ soit minimal dans $\varphi(K \setminus \{0\})$. Pour tout $u \in I$, il existe $q, r \in K$ tels que $u = vq + r$ et $\varphi(r) < \varphi(v)$. Or $r = u - vq \in I$ et donc $r = 0$ d'après le choix de v. Donc I est l'idéal engendré par v et K est principal. ∎

En particulier, on a la proposition suivante.

PROPOSITION 11.1.2. *L'anneau des entiers de Gauss est principal.*

Preuve. D'après la proposition 11.1.1, il suffit de prouver que K est euclidien. Soient $u \in K$ et $v \in K \setminus \{0\}$. Posons $u/v = w = x + iy$ et prenons des entiers m et n tels que $|x - m| \leq \frac{1}{2}$ et $|y - n| \leq \frac{1}{2}$ (ce qui est toujours possible). Si on pose $q = m + in$ et $r = u - vq$, il vient $u = vq + r$ et $|w - q|^2 \leq 1/2$, d'où

$$|r| = |u - vq| = |v||w - q| \leq \frac{1}{\sqrt{2}}|v| < |v|$$

Donc en prenant $\varphi(v) = |v|$, on en déduit que K est euclidien. ∎

Notons P l'ensemble des entiers de Gauss de la forme $a + ib$ avec $a > 0$ et $b \geq 0$. Soit $z = a + ib$ un élément non nul de K. Alors l'un au moins des éléments $z = a + ib$, $iz = -b + ia$, $-z = -a - ib$, $-iz = b - ia$ est élément de P. En fait, tout élément non nul z de K s'écrit de façon unique sous la forme $z = uz'$ avec $z' \in P$ et u inversible (c'est-à-dire égal à 1, i, -1 ou $-i$). Ceci permet de raisonner uniquement dans P pour les questions de divisibilité (à un élément inversible près). En particulier, on appellera pgcd de deux éléments non nuls z_1 et z_2 de K l'unique élément d de P tel que d divise z_1 et z_2, et tel que tout diviseur commun à z_1 et z_2 divise d. De même, on dira qu'un élément z de K est *irréductible* s'il appartient à $P \setminus \{1\}$ et si ses seuls diviseurs sont de la forme u ou uz avec u inversible. On peut alors énoncer le lemme de Gauss sous la forme suivante.

PROPOSITION 11.1.3. *Si z est irréductible et divise le produit $z_1 z_2$, alors z divise z_1 ou z divise z_2.*

Preuve. L'idéal engendré par z et z_1 est principal et est engendré par le pgcd d de z et de z_1. Comme z est irréductible, on a $d = z$ ou $d = 1$. Dans le premier cas, z divise z_1 et, dans le second cas, il existe $a, b \in K$ tels que $az + bz_1 = 1$, d'où, en multipliant par z_2, $azz_2 + bz_1z_2 = z_2$, et donc z divise z_2. ∎

Notons I l'ensemble des éléments irréductibles de K. Tout élément z de $K - \{0\}$ admet une décomposition unique de la forme

$$z = i^{c(z)} \prod_{x \in I} x^{c_x(z)}$$

avec $c(z) \in \{0, 1, 2, 3\}, c_x(z) \in \mathbb{N}$ et où l'ensemble des $x \in I$ tels que $c_x(z) \neq 0$ est fini. L'existence d'une telle décomposition est immédiate en raisonnant par récurrence sur $|z|$. Pour l'unicité, on raisonne à nouveau par récurrence sur $|z|$ en utilisant le lemme de Gauss. Par exemple, on a $2 = i^3(1 + i)^2$.

Reste à déterminer les éléments irréductibles. Pour cela quelques résultats d'arithmétique nous serons utiles. Commençons par le théorème de Wilson.

Proposition 11.1.4. *Un nombre entier p est premier si et seulement si $(p - 1)! \equiv -1 \bmod p$.*

Preuve. Si p n'est pas premier, il possède un diviseur non trivial, et donc $(p - 1)! \equiv 0 \bmod p$. Démontrons maintenant que si p est premier, alors $(p - 1)! \equiv -1 \bmod p$. Le cas $p = 2$ est trivial. Pour p impair, on peut démontrer le résultat de diverses manières. En voici deux. On pose $G = (\mathbb{Z}/p\mathbb{Z})^*$.

a) G est un groupe d'ordre $p - 1$, et donc tout élément g de G vérifie $g^{p-1} = 1$. Tous les éléments de G sont donc racines du polynôme $X^{p-1} - 1$. On a donc dans $\mathbb{Z}/p\mathbb{Z}[X]$:

$$X^{p-1} - 1 = (X - 1)(X - 2) \cdots (X - (p - 1))$$

En substituant 0 à X, on obtient

$$(-1)^{(p-1)}(p - 1)! \equiv -1 \bmod p$$

et puisque p est impair, $(-1)^{(p-1)} = 1$.

b) Mis à part 1 et $p - 1$, les éléments de G sont distincts de leurs inverses (puisque l'équation $x^2 = 1$ ne possède que 2 solutions dans $\mathbb{Z}/p\mathbb{Z}$). Dans le produit de tous les éléments de G, on peut donc regrouper chaque élément et son inverse : il reste donc le produit $1(p - 1)$ qui est congru à -1 modulo p. ∎

Proposition 11.1.5. *Pour tout nombre premier p tel que $p \not\equiv 3 \bmod 4$, il existe un unique entier naturel $d_p \leq p/2$ tel que $d_p^2 \equiv -1 \bmod p$.*

Preuve. Si $p = 2$, on peut prendre $d_p = 1$, et c'est évidemment la seule solution. Sinon, puisque $p \not\equiv 3 \bmod 4$, $\frac{p-1}{2}$ est pair, et on a

$$(p - 1)! \equiv 1 \cdot 2 \cdot \ \cdots \ \frac{p - 1}{2}(-\frac{p - 1}{2}) \cdots (-2)(-1) \equiv \left\{(\frac{p - 1}{2})!\right\}^2 \mod p$$

D'après le théorème de Wilson, le nombre $x = (\frac{p-1}{2})!$ vérifie $x^2 \equiv -1 \bmod p$. Soit s le reste de la division de x par p et soit $d_p = \min(s, p-s)$. On a $d_p \leq p/2$ par construction et $d_p^2 \equiv s^2 \equiv x^2 \equiv -1 \bmod p$.

Pour prouver l'unicité, remarquons que l'autre solution de l'équation $x^2 + 1 = 0$ (dans $\mathbb{Z}/p\mathbb{Z}$) est $d'_p = p - d_p$. Comme p est impair, on a $d'_p > p/2$. ■

Les deux propositions qui suivent donnent des propriétés élémentaires des entiers de Gauss irréductibles.

Proposition 11.1.6. *Si un entier naturel irréductible p divise $a + ib$, alors p divise a et b.*

Preuve. Puisque p irréductible, p est évidemment premier. Si p divise $z = a + ib$, alors $\bar{p} = p$ divise $\bar{z} = a - ib$. Donc p divise $z + \bar{z} = 2a$ et $-i(z - \bar{z}) = 2b$. Mais puisque 2 est réductible, p est impair, et donc p divise a et b. ■

Proposition 11.1.7. *Si un élément z de P, irréductible mais n'appartenant pas à $\mathbb{N}$, divise un entier naturel n, alors $|z|^2$ divise n.*

Preuve. Si z divise n, alors $\bar{z}$ divise $\bar{n} = n$. Puisque z et $\bar{z}$ sont irréductibles et distincts, $z\bar{z}$ divise n. ■

Nous pouvons à présent caractériser les entiers naturels irréductibles.

Proposition 11.1.8. *Un entier naturel p est irréductible si et seulement si p est un nombre premier congru à 3 modulo 4. De plus, tout nombre premier réductible est somme de deux carrés.*

Preuve. Supposons p irréductible. Alors p est évidemment premier. Si $p \not\equiv 3 \bmod 4$, il existe, d'après la proposition 11.1.5, un entier $d \leq p/2$ tel que $d^2 + 1 \equiv 0 \bmod p$. Alors p divise $(d-i)(d+i)$ et puisque p est irréductible, p divise $d - i$ ou p divise $d + i$. D'après la proposition 11.1.6, on en déduit p divise 1, ce qui est impossible.

Réciproquement, soit p un entier naturel premier mais réductible et soit z un diviseur irréductible de p. Si z est entier, alors z divise p dans $\mathbb{N}$ d'après la proposition 11.1.6, et comme p est premier, on a nécessairement $z = p$, ce qui contredit le fait que p est réductible. Si z n'est pas entier, alors $|z|^2$ divise p d'après la proposition 11.1.7. Comme p est premier, on a $p = |z|^2$ et donc p est somme de deux carrés. Maintenant, un carré est congru à 0 ou à 1 modulo 4, et donc une somme de deux carrés ne peut être congrue à 3 mod 4. ■

Les entiers de Gauss irréductibles autres que les entiers naturels se caractérisent de la façon suivante.

Proposition 11.1.9. *Soit z un élément de P de partie imaginaire non nulle. Les conditions suivantes sont équivalentes :*

(i) *z est irréductible,*

(*ii*) $|z|^2$ *est un nombre premier,*
(*iii*) $|z|^2$ *est un nombre premier non congru à* 3 *modulo* 4*.*

Preuve. Comme on l'a vu plus haut, une somme de deux carrés ne peut être congrue à 3 modulo 4. Donc (ii) et (iii) sont équivalents.

(i) entraîne (ii). Soit p un diviseur premier de $|z|^2 = z\bar{z}$. Par unicité de la décomposition, p est divisé soit par z, soit par $\bar{z}$. Dans les deux cas, $|z|^2$ divise p d'après la proposition 11.1.7 et donc $p = |z|^2$.

(ii) entraîne (i). Si $z = z_1z_2$ est une décomposition non triviale de z, on a, en prenant les modules $|z|^2 = |z_1|^2|z_2|^2$, ce qui montre que $|z|^2$ n'est pas premier. ∎

Pour décomposer un entier naturel n en produit d'entiers de Gauss irréductibles, on peut procéder comme suit. On commence par décomposer n en facteurs premiers, ce qui nous ramène à décomposer un nombre premier p. Si $p \equiv 3 \bmod 4$, la proposition 11.1.8 montre que p est irréductible. Si par contre $p \not\equiv 3 \bmod 4$, p se décompose, comme le montre le résultat suivant.

Proposition 11.1.10. *Pour tout nombre premier p non congru à* 3 *modulo* 4*, il existe une unique paire d'entiers positifs $\{a, b\}$ telle que $a^2 + b^2 = p$. La décomposition de p est alors donnée par la formule $p = i^3(a + ib)(b + ia)$.*

Preuve. Soit p un nombre premier p non congru à 3 modulo 4. D'après la proposition 11.1.8, p est somme de deux carrés. Réciproquement, si $p = a^2 + b^2$, où a et b sont des entiers positifs, on a $p = i^3(a+ib)(b+ai)$ et les éléments $a+ib$ et $b+ia$ sont irréductibles d'après la proposition 11.1.9. Donc $p = i^3(a + ib)(b + ai)$ est l'unique décomposition de p, ce qui détermine la paire $\{a, b\}$. ∎

On retrouve en particulier la décomposition de 2 donnée plus haut : $2 = i^3(1 + i)^2$.

Il reste à donner un algorithme pour obtenir la décomposition d'un entier de Gauss $z = u + iv$ non nul quelconque. Posons $d = \mathrm{pgcd}(u, v)$. Puisque $z = d((u/d) + i(v/d))$, on peut commencer par décomposer d en produit d'entiers irréductibles à l'aide de l'algorithme décrit ci-dessus. Autrement dit, quitte à remplacer u et v respectivement par u/d et v/d, on se ramène au cas où u et v sont premiers entre eux, ce qui entraîne, d'après la proposition 11.1.6, que z n'a aucun diviseur irréductible entier.

Soit x est un diviseur irréductible de z. Puisque x n'est pas un entier naturel, la proposition 11.1.9 montre que $|x|^2$ est un nombre premier non congru à 3 modulo 4 qui divise $|z|^2$. Réciproquement, soit p un facteur premier de $|z|^2$. D'après la proposition 11.1.10, p se décompose en $p = i^3(a + ib)(b + ia)$, avec $a, b > 0$. Il en résulte que $a + ib$ est diviseur de $|z|^2 = z\bar{z}$ et, d'après le lemme de Gauss, $a + ib$ divise soit z, soit $\bar{z}$. Par conséquent ou bien $a + ib$ divise z, ou bien $b + ia = i(a - ib)$ divise z. Il reste à déterminer le bon facteur.

Proposition 11.1.11. *Si $va - ub$ est un multiple de p, alors $a+ib$ divise z et, si $va - ub$ n'est pas un multiple de p, alors $b + ia$ divise z.*

Preuve. Si $p = 2$, on a $a = b = 1$ et le résultat est clair. Sinon, on a nécessairement $a \neq b$ sinon $p = a^2 + b^2$ ne serait pas premier. Montrons d'abord que si $a + ib$ divise z, alors p divise $va - ub$. Posons en effet $z = u + iv = (a + ib)(c + id)$. Alors $c + id = \frac{a-ib}{p}(u + iv)$ et donc $d = \frac{va-ub}{p}$, ce qui montre que p divise $va - ub$.

Réciproquement, si p divise $va - ub$ et si $a + ib$ ne divise pas z, alors $b + ia$ divise z et, par conséquent, p divise $vb - ua$ d'après ce qui précède. On en déduit que p divise d'une part $(va - ub) + (vb - ua) = (v - u)(a + b)$ et d'autre part $(va - ub) - (vb - ua) = (v + u)(a - b)$. Or p ne peut diviser $a + b$, puisque $a + b < a^2 + b^2 = p$ (le cas $a + b = a^2 + b^2$ donnant nécessairement $p = 2$, ce qui est exclu). Donc p divise $v - u$ et de même p divise $v + u$. On en déduit que p divise $2u$ et $2v$ et donc aussi u et v, puisque p est impair. Par conséquent p divise z et, comme $a + ib$ divise p, on arrive à une contradiction. ∎

En résumé, pour déterminer la décomposition de $z = u + iv$, on calcule donc la décomposition de $d = \mathrm{pgcd}(u, v)$ en facteurs premiers et on décompose éventuellement les facteurs premiers obtenus à l'aide de la proposition 11.1.10. On décompose ensuite $|z/d|^2$ en facteurs premiers (les facteurs premiers qui apparaissent dans cette décomposition étant nécessairement non congrus à 3 modulo 4, il est inutile de tester les nombres premiers de la forme $4k + 3$). Chaque facteur premier p de $|z/d|^2$ fournit un diviseur irréductible de z. Pour obtenir ce diviseur, on recherche d'abord la décomposition (unique) de p sous la forme $p = a^2 + b^2$, et on choisit ensuite entre $a + ib$ ou $b + ia$ à l'aide de la proposition 11.1.11.

11.1.3 Programme : entiers de Gauss

La bibliothèque de manipulation des nombres complexes doit être modifiée. Le type `complexe` est remplacé par un type `EntierGauss` :

```
TYPE
   EntierGauss = ARRAY[0..1] OF integer;
```

Les fonctions de manipulation classiques ne présentent pas de difficulté.

```
FUNCTION Re (z: EntierGauss): integer;                        ℜ(z)
   BEGIN
      Re := z[0]
   END; { de "Re" }
FUNCTION Im (z: EntierGauss): integer;                        ℑ(z)
   BEGIN
      Im := z[1]
   END; { de "Im" }
FUNCTION ModuleAuCarre (z: EntierGauss): integer;             |z|²
   BEGIN
      ModuleAuCarre := Sqr(Re(z)) + Sqr(Im(z))
   END; { de "ModuleAuCarre" }
FUNCTION Module (z: EntierGauss): real;                       |z|
```

```
      BEGIN
         Module := Sqrt(ModuleAuCarre(z))
      END; { de "Module"}
   PROCEDURE EGaussParEGauss (u, v: EntierGauss; VAR uv: EntierGauss);
      VAR
         a, b: integer;
      BEGIN
         a := Re(u) * Re(v) - Im(u) * Im(v);
         b := Re(u) * Im(v) + Im(u) * Re(v);
         CartesienEnEntierGauss(a, b, uv);
      END; { de "EGaussParEGauss" }
```

Les entrées-sorties sont réalisées par trois procédures :

```
   PROCEDURE CartesienEnEntierGauss (a, b: integer; VAR z: EntierGauss);
      BEGIN
         z[0] := a;
         z[1] := b
      END; { de "CartesienEnEntierGauss" }
   PROCEDURE EntrerEntierGauss (VAR z: EntierGauss; titre: texte);
      VAR
         a, b: integer;
      BEGIN
         writeln;
         writeln(titre);
         write('Partie reelle : ');
         readln(a);
         write('Partie imaginaire : ');
         readln(b);
         CartesienEnEntierGauss(a, b, z)
      END; { de "EntrerEntierGauss" }
   PROCEDURE EcrireEntierDeGauss (z: EntierGauss; titre: texte);
      VAR
         a, b: integer;
      BEGIN
         write(titre);
         a := Re(z);
         b := Im(z);
         IF b = 0 THEN                                    b = 0
            write(a : 1)
         ELSE BEGIN                                       b ≠ 0
            IF a <> 0 THEN write(a : 1);
            IF b < 0 THEN write(' - ')                    signe
            ELSE IF a <> 0 THEN write(' + ');
            IF abs(b) <> 1 THEN write(abs(b) : 1);
            write('i')
         END
      END; { de "EcrireEntierDeGauss" }
```

Contrairement à ce qui se passe dans $\mathbb{Z}$, on peut réaliser une division euclidienne de diverses façons. Voici une possibilité, utilisant la fonction `round`, qui, d'après Jensen et Wirth (Pascal, manuel de l'utilisateur) est définie ainsi :

$$round(s) = \begin{cases} +k & \text{s'il existe un entier } k > 0 \text{ tel que } k - 0{,}5 \le s < k + 0{,}5 \\ 0 & \text{si } -0{,}5 < s < 0{,}5 \\ -k & \text{s'il existe un entier } k > 0 \text{ tel que } -k - 0{,}5 < s \le -k + 0{,}5 \end{cases}$$

```
PROCEDURE DivisionEuclidienne (x, y: EntierGauss; VAR q, r: EntierGauss);
   Pour y ≠ 0, détermine q et r tels que x = qy + r avec |r| < |y|.
   BEGIN
      q[0] := round((Re(x) * Re(y) + Im(x) * Im(y)) / ModuleAuCarre(y));
      q[1] := round((Im(x) * Re(y) - Re(x) * Im(y)) / ModuleAuCarre(y));
      r[0] := Re(x) - (Re(y) * q[0] - Im(y) * q[1]);
      r[1] := Im(x) - (Im(y) * q[0] + Re(y) * q[1]);
   END; { de "DivisionEuclidienne" }
```

Notre programme de décomposition des entiers de Gauss en facteurs irréductibles fait appel à une procédure de décomposition des entiers naturels en facteurs premiers. Pour cela, le plus simple est d'engendrer une table des nombres premiers par un crible élémentaire. On peut ensuite, à l'aide de cette table, réaliser une procédure qui fournit le plus petit diviseur premier d'un entier. Bien entendu, cette méthode ne peut s'appliquer efficacement qu'à de très petites valeurs. Au-delà, si on vise l'efficacité, il faudrait avoir recours à des tests de primalité et de factorisation plus sophistiqués.

```
CONST
   NombreMax = 160;          Module maximal des entiers de Gauss à décomposer.
   ListePremiersMax = 35;    Nombre maximal de nombres premiers dans la table.
TYPE
   ListeNombresPremiers = ARRAY[1..ListePremiersMax] OF integer;
VAR
   Premiers: array[2..NombreMax] of boolean;
   NombresPremiers: ListeNombresPremiers;
PROCEDURE Crible;
   Engendre une table de nombres premiers par la méthode du crible.
   VAR
      p, i, j: integer;
      Premiers: ARRAY[2..NombreMax] OF boolean;
   BEGIN
      FOR i := 2 TO NombreMax DO
         Premiers[i] := True;
      p := 1;
      FOR i := 1 TO ListePremiersMax DO BEGIN
         REPEAT
            p := p + 1
         UNTIL Premiers[p];
         j := p;
```

```
            REPEAT
               Premiers[j] := false;
               j := j + p
            UNTIL j > NombreMax;
            NombresPremiers[i] := p;
         END;
      END; { de "Crible" }
```

Voici une façon d'implémenter la recherche du plus petit diviseur premier d'un entier :

```
   FUNCTION FacteurPremier (n: integer): integer;
      Donne le plus petit diviseur premier de n.
      VAR
         racine, i, p: integer;
         EstCompose: boolean;
      BEGIN
         FacteurPremier := n;
         EstCompose := false;
         racine := round(sqrt(n));
         i := 1;
         REPEAT
            p := NombresPremiers[i];
            EstCompose := n MOD p = 0;
            IF EstCompose THEN
               FacteurPremier := p
            ELSE
               i := i + 1;
         UNTIL (p > racine) OR EstCompose;
      END; { de "FacteurPremier" }
```

On peut facilement convertir la procédure précédente en test de primalité :

```
   FUNCTION EstPremier (n: integer): boolean;
      BEGIN
         EstPremier := FacteurPremier(n) = n
      END; { de "EstPremier" }
```

La recherche de la décomposition d'un nombre premier $p \not\equiv 3 \bmod 4$ comme somme de deux carrés peut s'effectuer de façon exhaustive de la façon suivante :

```
   PROCEDURE DeuxCarres (p: integer; VAR a, b: integer);
      Fournit la décomposition d'un nombre premier p ≢ 3 mod 4 comme somme de deux
      carrés : p = a² + b².
      BEGIN
         a := 0;
         REPEAT
            a := a + 1
         UNTIL (sqr(a) + sqr(trunc(sqrt(p - sqr(a)))) = p);
         b := trunc(sqrt(p - sqr(a)));
      END; { de "DeuxCarres" }
```

Sous les mêmes hypothèses pour p, la recherche de l'unique entier $d_p \leq (p-1)/2$ tel que $d_p^2 \equiv -1 \bmod p$ s'effectue de façon exhaustive. La seule difficulté est d'éviter le dépassement de capacité. Pour cela, on réduit modulo p à chaque étape et le passage de $d_p^2 + 1$ à $(d_p + 1)^2 + 1$ s'effectue par addition de $2d_p + 1$.

```
FUNCTION RacineDeMoinsUn (p: integer): integer;
   Si p est premier non congru à 3 mod 4, cette fonction retourne l'unique entier d_p ≤
   (p − 1)/2 tel que d_p^2 ≡ −1 mod p.
   VAR
      d, d1: integer;
   BEGIN
      d := 0;
      d1 := 1;                            d_1 = d_p^2 + 1.
      REPEAT
         d1 := (d1 + 2 * d + 1) MOD p;    Pour éviter le dépassement de capacité.
         d := d + 1;
      UNTIL d1 = 0;
      RacineDeMoinsUn := d
   END; { de "RacineDeMoinsUn" }
```

Pour savoir si un entier de Gauss z est irréductible, deux cas sont possibles. Si z est un entier naturel, on teste si z est premier et congru à 3 modulo 4. Sinon, on teste si $|z|^2$ est premier.

```
FUNCTION EstIrreductible (z: EntierGauss): boolean;    Teste si z est irréductible.
   VAR
      a, b: integer;
   BEGIN
      a := Re(z);
      b := Im(z);
      IF b = 0 THEN
         EstIrreductible := EstPremier(a) AND ((a MOD 4) = 3)
      ELSE
         EstIrreductible := EstPremier(ModuleAuCarre(z));
   END; { de "EstIrreductible" }
```

Pour décomposer un entier de Gauss z en produit d'entiers de Gauss irréductibles, on gère une liste de ses facteurs irréductibles et une variable `cz`, dont la valeur représente l'exposant de i dans la décomposition de z. On mémorise également le nombre de facteurs de la décomposition.

```
CONST
   ListeMax = 100;          Nombre maximal de facteurs irréductibles dans les listes.
TYPE
   ListeEntiersGauss = array[1..ListeMax] of EntierGauss;
PROCEDURE FactoriserEntierGauss (z: EntierGauss;
   VAR Liste: ListeEntiersGauss; VAR cz, NbFacteurs: integer);
   VAR
```

```
    a, b, d, n, p: integer;
    x, w, r: EntierGauss;
```

La procédure de décomposition d'un entier de Gauss en produit de facteurs irréductibles fait appel à plusieurs sous-procédures. Une première procédure permet d'ajouter un facteur à la liste des facteurs déjà trouvés et de mettre à jour l'exposant `cz`.

```
PROCEDURE AjouterFacteur (cx: integer; x: EntierGauss);
  BEGIN
    cz := (cz + cx) MOD 4;
    NbFacteurs := NbFacteurs + 1;
    Liste[NbFacteurs] := x;
  END; { de "AjouterFacteur"}
```

Une autre procédure donne la factorisation complète d'un nombre premier. Rappelons que si $p \equiv 3 \bmod 4$, p est irréductible. Si $p \equiv 1 \bmod 4$, on a $p = i^3(a+ib)(b+ia)$, où $p = a^2 + b^2$ est l'unique décomposition de p comme somme de deux carrés.

```
PROCEDURE FactoriserPremier (p: integer);
  VAR
    a, b: integer;
    x: EntierGauss;
  BEGIN
    IF (p MOD 4) = 3 THEN BEGIN
      CartesienEnEntierGauss(p, 0, x);
      AjouterFacteur(0, x)
    END
    ELSE BEGIN
      DeuxCarres(p, a, b);
      CartesienEnEntierGauss(a, b, x);
      AjouterFacteur(0, x);
      CartesienEnEntierGauss(b, a, x);
      AjouterFacteur(3, x)
    END
  END; { de "FactoriserPremier" }
```

Enfin, une procédure permet de décomposer un entier naturel n en produit d'entiers de Gauss irréductibles.

```
PROCEDURE FactoriserNaturel (n: integer);
    Décomposition d'un entier naturel en produit d'entiers de Gauss irréductibles.
  VAR
    p: integer;
  BEGIN
    WHILE n <> 1 DO BEGIN
      p := FacteurPremier(n);
      FactoriserPremier(p);
      n := n DIV p
    END
```

```
        END; { de "FactoriserNaturel" }
```

Finalement la procédure de factorisation d'un entier de Gauss $z = u + iv$ peut être réalisée ainsi. On commence par décomposer le pgcd de u et de v, ce qui permet de se ramener au cas où ce pgcd est égal à 1. On recherche alors les facteurs premiers de $|z|^2$, qui sont nécessairement non congrus à 3 modulo 4. Si p est un tel facteur et si $p = a^2 + b^2$ est l'unique décomposition de p comme somme de deux carrés, l'un des nombres $a + ib$ ou $a - ib = -i(b + ia)$ est diviseur de z, et on applique la proposition 11.1.11 pour déterminer le bon facteur.

```
      BEGIN { de "FactoriserEntierGauss" }
         NbFacteurs := 0;
         cz := 0;
         d := PGCD(Re(z), Im(z));
         FactoriserNaturel(d);
         CartesienEnEntierGauss(Re(z) DIV d, Im(z) DIV d, w);
         On se ramène au cas z = u + iv avec pgcd(u, v) = 1.
         n := ModuleAuCarre(w);
         WHILE n <> 1 DO BEGIN
            p := FacteurPremier(n);
            DeuxCarres(p, a, b);
            Dans ce cas, a + ib ou a − ib = −i(b + ia) est diviseur de w.
            IF ((Im(w) * a - Re(w) * b) MOD p) = 0 THEN
               CartesienEnEntierGauss(a, b, x)
            ELSE
               CartesienEnEntierGauss(b, a, x);
            AjouterFacteur(0, x);
            DivisionEuclidienne(w, x, w, r);
            n := ModuleAuCarre(w);
         END;
         IF Re(w) = 0 THEN                      Reste un nombre w de module 1.
            cz := (cz + 2 - Im(w)) MOD 4
            Si w = i, cz := cz + 1, et si w = −i, cz := cz + 3.
         ELSE
            cz := (cz + 1 - Re(w)) MOD 4;
            Si w = 1, cz ne change pas, et si z = −1, cz := cz + 2.
     END; { de "FactoriserEntierGauss" }
```

Pour gérer les diverses procédures, on peut utiliser le menu suivant :

```
   PROCEDURE Menu;
      VAR
         u, v, q, r, z: EntierGauss;
         Liste: ListeEntiersGauss;
         p, n, NbFacteurs, cz, choix: integer;
      BEGIN
         writeln('Menu');
         writeln('(1) Division euclidienne');
         writeln('(2) Calcul d'' une racine de - 1 modulo p ');
```

```
      writeln('(3) Teste d''irreductibilité');
      writeln('(4) Décomposition d''un entier de Gauss');
      write('Votre choix : ');
      readln(choix);
      CASE choix OF
         1:
            BEGIN
            EntrerEntierGauss(u, 'Donner u = a + ib :');
            EntrerEntierGauss(v, 'Donner v = a + ib :');
            DivisionEuclidienne(u, v, q, r);
            EcrireEntierDeGauss(u, '');
            EcrireEntierDeGauss(q, ' = (');
            EcrireEntierDeGauss(v, ')(');
            EcrireEntierDeGauss(r, ') + (');
            writeln(')');
         END;
         2:
            BEGIN
            write('Donner p (nombre premier non congru à 3 mod 4) : ');
            readln(p);
            writeln('Racine de -1 mod p = ', RacineDeMoinsUn(p) : 1)
         END;
         3:
            BEGIN
            EntrerEntierGauss(z, 'Donner z = a + ib :');
            IF EstIrreductible(z) THEN
               writeln('z est irréductible')
            ELSE
               writeln('z est réductible')
         END;
         4:
            BEGIN
            EntrerEntierGauss(z, 'Donner z = a + ib  (|z| < 150) : ');
            IF EstIrreductible(z) THEN
               writeln('z est irréductible');
            Factorisation(z, Liste, cz, NbFacteurs);
            write('i^', cz : 1);
            EcrireListe(Liste, NbFacteurs);
            writeln
         END;
         OTHERWISE
      END
   END;
```

Voici quelques résultats numériques.

Pour $p = 7001$, on trouve $d_p = 1198$. L'entier de Gauss $6 + 11i$ est irréductible, car $36 + 121 = 157$ est premier. En revanche, 17 est réductible, car $17 \not\equiv 3 \bmod 4$. On a en fait $17 = i^3(1 + 4i)(4 + i)$. On trouve également $100 = (1 + i)^4(1 + 2i)^2(2 + i)^2$,

$56+98i = i^3(7)(1+i)^2(2+i)(3+2i)$ et $19+61i = i^3(1+i)(2+3i)(6+11i)$.

Pour finir, voici les décompositions des 100 premiers entiers naturels :

$2 = i^3(1+i)^2$	$3 = (3)$
$4 = i^2(1+i)^4$	$5 = i^3(1+2i)(2+i)$
$6 = i^3(1+i)^2(3)$	$7 = (7)$
$8 = i^1(1+i)^6$	$9 = (3^2)$
$10 = i^2(1+i)^2(1+2i)(2+i)$	$11 = (11)$
$12 = i^2(1+i)^4(3)$	$13 = i^3(2+3i)(3+2i)$
$14 = i^3(1+i)^2(7)$	$15 = i^3(3)(1+2i)(2+i)$
$16 = (1+i)^8$	$17 = i^3(1+4i)(4+i)$
$18 = i^3(1+i)^2(3^2)$	$19 = (19)$
$20 = i^1(1+i)^4(1+2i)(2+i)$	$21 = (3)(7)$
$22 = i^3(1+i)^2(11)$	$23 = (23)$
$24 = i^1(1+i)^6(3)$	$25 = i^2(1+2i)^2(2+i)^2$
$26 = i^2(1+i)^2(2+3i)(3+2i)$	$27 = (3^3)$
$28 = i^2(1+i)^4(7)$	$29 = i^3(2+5i)(5+2i)$
$30 = i^2(1+i)^2(3)(1+2i)(2+i)$	$31 = (31)$
$32 = i^3(1+i)^{10}$	$33 = (3)(11)$
$34 = i^2(1+i)^2(1+4i)(4+i)$	$35 = i^3(1+2i)(2+i)(7)$
$36 = i^2(1+i)^4(3^2)$	$37 = i^3(1+6i)(6+i)$
$38 = i^3(1+i)^2(19)$	$39 = i^3(3)(2+3i)(3+2i)$
$40 = (1+i)^6(1+2i)(2+i)$	$41 = i^3(4+5i)(5+4i)$
$42 = i^3(1+i)^2(3)(7)$	$43 = (43)$
$44 = i^2(1+i)^4(11)$	$45 = i^3(3^2)(1+2i)(2+i)$
$46 = i^3(1+i)^2(23)$	$47 = (47)$
$48 = (1+i)^8(3)$	$49 = (7^2)$
$50 = i^1(1+i)^2(1+2i)^2(2+i)^2$	$51 = i^3(3)(1+4i)(4+i)$
$52 = i^1(1+i)^4(2+3i)(3+2i)$	$53 = i^3(2+7i)(7+2i)$
$54 = i^3(1+i)^2(3^3)$	$55 = i^3(1+2i)(2+i)(11)$
$56 = i^1(1+i)^6(7)$	$57 = (3)(19)$
$58 = i^2(1+i)^2(2+5i)(5+2i)$	$59 = (59)$
$60 = i^1(1+i)^4(3)(1+2i)(2+i)$	$61 = i^3(5+6i)(6+5i)$
$62 = i^3(1+i)^2(31)$	$63 = (3^2)(7)$
$64 = i^2(1+i)^{12}$	$65 = i^2(1+2i)(2+i)(2+3i)(3+2i)$
$66 = i^3(1+i)^2(3)(11)$	$67 = (67)$
$68 = i^1(1+i)^4(1+4i)(4+i)$	$69 = (3)(23)$
$70 = i^2(1+i)^2(1+2i)(2+i)(7)$	$71 = (71)$
$72 = i^1(1+i)^6(3^2)$	$73 = i^3(3+8i)(8+3i)$
$74 = i^2(1+i)^2(1+6i)(6+i)$	$75 = i^2(3)(1+2i)^2(2+i)^2$
$76 = i^2(1+i)^4(19)$	$77 = (7)(11)$
$78 = i^2(1+i)^2(3)(2+3i)(3+2i)$	$79 = (79)$
$80 = i^3(1+i)^8(1+2i)(2+i)$	$81 = (3^4)$
$82 = i^2(1+i)^2(4+5i)(5+4i)$	$83 = (83)$
$84 = i^2(1+i)^4(3)(7)$	$85 = i^2(1+2i)(2+i)(1+4i)(4+i)$
$86 = i^3(1+i)^2(43)$	$87 = i^3(3)(2+5i)(5+2i)$
$88 = i^1(1+i)^6(11)$	$89 = i^3(5+8i)(8+5i)$
$90 = i^2(1+i)^2(3^2)(1+2i)(2+i)$	$91 = i^3(7)(2+3i)(3+2i)$

$92 = i^2(1+i)^4(23)$
$93 = (3)(31)$
$94 = i^3(1+i)^2(47)$
$95 = i^3(1+2i)(2+i)(19)$
$96 = i^3(1+i)^{10}(3)$
$97 = i^3(4+9i)(9+4i)$
$98 = i^3(1+i)^2(7^2)$
$99 = (3^2)(11)$
$100 = (1+i)^4(1+2i)^2(2+i)^2$

11.2 Arithmétique modulaire

11.2.1 Enoncé : arithmétique modulaire

Soient $m_1, m_2, \ldots, m_r$ des entiers positifs deux à deux premiers entre eux et soit $m = m_1 m_2 \cdots m_r$. Le théorème des restes chinois établit l'isomorphisme entre $\mathbb{Z}/m\mathbb{Z}$ et $\mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_r\mathbb{Z}$. Pour tout entier $u \in \{0, \ldots, m-1\}$, le r-uplet $(u_1, \ldots, u_r)$ d'entiers avec $0 \leq u_i < m_i$ et $u \equiv u_i \bmod m_i$ pour $i = 1, \ldots, r$ est la *représentation modulaire* de u.

La *représentation en base* b de u est un vecteur U d'entiers dont chaque composante U_i est comprise entre 0 et $b-1$ et tel que $u = \sum_{i \geq 0} U_i b^i$.

Dans la rédaction des programmes, on pourra supposer que les m_i sont choisis de sorte que m_i^2 soit représentable par un entier simple précision de l'ordinateur.

1.– Démontrer que pour tout couple d'entiers i, j avec $0 \leq i < j \leq r$, il existe un entier $c_{i,j}$ avec $0 \leq c_{i,j} < m_j$ tel que

$$c_{i,j} m_i \equiv 1 \pmod{m_j}$$

et donner un algorithme pour calculer $c_{i,j}$ à partir de m_i et m_j. Ecrire une procédure qui affiche la table des $c_{i,j}$ pour les valeurs numériques données plus bas.

2.– Soit $(u_1, \ldots, u_r)$ la représentation modulaire de u. Pour calculer la représentation en base b de u, on définit $v_1, \ldots, v_r$ avec $0 \leq v_i < m_i$ par

$$\begin{aligned}
v_1 &= u_1 \\
v_2 &\equiv (u_2 - v_1)c_{1,2} \pmod{m_2} \\
v_3 &\equiv ((u_3 - v_1)c_{1,3} - v_2)c_{2,3} \pmod{m_3} \\
&\vdots \\
v_r &\equiv (\ldots((u_r - v_1)c_{1,r} - v_2)c_{2,r} - \ldots - v_{r-1})c_{r-1,r} \pmod{m_r}
\end{aligned}$$

Démontrer que

$$u = v_r m_{r-1} \cdots m_1 + \ldots + v_3 m_2 m_1 + v_2 m_1 + v_1$$

3.– Ecrire une procédure qui permet de passer de la représentation modulaire à la représentation en base b (avec $b = 10$ ou $b = 100$ par exemple). Ecrire une procédure qui permet de passer de la représentation en base b à la représentation modulaire.

4.– Ecrire des procédures qui prennent en argument deux entiers en représentation modulaire et qui fournissent leur somme, différence, produit modulo m en représentation modulaire.

5.– Les *nombres de Catalan* c_n sont définis par $c_0 = 1$ et $c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$ pour $n \geq 0$. Ecrire une procédure qui calcule une table des 20 premiers nombres de Catalan en utilisant les procédures de la question précédente et qui affiche les résultats en base b.

Pour les exemples numériques, on pourra prendre $r = 8$ et $m_1 = 99$, $m_2 = 97$, $m_3 = 95$, $m_4 = 91$, $m_5 = 89$, $m_6 = 83$, $m_7 = 79$, $m_8 = 73$. ($m_1 \cdots m_8 = 3536632852868115$)

6.– Démontrer que

$$c_n = \frac{1}{n+1}\binom{2n}{n}$$

(On note $\binom{k}{p} = \frac{k!}{p!(k-p)!}$). En déduire une estimation du plus grand nombre de Catalan que l'on peut calculer avec les valeurs des m_i données ci-dessus et afficher les nombres de Catalan jusqu'à cette valeur.

7.– Une *suite de Catalan* de longueur $2n+1$ est une suite $(a_0, a_1, \ldots, a_{2n})$ d'entiers positifs ou nuls tels que $a_0 = a_{2n} = 0$ et $|a_{i+1} - a_i| = 1$ pour $i = 0, \ldots, 2n-1$. Démontrer que le nombre de suites de Catalan de longueur $2n+1$ est c_n.

11.2.2 Solution : arithmétique modulaire

Soient $m_1, m_2, \ldots, m_r$ des entiers positifs deux à deux premiers entre eux et soit $m = m_1 m_2 \cdots m_r$. Le théorème des restes chinois établit l'isomorphisme entre $\mathbb{Z}/m\mathbb{Z}$ et $\mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_r\mathbb{Z}$:

Théorème 11.2.1 (des restes chinois). *Soient $m_1, m_2, \ldots, m_r$ des entiers positifs premiers entre eux deux à deux et soit $m = m_1 m_2 \cdots m_r$. Soient $u_1, \ldots, u_r$ des entiers. Alors il existe un entier u et un seul tel que $0 \leq u \leq m-1$ et*

$$u \equiv u_j \pmod{m_j} \qquad pour 1 \leq j \leq r$$

Preuve. Par le théorème de Bezout, il existe pour tout $i \neq j$, avec $1 \leq i, j \leq r$, un entier $c_{i,j}$ (unique si $0 < c_{i,j} < m_j$) tel que

$$c_{i,j} m_i \equiv 1 \pmod{m_j} \tag{2.1}$$

Posons pour $1 \leq j \leq r$

$$M_j = \prod_{\substack{1 \leq i \leq r \\ i \neq j}} m_i c_{i,j}$$

Alors $M_j \equiv 1 \pmod{m_j}$ et $M_j \equiv 0 \pmod{m_i}$ pour $i \neq j$, de sorte que le reste u de $u_1 M_1 + \cdots + u_r M_r$ modulo m vérifie les conclusions du théorème. L'unicité est immédiate. ∎

L'application
$$(u_1, \ldots, u_r) \mapsto u_1 M_1 + \cdots + u_r M_r$$
est un isomorphisme d'anneau parce que les M_j sont «orthogonaux» : pour $i \neq j$, on a $M_i M_j \equiv 0 \pmod{m}$ et par ailleurs $M_j M_j \equiv 1 \pmod{m}$.

Pour tout entier $u \in \{0, \ldots, m-1\}$, le r-uplet $(u_1, \ldots, u_r)$ d'entiers avec $0 \leq u_i < m_i$ et $u \equiv u_i \pmod{m_i}$ pour $1 \leq i \leq r$ est la *représentation modulaire* de u. La *représentation en base* b de u est un vecteur U d'entiers dont chaque composante U_i est comprise entre 0 et $b-1$ et tel que $u = \sum_{i \geq 0} U_i b^i$.

L'arithmétique dite modulaire concerne les opérations arithmétiques de l'anneau $\mathbb{Z}$. Ces opérations sont en fait effectuées modulo m, où m est un entier choisi suffisamment grand pour que le résultat modulo m soit égal au résultat dans $\mathbb{Z}$. Les opérations modulo m sont, à leur tour, effectuées dans la représentation modulaire, c'est-à-dire composante par composante. Les opérations arithmétiques sont très peu coûteuses ; en revanche, pour que le résultat soit lisible, il doit être converti en une représentation plus commode, comme la représentation décimale. La programmation des opérations arithmétiques modulaires est facile. Reste à considérer la conversion entre les deux représentations. On utilise pour cela la proposition qui suit.

Proposition 11.2.2. *Soit $(u_1, \ldots, u_r)$ la représentation modulaire de u, et soient $v_1, \ldots, v_r$ avec $0 \leq v_i < m_i$ définis par*

$$\begin{aligned}
v_1 &= u_1 \\
v_2 &\equiv (u_2 - v_1)c_{1,2} \pmod{m_2} \\
v_3 &\equiv ((u_3 - v_1)c_{1,3} - v_2)c_{2,3} \pmod{m_3} \\
&\vdots \\
v_r &\equiv (\ldots((u_r - v_1)c_{1,r} - v_2)c_{2,r} - \ldots - v_{r-1})c_{r-1,r} \pmod{m_r}
\end{aligned}$$

Alors
$$u = v_r m_{r-1} \cdots m_1 + \ldots + v_3 m_2 m_1 + v_2 m_1 + v_1$$

Preuve. D'après la définition de la représentation modulaire, il suffit de prouver que

$$u_i \equiv v_i m_{i-1} \cdots m_1 + \ldots + v_3 m_2 m_1 + v_2 m_1 + v_1 \pmod{m_i} \qquad 1 \leq i \leq r \tag{2.2}$$

Considérons

$$v_i \equiv (\ldots((u_i - v_1)c_{1,i} - v_2)c_{2,i} - \ldots - v_{i-1})c_{i-1,i} \pmod{m_i}$$

En multipliant par m_{i-1}, on obtient en utilisant (2.1)

$$v_i m_{i-1} + v_{i-1} \equiv (\ldots((u_i - v_1)c_{1,i} - v_2)c_{2,i} - \ldots - v_{i-2})c_{i-2,i} \pmod{m_i}$$

et plus généralement, pour $1 \leq k \leq i-1$,

$$\begin{aligned}
&v_i m_{i-1} \cdots m_{i-k} + \cdots + v_{i-k+1} m_{i-k} + v_{i-k} \equiv \\
&\qquad (\ldots((u_i - v_1)c_{1,i} - v_2)c_{2,i} - \ldots - v_{i-(k+1)})c_{i-(k+1),i} \pmod{m_i}
\end{aligned}$$

ce qui donne (2.2) pour $k = i - 1$. ■

L'expression donnée pour u dans cette proposition en permet l'évaluation par un schéma de Horner : l'opération répétée consiste à réaliser $s := sm_i + v_i$. Ceci peut être fait sans difficulté sur des entiers en multiprécision, si m_i et v_i sont des entiers simple précision.

Réciproquement, pour passer d'une représentation en base b à la représentation modulaire, il suffit d'évaluer $u = \sum_{i \geq 0} U_i b^i$ modulo m_j pour chaque j, encore une fois par le schéma de Horner.

Comme exemple de calcul, considérons les *nombres de Catalan*. Ils sont définis par

$$c_n = \frac{1}{n+1}\binom{2n}{n} \tag{2.3}$$

(On note $\binom{k}{p} = \frac{k!}{p!(k-p)!}$.) Les nombres de Catalan interviennent en combinatoire dans de très nombreuses situations. Nous les avons déjà rencontrés au chapitre 9, où nous avons prouvé qu'ils comptent les mots de Lukasiewicz. Voici un exemple similaire : appelons *suite de Catalan* de longueur $n + 1$ une suite $(a_0, a_1, \ldots, a_n)$ d'entiers positifs ou nuls tels que $a_0 = a_n = 0$ et $|a_{i+1} - a_i| = 1$ pour $i = 0, \ldots, n - 1$. En fait, l'entier n est nécessairement pair.

Proposition 11.2.3. *Le nombre de suites de Catalan de longueur* $2k + 1$ *est* c_k.

Preuve. Soit $u = (u_0, \ldots, u_n)$ une suite d'entiers avec $u_i \in 0, 1$. On pose

$$s(u, j) = \sum_{i=0}^{j} u_i \quad j = 0, \ldots, n$$

Avec la terminologie du chapitre 9, la suite u est un *mot de Lukasiewicz* si et seulement si $s(u, n) = -1$ et $s(u, j) \geq 0$ pour $0 \leq j \leq n - 1$. Si u est un mot de Lukasiewicz, alors la suite $(0, s(u, 0), \ldots, s(0, n - 1))$ est une suite de Catalan. Réciproquement, si $(a_0, a_1, \ldots, a_n)$ est une suite de Catalan, alors $(a_1 - a_0, a_2 - a_1, \ldots, a_n - a_{n-1}, -1)$ est un mot de Lukasiewicz. Il y a donc bijection entre les mots de Lukasiewicz de longueur n et les suites de Catalan de longueur n. La proposition est alors une conséquence du corollaire 9.2.7. ■

Bien entendu, on peut aussi prouver la proposition directement. Pour le calcul en arithmétique modulaire, la formule (2.3) n'est utilisable que si les dénominateurs sont inversibles pour tous les modules m_j, ce qui n'est pas le cas si l'on choisit par exemple $m_1 = 99$. Les calculs doivent donc être organisés différemment. Pour cela, on prouve que

$$c_n = \sum_{k=0}^{n-1} c_k c_{n-k-1} \tag{2.4}$$

On peut vérifier cette formule directement ou encore considérer la série

$$c(z) = \sum_{n=0}^{\infty} c_n z^n$$

Par la formule de Stirling ($n! \sim n^n e^{-n}\sqrt{2\pi n}$), on a

$$c_n \sim \frac{4^n}{(n+1)\sqrt{\pi n}}$$

donc la série converge pour $|z|$ suffisamment petit et on a

$$c(z) = \frac{1-\sqrt{1-4z}}{2z}$$

Par conséquent, $c(z)$ vérifie l'équation

$$c(z) = 1 + z(c(z))^2$$

d'où l'on déduit par identification $c_0 = 1$ et la formule (2.4). Cette expression est longue à évaluer mais se prête au calcul en arithmétique modulaire. Si l'on prend $r = 8$ et $m_1 = 99$, $m_2 = 97$, $m_3 = 95$, $m_4 = 91$, $m_5 = 89$, $m_6 = 83$, $m_7 = 79$, $m_8 = 73$, alors $m = m_1 \cdots m_8 = 3536632852868115$, et l'expression asymptotique des nombres de Catalan montre, avec un petit calcul, que l'on peut évaluer c_n pour $n \leq 29$.

11.2.3 Programme : arithmétique modulaire

Soit $m > 1$ un entier et soit a un entier premier à m. Le théorème de Bezout affirme l'existence d'entiers uniques x et y tels que $0 < x < m$, $0 < y < a$ et

$$ax + my = 1 \tag{2.5}$$

Il en résulte que $ax \equiv 1 \bmod m$. Le calcul de x peut être fait en recherchant les couples (x, y) vérifiant (2.5). Pour cela, considérons la suite des divisions euclidiennes successives réalisées pour le calcul du pgcd de a et de m : on pose $r_0 = a$, $r_1 = m$, et on définit q_k et r_{k+2}, pour $k \geq 0$, par

$$r_k = q_k r_{k+1} + r_{k+2} \text{ avec } r_{k+1} \neq 0 \text{ et } 0 \leq r_{k+2} < r_{k+1}$$

Soient alors $(x_0, x_1) = (1, 0)$, $(y_0, y_1) = (0, 1)$ et, pour $k \geq 0$,

$$\begin{aligned} x_{k+2} &= x_k - x_{k+1} q_k \\ y_{k+2} &= y_k - y_{k+1} q_k \end{aligned}$$

On vérifie alors que pour tout k tel que r_k est défini, on a

$$r_k = x_k a + y_k m$$

Notamment, si k est le plus petit entier tel que $r_{k+1} = 0$, alors $r_k = \text{pgcd}(a, m)$ et les entiers x_k,y_k sont des coefficients pour l'identité de Bezout. Dans le cas particulier où a et m sont premiers entre eux, on a

$$1 \equiv x_k a \pmod m$$

Ceci conduit à la procédure :

```
FUNCTION Inverse (x, m: integer): integer;
   Calcule l'inverse de x (mod m) par adaptation de l'algorithme d'Euclide. On suppose x
   et m premiers entre eux.
   VAR
      r0, r1, r2, x0, x1, x2, q: integer;
   BEGIN
      x0 := 1; x1 := 0; r0 := x; r1 := m;          Initialisation.
      WHILE r1 <> 0 DO BEGIN
         q := r0 DIV r1;
         r2 := r0 - q * r1; x2 := x0 - q * x1;     Calcul.
         x0 := x1; x1 := x2; r0 := r1; r1 := r2    Copie.
      END;
      x0 := x0 MOD m;
      IF x0 < 0 THEN x0 := x0 + m;                 Pour rétablir le modulo.
      Inverse := x0
   END; { de "Inverse" }
```

Les représentations modulaires sont des tableaux d'entiers. On définit donc :

```
CONST
   TailleMod = 8;
TYPE
   EntierMod = ARRAY[1..TailleMod] OF integer;
VAR
   m: EntierMod;
   c: ARRAY[1..TailleMod, 1..TailleMod] OF integer;
```

On déclare aussi deux tableaux spécifiques, l'un qui contient les modules m_j, l'autre qui contient les coefficients $c_{i,j}$ nécessaires à la conversion. Nous initialisons, pour fixer les idées,

```
PROCEDURE InitModules;
   BEGIN
      m[1] := 99;   m[2] := 97;   m[3] := 95;   m[4] := 91;
      m[5] := 89;   m[6] := 83;   m[7] := 79;   m[8] := 73
   END; { de "InitModules" }
```

et calculons la table des $c_{i,j}$ (il suffit de la calculer pour $1 \leq i < j \leq r$) par :

```
PROCEDURE InitInverses;
   VAR
      i, j: integer;
   BEGIN
      FOR i := 1 TO TailleMod DO
         FOR j := i + 1 TO TailleMod DO
            c[i, j] := Inverse(m[i], m[j])
   END; { de "InitInverses" }
```

On obtient :

```
La table des inverses :
 49 24 57  9 26  4 59
    48 76 78  6 22 70
       23 15  7  5 10
          45 52 33 69
             14  8 32
                20 22
                   61
```

Considérons d'abord les conversions. Des entiers en multiprécision, tels que présentés au chapitre suivant, on n'utilise que le type :

```
TYPE
        chiffre = integer;
        entier = ARRAY[-2..TailleMax] OF chiffre;
```

et les procédures :

```
PROCEDURE EntierPlusChiffre (u: entier; x: chiffre; VAR w: entier);
PROCEDURE EntierParChiffre (u: entier; x: chiffre; VAR w: entier);
PROCEDURE ChiffreEnEntier (x: chiffre; VAR u: entier);
FUNCTION Taille (VAR u: entier): integer;
PROCEDURE FixerSigne (VAR u: entier; s: integer);
PROCEDURE EcrireEntier (VAR u: entier; titre: texte);
```

La conversion de la représentation en base b en représentation modulaire se fait comme suit :

```
PROCEDURE EntierEnEntierMod (a: Entier; VAR u: EntierMod);
   VAR
      k: integer;
   BEGIN
      FOR k := 1 TO TailleMod DO
         u[k] := EvalueMod(a, m[k])
   END; { de "EntierEnEntierMod" }
```

La procédure `EvalueMod` évalue l'entier a modulo son deuxième argument, par le schéma de Horner :

```
FUNCTION EvalueMod (VAR a: Entier; m: integer): integer;
   VAR
      k, aModm: integer;
   BEGIN
      aModm := 0;
      FOR k := Taille(a) DOWNTO 0 DO
         aModm := (aModm * base + a[k]) MOD m;
      EvalueMod := aModm
   END; { de "EvalueMod" }
```

Réciproquement, le calcul de la représentation en base b à partir de la représentation modulaire se fait en deux étapes. Dans une première phase, on calcule les coefficients v_j donnés par la proposition 11.2.2 :

```
PROCEDURE Conversion (u: EntierMod; VAR v: EntierMod);
   VAR
      j, k, z: integer;
   BEGIN
      FOR k := 1 TO TailleMod DO BEGIN
         z := u[k] MOD m[k];
         FOR j := 1 TO k - 1 DO
            z := ((z - v[j]) * c[j, k]) MOD m[k];
         IF z < 0 THEN
            v[k] := z + m[k]
         ELSE
            v[k] := z
      END;
   END; { de "Conversion" }
```

Ensuite, on utilise ces coefficients pour évaluer, essentiellement par un schéma de Horner, la représentation en base b. La procédure suivante contient l'appel à la procédure de conversion :

```
PROCEDURE EntierModEnEntier (u: EntierMod; VAR a: Entier);
   VAR
      k: integer;
      v: EntierMod;
   BEGIN
      Conversion(u, v);                          Coefficients de conversion.
      ChiffreEnEntier(v[TailleMod], a);          a := v_r.
      FixerSigne(a, 1);
      FOR k := TailleMod - 1 DOWNTO 1 DO BEGIN
         EntierParChiffre(a, m[k], a);           a := a m_k + v_k.
         EntierPlusChiffre(a, v[k], a)
      END;
   END; { de "EntierModEnEntier" }
```

Maintenant que nous disposons des procédures de conversion, nous pouvons considérer l'arithmétique modulaire proprement dite. Les opérations, avons-nous dit, se font coefficient par coefficient. On écrit donc :

```
PROCEDURE EntierModPlusEntierMod (u, v: EntierMod; VAR w: EntierMod);
   VAR
      k: integer;
   BEGIN
      FOR k := 1 TO TailleMod DO
         w[k] := SommeMod(u[k], v[k], m[k]);
   END; { de "EntierModPlusEntierMod" }
PROCEDURE EntierModMoinsEntierMod (u, v: EntierMod; VAR w: EntierMod);
   VAR
      k: integer;
   BEGIN
      FOR k := 1 TO TailleMod DO
```

```
            w[k] := DifferenceMod(u[k], v[k], m[k]);
      END; { de "EntierModMoinsEntierMod" }
   PROCEDURE EntierModParEntierMod (u, v: EntierMod; VAR w: EntierMod);
      VAR
         k: integer;
      BEGIN
         FOR k := 1 TO TailleMod DO
            w[k] := ProduitMod(u[k], v[k], m[k]);
      END; { de "EntierModParEntierMod" }
```

Les opérations élémentaires sont :

```
   FUNCTION SommeMod (x, y, m: integer): integer;
      BEGIN
         IF x + y >= m THEN SommeMod := x + y - m ELSE SommeMod := x + y
      END; { de "SommeMod" }
   FUNCTION DifferenceMod (x, y, m: integer): integer;
      BEGIN
         IF x < y THEN DifferenceMod := x - y + m ELSE DifferenceMod := x - y
      END; { de "DifferenceMod" }
   FUNCTION ProduitMod (x, y, m: integer): integer;
      BEGIN
         ProduitMod := (x * y) MOD m
      END; { de "ProduitMod" }
```

Pour calculer commodément la représentation modulaire d'un entier en simple précision, on utilise :

```
   PROCEDURE ChiffreEnEntierMod (x: chiffre; VAR a: EntierMod);
      VAR
         k: integer;
      BEGIN
         FOR k := 1 TO TailleMod DO
            a[k] := x MOD m[k]
      END; { de "ChiffreEnEntierMod" }
```

Le calcul des nombres de Catalan se fait dans une table, définie par :

```
   CONST
      TailleSuite = 30;
   TYPE
      SuiteEntierMod = ARRAY[0..TailleSuite] OF EntierMod;
```

La procédure s'écrit :

```
   PROCEDURE SuiteCatalan (VAR c: SuiteEntierMod; r: integer);
      VAR
         n, k: integer;
         p, s: EntierMod;
```

```
BEGIN
   ChiffreEnEntierMod(1, c[0]);                           c0 = 1.
   FOR n := 1 TO r DO BEGIN
      ChiffreEnEntierMod(0, s);                           s := 1.
      FOR k := 0 TO n - 1 DO BEGIN
         EntierModParEntierMod(c[k], c[n - 1 - k], p);    p := c_k c_{n-1-k}.
         EntierModPlusEntierMod(s, p, s)                  s := s + p.
      END;
      c[n] := s
   END;
END; { de "SuiteCatalan" }
```

On obtient

```
c[0] = 1
c[1] = 1
c[2] = 2
c[3] = 5
c[4] = 14
c[5] = 42
c[6] = 1 32
c[7] = 4 29
c[8] = 14 30
c[9] = 48 62
c[10] = 1 67 96
c[11] = 5 87 86
c[12] = 20 80 12
c[13] = 74 29 00
c[14] = 2 67 44 40
c[15] = 9 69 48 45
c[16] = 35 35 76 70
c[17] = 1 29 64 47 90
c[18] = 4 77 63 87 00
c[19] = 17 67 26 31 90
c[20] = 65 64 12 04 20
c[21] = 2 44 66 26 70 20
c[22] = 9 14 82 56 36 40
c[23] = 34 30 59 61 36 50
c[24] = 1 28 99 04 14 73 24
c[25] = 4 86 19 46 40 14 52
c[26] = 18 36 73 53 07 21 52
c[27] = 69 53 35 50 91 60 04
c[28] = 2 63 74 79 51 75 03 60
c[29] = 10 02 24 22 16 65 13 68
c[30] = 2 78 35 36 49 22 41 89
```

La dernière valeur n'est juste que modulo 3536632852868115.

Notes bibliographiques

Les entiers de Gauss sont traités dans de très nombreux ouvrages d'algèbre ou de théorie des nombres. Pour les replacer dans un contexte un peu plus général, on pourra consulter le petit livre de P. Samuel :

P. Samuel, *Théorie algébrique des nombres*, Paris, Hermann, 1967.

Chapitre 12

Grands nombres

12.1 Entiers en multiprécision

12.1.1 Enoncé : entiers en multiprécision

Les opérations arithmétiques sur de grands entiers se réalisent de la façon suivante. Soient $N > 0$ et $b \geq 2$ des entiers; tout entier positif $u < b^N$ s'écrit de façon unique

$$u = \sum_{i=0}^{n} u_i b^i$$

avec $0 \leq n < N$, $0 \leq u_i < b$ et $u_n \neq 0$. La suite $(u_n, \ldots, u_0)$ est la *représentation* de u (en base b). On écrira $u = (u_n, \ldots, u_0)$. La représentation de 0 est (0). Tout entier de valeur absolue $< b^N$ est décrit par son signe et la représentation de sa valeur absolue.

Les opérations arithmétiques sont à implémenter dans cette représentation. On pourra prendre $b = 100$ et $N = 10$ (ou plus).

1.– Ecrire une procédure pour l'addition de grands entiers.

Les *nombres de Fibonacci* sont définis par $F_0 = 0$, $F_1 = 1$ et, pour $n \geq 0$, $F_{n+2} = F_{n+1} + F_n$.

2.– Jusqu'à quel indice peut-on calculer les nombres de Fibonacci lorsque $b = 100$ et $N = 10$? Ecrire une procédure calculant ces nombres de Fibonacci.

3.– Ecrire des procédures pour la comparaison, la soustraction, la multiplication de grands entiers.

Soient $u = (u_{n+1}, \ldots, u_0)$ et $v = (v_n, \ldots, v_0)$ tels que $q = [u/v] < b$, et soit

$$\hat{q} = \min(b - 1, \left[\frac{u_{n+1}b + u_n}{v_n}\right])$$

(On note $[x]$ la partie entière de x.)

4.– Démontrer que $\hat{q} \geq q$.

5.– Démontrer que si $v_n \geq [b/2]$, alors $\hat{q} - 2 \leq q$.

6.– Soit $d = [b/(1 + v_n)]$ et $v' = dv$. Démontrer que $v' = (v'_n, \ldots, v'_0)$ et que $v'_n \geq [b/2]$.

7.– En déduire un algorithme qui calcule le quotient et le reste de la division entière de $u = (u_m, \ldots, u_0)$ par $v = (v_n, \ldots, v_0)$.

8.– Ecrire une procédure calculant le pgcd de deux grands entiers. Utilisez-la pour vérifier expérimentalement l'égalité

$$\mathrm{pgcd}(F_n, F_m) = F_{\mathrm{pgcd}(n,m)}$$

9.– Démontrer cette formule.

12.1.2 Solution : entiers en multiprécision

En Pascal, les entiers représentables par le type `integer` sont de taille limitée (entre -2^{15} et $2^{15}-1$ sur un micro). La manipulation de grands entiers exige donc la programmation d'un certain nombre de procédures qui effectuent les opérations arithmétiques sur des représentations appropriées d'entiers de grande taille. Nous considérons le cadre que voici.

Soient $N > 0$ et $b \geq 2$ deux entiers; tout entier positif $u < b^N$ s'écrit de façon unique

$$u = \sum_{i=0}^{n} u_i b^i$$

avec $0 \leq n < N$, $0 \leq u_i < b$ et $u_n \neq 0$. La suite $(u_n, \ldots, u_0)$ est la *représentation* de u (en base b). On écrira $u = (u_n, \ldots, u_0)$. Chaque u_i sera appelé un *chiffre*. La représentation de 0 est (0). Tout entier de valeur absolue $< b^N$ est décrit par son signe et la représentation de sa valeur absolue. Dans la pratique, on voudrait que la base b soit la plus grande possible, mais pour effectuer simplement les opérations sur les chiffres, on demande que b^2 soit majoré par le plus grand entier représentable. De plus, pour éviter une conversion longue, on choisit pour b une puissance de 10. Cela ne laisse que $b = 10$ ou $b = 100$. Si l'on se permet l'emploi du type `longint`, on peut prendre $b = 10\,000$, ce qui est plus satisfaisant. En codant les entiers par des réels, on peut aller un peu plus loin encore au moyen de quelques acrobaties que nous ne considérons pas ici.

Les opérations d'addition, de soustraction et de multiplication de grands entiers se réalisent en transcrivant l'algorithme scolaire qui opère sur les chiffres. Pour l'addition de deux entiers naturels $(u_n, \ldots, u_0)$ et $(v_n, \ldots, v_0)$ par exemple, que nous supposons pour simplifier de même taille, on évalue les chiffres

$$\begin{aligned} w_i &= u_i + v_i + r_i \pmod b \\ r_{i+1} &= \lfloor (u_i + v_i + r_i)/b \rfloor \end{aligned}$$

où $r_0 = 0$ (on note $\lfloor x \rfloor$ la partie entière de x). Cette opération n'est pas difficile à programmer.

La division de deux entiers naturels est plus délicate. La méthode scolaire consiste à estimer, de manière un peu empirique, le chiffre suivant du quotient, puis à vérifier qu'il est correct. Nous allons procéder de manière similaire.

La première méthode qui vient à l'esprit est très longue : pour déterminer un chiffre, disons le premier, du quotient d'un entier $u = (u_n, \ldots, u_0)$ par un entier $v = (v_m, \ldots, v_0)$, on peut soustraire de u le nombre $v \cdot b^{n-m}$ tant que le résultat de la soustraction reste positif. Il est clair que le nombre de soustractions est considérable ($b/2$ pour un chiffre en moyenne). Nous cherchons un procédé plus rapide.

Réduit à son essence, le problème est donc de déterminer, de façon économique, le quotient entier q de deux nombres $u = (u_{n+1}, \ldots, u_0)$ et $v = (v_n \ldots, v_0)$. Notons d'abord que l'entier q dépend de tous les chiffres de u et ne peut donc être déterminé sans examiner u en entier. Ainsi, en base 10, le quotient de $u = 100\,002$ par $v = 50\,001$ est 2, alors que le quotient entier de 100 001 par v est 1.

Il existe un algorithme qui, moyennant un prétraitement des nombres u et v, permet d'estimer un chiffre du quotient avec une erreur d'au plus 2. Le nombre de soustractions à faire est donc majoré par 3 et, surtout, est indépendant de la base. La méthode procède en deux étapes. On calcule d'abord un quotient approché, par la formule

$$p = \lfloor (u_{n+1}b + u_n)/v_n \rfloor \tag{1.1}$$

puis on l'ajuste pour obtenir le quotient exact q. Si l'on n'y prend garde, la valeur de p peut être très éloignée de celle de q. Ainsi, en base 100, pour

$$u = (1, 80, \ldots) \qquad v = (2, \ldots)$$

on a $p = 90$, mais le quotient exact est $q = 60$ si

$$u = 1\ 80\ 00\ 00\ 00 \quad \text{et} \quad v = 2\ 99\ 99$$

et toutes les valeurs intermédiaires entre 60 et 90 sont possibles. En revanche, et toujours en base 100, pour

$$u = (30, 60, \ldots) \qquad v = (51, \ldots)$$

on a $p = 60$, et $q = 58$ si $u = 30\ 60\ 00$ et $v = 51\ 99$. Nous allons montrer que ce dernier phénomène, à savoir $p - q \leq 2$, est général moyennant une condition qui n'est pas trop difficile à réaliser.

Soient donc

$$u = (u_{n+1}, \ldots, u_0) \quad v = (v_n, \ldots, v_0)$$

avec $v_n \neq 0$, et supposons $\lfloor u/v \rfloor \leq b - 1$. Soit

$$u = qv + r \text{ avec } 0 \leq r < v$$

la division euclidienne de u par v, avec $q = \lfloor u/v \rfloor$. Avec le quotient approché p donné par l'équation (1.1), posons

$$\hat{q} = \min(b-1, p)$$

Les deux propositions suivantes montrent que $\hat{q}$ approche assez bien le quotient exact q.

Proposition 12.1.1. *On a $\hat{q} \geq q$.*

Preuve. Comme $q \leq b-1$, l'inégalité est vraie si $p > b$. Si $p \leq b-1$, alors notons que

$$p \geq \frac{u_{n+1}b + u_n}{v_n} - \frac{v_n - 1}{v_n}$$

d'où $v_n p \geq u_{n+1}b + u_n - v_n + 1$. Il en résulte que

$$u - pv \leq u - pv_n b^n \leq (u - u_{n+1}b^{n+1} - u_n b^n) + v_n b^n - b^n < b^n + v_n b^n - b^n \leq v$$

On a donc $u - pv < v$, d'où $u - pv \leq u - qv$, soit $q \leq p$. ∎

Proposition 12.1.2. *Si $v_n \geq \lfloor b/2 \rfloor$, alors $\hat{q} - 2 \leq q$.*

Preuve. Si $v = b^n$, alors

$$p = \left\lfloor \frac{u_{n+1}b^{n+1} + u_n b^n}{v_n b^n} \right\rfloor = \left\lfloor \frac{u}{b^n} \right\rfloor = q$$

et $\hat{q} = q$. On peut donc supposer $v \neq b^n$. On a alors

$$\hat{q} \leq \frac{u_{n+1}b + u_n}{v_n} = \frac{u_{n+1}b^{n+1} + u_n b^n}{v_n b^n} \leq \frac{u}{v_n b^n} < \frac{u}{v - b^n}$$

la dernière inégalité provenant de ce que $v < (v_n + 1)b^n$.
Supposons alors la conclusion fausse, soit $q + 3 \leq \hat{q}$. Comme $\hat{q} \leq b-1$, il en résulte que

$$q = \left\lfloor \frac{u}{v} \right\rfloor \leq b - 4 \tag{1.2}$$

Par ailleurs, on a $q \geq u/v - 1$, d'où

$$3 \leq \hat{q} - q < \frac{u}{v - b^n} - \frac{u}{v} + 1 = \frac{ub^n}{(v - b^n)v} + 1, \text{ soit } 2 < \frac{u}{v}\frac{b^n}{v - b^n},$$

ou encore

$$\frac{u}{v} > 2\frac{v - b^n}{b^n} \geq 2(v_n - 1)$$

et comme $v_n > b/2 - 1$, il en résulte que $\lfloor u/v \rfloor \geq b - 3$, en contradiction avec (1.2). ∎

Ces deux propositions montrent que l'un des trois nombres $\hat{q}$, $\hat{q} - 1$, $\hat{q} - 2$ est le quotient exact. Il reste à trouver une façon simple de satisfaire la condition de la proposition 12.1.2.

Proposition 12.1.3. *Soit* $v' = v\lfloor b/(1+v_n)\rfloor$. *Alors* $v' < b^n$ *et on a* $v'_n \geq \lfloor b/2\rfloor$, *avec* $v' = (v'_n, \ldots, v'_0)$.

Preuve. Posons $d = \lfloor b/(1+v_n)\rfloor$ et effectuons la multiplication de v par d : on a $v_0 d = r_1 b + v'_0$, avec $0 \leq v'_0 < b$ et $r_1 < d$. Plus généralement, supposant $r_k < d$, pour $k \geq 1$, on a

$$v_k d + r_k = r_{k+1} b + v'_k \qquad 0 \leq v'_k < b, \quad 0 \leq r_{k+1} < d$$

Comme

$$v_n d + r_n < (v_n + 1)\lfloor b/(1+v_n)\rfloor \leq b$$

on a $r_{n+1} = 0$ et $v' < b^n$.

Par ailleurs, on a $v'_n \geq v_n \lfloor b/(1+v_n)\rfloor$. Pour montrer que $v'_n \geq \lfloor b/2\rfloor$, nous vérifions que $x\lfloor b/(1+x)\rfloor \geq \lfloor b/2\rfloor$ pour tout entier x, avec $1 \leq x < b$. Ceci est vrai si $x \geq \lfloor b/2\rfloor$. Si $1 \leq x < \lfloor b/2\rfloor$, alors

$$x\left\lfloor \frac{b}{1+x}\right\rfloor - \left\lfloor \frac{b}{2}\right\rfloor > x\left(\frac{b}{1+x} - 1\right) - \frac{b}{2} = \frac{x-1}{x+1}\left(\frac{b}{2} - x - 1\right) - 1 \geq -1$$

puisque $b/2 \geq x + 1$. On a donc $x\lfloor b/(1+x)\rfloor > \lfloor b/2\rfloor - 1$. Ceci prouve la deuxième assertion. ∎

Disons que v est *normalisé* si $v_n \geq \lfloor b/2\rfloor$. La division de $u = (u_{n+1}, \ldots, u_0)$ par $v = (v_n, \ldots, v_0)$, lorsque $\lfloor u/v\rfloor \leq b - 1$ et v est normalisé, peut donc se faire comme suit : dans une première étape, calculer $\hat{q}$; ensuite, si $u - \hat{q}v \geq 0$, alors $q = \hat{q}$, sinon, si $u - (\hat{q}-1)v \geq 0$, alors $q = \hat{q} - 1$, et sinon $q = \hat{q} - 2$. Cet algorithme est mis en œuvre sur des couples (u, v) quelconques en normalisant d'abord v et en multipliant u par le même facteur; ensuite, les chiffres du quotient sont déterminés en divisant u par des «décalés» vb^k de v. Enfin, le reste de la division est obtenu en annulant l'effet de la normalisation.

12.1.3 Programme : entiers en multiprécision

Soit $b \geq 2$ la *base* choisie. Un entier positif u admet un développement unique

$$u = \sum_{k=0}^{n} u_k b^k, \qquad 0 \leq u_k < b$$

dans cette base. La représentation

$$(u_n, \ldots, u_0)$$

de u est rangée dans un tableau. En vue d'éviter des calculs inutiles, on mémorise également l'indice n du premier chiffre non nul du développement de u en base b. Appelons cet indice la *taille* de u. (Si $u = 0$, sa taille est fixée arbitrairement à un entier négatif.) Un entier naturel est donc décrit par un tableau de chiffres (relativement à la base) et par sa taille; un entier relatif comporte, en plus, son signe. Ceci conduit à la structure suivante :

```
CONST
        base = 100;
        DecimauxParChiffre = 2;      Nombre de chiffres décimaux pour écrire un «chiffre».
        TailleMax = 20;
        TailleEntierNul = -1;
TYPE
        chiffre = integer;
        entier = ARRAY[-2..TailleMax] OF chiffre;
```

Le choix de la base n'est pas gratuit. D'abord, une base qui est une puissance de 10 permet de lire et d'écrire des entiers (presque) sans conversion. Ensuite, la base b doit être telle que l'entier b^2 est inférieur au plus grand entier représentable dans la machine, ceci pour pouvoir au moins faire le produit de deux chiffres (relativement à b) sans débordement. Ceci conduit au choix fait. Toutefois, nous allons également nous permettre la base 10 000, en manipulant des chiffres définis comme `longint`. L'introduction du type chiffre permet une conversion instantanée à cette situation.

Dans l'emplacement d'indice -1 du tableau d'un entier on range sa taille, et dans l'emplacement d'indice -2 son signe. Il y a évidemment de l'arbitraire dans cette convention, mais de toute façon le choix est masqué par les fonctions d'accès et de modification de taille et de signe qui sont les suivantes :

```
FUNCTION Taille (VAR u: entier): integer;
PROCEDURE FixerTaille (VAR u: entier; p: integer);
FUNCTION LeSigne (VAR u: entier): integer;
PROCEDURE FixerSigne (VAR u: entier; s: integer);
FUNCTION EstEntierNul (VAR u: entier): boolean;
```

et qui se réalisent, dans notre représentation, par :

```
FUNCTION Taille (VAR u: entier): integer;
   BEGIN
      Taille := u[-1]
   END; { de "Taille" }
PROCEDURE FixerTaille (VAR u: entier; p: integer);
   BEGIN
      u[-1] := p
   END; { de "FixerTaille" }
FUNCTION LeSigne (VAR u: entier): integer;
   BEGIN
      LeSigne := u[-2]
   END; { de "LeSigne" }
PROCEDURE FixerSigne (VAR u: entier; s: integer);
   BEGIN
      u[-2] := s
   END; { de "FixerSigne" }
FUNCTION EstEntierNul (VAR u: entier): boolean;
```

```
   BEGIN
      EstEntierNul := Taille(u) = TailleEntierNul
   END; { de "EstEntierNul" }
```

Ainsi, pour affecter l'entier nul à une variable, on utilise la procédure :

```
PROCEDURE EntierNul (VAR u: entier);
   BEGIN
      FixerTaille(u, TailleEntierNul);
      FixerSigne(u, 1);
   END; { de "EntierNul" }
```

Les opérations arithmétiques sur les entiers se réalisent en transcrivant l'algorithme scolaire qui, lui, opère sur les chiffres. Ainsi l'addition de deux chiffres x et y, en présence d'une retenue r, se fait par les formules

$$
\begin{aligned}
z &= x + y + r \pmod b \\
r' &= \lfloor (x + y + r)/b \rfloor
\end{aligned}
$$

où r' est la nouvelle retenue. Cette opération élémentaire et ses trois consœurs sont prises en charge par les procédures :

```
PROCEDURE SommeChiffres (x, y: chiffre; VAR z, r: chiffre);
PROCEDURE DifferenceChiffres (x, y: chiffre; VAR z, r: chiffre);
PROCEDURE ProduitChiffres (x, y: chiffre; VAR z, r: chiffre);
PROCEDURE DivisionChiffres (x, y: chiffre; VAR z, r: chiffre);
```

Elles s'écrivent comme suit :

```
PROCEDURE SommeChiffres (x, y: chiffre; VAR z, r: chiffre);
   BEGIN
      z := x + y + r;
      IF z >= base THEN BEGIN
         z := z - base; r := 1
      END
      ELSE
         r := 0;
   END; { de "SommeChiffres" }
PROCEDURE DifferenceChiffres (x, y: chiffre; VAR z, r: chiffre);
   BEGIN
      z := x - y - r;
      IF z < 0 THEN BEGIN
         z := z + base; r := 1
      END
      ELSE
         r := 0;
   END; { de "DifferenceChiffres" }
PROCEDURE ProduitChiffres (x, y: chiffre; VAR z, r: chiffre);
   VAR
```

```
      s: chiffre;
   BEGIN
      s := x * y + r;
      z := s MOD base; r := s DIV base
   END; { de "ProduitChiffres" }
PROCEDURE DivisionChiffres (x, y: chiffre; VAR z, r: chiffre);
   VAR
      s: chiffre;
   BEGIN
      s := x + r * base;
      z := s DIV y; r := s MOD y
   END; { de "DivisionChiffres" }
```

A l'aide de ces procédures, nous mettons en place des procédures de manipulation d'entiers, sans considération du signe, à savoir :

```
PROCEDURE NaturelPlusNaturel (VAR u, v, w: entier);
PROCEDURE NaturelMoinsNaturel (VAR u, v, w: entier);
PROCEDURE NaturelParNaturel (u, v: entier; VAR w: entier);
PROCEDURE NaturelDivEuclNaturel (u, v: entier; VAR q, r: entier);
FUNCTION CompareNaturel (VAR u, v: entier): integer;
```

Ces procédures sont ensuite étendues aux entiers avec signe. Les noms correspondants sont :

```
PROCEDURE EntierPlusEntier (u, v: entier; VAR w: entier);
PROCEDURE EntierMoinsEntier (u, v: entier; VAR w: entier);
PROCEDURE EntierParChiffre (u: entier; x: chiffre; VAR w: entier);
PROCEDURE EntierParEntier (u, v: entier; VAR w: entier);
PROCEDURE EntierDivEuclEntier (u, v: entier; VAR q, r: entier);
PROCEDURE EntierModEntier (u, v: entier; VAR w: entier);
PROCEDURE EntierDivEntier (u, v: entier; VAR w: entier);
```

Les procédures suivantes constituent des cas particuliers, parfois utiles :

```
PROCEDURE NaturelParChiffre (u: entier; x: chiffre; VAR w: entier);
PROCEDURE NaturelPlusChiffre (u: entier; x: chiffre; VAR w: entier);
PROCEDURE NaturelParMonome (u: entier; x:chiffre; n: integer; VAR w: entier);
PROCEDURE NaturelSurChiffre (u: entier; x: chiffre; VAR w: entier);
```

Voici une réalisation de l'addition et de la soustraction :

```
PROCEDURE NaturelPlusNaturel (VAR u, v, w: entier);          w = u + v
   VAR
      i, m: integer;
      retenue: chiffre;
   BEGIN
      m := min(Taille(u), Taille(v));
      retenue := 0;
      FOR i := 0 TO m DO
```

```
      SommeChiffres(u[i], v[i], w[i], retenue);
   IF Taille(u) > m THEN
      FOR i := m + 1 TO Taille(u) DO
         SommeChiffres(u[i], 0, w[i], retenue)
   ELSE
      FOR i := m + 1 TO Taille(v) DO
         SommeChiffres(0, v[i], w[i], retenue);
   m := max(Taille(u), Taille(v));
   IF retenue = 1 THEN BEGIN
      m := m + 1;
      w[m] := retenue
   END;
   FixerTaille(w, m)
END; { de "NaturelPlusNaturel" }
```

La relative difficulté de cette procédure vient du choix de la représentation : pour chacun des deux opérandes u et v, on ne peut être sûr que les chiffres au-delà de la taille sont nuls, et il faut donc séparer les cas. La soustraction se réalise de la même façon :

```
PROCEDURE NaturelMoinsNaturel (VAR u, v, w: entier);          w = u − v
   On suppose u ≥ v.
   VAR
      i: integer;
      retenue: chiffre;
   BEGIN
      retenue := 0;
      FOR i := 0 TO Taille(v) DO
         DifferenceChiffres(u[i], v[i], w[i], retenue);
      FOR i := Taille(v) + 1 TO Taille(u) DO
         DifferenceChiffres(u[i], 0, w[i], retenue);
      i := Taille(u);                          Calcul de la taille de w.
      WHILE (i >= 0) AND (w[i] = 0) DO         Des chiffres annulés ?
         i := i - 1;                           Si i = −1, alors w = 0.
      FixerTaille(w, i)
   END; { de "NaturelMoinsNaturel" }
```

La multiplication se fait en deux étapes : calcul du produit d'un entier par un chiffre, puis addition au résultat partiel déjà obtenu. La première étape s'écrit :

```
PROCEDURE NaturelParMonome (u: entier; x:chiffre; n: integer; VAR w: entier);
   Calcul de w = u·xb^n.
   VAR
      i: integer;
      retenue: chiffre;
   BEGIN
      IF x = 0 THEN
         EntierNul(w)
      ELSE BEGIN
         FOR i := 0 TO n - 1 DO                Décalage.
```

```
            w[i] := 0;
         retenue := 0;
         FOR i := 0 TO Taille(u) DO
            ProduitChiffres(u[i], x, w[n + i], retenue);
         IF retenue > 0 THEN BEGIN             La taille de w dépend
            w[1 + Taille(u) + n] := retenue;   de la dernière retenue.
            FixerTaille(w, 1 + Taille(u) + n)
         END
         ELSE
            FixerTaille(w, Taille(u) + n);
      END
   END; { de "NaturelParMonome" }
```

La deuxième étape simule la méthode scolaire, sauf que l'addition se fait au fur et à mesure, pour ne pas dépasser la taille des chiffres :

```
PROCEDURE NaturelParNaturel (u, v: entier; VAR w: entier);     w = u v
   VAR
      i: integer;
      d: entier;
   BEGIN
      FixerTaille(w, TailleEntierNul);             Mise à zéro de w,
      FOR i := 0 TO 1 + Taille(u) + Taille(v) DO   même ses chiffres.
         w[i] := 0;
      FOR i := 0 TO Taille(v) DO BEGIN
         NaturelParMonome(u, v[i], i, d);          Multiplication par le
         NaturelPlusNaturel(w, d, w);              chiffre, puis addition.
      END;
   END; { de "NaturelParNaturel" }
```

La comparaison de deux entiers naturels se fait en comparant d'abord leur taille et, à taille égale, en cherchant le chiffre de plus haut indice où ils diffèrent. S'il n'y en a pas, les entiers sont égaux. La fonction suivante rend -1, 0 ou 1 selon que le premier argument est plus petit, égal ou plus grand que le deuxième.

```
FUNCTION CompareNaturel (VAR u, v: entier): integer;
   VAR
      i: integer;
   BEGIN
      IF Taille(u) <> Taille(v) THEN
         CompareNaturel := signe(Taille(u) - Taille(v))
      ELSE BEGIN                                   u et v ont même taille.
         i := Taille(u);
         WHILE (i >= 0) AND (u[i] = v[i]) DO { AND séquentiel }
            i := i - 1;
         IF i = -1 THEN
            CompareNaturel := 0
         ELSE
            CompareNaturel := signe(u[i] - v[i])
```

```
      END
   END;  { de "CompareNaturel"}
```

Comme nous l'avons vu, la division euclidienne de deux entiers naturels se ramène à la division *normalisée*. La normalisation consiste à multiplier les deux opérandes par un chiffre particulier. Voici donc la procédure principale :

```
PROCEDURE NaturelDivEuclNaturel (u, v: entier; VAR q, r: entier);     u = qv+r
   VAR
      omega: chiffre;
   BEGIN
      IF CompareNaturel(u, v) = -1 THEN BEGIN      Cas facile.
         EntierNul(q);
         r := u
      END
      ELSE BEGIN
         omega := base DIV (1 + v[Taille(v)]);    Facteur de normalisation.
         NaturelParChiffre(u, omega, u);          Normalisation des opérandes.
         NaturelParChiffre(v, omega, v);
         DivisionNormalisee(u, v, q, r);          Division euclidienne.
         NaturelSurChiffre(r, omega, r);          Dénormalisation du reste.
      END
   END; { de "NaturelDivEuclNaturel" }
```

Cette procédure utilise trois procédures auxiliaires : considérons d'abord la multiplication et la division par un chiffre, qui s'écrivent comme suit :

```
PROCEDURE NaturelParChiffre (u: entier; x: chiffre; VAR w: entier);
   Calcule le produit de l'entier u par le chiffre x.
   VAR
      i: integer;
      retenue: chiffre;
   BEGIN
      IF x = 0 THEN
         EntierNul(w)
      ELSE BEGIN
         retenue := 0;
         FOR i := 0 TO Taille(u) DO
            ProduitChiffres(u[i], x, w[i], retenue);
         IF retenue > 0 THEN BEGIN
            w[1 + Taille(u)] := retenue;
            FixerTaille(w, 1 + Taille(u))
         END
         ELSE
            FixerTaille(w, Taille(u));
      END
   END; { de "NaturelParChiffre" }
PROCEDURE NaturelSurChiffre (u: entier; x: chiffre; VAR w: entier);
   Calcule le quotient entier de u par le chiffre x > 0.
```

```
   VAR
      retenue: chiffre;
      k: integer;
   BEGIN
      IF EstEntierNul(u) THEN
         w := u
      ELSE BEGIN
         retenue := 0;
         FOR k := Taille(u) DOWNTO 0 DO
            DivisionChiffres(u[k], x, w[k], retenue);
         IF w[Taille(u)] = 0 THEN
            FixerTaille(w, Taille(u) - 1)
         ELSE
            FixerTaille(w, Taille(u));
      END
   END; { de "NaturelSurChiffre" }
```

La division normalisée suit la méthode exposée dans la section précédente :

```
PROCEDURE DivisionNormalisee (u, v: entier; VAR q, r: entier);
```

Division euclidienne de u par v. L'entier v est normalisé. Le quotient entier est q et le reste est r.

```
   VAR
      qEstime: chiffre;
      n, na, m: integer;
      qvprime, vprime: entier;
   BEGIN
      na := Taille(u);
      m := Taille(v);
      u[na + 1] := 0;
      FOR n := na DOWNTO m DO BEGIN
         IF u[n + 1] = v[m] THEN                            Estimation de q̂.
            qEstime := base - 1
         ELSE
            qEstime := (u[n + 1] * base + u[n]) DIV v[m];
         NaturelParMonome(v, 1, n - m, vprime);             v' = vb^(n-m).
         NaturelParChiffre(vprime, qEstime, qvprime);       q̂v'.
         WHILE CompareNaturel(u, qvprime) = -1 DO BEGIN     u < q̂v' ?
            qEstime := qEstime - 1;                         q̂ := q̂ - 1.
            NaturelMoinsNaturel(qvprime, vprime, qvprime);
         END;
         NaturelMoinsNaturel(u, qvprime, u);                u := u - q̂v'.
         q[n - m] := qEstime                                Le bon quotient.
      END;
      IF q[na - m] = 0 THEN
         FixerTaille(q, na - m - 1)
      ELSE
         FixerTaille(q, na - m);
      FixerSigne(q, 1);
```

```
      r := u;                                              Le reste.
   END; { de "DivisionNormalisee" }
```

Il est maintenant facile de réaliser les trois opérations arithmétiques sur des entiers relatifs : il suffit de tenir compte du signe.

```
PROCEDURE EntierPlusEntier (u, v: entier; VAR w: entier);
   BEGIN
      IF EstEntierNul(u) THEN
         w := v
      ELSE IF EstEntierNul(v) THEN
         w := u
      ELSE IF LeSigne(u) = LeSigne(v) THEN BEGIN
         NaturelPlusNaturel(u, v, w);
         FixerSigne(w, LeSigne(u))
      END
      ELSE                                        u et v de signe opposé.
         CASE CompareNaturel(u, v) OF
            1: BEGIN                              |u| > |v|
                  NaturelMoinsNaturel(u, v, w);
                  FixerSigne(w, LeSigne(u))
               END;
            0: EntierNul(w);                      u = −v
           -1: BEGIN                              |u| < |v|
                  NaturelMoinsNaturel(v, u, w);
                  FixerSigne(w, LeSigne(v))
               END
         END
   END; { de "EntierPlusEntier" }
```

La soustraction et la multiplication sont plus simples :

```
PROCEDURE EntierMoinsEntier (u, v: entier; VAR w: entier);
   BEGIN
      FixerSigne(v, -LeSigne(v));
      EntierPlusEntier(u, v, w)
   END; { de "EntierMoinsEntier" }
PROCEDURE EntierParEntier (u, v: entier; VAR w: entier);
   BEGIN
      FixerSigne(w, LeSigne(u) * LeSigne(v));
      NaturelParNaturel(u, v, w);
   END; { de "EntierParEntier" }
```

La division euclidienne de deux entiers se ramène à celle de deux entiers naturels, avec une petite complication si l'on veut que le reste soit toujours positif ou nul, même si le dividende ou le diviseur sont négatifs. On obtient :

```
PROCEDURE EntierDivEuclEntier (u, v: entier; VAR q, r: entier);
   VAR
```

```
      su, sv: integer;
   BEGIN
      su := LeSigne(u);
      sv := LeSigne(v);
      FixerSigne(u, 1);
      FixerSigne(v, 1);
      NaturelDivEuclNaturel(u, v, q, r);
      IF (su = -1) AND NOT EstEntierNul(r) THEN BEGIN
         NaturelMoinsNaturel(v, r, r);
         NaturelPlusChiffre(q, 1, q);
      END;
      FixerSigne(q, su * sv)
   END; { de "EntierDivEuclEntier" }
```

La procédure `NaturelPlusChiffre` s'écrit :

```
PROCEDURE NaturelPlusChiffre (u: entier; x: chiffre; VAR w: entier);
   VAR
      i: integer;
      retenue: chiffre;
   BEGIN
      i := 0;
      retenue := x;
      REPEAT
         SommeChiffres(u[i], 0, w[i], retenue);
         i := i + 1
      UNTIL (retenue = 0) OR (i = 1 + Taille(u));
      IF retenue > 0 THEN BEGIN
         w[i] := retenue;
         FixerTaille(w, i)
      END
      ELSE
         FixerTaille(w, Taille(u));
   END; { "NaturelPlusChiffre" }
```

Si seul le reste de la division euclidenne est demandé, on utilise la variante suivante :

```
PROCEDURE EntierModEntier (u, v: entier; VAR w: entier);
   VAR
      q: entier;
   BEGIN
      EntierDivEuclEntier(u, v, q, w)
   END; { de "EntierModEntier" }
```

Il est maintenant très facile d'écrire une procédure de calcul du pgcd de deux entiers :

```
PROCEDURE pgcdEntier (u, v: entier; VAR d: entier);
   VAR
      w: entier;
   BEGIN
```

```
      WHILE NOT EstEntierNul(v) DO BEGIN
         EntierModEntier(u, v, w);
         u := v;
         v := w
      END;
      d := u
   END; { de "pgcdEntier" }
```

Une première application est le calcul d'une table de nombres de Fibonacci, par simple application de la formule de définition.

```
VAR
   F: ARRAY[0..110] OF entier;            Pour ranger les nombres.
PROCEDURE CalculerNombresFibonacci (n: integer);
   Calcul les nombres de Fibonacci F_i pour i = 0, ..., n.
   VAR
      i: integer;
   BEGIN
      EntierNul(F[0]);                    F_0 = 0.
      ChiffreEnEntier(1, F[1]);           F_1 = 1.
      FOR i := 2 TO n DO                  La formule F_i = F_{i-1} + F_{i-2}.
         EntierPlusEntier(F[i - 1], F[i - 2], F[i])
   END; { de "CalculerNombresFibonacci" }
```

avec

```
PROCEDURE ChiffreEnEntier (x: chiffre; VAR u: entier);
   BEGIN
      FixerSigne(u, signe(x));
      FixerTaille(u, 0);
      u[0] := abs(x);
   END; { de "ChiffreEnEntier" }
```

Notons que

$$F_n \sim \frac{1}{\sqrt{5}}\alpha \text{ avec } \alpha = (1+\sqrt{5})/2$$

Si la taille maximale est 10, on peut calculer les nombres de Fibonacci inférieurs à 100^{11}, soit d'indice majoré par

$$\frac{10 \ln 100 + \ln \sqrt{5}}{\ln \alpha}$$

ce qui donne 106. Voici ces nombres :

```
Nombres de Fibonacci d'indices de  80  à  106

F(80) = 2 34 16 72 83 48 46 76 85
F(81) = 3 78 89 06 23 73 14 39 06
F(82) = 6 13 05 79 07 21 61 15 91
F(83) = 9 91 94 85 30 94 75 54 97
F(84) = 16 05 00 64 38 16 36 70 88
```

```
F(85) = 25 96 95 49 69 11 12 25 85
F(86) = 42 01 96 14 07 27 48 96 73
F(87) = 67 98 91 63 76 38 61 22 58
F(88) = 1 10 00 87 77 83 66 10 19 31
F(89) = 1 77 99 79 41 60 04 71 41 89
F(90) = 2 88 00 67 19 43 70 81 61 20
F(91) = 4 66 00 46 61 03 75 53 03 09
F(92) = 7 54 01 13 80 47 46 34 64 29
F(93) = 12 20 01 60 41 51 21 87 67 38
F(94) = 19 74 02 74 21 98 68 22 31 67
F(95) = 31 94 04 34 63 49 90 09 99 05
F(96) = 51 68 07 08 85 48 58 32 30 72
F(97) = 83 62 11 43 48 98 48 42 29 77
F(98) = 1 35 30 18 52 34 47 06 74 60 49
F(99) = 2 18 92 29 95 83 45 55 16 90 26
F(100) = 3 54 22 48 48 17 92 61 91 50 75
F(101) = 5 73 14 78 44 01 38 17 08 41 01
F(102) = 9 27 37 26 92 19 30 78 99 91 76
F(103) = 15 00 52 05 36 20 68 96 08 32 77
F(104) = 24 27 89 32 28 39 99 75 08 24 53
F(105) = 39 28 41 37 64 60 68 71 16 57 30
F(106) = 63 56 30 69 93 00 68 46 24 81 83
```

L'affichage d'un entier se fait en affichant les chiffres, par exemple séparés par un blanc, comme ci-dessus. On obtient donc :

```
PROCEDURE EcrireEntier (VAR u: entier; titre: texte);
   VAR
      i: integer;
   BEGIN
      write(titre);
      IF EstEntierNul(u) THEN                    L'entier nul.
         writeln(' 0')
      ELSE BEGIN
         IF LeSigne(u) = -1 THEN                 Le signe, si nécessaire.
            write('- ');
         write(u[Taille(u)] : 1);                Le premier chiffre,
         FOR i := Taille(u) - 1 DOWNTO 0 DO
            EcrireChiffre(u[i]);                 et les suivants.
         writeln
      END
   END; { de "EcrireEntier" }
```

Pour l'affichage d'un chiffre, observons que s'il est nul, ou s'il ne contient pas assez de chiffres décimaux, il convient d'afficher des zéros (sauf pour le premier).

```
PROCEDURE EcrireChiffre (a: chiffre);
   VAR
      i, n: integer;
```

```
    FUNCTION LargeurChiffre (a: chiffre): integer;
       BEGIN
          IF a = 0 THEN
             LargeurChiffre := 1
          ELSE
             LargeurChiffre := 1 + trunc(log10(a))
       END; { de "LargeurChiffre" }
    BEGIN
       write(' ');
       n := LargeurChiffre(a);                    Nombre de chiffres décimaux de a.
       FOR i := 1 TO DecimauxParChiffre - n DO
          write('0' : 1);
       write(a : 1)
    END; { "EcrireChiffre" }
```

La lecture d'un entier au clavier est plus difficile à rédiger, si l'on veut que la saisie se fasse agréablement, c'est-à-dire sans introduire de blanc entre les chiffres. On saisit alors les chiffres décimaux (appelons-les «digits») tapés comme des caractères, puis on les convertit et les regroupe en «chiffres» par rapport à la base. Voici une réalisation :

```
 PROCEDURE EntrerEntier (VAR u: entier; titre: texte);
    CONST
       LongueurMax = 80;
    VAR
       e: ARRAY[0..LongueurMax] OF integer;       Pour la saisie des chiffres décimaux.
       c: char;
       t, k, i, m, n: integer;
       x: chiffre;
    BEGIN
       writeln;
       writeln(titre);
       read(c);                               Lecture du premier caractère.
       IF c = '+' THEN BEGIN                  Si c'est le signe +,
          FixerSigne(u, 1);                   on en tient compte,
          read(c)                             et on lit le caractère suivant.
       END
       ELSE IF c = '-' THEN BEGIN             Signe −, on procède de même.
          FixerSigne(u, -1);
          read(c)
       END
       ELSE                                   Sinon, c'est un entier positif.
          FixerSigne(u, 1);
       n := 1;
       e[n] := ord(c) - ord('0');             Conversion du premier caractère en digit.
       WHILE NOT eoln DO BEGIN                Lecture et conversion des autres.
          n := n + 1;
          read(c);
          e[n] := ord(c) - ord('0')
```

```
      END;
      readln;                              Au total, n digits lus.
      IF (n = 1) AND (e[n] = 0) THEN       L'entier nul.
         FixerTaille(u, TailleEntierNul)
      ELSE BEGIN
         t := (n - 1) DIV DecimauxParChiffre;
         FixerTaille(u, t);                La taille de l'entier.
         i := n MOD DecimauxParChiffre;
         i := (DecimauxParChiffre - i) MOD DecimauxParChiffre;
         x := 0;                           On groupe les digits
         m := t;                           par paquet de DecimauxParChiffre.
         FOR k := 1 TO n DO BEGIN
            x := 10 * x + e[k];            Schéma de Horner, bien sûr.
            i := i + 1;
            IF i = DecimauxParChiffre THEN BEGIN
               u[m] := x;                  Chiffre suivant.
               x := 0; i := 0; m := m - 1
            END
         END
      END;
   END; { de "EntrerEntier" }
```

Considérons une autre application. Les *nombres de Catalan* sont définis par

$$c_n = \frac{1}{n+1}\binom{2n}{n} \qquad (n \geq 0)$$

Nous les avons déjà rencontrés au chapitre 9 et nous les avons calculés en arithmétique modulaire au chapitre précédent. Il n'est pas difficile de vérifier que les nombres de Catalan satisfont la relation de récurrence

$$c_n = c_{n-1}\frac{4n-2}{n+1} \qquad (n \geq 1)$$

On peut donc les calculer au moyen de la procédure :

```
PROCEDURE CalculerNombresCatalan (nn: integer);
   VAR
      n: integer;
   BEGIN
      ChiffreEnEntier(1, Cat[0]);
      FOR n := 1 TO nn DO BEGIN
         EntierParChiffre(Cat[n - 1], 4 * n - 2, Cat[n]);
         EntierSurChiffre(Cat[n], n + 1, Cat[n])
      END;
   END; { "CalculerNombresCatalan" }
```

où `Cat` est un tableau d'entiers de taille appropriée. Voici une table, obtenue en choisissant la base égale à 1000 (bien entendu, `chiffre=longint`) :

```
Nombres de Catalan d'indices de  0  à  50

C(0) = 1
C(1) = 1
C(2) = 2
C(3) = 5
C(4) = 14
C(5) = 42
C(6) = 132
C(7) = 429
C(8) = 1 430
C(9) = 4 862
C(10) = 16 796
C(11) = 58 786
C(12) = 208 012
C(13) = 742 900
C(14) = 2 674 440
C(15) = 9 694 845
C(16) = 35 357 670
C(17) = 129 644 790
C(18) = 477 638 700
C(19) = 1 767 263 190
C(20) = 6 564 120 420
C(21) = 24 466 267 020
C(22) = 91 482 563 640
C(23) = 343 059 613 650
C(24) = 1 289 904 147 324
C(25) = 4 861 946 401 452
C(26) = 18 367 353 072 152
C(27) = 69 533 550 916 004
C(28) = 263 747 951 750 360
C(29) = 1 002 242 216 651 368
C(30) = 3 814 986 502 092 304
C(31) = 14 544 636 039 226 909
C(32) = 55 534 064 877 048 198
C(33) = 212 336 130 412 243 110
C(34) = 812 944 042 149 730 764
C(35) = 3 116 285 494 907 301 262
C(36) = 11 959 798 385 860 453 492
C(37) = 45 950 804 324 621 742 364
C(38) = 176 733 862 787 006 701 400
C(39) = 680 425 371 729 975 800 390
C(40) = 2 622 127 042 276 492 108 820
C(41) = 10 113 918 591 637 898 134 020
C(42) = 39 044 429 911 904 443 959 240
C(43) = 150 853 479 205 085 351 660 700
C(44) = 583 300 119 592 996 693 088 040
C(45) = 2 257 117 854 077 248 073 253 720
```

```
C(46) = 8 740 328 711 533 173 390 046 320
C(47) = 33 868 773 757 191 046 886 429 490
C(48) = 131 327 898 242 169 365 477 991 900
C(49) = 509 552 245 179 617 138 054 608 572
C(50) = 1 978 261 657 756 160 653 623 774 456
```

On utilisera ces procédures aussi pour le calcul de π par arctangente.

12.2 Arithmétique flottante

12.2.1 Enoncé : arithmétique flottante

On se propose de calculer certains nombres, comme $\sqrt{2}$, avec une trentaine de décimales. Cela ne peut pas se faire avec un seul objet de type "real" puisque leur précision est limitée.

On effectue les calculs dans une "arithmétique flottante" en base B ($B = 100$ pour fixer les idées). Dans ce contexte, on appelle *chiffre* tout entier compris entre 0 et $B - 1$. Un *réel flottant* est composé d'un *exposant* e (un entier), d'un *signe* $s \in \{+1, -1\}$ et d'une suite $d_1, \ldots, d_N$ de chiffres, avec $d_1 \neq 0$ sauf si le nombre est nul. Le signe de zéro est $+1$. La valeur du réel flottant est

$$s * B^e * \sum_{i=1}^{N} d_i B^{-i}$$

On fixera $N = 16$. Un réel flottant peut donc être défini comme un tableau d'entiers dont les indices vont de -1 à N.

1.– Ecrire les procédures suivantes :
- comparaison de deux réels flottants;
- addition et soustraction de deux réels flottants;
- multiplication de deux réels flottants;
- division d'un réel flottant par un chiffre.

Le calcul d'un zéro d'une fonction f dérivable peut se faire par la méthode de Newton, c'est-à-dire en calculant une suite $(x_n)_{n\geq 0}$ par

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

2.– Démontrer que, pour calculer $1/\sqrt{a}$, on obtient le schéma itératif

$$x_{n+1} = (3x_n - ax_n^3)/2$$

Démontrer que x_n tend vers $1/\sqrt{a}$ lorsque $x_0 \in]0, \sqrt{\frac{3}{a}}[$. Ecrire une procédure qui calcule et affiche $1/\sqrt{2}$ à 30 décimales.

3.– Le calcul de l'inverse d'un réel flottant $a \neq 0$ peut se faire au moyen du schéma itératif

$$x_{n+1} = 2x_n - ax_n^2$$

Déterminer les valeurs de x_0 pour lesquelles x_n tend vers $1/a$. Ecrire une procédure qui calcule l'inverse d'un réel flottant. Vérifiez vos procédures en calculant l'inverse du réel flottant de la question précédente et en multipliant ces deux réels flottants.

4.– Le calcul de la racine carrée d'un réel flottant peut se faire au moyen du procédé itératif

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

Démontrer que x_n tend vers $\sqrt{a}$. Comment convient-il de choisir x_0 ? Ecrire une procédure qui calcule la racine carrée d'un réel flottant et calculer $\sqrt{2}$ avec 30 décimales.

12.2.2 Solution : arithmétique flottante

La *méthode de Newton* pour le calcul d'un zéro d'une fonction réelle dérivable revient à calculer une suite $(x_n)_{n\geq 0}$ de nombres réels par la formule

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

en partant d'un point x_0 convenablement choisi. On suppose bien entendu que f' ne s'annule en aucun des points x_n. Rappelons l'interprétation géométrique de la formule : x_{n+1} est l'abscisse de l'intersection de la tangente à la courbe $y = f(x)$ au point $(x_n, f(x_n))$ avec l'axe des x.

Si la suite converge vers un point x et si $f'(x) \neq 0$, alors $f(x) = 0$, comme on le voit en faisant tendre n vers l'infini dans l'équation ci-dessus. Nous renvoyons à la littérature pour une discussion plus détaillée de la méthode de Newton (voir les notes en fin de chapitre).

Considérons quelques cas particuliers. Soit a un nombre réel positif et soit f définie par

$$f(x) = 1 - \frac{1}{ax^2}$$

Alors $1/\sqrt{a}$ est la solution de $f(x) = 0$ et comme $f'(x) = 2/ax^3$, on a

$$x - \frac{f(x)}{f'(x)} = (3x - ax^3)/2$$

La fonction f étant concave, on a $0 < x_{n+1} < 1/\sqrt{a}$ si $0 < x_n < 1/\sqrt{a}$, et $x_{n+1} < 1/\sqrt{a}$ si $1/\sqrt{a} < x_n$. Il faut s'assurer que $0 < x_{n+1}$, et en fait que $0 < x_1$. Or $x_1 = 0$ si et seulement si $x_0 = f(x_0)/f'(x_0)$, donc si et seulement si $x_0 = \sqrt{3/a}$. Il en résulte que la méthode converge si x_0 est pris dans l'intervalle $]0, \sqrt{3/a}[$. En particulier, si $a = 2$, on peut prendre $x_0 = 1$.

Cherchons à calculer l'inverse d'un réel positif a. Pour cela, on considère la fonction

$$f(x) = 1 - \frac{1}{ax}$$

qui admet le zéro $1/a$. On a ici

$$x - \frac{f(x)}{f'(x)} = 2x - ax^2$$

à nouveau la fonction f est concave, et $x_0 = f(x_0)/f'(x_0)$ si et seulement si $x_0 = 2/a$. Il en résulte que la méthode converge si x_0 est pris dans l'intervalle $]0, 2/a[$.

Ces deux schémas itératifs ont l'avantage de ne pas faire intervenir de division autre que par une petite constante, et sont donc intéressants pour le calcul sur les flottants.

Le calcul de la racine carrée d'un réel positif a peut se faire en considérant la fonction

$$f(x) = ax^2 - 1$$

On a alors

$$x - \frac{f(x)}{f'(x)} = \frac{1}{2}(x + \frac{a}{x})$$

De plus, la fonction f est convexe (c'est une parabole) et la méthode de Newton converge pour tout x_0 positif. Bien sûr, on a intérêt à choisir le point de départ le plus près possible de la racine.

12.2.3 Programme : arithmétique flottante

On se propose de calculer certains nombres réels avec beaucoup de décimales. Cela ne peut pas se faire avec un seul objet de type «real» puisque leur précision est limitée. On effectue les calculs dans une «arithmétique flottante» qui ressemble à celle des entiers en multiprécision; et on pourra utiliser un certain nombre de procédures développées dans ce cadre.

On se fixe une base B qui sera une puissance de 10 pour faciliter les lectures et écritures. On appelle *chiffre* tout entier compris entre 0 et $B-1$. Un *réel flottant* est composé d'un *exposant* e (un entier), d'un *signe* $s \in \{+1, -1\}$ et d'une suite $d_1, \ldots, d_N$ de chiffres, avec $d_1 \neq 0$ sauf si le nombre est nul. Le signe de zéro est $+1$. La valeur du réel flottant est

$$s * B^e * \sum_{i=1}^{N} d_i B^{-i}$$

Appelons *normalisée* l'écriture d'un flottant avec $d_1 \neq 0$. Les opérations arithmétiques ne fournissent pas toujours une représentation normalisée, et il faudra donc les normaliser.

Un réel flottant peut être défini comme un tableau de chiffres dont les indices vont de -1 à N.

Les types de départ sont :

```
CONST
   base = 100;
   DecimauxParChiffre = 2;                Nombre de «digits» pour écrire un chiffre.
   TailleMax = 20;
TYPE
   chiffre = integer;
   Flottant = ARRAY[-1..TailleMax] OF chiffre;
VAR
   FlottantNul : Flottant;
```

Contrairement aux choix faits pour les entiers en multiprécision, nous définissons le flottant nul comme objet et non pas comme procédure. La différence est mince, donc pour varier... Dans les emplacements d'indice -1 et 0 sont rangés respectivement le signe et l'exposant du flottant ; comme pour les grands entiers, on utilisera des procédures pour accéder ou modifier ces objets :

```
FUNCTION Exposant (VAR u: Flottant): integer;
PROCEDURE FixerExposant (VAR u: Flottant; p: integer);
FUNCTION LeSigne (VAR u: Flottant): integer;
PROCEDURE FixerSigne (VAR u: Flottant; s: integer);
FUNCTION EstFlottantNul (VAR u: Flottant): boolean;
```

On se propose d'écrire, dans un premier temps, des procédures pour réaliser les quatre opérations arithmétiques, disons :

```
PROCEDURE FlottantPlusFlottant (u, v: Flottant; VAR w: Flottant);
PROCEDURE FlottantMoinsFlottant (u, v: Flottant; VAR w: Flottant);
PROCEDURE FlottantParFlottant (u, v: Flottant; VAR w: Flottant);
PROCEDURE FlottantSurFlottant (u, v: Flottant; VAR w: Flottant);
```

Les trois premières procédures sont faciles à écrire, en s'inspirant de la méthode scolaire ; pour la quatrième, on calcule l'inverse du diviseur par la méthode de Newton, puis on multiplie le résultat par le dividende.

Revenons aux procédures de base. Elles se réalisent comme suit :

```
FUNCTION Exposant (VAR u: Flottant): chiffre;
   BEGIN
      Exposant := u[0]
   END; { de "Exposant" }
PROCEDURE FixerExposant (VAR u: Flottant; p: chiffre);
   BEGIN
      u[0] := p
   END; { de "FixerExposant" }
FUNCTION LeSigne (VAR u: Flottant): chiffre;
   BEGIN
      LeSigne := u[-1]
   END; { de "LeSigne" }
PROCEDURE FixerSigne (VAR u: Flottant; s: chiffre);
```

```
    BEGIN
       u[-1] := s
    END; { de "FixerSigne" }
FUNCTION EstFlottantNul (VAR u: Flottant): boolean;
    BEGIN
       EstFlottantNul := u[1] = 0
    END; { de "EstFlottantNul" }
```

On reconnaît donc qu'un flottant est nul par le fait qu'il n'est pas normalisé. Le flottant nul est initialisé dans une procédure :

```
PROCEDURE InitFlottants;
    VAR
       i: integer;
    BEGIN
       FOR i := 1 TO TailleMax DO
          FlottantNul[i] := 0;
       FixerSigne(FlottantNul, 1);
       FixerExposant(FlottantNul, -(maxint DIV 2));
    END; { de "InitFlottants" }
```

On a choisi, comme exposant du flottant nul, un entier négatif grand; le choix de `maxint` au lieu de `maxint DIV 2` peut provoquer des messages d'erreurs si on fait des opérations sur les signes.

Les *opérations arithmétiques* se font en trois parties : les opérations sur les mantisses, les opérations sur les exposants et les opérations sur les signes. Les opérations sur les mantisses sont cause d'erreurs d'arrondi; ces erreurs sont inévitables, puisqu'elles sont liées à la limitation même de la représentation. On peut les atténuer, en arrondissant au nombre le plus proche plutôt que d'arrondir par défaut. Nous avons fait le deuxième choix parce qu'il simplifie les procédures.

Commençons par l'*addition*. L'addition de deux flottants positifs (ou de même signe) se fait en additionnant les mantisses, mais après avoir procédé au décalage nécessaire pour se ramener à des exposants égaux. On utilise des procédures de décalage vers la gauche et vers la droite de la mantisse; les places devenant libres sont remplies avec le chiffre 0 :

```
PROCEDURE DecalerMantisseDroite (d: integer; VAR u: Flottant);
    Décale vers la droite : (u_1, ..., u_N) → (0, ..., 0, u_1, ..., u_{N-d}).
    VAR
       k: integer;
    BEGIN
       FOR k := TailleMax DOWNTO d + 1 DO
          u[k] := u[k - d];
       FOR k := d DOWNTO 1 DO
          u[k] := 0;
    END; { de "DecalerMantisseDroite" }
```

```
PROCEDURE DecalerMantisseGauche (d: integer; VAR u: Flottant);
   Décale vers la gauche : (u_1, ..., u_N) → (u_{d+1}, ..., u_N, 0, ..., 0).
   VAR
      k: integer;
   BEGIN
      FOR k := 1 TO TailleMax - d DO
         u[k] := u[k + d];
      FOR k := TailleMax - d + 1 TO TailleMax DO
         u[k] := 0;
   END; { de "DecalerMantisseGauche" }
```

Moyennant ces procédures, on peut écrire l'addition de deux flottants positifs. Notons — et c'est là encore une conséquence de la limitation de la taille des flottants — que l'addition n'a pas d'effet si les exposants sont trop différents l'un de l'autre.

```
PROCEDURE PositifPlusPositif (u, v: Flottant; VAR w: Flottant);
   w = u + v, on suppose u et v de même signe.
   VAR
      delta: integer;
   BEGIN
      delta := Exposant(u) - Exposant(v);
      IF delta >= TailleMax THEN                Pas de calcul.
         w := u
      ELSE IF delta <= -TailleMax THEN          Pas de calcul.
         w := v
      ELSE IF delta >= 0 THEN
         PositifEtPositif(delta, u, v, w)       Une même procédure pour les
      ELSE                                      deux cas; le premier argument
         PositifEtPositif(-delta, v, u, w);     est toujours plus grand.
   END; { de "PositifPlusPositif" }
```

Pour économiser de l'écriture, on utilise une même procédure :

```
PROCEDURE PositifEtPositif (delta: integer; VAR u, v, w: Flottant);
   w = u + v, on suppose u > v et δ est la différence des exposants de u et v.
   VAR
      k: integer;
      retenue: chiffre;
   BEGIN
      DecalerMantisseDroite(delta, v);                Aligne l'exposant de v.
      retenue := 0;
      FOR k := TailleMax DOWNTO 1 DO                  Calcul de la somme,
         SommeChiffres(u[k], v[k], w[k], retenue);    chiffre par chiffre.
      IF retenue = 1 THEN BEGIN                       La dernière retenue est
         DecalerMantisseDroite(1, w);                 inscrite en mantisse...
         w[1] := retenue;
         FixerExposant(w, 1 + Exposant(u))            et en exposant.
      END
      ELSE
```

```
      FixerExposant(w, Exposant(u));
   END; { de "PositifEtPositif" }
```

La procédure fait appel à la procédure *SommeChiffres* déjà vue avec les grands entiers.

La *soustraction* de deux flottants de même signe pose le problème inverse du décalage, à savoir de la normalisation : si le flottant n'est pas nul, son premier chiffre doit être non nul. On obtient cette normalisation en décalant la mantisse. Ceci introduit des zéros en fin de représentation. Voici comment on réalise la normalisation :

```
PROCEDURE NormaliserFlottant (VAR u: Flottant);
   VAR
      d: integer;
   BEGIN
      d := 1;
      WHILE (d <= TailleMax) AND (u[d] = 0) DO      Chercher le premier
         d := d + 1;                                chiffre non nul.
      IF d = 1 + TailleMax THEN                     Si u = 0, ne rien faire,
      ELSE IF d > 1 THEN BEGIN
         DecalerMantisseGauche(d - 1, u);           sinon, décaler de d positions
         FixerExposant(u, Exposant(u) - d + 1)      et répercuter sur l'exposant.
      END;
   END; { de "NormaliserFlottant" }
```

La procédure suivante réalise la soustraction de deux flottants de même signe, le premier étant supposé plus grand en valeur absolue que le deuxième. On utilise la procédure *DifferenceChiffres*.

```
PROCEDURE PositifDePositif (delta: integer; VAR u, v, w: Flottant);
   w = u − v, on suppose u > v et δ est la différence des exposants de u et v.
   VAR
      k, retenue: integer;
   BEGIN
      DecalerMantisseDroite(delta, v);
      retenue := 0;
      FOR k := TailleMax DOWNTO 1 DO
         DifferenceChiffres(u[k], v[k], w[k], retenue);
      FixerExposant(w, Exposant(u));
      NormaliserFlottant(w);                        On normalise le résultat.
   END; { de "PositifDePositif" }
```

La procédure de soustraction est alors, comme pour l'addition, un simple aiguillage entre les diverses possibilités :

```
PROCEDURE PositifMoinsPositif (u, v: Flottant; VAR w: Flottant);
   VAR
      delta: integer;
   BEGIN
      delta := Exposant(u) - Exposant(v);
      IF delta >= TailleMax THEN                    Pas de calcul.
```

```
            w := u
         ELSE IF delta <= -TailleMax THEN                Pas de calcul.
            w := v
         ELSE IF delta >= 0 THEN
            PositifDePositif(delta, u, v, w)
         ELSE
            PositifDePositif(-delta, v, u, w);
      END; { de "PositifMoinsPositif" }
```

Comme pour les grands entiers, pour tenir compte du signe, il faut savoir comparer deux flottants ; l'algorithme est le même, la réalisation aussi, sauf que l'on compare l'exposant au lieu de la taille (mais est-ce vraiment différent ?) :

```
   FUNCTION ComparePositif (VAR u, v: Flottant): integer;
      VAR
         k: integer;
      BEGIN
         IF Exposant(u) <> Exposant(v) THEN
            ComparePositif := signe(Exposant(u) - Exposant(v))
         ELSE BEGIN
            k := 1;
            WHILE (k <= TailleMax) AND (u[k] = v[k]) DO k := k + 1;
            IF k = TailleMax + 1 THEN
               ComparePositif := 0
            ELSE ComparePositif := signe(u[k] - v[k])
         END
      END; { de "ComparePositif" }
```

Les procédures d'addition et de soustraction de deux réels flottants sont de même nature que pour les grands entiers :

```
   PROCEDURE FlottantPlusFlottant (u, v: Flottant; VAR w: Flottant);
      BEGIN
         IF LeSigne(u) = LeSigne(v) THEN BEGIN
            PositifPlusPositif(u, v, w);
            FixerSigne(w, LeSigne(u))
         END
         ELSE
            CASE ComparePositif(u, v) OF
               1: BEGIN
                     PositifMoinsPositif(u, v, w);
                     FixerSigne(w, LeSigne(u))
                  END;
               0: w := FlottantNul;
              -1: BEGIN
                     PositifMoinsPositif(v, u, w);
                     FixerSigne(w, LeSigne(v))
                  END
            END
```

```
   END; { de "FlottantPlusFlottant" }

PROCEDURE FlottantMoinsFlottant (u, v: Flottant; VAR w: Flottant);
   BEGIN
      FixerSigne(v, -LeSigne(v));
      FlottantPlusFlottant(u, v, w)
   END; { de "FlottantMoinsFlottant" }
```

La *multiplication* se réalise en multipliant chaque chiffre de l'un des facteurs par l'autre facteur et en additionnant le résultat, décalé du nombre approprié de places, au produit partiel déjà obtenu. Ceci donne la procédure que voici :

```
PROCEDURE FlottantParFlottant (u, v: Flottant; VAR w: Flottant);
   VAR
      k: integer;
      z: Flottant;
   BEGIN
      w := FlottantNul;
      IF NOT (EstFlottantNul(u) OR EstFlottantNul(v)) THEN BEGIN
         FOR k := TailleMax DOWNTO 1 DO
            IF u[k] <> 0 THEN BEGIN
               z:=v;
               FlottantParChiffre(z, u[k]);                 Multiplication.
               FixerExposant(z, Exposant(z) - k);           Décalage.
               PositifPlusPositif(w, z, w);                 Addition.
            END;
         FixerExposant(w, Exposant(u) + Exposant(w));
         FixerSigne(w, LeSigne(u) * LeSigne(v))
      END { de uv<>0 }
   END; { de "FlottantParFlottant" }
```

Le produit d'un flottant par un chiffre non nul se fait par :

```
PROCEDURE FlottantParChiffre (VAR u: Flottant; x: chiffre);
   Multiplie u par le chiffre x > 0.
   VAR
      k: integer;
      retenue: chiffre;
   BEGIN
      retenue := 0;
      FOR k := TailleMax DOWNTO 1 DO
         ProduitChiffres(u[k], x, u[k], retenue);
      IF retenue > 0 THEN BEGIN
         DecalerMantisseDroite(1, u);
         u[1] := retenue;
         FixerExposant(u, 1 + Exposant(u))
      END
   END; { de "FlottantParChiffre" }
```

L'opération «inverse» est la division d'un flottant par un chiffre :

```
PROCEDURE FlottantSurChiffre (VAR u: Flottant; x: chiffre);
   Divise u par le chiffre x > 0.
   VAR
      k: integer;
      retenue: chiffre;
   BEGIN
      IF EstFlottantNul(u) THEN
      ELSE BEGIN
         retenue := 0;
         FOR k := 1 TO TailleMax DO
            DivisionChiffres(u[k], x, u[k], retenue);
         NormaliserFlottant(u)
      END
   END; { de "FlottantSurChiffre" }
```

Il est utile, enfin, de disposer d'une procédure qui définit un flottant de valeur donnée :

```
PROCEDURE ChiffreEnFlottant (VAR u: Flottant; x: chiffre);
   Donne u = x, où x > 0 est un chiffre.
   BEGIN
      u := FlottantNul;
      FixerExposant(u, 1);
      u[1] := x;
   END; { de "ChiffreEnFlottant" }
```

Le *calcul de l'inverse* d'un flottant a se fait en utilisant la méthode de Newton. La formule d'itération est :

$$x_{n+1} = 2x_n - ax_n^2 \qquad (n \geq 0)$$

La seule difficulté est l'initialisation : on doit avoir $0 < |x_0| < 2/|a|$. Supposons $a > 0$, donné par

$$a = B^e \sum_{k=1}^{N} a_i B^{-i}$$

Alors $0 < a_1 < B$. Soit $x_0 = B^{-e}k$, avec $k = \lfloor B/(a_1+1) \rfloor$. Comme $a < B^{e-1}(a_1+1)$, on a

$$x_0 a < B^{-1}k(a_1+1) \leq B^{-1}\frac{B}{a_1+1}(a_1+1) = 1$$

Donc cette valeur de x_0 convient. La procédure suivante réalise le processus itératif :

```
PROCEDURE InverseFlottant (a: Flottant; VAR b: Flottant);
   BEGIN
      ChiffreEnFlottant(b, base DIV (a[1] + 1));
      FixerExposant(b, 1 - Exposant(a));          Valeur initiale d'itération,
      FixerSigne(b, LeSigne(a));
      FlottantInverseIteration(a, b)              puis les itérations.
   END; { de "InverseFlottant" }
```

Elle initialise un flottant à une valeur qui assure la convergence de la méthode de Newton. L'itération proprement dite se fait dans la procédure que voici :

```
PROCEDURE FlottantInverseIteration (a: Flottant; VAR x: Flottant);
   VAR
      y, z: Flottant;
   BEGIN
      REPEAT
         y := x;
         FlottantParChiffre(x, 2);                  x := 2x
         FlottantParFlottant(a, y, z);
         FlottantParFlottant(z, y, z);              z := ax^2
         FlottantMoinsFlottant(x, z, x);            x := 2x - ax^2
      UNTIL FlottantProches(y, x)
   END; { de "FlottantInverseIteration" }
```

Nous avons séparé les deux étapes parce qu'il arrive assez souvent qu'une valeur initiale meilleure que la nôtre soit connue grâce à un autre calcul. On arrête les itérations lorsque deux valeurs consécutives sont très proches. On peut choisir comme critère qu'elles ne diffèrent que sur le dernier chiffre ; nous sommes plus exigeants dans la réalisation suivante :

```
FUNCTION FlottantProches (VAR u, v: Flottant): boolean;
   VAR
      w: Flottant;
   BEGIN
      FlottantMoinsFlottant(u, v, w);
      FlottantProches := EstFlottantNul(w) OR
      ((Exposant(u) - Exposant(w) = TailleMax - 1) AND (w[1] <= base / 10))
   END; { de "FlottantProches" }
```

Le quotient de deux flottants s'obtient par :

```
PROCEDURE FlottantSurFlottant (u, v: Flottant; VAR w: Flottant);
   BEGIN
      InverseFlottant(v, w);
      FlottantParFlottant(u, w, w)
   END; { de  "FlottantSurFlottant" }
```

Voici deux exemples de calculs, (avec des résultats intermédiaires indiqués par le signe '='); on voit bien la convergence quadratique de la méthode de Newton, qui double pratiquement le nombre de chiffres exacts à chaque itération.

```
  a = 0. 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E 0
    = 0. 03 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E 1
    = 0. 03 33 30 00 00 00 00 00 00 00 00 00 00 00 00 00 E 1
    = 0. 03 33 33 33 30 00 00 00 00 00 00 00 00 00 00 00 E 1
    = 0. 03 33 33 33 33 33 33 33 30 00 00 00 00 00 00 00 E 1
    = 0. 03 33 33 33 33 33 33 33 33 33 33 33 33 33 33 34 E 1
    = 0. 03 33 33 33 33 33 33 33 33 33 33 33 33 33 33 34 E 1
1/a = 0. 03 33 33 33 33 33 33 33 33 33 33 33 33 33 33 34 E 1
  a = 0. 70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E 3
```

```
     = 0. 01 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E -2
     = 0. 01 41 70 00 00 00 00 00 00 00 00 00 00 00 00 00 E -2
     = 0. 01 42 84 77 70 00 00 00 00 00 00 00 00 00 00 00 E -2
     = 0. 01 42 85 71 42 24 21 89 70 00 00 00 00 00 00 00 E -2
     = 0. 01 42 85 71 42 85 71 42 85 44 95 68 54 44 49 74 E -2
     = 0. 01 42 85 71 42 85 71 42 85 71 42 85 71 42 85 72 E -2
     = 0. 01 42 85 71 42 85 71 42 85 71 42 85 71 42 85 72 E -2
 1/a = 0. 01 42 85 71 42 85 71 42 85 71 42 85 71 42 85 72 E -2
```

On voit l'influence des erreurs d'arrondis sur la dernière décimale : on devrait avoir 71 au lieu de 72, puisque le développement est périodique.

Pour calculer $1/\sqrt{a}$, on procède de même en traduisant la formule de récurrence :

$$x_{n+1} = (3x_n - ax_n^3)/2 \qquad (n \geq 0)$$

Là encore, il faut choisir une valeur initiale convenable. Considérons à nouveau

$$a = B^e(a_1B^{-1} + a_2B^{-2} + \cdots)$$

Si e est impair, alors $\sqrt{a} \simeq B^{(e-1)/2}\sqrt{a_1}$ et donc

$$\frac{1}{\sqrt{a}} \simeq B^{-\frac{e-1}{2}} \frac{B}{\sqrt{a_1}} B^{-1}$$

et si e est pair,

$$\frac{1}{\sqrt{a}} \simeq B^{-\frac{e-2}{2}} \frac{B}{\sqrt{a_1B + a_2}} B^{-1}$$

La procédure générale est :

```
PROCEDURE UnSurRacineFlottant (a: Flottant; VAR b: Flottant);
   VAR
      e: chiffre;
   BEGIN
      b := FlottantNul;
      e := Exposant(a);
      IF odd(e) THEN BEGIN
         FixerExposant(b, -(e - 1) DIV 2);
         b[1] := min(base - 1, round(base / sqrt(a[1])))
      END
      ELSE BEGIN
         FixerExposant(b, -(e - 2) DIV 2);
         b[1] := round(base / sqrt(a[1] * base + a[2]));
      END;
      FlottantUnSurRacineIteration(a, b)
   END; { de "UnSurRacineFlottant" }
```

```
PROCEDURE FlottantUnSurRacineIteration (a: Flottant; VAR x: Flottant);
```

Calcule $x = 1/\sqrt{a}$ à partir d'une valeur approchée x donnée.

```
   VAR
      y, z: Flottant;
   BEGIN
      REPEAT
         y := x;
         FlottantParChiffre(x, 3);                3x
         FlottantParFlottant(a, y, z);
         FlottantParFlottant(z, y, z);
         FlottantParFlottant(z, y, z);            z := ax^3
         FlottantMoinsFlottant(x, z, x);          x := 3x - ax^3
         FlottantSurChiffre(x, 2)
      UNTIL FlottantProches(y, x);
   END; { de "FlottantUnSurRacineIteration" }
```

et son application particulière donne :

```
PROCEDURE FlottantInverseRacine2 (VAR b: Flottant);
   VAR
      a: Flottant;
   BEGIN
      ChiffreEnFlottant(a, 2);       a := 2
      UnSurRacineFlottant(a, b)
   END; { de "FlottantInverseRacine2" }
```

Voici des résultats numériques :

```
 = 0. 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E 0
 = 0. 62 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E 0
 = 0. 69 33 59 37 50 00 00 00 00 00 00 00 00 00 00 00 E 0
 = 0. 70 67 08 46 84 96 79 94 68 99 41 40 62 50 00 00 E 0
 = 0. 70 71 06 44 46 95 90 70 75 51 17 30 67 65 93 00 E 0
 = 0. 70 71 06 78 11 86 30 73 35 92 54 35 93 12 38 00 E 0
 = 0. 70 71 06 78 11 86 54 75 24 40 08 44 23 97 25 00 E 0
 = 0. 70 71 06 78 11 86 54 75 24 40 08 44 36 21 05 00 E 0
 = 0. 70 71 06 78 11 86 54 75 24 40 08 44 36 21 05 00 E 0
1/sqrt(2) = 0. 70 71 06 78 11 86 54 75 24 40 08 44 36 21 05 00 E 0
```

Les deux derniers chiffres de $1/\sqrt{2}$ devraient être 04 85 au lieu de 05 00.

On peut calculer directement la racine carrée d'un flottant, par la suite décrite au paragraphe précédent. Cette suite fait une division à chaque itération, donc est lente. Par conséquent, on a intérêt à choisir une valeur de départ proche de la solution. Or, si

$$a = B^N \sum_{k=1}^{N} a_i B^{-i} = a_1 B^{N-1} + a_2 B^{N-2} + \cdots$$

alors

$$\sqrt{a} \simeq \begin{cases} \sqrt{a_1} B^{(N-1)/2} & \text{si } N \text{ est impair,} \\ \sqrt{a_1 B + a_2} B^{(N-2)/2} & \text{si } N \text{ est pair.} \end{cases}$$

On peut donc prendre cette valeur comme départ. Ceci est fait dans :

```
PROCEDURE RacineFlottant (a: Flottant; VAR b: Flottant);
   BEGIN
      IF odd(Exposant(a)) THEN BEGIN
         ChiffreEnFlottant(b, round(sqrt(a[1])));
         FixerExposant(b, 1 + (Exposant(a) - 1) DIV 2)
      END
      ELSE BEGIN
         ChiffreEnFlottant(b, round(sqrt(a[1] * base + a[2])));
         FixerExposant(b, 1 + (Exposant(a) - 2) DIV 2)
      END;
      FlottantRacineIteration(a, b);
   END; { de "RacineFlottant" }
```

avec

```
PROCEDURE FlottantRacineIteration (a: Flottant; VAR x: Flottant);
   VAR
      y: Flottant;
   BEGIN
      REPEAT
         y := x;
         FlottantSurFlottant(a, y, x);            a/x
         FlottantPlusFlottant(x, y, x);           x := x + a/x
         FlottantSurChiffre(x, 2)
         UNTIL FlottantProches(y, x)
   END; { de "FlottantRacineIteration" }
```

En fait, on a plutôt intérêt à calculer $\sqrt{a}$ en faisant le produit de a par $1/\sqrt{a}$. C'est nettement plus rapide.

12.3 Calcul de π par arctangente

12.3.1 Enoncé : calcul de π par arctangente

Le calcul de π avec beaucoup de décimales peut se faire au moyen de formules impliquant l'arctangente, comme par exemple

$$\begin{aligned}\frac{\pi}{4} &= \mathrm{Arctan}(1/2) + \mathrm{Arctan}(1/3) \\ &= 2\mathrm{Arctan}(1/3) + \mathrm{Arctan}(1/7) \\ &= 2\mathrm{Arctan}(1/4) + \mathrm{Arctan}(1/7) + 2\mathrm{Arctan}(1/13) \\ &= 5\mathrm{Arctan}(1/8) + 2\mathrm{Arctan}(1/18) + 3\mathrm{Arctan}(1/57)\end{aligned}$$

Les calculs se feront en multiprécision, dans une base b telle que b^2 soit encore un entier simple précision (par exemple $b = 100$) : un nombre y avec $|y| < b$ à p «chiffres» en base

b sera représenté par la suite $(y_0, \ldots, y_p)$ d'entiers tels que

$$y = \sum_{i=0}^{p} y_i b^{-i}$$

avec $0 \leq y_i < b$ pour $i = 1, \ldots, p$ et $0 \leq |y_0| < b$.

On pose

$$S_n(u) = \sum_{k=0}^{n} (-1)^k u^{2k+1}/(2k+1)$$

et

$$R_n(u) = |\mathrm{Arctan}(u) - S_n(u)|$$

1.– Démontrer que pour avoir p «chiffres» (en base b) de $\mathrm{Arctan}(1/x)$, où $x > 1$ est entier, il suffit de calculer $S_n(1/x)$ avec $n > (p \ln b / \ln x - 3)/2$.

2.– Démontrer que si $x < b$ et $2n + 1 < b$, $S_n(1/x)$ peut être évalué par le schéma de Horner, et que seules des multiplications ou divisions par des entiers b sont nécessaires.

3.– Ecrire une procédure qui prend en argument un entier $x > 1$ et un entier p, et qui calcule p chiffres (en base b) de $\mathrm{Arctan}(1/x)$.

4.– Ecrire un programme qui lit une suite $\alpha_1, x_1, \ldots, \alpha_k, x_k$ d'entiers, avec $1 < x_i < b$, et qui calcule p chiffres en base b de

$$4\Big(\sum_{i=1}^{k} \alpha_i \mathrm{Arctan}(1/x_i)\Big)$$

et l'utiliser pour calculer 20 décimales de π.

($\pi = 3.\ 1415\ 9265\ 3589\ 7932\ 3846\ 2643\ 3832\ldots$)

5.– Apporter les modifications nécessaires au programme pour pouvoir utiliser des formules comme celle de Machin :

$$\frac{\pi}{4} = 4\mathrm{Arctan}(1/5) - \mathrm{Arctan}(1/239)$$

dans lesquelles $\pi/4$ est combinaison linéaire d'expressions $\mathrm{Arctan}(1/x)$, où l'entier x est encore un entier simple précision, mais où son carré ne l'est plus nécessairement.

On pose $\bar{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$. On définit une bijection t de $\mathbb{R}/\pi\mathbb{Z}$ sur $\bar{\mathbb{R}}$ par

$$t(x) = \begin{cases} \tan x & \text{si } x \neq \pi/2 \\ \infty & \text{si } x = \pi/2 \end{cases}$$

et on pose, pour $x, y \in \bar{\mathbb{R}}$,

$$x \oplus y = t(t^{-1}(x) + t^{-1}(y))$$

6.– Démontrer que $(\bar{\mathbb{R}}, \oplus)$ est un groupe isomorphe à $(\mathbb{R}/\pi\mathbb{Z}, +)$.

7.– Démontrer que pour $x, y \in \mathbb{R}$ et $xy \neq 1$, on a $x \oplus y = \frac{x+y}{1-xy}$. Donner la valeur de $x \oplus y$ dans les autres cas.

8.– Démontrer que $1/x = 1/(x+1) \oplus 1/(x^2+x+1)$. On pose $F_0 = F_1 = 1$ et pour $n \geq 0$, $F_{n+2} = F_{n+1} + F_n$. Démontrer que pour $n > 0$, on a $1/F_{2n-1} = 1/F_{2n} \oplus 1/F_{2n+1}$.

9.– Dans cette question, les rationnels seront représentés par une paire (numérateur, dénominateur) d'entiers premiers entre eux. Ecrire une procédure qui, pour $x, y \in \mathbb{Q}$, avec $xy \neq 1$, calcule $x \oplus y$, et utiliser cette procédure pour vérifier les 3 premières des formules données au début du texte.

Soit $(O, \vec{\imath}, \vec{\jmath})$ un repère orthonormé du plan. Soient A et B deux points d'un cercle $\mathcal{C}$ de centre O. On note (MA, MB) l'angle orienté des droites MA et MB, où $M \neq A, B$ est un point quelconque de $\mathcal{C}$, et on note $\widehat{AB} = \tan(MA, MB)$.

a) Démontrer que si A est à coordonnées rationnelles et si $\widehat{AB}$ est rationnel, alors B est à coordonnées rationnelles.

b) Soient $a_1, \ldots, a_k$ des nombres rationnels tels que $1 = a_1 \oplus \cdots \oplus a_k$. Démontrer qu'il existe un cercle $\mathcal{C}$ de centre O et des points $A_0, A_1, \ldots, A_k$ à coordonnées entières du cercle tels que $a_i = \widehat{A_{i-1}A_i}$.

12.3.2 Solution : calcul de π par arctangente

Le calcul de π avec beaucoup de décimales peut se faire au moyen de formules impliquant l'arctangente, comme par exemple

$$\begin{aligned}
\frac{\pi}{4} &= \mathrm{Arctan}(1/2) + \mathrm{Arctan}(1/3) \\
&= 2\mathrm{Arctan}(1/3) + \mathrm{Arctan}(1/7) \\
&= 2\mathrm{Arctan}(1/4) + \mathrm{Arctan}(1/7) + 2\mathrm{Arctan}(1/13) \\
&= 5\mathrm{Arctan}(1/8) + 2\mathrm{Arctan}(1/18) + 3\mathrm{Arctan}(1/57)
\end{aligned}$$

La plus connue de ces formules est celle de Machin :

$$\frac{\pi}{4} = 4\mathrm{Arctan}(1/5) - \mathrm{Arctan}(1/239).$$

Notre but est d'expliquer comment obtenir de telles formules. On pose $\bar{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$. On définit une bijection t de $\mathbb{R}/\pi\mathbb{Z}$ sur $\bar{\mathbb{R}}$ par

$$t(x) = \begin{cases} \tan x & \text{si } x \neq \pi/2 \\ \infty & \text{si } x = \pi/2 \end{cases}$$

et on pose, pour $x, y \in \bar{\mathbb{R}}$,

$$x \oplus y = t(t^{-1}(x) + t^{-1}(y)) \tag{3.1}$$

Par transfert de structure, $\bar{\mathbb{R}}$ muni de la loi de composition interne $\oplus$ est un groupe commutatif isomorphe à $(\mathbb{R}/\pi\mathbb{Z}, +)$, d'élément neutre $t(0) = 0$. La table de la loi $\oplus$ est donnée dans la proposition suivante.

PROPOSITION 12.3.1. *Soient x, y deux réels tels que $xy \neq 1$. Alors*

$$x \oplus y = \frac{x+y}{1-xy} \tag{3.2}$$

De plus, pour x réel non nul, on a

$$\infty \oplus x = \frac{-1}{x} \quad \textit{et} \quad x \oplus \frac{1}{x} = \infty \tag{3.3}$$

et enfin

$$\infty \oplus \infty = 0 \quad \textit{et} \quad \infty \oplus 0 = \infty \tag{3.4}$$

Preuve. Le transfert, via (3.1), de la relation bien connue

$$\tan(\theta + \phi) = \frac{\tan\theta + \tan\phi}{1 - \tan\theta\tan\phi}$$

conduit à la relation (3.2). De même les relations (3.3) sont obtenues par transfert des relations

$$\tan(\pi/2 + \theta) = -\frac{1}{\tan(\theta)} \quad \text{et} \quad t(\theta + \pi/2 - \theta) = \infty$$

Enfin la dernière relation est obtenue par transfert de $\tan(\pi/2 + \pi/2) = \tan\pi = 0$. ■

De toute relation de la forme

$$1 = a_1 \oplus a_2 \cdots \oplus a_k \tag{3.5}$$

où les a_i sont rationels, on déduit une expression de $\pi/4$ comme combinaison linéaire d'arctangentes

$$\frac{\pi}{4} = \text{Arctan } a_1 + \cdots + \text{Arctan } a_k$$

(valable, naturellement, à π près), potentiellement utilisable pour le calcul de π en grande précision. Commençons par donner deux exemples, purement algébriques, de génération de formules du type (3.5).

PROPOSITION 12.3.2. *Pour x non nul on a*

$$\frac{1}{x} = \frac{1}{x+1} \oplus \frac{1}{x^2+x+1}$$

Preuve. En vertu de (3.2)

$$
\begin{aligned}
\frac{1}{x+1} \oplus \frac{1}{x^2+x+1} &= \frac{\dfrac{1}{x+1} + \dfrac{1}{x^2+x+1}}{1 - \dfrac{1}{x+1}\dfrac{1}{x^2+x+1}} \\
&= \frac{x^2+2x+2}{(x+1)(x^2+x+1)-1} \\
&= \frac{x^2+2x+2}{x^3+2x^2+2x} = \frac{1}{x}
\end{aligned}
$$

■

On obtient ainsi les relations

$$
\begin{aligned}
1 &= \frac{1}{2} \oplus \frac{1}{3} \\
&= \frac{1}{2} \oplus \frac{1}{4} \oplus \frac{1}{13} \\
&= \frac{1}{2} \oplus \frac{1}{4} \oplus \frac{1}{14} \oplus \frac{1}{183} \\
&= \dots
\end{aligned}
$$

Le second exemple est fondé sur la suite de Fibonacci (F_n), définie par la relation de récurrence

$$
\begin{cases} F_0 = F_1 = 1 \\ F_{n+2} = F_{n+1} + F_n \text{ pour } n \geq 0 \end{cases} \tag{3.6}
$$

Les premiers termes de cette suite sont : $1, 1, 2, 3, 5, 8, 13, 21, \dots$.

Proposition 12.3.3. *On pose $F_0 = F_1 = 1$ et pour $n \geq 0$, $F_{n+2} = F_{n+1} + F_n$. Alors*

$$
\frac{1}{F_{2n-1}} = \frac{1}{F_{2n}} \oplus \frac{1}{F_{2n+1}}.
$$

Preuve. La relation à démontrer est équivalente, en vertu de (3.2), à la relation :

$$
(F_{2n} + F_{2n+1})F_{2n-1} = F_{2n}F_{2n+1} - 1
$$

ou encore, en utilisant (3.6),

$$
F_{2n+2}F_{2n-1} - F_{2n}F_{2n-1} = -1
$$

Nous démontrons cette dernière relation par récurrence sur n. En utilisant la relation de récurrence (3.6), il vient :

$$
\begin{aligned}
F_{n+3}F_n - F_{n+1}F_{n+2} &= (F_{n+2} + F_{n+1})F_n - (F_n + F_{n-1})F_{n+2} \\
&= F_{n+1}F_n - F_{n-1}F_{n+2} \\
&= -(F_{n+2}F_{n-1} - F_nF_{n+1})
\end{aligned}
$$

d'où par itération

$$F_{n+3}F_n - F_{n+1}F_{n+2} = (-1)^n(F_3F_0 - F_1F_2) = (-1)^n \qquad \blacksquare$$

On obtient ainsi les relations

$$\begin{aligned} 1 &= \frac{1}{2} \oplus \frac{1}{3} \\ &= \frac{1}{2} \oplus \frac{1}{5} \oplus \frac{1}{8} \\ &= \frac{1}{2} \oplus \frac{1}{5} \oplus \frac{1}{13} \oplus \frac{1}{21} \\ &= \dots \end{aligned}$$

En fait la recherche de formule du type (3.5) est équivalente à la recherche des points à coordonnées rationnelles du cercle unité. Nous expliquons maintenant cette équivalence.

Soit $(O, \vec{\imath}, \vec{\jmath})$ un repère orthonormé direct du plan et soit $\mathcal{C}$ le cercle unité de centre O. Soient A et B deux points de $\mathcal{C}$ et soit C l'image de A par la rotation de centre O et d'angle $+\pi/2$. On note (MA, MB) l'angle orienté des droites MA et MB, où $M \neq A, B$ est un point quelconque de $\mathcal{C}$. En vertu de la proposition de l'arc capable, cet angle est indépendant du point M et est relié à l'angle orienté des vecteurs $\overrightarrow{OA}$ et $\overrightarrow{OB}$, noté $(\overrightarrow{OA}, \overrightarrow{OB})$, par la relation

$$(\overrightarrow{OA}, \overrightarrow{OB}) = 2(MA, MB).$$

Soit α une mesure de l'angle (MA, MB); les coordonnées du point B dans le repère orthonormé direct $(O, \overrightarrow{OA}, \overrightarrow{OC})$ sont alors $\cos 2\alpha$ et $\sin 2\alpha$; les relations trigonométriques

$$\cos 2\alpha = \frac{1 - \tan^2 \alpha}{1 + \tan^2 \alpha}, \quad \sin 2\alpha = \frac{2 \tan \alpha}{1 + \tan^2 \alpha} \quad \text{et} \quad \tan \alpha = \frac{1 - \cos 2\alpha}{\sin 2\alpha} \tag{3.7}$$

montrent clairement que $\tan \alpha$ est rationnel si et seulement si $\cos 2\alpha$ et $\sin 2\alpha$ sont rationnels; en d'autres termes si et seulement si les coordonnées de B, dans le repère $(O, \overrightarrow{OA}, \overrightarrow{OC})$, sont rationnelles.

Soit maintenant $A_1, \dots, A_{k-1}$ une suite de $k-1$ points à coordonnées rationnelles du cercle $\mathcal{C}$ et M un point de $\mathcal{C}$; soit A_k l'image de A_1 par la rotation de centre O et d'angle $+\pi/2$. En vertu de ce qui précéde les nombres $\widehat{A_iA_{i+1}} = \tan(MA_i, MA_{i+1})$, pour $i = 1, \dots, k-1$, sont rationnels. La relation suivante, obtenue par application de la relation de Chasles,

$$(MA_1, MA_2) + (MA_2, MA_3) + \cdots + (MA_{k-1}, MA_k) = (MA_1, MA_k)$$

conduit alors à la relation

$$\widehat{A_1A_2} \oplus \cdots \oplus \widehat{A_{k-1}A_k} = 1$$

En pratique il est préférable de travailler avec des points à coordonnées entières sur un cercle qui n'est alors plus de rayon unité; d'autre part il est bon d'observer que si $A = (x, y)$ et $B = (x', y')$ sont deux points du cercle alors

$$\widehat{AB} = \frac{x - x'}{y + y'}$$

Par exemple le cercle de rayon 5 contient les points $A_1 = (5, 0)$, $A_2 = (4, 3)$, $A_3 = (3, 4)$ et $A_4 = (0, 5)$; un simple calcul donne $\widehat{A_1A_2} = \widehat{A_3A_4} = 1/3$ et $\widehat{A_2A_3} = 1/7$; d'où la relation

$$1 = \frac{1}{3} \oplus \frac{1}{3} \oplus \frac{1}{7}$$

Réciproquement toute relation de type (3.5) peut être obtenue de la sorte.

Proposition 12.3.4. *Soient $a_1, \ldots, a_{k-1}$ des nombres rationnels tels que $1 = a_1 \oplus \cdots \oplus a_{k-1}$. Alors il existe des points $A_1, A_2, \ldots, A_k$ du cercle unité, à coordonnées rationnelles, tels que $a_i = \widehat{A_iA_{i+1}}$.*

Preuve. Soit A_1 le point de coordonnées $(1, 0)$ et définissons successivement les points $A_2, \ldots, A_k$ du cercle unité par

$$\widehat{A_1A_2} = a_1, \ldots, \widehat{A_{k-1}A_k} = a_{k-1}$$

En vertu des relations (3.7) la suite des points A_i est bien définie et les A_i sont à coordonnées rationnelles. ∎

L'analyse qui précède a montré l'équivalence du problème de la recherche des formules «à la Machin» et de la résolution l'équation diophantienne :

$$a^2 + b^2 = n^2 \tag{3.8}$$

Les entiers de Gauss (voir chapitre 11) permettent de résoudre cette équation.

Soit $n = i^{c(n)} z_1 z_2 \cdots z_k$ la décomposition de n en produit de facteurs irréductibles sur l'anneau des entiers de Gauss et soit Z le nombre complexe

$$Z = z'_1 \cdots z'_k$$

où z'_i est égal soit à z_i, soit à son conjugué $\overline{z}_i$. Les parties réelle et imaginaire a et b de Z sont alors solutions de l'équation (3.8); en effet on a

$$n^2 = Z\overline{Z} = a^2 + b^2$$

Par exemple de $65 = i^2(1 + 2i)(2 + i)(2 + 3i)(3 + 2i)$, que l'on préférera écrire sous la forme

$$65 = (1 + 2i)(1 - 2i)(2 + 3i)(2 - 3i)$$

qui met en évidence les facteurs conjugués, on obtient essentiellement quatre valeurs possibles de Z (à conjugaison et unité près)

$$\begin{aligned}
Z_1 &= (1+2i)(1-2i)(2+3i)(2-3i) = 65\\
Z_2 &= (1+2i)(1+2i)(2+3i)(2-3i) = -39+52i\\
Z_3 &= (1+2i)(1-2i)(2+3i)(2+3i) = -25+60i\\
Z_4 &= (1+2i)(1+2i)(2+3i)(2+3i) = -33-56i
\end{aligned}$$

d'où les décompositions

$$65^2 = 65^2+0^2 = 39^2+52^2 = 33^2+56^2 = 25^2+60^2$$

Posons $A_1 = (65,0), A_2 = (39,52), A_3 = (33,56), A_4 = (25,60)$ et $A_5 = (0,65)$; un simple calcul donne $\widehat{A_1A_2} = 1/2$, $\widehat{A_2A_3} = 1/18$, $\widehat{A_3A_4} = 2/29$, $\widehat{A_4A_5} = 1/5$ d'où la relation «à la Machin»

$$1 = \frac{1}{2} \oplus \frac{1}{18} \oplus \frac{2}{29} \oplus \frac{1}{5}$$

12.3.3 Programme : calcul de π par arctangente

On rappelle que

$$\operatorname{Arctan}(u) = \sum_{k=0}^{\infty} (-1)^k u^{2k+1}/(2k+1) \qquad |u| \le 1$$

Avec

$$S_n(u) = \sum_{k=0}^{n} (-1)^k u^{2k+1}/(2k+1)$$

l'erreur $R_n(u) = |\operatorname{Arctan}(u) - S_n(u)|$ est majorée par le premier terme négligé, parce que la série est alternée. On a donc

$$R_n(u) < \frac{|u|^{2n+3}}{2n+3}$$

Pour que ce nombre soit inférieur à b^{-p}, il suffit que $|u|^{2n+3} < b^{-p}$ et, avec $u = 1/x$, cela conduit à

$$n \ge \lceil (p \ln b / \ln x - 3)/2 \rceil$$

Le calcul de $S_n(1/x)$ peut se faire par le schéma suivant, proche du schéma de Horner : on pose

$$t_n = \frac{2n-1}{2n+1}\frac{1}{x^2}$$

et

$$t_k = \frac{2k-1}{2k+1}\frac{1}{x^2}(1 - t_{k+1})$$

pour $1 \leq k \leq n-1$. Alors

$$S_n(1/x) = \frac{1}{x}(1 - t_1)$$

La réalisation de ces calculs se fait en utilisant le type `entier` que nous avons défini au début de ce chapitre. Plus précisément, tout nombre réel r, avec $|r| < b$, est remplacé par l'entier $\lfloor rb^N \rfloor$, où N est la taille maximale choisie. Les opérations sur ces réels sont approchées par les opérations sur les entiers associés. Ainsi, l'évaluation de $S_n(1/x)$ se réalise comme suit :

```
PROCEDURE atan (x: chiffre; n: integer; VAR a: entier);        Version simple.
   Calcule a = Arctan(1/x) en évaluant les n premiers termes du développement.
   VAR
      k: integer;
   BEGIN
      EntierNul(a);
      FOR k := 0 TO TailleMax - 1 DO
         a[k] := 0;
      a[TailleMax] := 1;                         a := 1 · b^TailleMax.
      FixerTaille(a, TailleMax);
      FixerSigne(a, 1);
      FOR k := n DOWNTO 1 DO BEGIN
         NaturelParChiffre(a, 2 * k - 1, a);     a := (2k-1)/(2k+1) a.
         NaturelSurChiffre(a, 2 * k + 1, a);
         NaturelSurChiffre(a, x, a);             a := a/x^2.
         NaturelSurChiffre(a, x, a);
         ComplementerNaturel(a, a)               a := 1 · b^TailleMax - a.
      END;
      NaturelSurChiffre(a, x, a);                a := a/x.
   END; { de "atan" }
```

La procédure de complémentation s'écrit :

```
PROCEDURE ComplementerNaturel (a: entier; VAR ma: entier);
   VAR
      k: integer;
   BEGIN
      ma[0] := base - a[0];
      FOR k := 1 TO TailleMax - 1 DO
         ma[k] := base - 1 - a[k];
      ma[TailleMax] := 0;
      k := TailleMax - 1;
      WHILE (k >= 0) AND (ma[k] = 0) DO
         k := k - 1;
      FixerTaille(ma, k)
   END; { de "ComplementerNaturel" }
```

On peut accélérer le calcul de l'arctangente, en essayant de grouper des divisions par des chiffres si c'est possible, en particulier si $x^2 < b$, où b est la base. Voici une façon de faire :

```
PROCEDURE atan (x: chiffre; n: integer; VAR a: entier);     Version accélérée.
   VAR
      k: integer;
      y: chiffre;
   BEGIN
      EntierNul(a);
      FOR k := 0 TO TailleMax - 1 DO
         a[k] := 0;
      a[TailleMax] := 1;
      FixerTaille(a, TailleMax);
      FixerSigne(a, 1);
      IF sqr(x) < base THEN BEGIN
         y := sqr(x);
         FOR k := n DOWNTO 1 DO BEGIN
            NaturelParChiffre(a, 2 * k - 1, a);
            IF y * (2 * k + 1) < base THEN
               NaturelSurChiffre(a, y * (2 * k + 1), a)
            ELSE BEGIN
               NaturelSurChiffre(a, 2 * k + 1, a);
               NaturelSurChiffre(a, y, a)
            END;
            ComplementerNaturel(a, a)
         END
      END
      ELSE
         FOR k := n DOWNTO 1 DO BEGIN
            NaturelParChiffre(a, 2 * k - 1, a);
            NaturelSurChiffre(a, 2 * k + 1, a);
            NaturelSurChiffre(a, x, a);
            NaturelSurChiffre(a, x, a);
            ComplementerNaturel(a, a)
         END;
      NaturelSurChiffre(a, x, a);
   END; { de "atan" }
```

Pour le calcul de π comme combinaisons linéaires d'arctangentes, on doit évaluer

$$4\Big(\sum_{i=1}^{k} \alpha_i \mathrm{Arctan}(1/x_i)\Big)$$

où $\alpha_1, x_1, \ldots, \alpha_k, x_k$ sont des entiers, avec $1 < x_i < b$. Le nombre de termes du développement de $\mathrm{Arctan}(1/x_i)$ à calculer pour avoir p places dépend bien entendu de x_i, et est donc évalué séparément :

```
PROCEDURE CalculPi (places, n: integer; alpha, x: Coefficients;
   VAR p: entier);
   Calcul de π par l'expression 4(α1 Arctan(x1) + ··· + αn Arctan(xn)).
   VAR
```

```
      k, m: integer;
      a: entier;
   BEGIN
      EntierNul(p);
      FOR k := 1 TO n DO BEGIN
         m := 1 + trunc(((places * ln(base)) / ln(x[k]) - 3) / 2);
         atan(x[k], m, a);                         Calcul de Arctan(x_k).
         EntierParChiffre(a, alpha[k], a);         Multiplication par α_k.
         EntierPlusEntier(p, a, p);                Sommation.
      END;
      NaturelParChiffre(p, 4, p);                  Multiplication finale par 4.
   END; { de "CalculPi" }
```

Voici 2 000 décimales de π, calculées sur un Macintosh IIcx, en choisissant la base $b = 10\,000$ et en utilisant la formule

$$\frac{\pi}{4} = 12\mathrm{Arctan}(1/18) + 8\mathrm{Arctan}(1/57) - 5\mathrm{Arctan}(1/239)$$

formule qui a déjà été donnée par Gauss :

```
3 1415 9265 3589 7932 3846 2643 3832 7950 2884 1971
  6939 9375 1058 2097 4944 5923 0781 6406 2862 0899
  8628 0348 2534 2117 0679 8214 8086 5132 8230 6647
  0938 4460 9550 5822 3172 5359 4081 2848 1117 4502
  8410 2701 9385 2110 5559 6446 2294 8954 9303 8196
  4428 8109 7566 5933 4461 2847 5648 2337 8678 3165
  2712 0190 9145 6485 6692 3460 3486 1045 4326 6482
  1339 3607 2602 4914 1273 7245 8700 6606 3155 8817
  4881 5209 2096 2829 2540 9171 5364 3678 9259 0360
  0113 3053 0548 8204 6652 1384 1469 5194 1511 6094
  3305 7270 3657 5959 1953 0921 8611 7381 9326 1179
  3105 1185 4807 4462 3799 6274 9567 3518 8575 2724
  8912 2793 8183 0119 4912 9833 6733 6244 0656 6430
  8602 1394 9463 9522 4737 1907 0217 9860 9437 0277
  0539 2171 7629 3176 7523 8467 4818 4676 6940 5132
  0005 6812 7145 2635 6082 7785 7713 4275 7789 6091
  7363 7178 7214 6844 0901 2249 5343 0146 5495 8537
  1050 7922 7968 9258 9235 4201 9956 1121 2902 1960
  8640 3441 8159 8136 2977 4771 3099 6051 8707 2113
  4999 9998 3729 7804 9951 0597 3173 2816 0963 1859
  5024 4594 5534 6908 3026 4252 2308 2533 4468 5035
  2619 3118 8171 0100 0313 7838 7528 8658 7533 2083
  8142 0617 1776 6914 7303 5982 5349 0428 7554 6873
  1159 5628 6388 2353 7875 9375 1957 7818 5778 0532
  1712 2680 6613 0019 2787 6611 1959 0921 6420 1989
  3809 5257 2010 6548 5863 2788 6593 6153 3818 2796
  8230 3019 5203 5301 8529 6899 5773 6225 9941 3891
  2497 2177 5283 4791 3151 5574 8572 4245 4150 6959
```

```
5082 9533 1168 6172 7855 8890 7509 8381 7546 3746
4939 3192 5506 0400 9277 0167 1139 0098 4882 4012
8583 6160 3563 7076 6010 4710 1819 4295 5596 1989
4676 7837 4494 4825 5379 7747 2684 7104 0475 3464
6208 0466 8425 9069 4912 9331 3677 0289 8915 2104
7521 6205 6966 0240 5803 8150 1935 1125 3382 4300
3558 7640 2474 9647 3263 9141 9927 2604 2699 2279
6782 3547 8163 6009 3417 2164 1219 9245 8631 5030
2861 8297 4555 7067 4983 8505 4945 8858 6926 9956
9092 7210 7975 0930 2955 3211 6534 4987 2027 5596
0236 4806 6549 9119 8818 3479 7753 5663 6980 7426
5425 2786 2551 8184 1757 4672 8909 7777 2793 8000
8164 7060 0161 4524 9192 1732 1721 4772 3501 4144
1973 5685 4816 1361 1573 5255 2133 4757 4184 9468
4385 2332 3907 3941 4333 4547 7624 1686 2518 9835
6948 5562 0992 1922 2184 2725 5025 4256 8876 7179
0494 6016 5346 6804 9886 2723 2791 7860 8578 4383
8279 6797 6681 4541 0095 3883 7863 6095 0680 0642
2512 5205 1173 9298 4896 0841 2848 8626 9456 0424
1965 2850 2221 0661 1863 0674 4278 6220 3919 4945
0471 2371 3786 9609 5636 4371 9172 8746 7764 6575
7396 2413 8908 6583 2645 9958 1339 0478 0275 8968
Le calcul a pris 87.717 secondes
```

La valeur exacte des 4 derniers chiffres est 9009. Le calcul de Arctan(1/18) a demandé l'évaluation de 796 termes, celui de Arctan(1/57) le calcul de 569 termes et celui de Arctan(1/239) le calcul de 419 termes.

12.4 La formule de Brent-Salamin

12.4.1 Enoncé : la formule de Brent-Salamin

Soient a, b deux nombres réels positifs tels que $a > b$. On définit deux suites (a_n),(b_n) pour $n \in \mathbb{N}$ par $a_0 = a$, $b_0 = b$ et, pour $n \geq 1$,

$$\begin{aligned} a_n &= (a_{n-1} + b_{n-1})/2 \\ b_n &= \sqrt{a_{n-1} b_{n-1}} \end{aligned}$$

1.– Démontrer que les suites (a_n) et (b_n) admettent une limite commune. Cette limite est notée $\mu(a, b)$.

2.– On pose $c_n = \sqrt{a_n^2 - b_n^2}$. Démontrer que

$$\begin{aligned} c_{n+1} &= (a_n - b_n)/2 \qquad & a_{n+1} + c_{n+1} &= a_n \\ a_{n+1} c_{n+1} &= c_n^2/4 & a_{n+1} - c_{n+1} &= b_n \end{aligned}$$

En déduire que la vitesse de convergence de la suite (c_n) est quadratique.

3.– On pose $\Delta(a,b,\phi) = (a^2\cos^2\phi + b^2\sin^2\phi)^{+1/2}$, puis

$$I(a,b) = \int_0^{\pi/2} 1/\Delta(a,b,\phi)\,d\phi \quad \text{et} \quad J(a,b) = \int_0^{\pi/2} \Delta(a,b,\phi)\,d\phi$$

En utilisant les égalités

$$I(a,b) = I(a_1,b_1) \qquad (*)$$

$$J(a,b) + abI(a,b) = 2J(a_1,b_1) \qquad (**)$$

démontrer que

$$I(a,b) = \frac{\pi}{2\mu(a,b)} \quad \text{et} \quad J(a,b) = \left(a^2 - \frac{1}{2}\sum_{n=0}^{\infty} 2^n c_n^2\right) I(a,b)$$

4.– Utiliser l'égalité

$$2I(1,\sqrt{2})J(1,1/\sqrt{2}) - I^2(1,\sqrt{2}) = \pi/2$$

pour démontrer la formule

$$\pi = \frac{4\mu^2(1,1/\sqrt{2})}{1 - \sum_{j=1}^{\infty} 2^{j+1}c_j^2}$$

5.– Utiliser l'approximation

$$\pi_n = \frac{4a_{n+1}^2}{1 - \sum_{j=1}^{n} 2^{j+1}c_j^2}$$

de π pour écrire un programme qui calcule π à N décimales.

6.– On se propose maintenant de démontrer $(*)$ et $(**)$. A cette fin on utilise le changement de variable

$$\tan(\phi_1 - \phi) = \frac{b}{a}\tan\phi$$

a) Démontrer les relations

$$\begin{aligned}
a_1\cos(2\phi-\phi_1) - c_1\cos\phi_1 &= ab/\Delta(a,b,\phi)\\
a_1\cos(2\phi-\phi_1) + c_1\cos\phi_1 &= \Delta(a,b,\phi)\\
a_1\sin(2\phi-\phi_1) - c_1\sin\phi_1 &= 0\\
a_1\cos(2\phi-\phi_1) &= \Delta(a_1,b_1,\phi_1)
\end{aligned}$$

b) En déduire que ϕ et ϕ_1 vérifient l'équation différentielle

$$2\Delta(a_1,b_1,\phi_1)\,d\phi = \Delta(a,b,\phi)\,d\phi_1$$

c) Démontrer $(*)$ puis $(**)$.

12.4.2 Solution : la formule de Brent-Salamin

En 1976, Brent et Salamin ont observé, de manière indépendante, que l'algorithme classique de calcul des intégrales elliptiques, utilisant la moyenne arithmético-géométrique, peut être combiné avec l'identité de Legendre pour fournir un algorithme quadratique de calcul de π. La mise en œuvre de cet algorithme a conduit au début des années 80 à la détermination de 16 millions de décimales de π. Actuellement, on en connaît un milliard de décimales.

La découverte et l'étude de la moyenne arithmético-géométrique sont associées aux noms de Lagrange et de Gauss. La connexion avec les intégrales elliptiques est en particulier due au mathématicien anglais du dix-huitième siècle John Landen, au travers de la transformation qui porte son nom et qui nous servira de cheval de bataille pour établir deux des trois formules clés conduisant à l'expression de π utilisée par Brent et Salamin.

Nous commençons par introduire la moyenne arithmético-géométrique.

Soient a et b deux réels positifs tels que $a > b$. Les suites (a_n) et (b_n) des moyennes arithmétiques et géométriques sont définies par la récurrence :

$$\begin{array}{rclrcl} a_0 &=& a & b_0 &=& b \\ a_1 &=& \frac{1}{2}(a_0+b_0) & b_1 &=& \sqrt{a_0 b_0} \\ &\vdots& & &\vdots& \\ a_{n+1} &=& \frac{1}{2}(a_n+b_n) & b_{n+1} &=& \sqrt{a_n b_n} \\ &\vdots& & &\vdots& \end{array} \tag{4.1}$$

Observons, en usant en particulier de l'inégalité de la moyenne $\sqrt{ab} < \frac{a+b}{2}$, que

$$b < b_1 < a_1 < a$$

et que

$$a_1 - b_1 \leq a_1 - b = \frac{1}{2}(a-b)$$

D'où par itération

$$b < b_1 < b_2 < b_3 < \cdots < a_3 < a_2 < a_1 < a$$

et

$$a_n - b_n \leq \left(\frac{1}{2}\right)^n (a-b)$$

Ainsi, (b_n) est croissante majorée et (a_n) est décroissante minorée. Les deux suites sont alors convergentes et la dernière inégalité montre qu'elles admettent la même limite. Par définition la limite commune des suites (a_n) et (b_n), notée $\mu(a,b)$, est *la moyenne arithmético-géométrique* de a et b.

Pour évaluer la rapidité de convergence des suites (a_n) et (b_n) nous introduisons la suite (c_n), à termes positifs, définie par

$$c_n^2 = a_n^2 - b_n^2 \tag{4.2}$$

PROPOSITION 12.4.1. *Les suites $(a_n), (b_n)$ et (c_n) sont liées par les relations*

$$\begin{array}{ll} c_{n+1} \quad = (a_n - b_n)/2 & a_{n+1} + c_{n+1} = a_n \\ a_{n+1}c_{n+1} = c_n^2/4 & a_{n+1} - c_{n+1} = b_n \end{array} \tag{4.3}$$

Preuve. En effet, en utilisant (4.1), il vient

$$c_{n+1}^2 = a_{n+1}^2 - b_{n+1}^2 = \frac{1}{4}(a_n + b_n)^2 - a_n b_n = \frac{1}{4}(a_n - b_n)^2$$

Puis

$$a_{n+1} + c_{n+1} = \frac{1}{2}(a_n + b_n) + \frac{1}{2}(a_n - b_n) = a_n$$

$$a_{n+1} - c_{n+1} = \frac{1}{2}(a_n + b_n) - \frac{1}{2}(a_n - b_n) = b_n$$

et enfin

$$a_{n+1}c_{n+1} = \frac{1}{4}(a_n + b_n)(a_n - b_n) = \frac{1}{4}c_n^2$$

∎

Cette dernière égalité montre que la convergence de la suite (c_n) (et par suite également la convergence des suites (a_n) et (b_n)) est quadratique ou encore du second ordre; en effet, en usant de la décroissance de la suite (a_n), il vient

$$c_{n+1} \leq \frac{1}{4\mu(a,b)} c_n^2$$

Plus généralement la convergence d'une suite (u_n) de limite l est dite d'ordre $m \geq 1$ s'il existe une constante $C > 0$ telle que

$$|u_{n+1} - l| \leq C|u_n - l|^m$$

Pour C voisin de 1 une itération fait passer de la précision 10^{-p} à la précision 10^{-mp}. Les suites $(a_n), (b_n)$ et (c_n) obtenues pour $a = 1, b = 1/\sqrt{2}$ permettent de calculer π en utilisant la formule de Brent-Salamin :

$$\pi = \frac{4\mu^2(1, 1/\sqrt{2})}{1 - \displaystyle\sum_{n=1}^{\infty} 2^{n+1} c_n^2}$$

Les intégrales elliptiques du premier et du second ordre sont définies par

$$I(a,b) = \int_0^{\pi/2} 1/\Delta(a,b,\phi)\, d\phi \quad \text{et} \quad J(a,b) = \int_0^{\pi/2} \Delta(a,b,\phi)\, d\phi \tag{4.4}$$

où

$$\Delta(a,b,\phi) = (a^2 \cos^2\phi + b^2 \sin^2\phi)^{+1/2} \tag{4.5}$$

Les deux théorèmes suivants établissent le lien entre la moyenne arithmético-géométrique et les intégrales elliptiques du premier et du second ordre.

Théorème 12.4.2. *Soient $a > b > 0$. Alors*

$$I(a,b) = I(a_1, b_1) \tag{4.6}$$

et

$$J(a,b) + abI(a,b) = 2J(a_1, b_1) \tag{4.7}$$

où a_1 et b_1 sont définis par (4.1).

Théorème 12.4.3. *Soient $a > b > 0$. Alors*

$$I(a,b) = \frac{\pi}{2\mu(a,b)} \quad \text{et} \quad J(a,b) = \left(a^2 - \frac{1}{2}\sum_{n=0}^{\infty} 2^n c_n^2\right) I(a,b)$$

où c_n est définie par (4.2).

Pour établir le premier théorème nous introduisons la transformation de Landen qui associe au réel ϕ, de l'intervalle $[0, \pi/2]$, le réel ϕ_1 défini par

$$\phi_1 = \phi + \operatorname{Arctan}\left(\frac{b}{a}\tan\phi\right) \tag{4.8}$$

Des propriétés élémentaires de la fonction tangente, nous déduisons que la transformation de Landen est un difféomorphisme croissant de l'intervalle $[0, \pi/2]$ sur l'intervalle $[0, \pi]$. Observons que $(\phi_1 - \phi)$ est l'argument du nombre complexe $a\cos\phi + i\, b\sin\phi$

$$a\cos\phi + i\, b\sin\phi = \Delta(a,b,\phi)\, e^{i(\phi_1 - \phi)}$$

La proposition suivante rassemble les propriétés utiles à notre propos de la transformation de Landen.

Proposition 12.4.4. *La transformation de Landen $\phi \mapsto \phi_1$ vérifie les équations*

$$\begin{aligned}
a_1\cos(2\phi - \phi_1) - c_1\cos\phi_1 &= ab/\Delta(a,b,\phi)\\
a_1\cos(2\phi - \phi_1) + c_1\cos\phi_1 &= \Delta(a,b,\phi)\\
a_1\sin(2\phi - \phi_1) - c_1\sin\phi_1 &= 0\\
a_1\cos(2\phi - \phi_1) &= \Delta(a_1,b_1,\phi_1)
\end{aligned} \tag{4.9}$$

ainsi que l'équation différentielle

$$\frac{d\phi}{\Delta(a,b,\phi)} = \frac{1}{2}\frac{d\phi_1}{\Delta(a_1,b_1,\phi_1)} \tag{4.10}$$

Preuve. Pour simplifier les notations posons $\Delta = \Delta(a,b,\phi)$ et $\Delta_1 = \Delta(a_1,b_1,\phi_1)$.

Les deux premières équations se déduisent l'une de l'autre, sous l'hypothèse de validité de la quatrième équation, en remarquant que le produit de leurs membres gauches égale ab; en effet

$$\Delta_1^2 - (c_1\cos\phi_1)^2 = (a_1^2 - c_1^2)\cos^2\phi_1 + b_1^2\sin^2\phi_1 = b_1^2 = ab$$

D'autre part la quatrième équation se déduit de la troisième, comme le montre le calcul suivant

$$a_1^2 \cos^2(2\phi - \phi_1) = a_1^2 - a_1^2 \sin^2(2\phi - \phi_1) = a_1^2 - c_1^2 \sin^2 \phi_1 = \Delta_1^2$$

et le fait que $\cos(2\phi - \phi_1) \geq 0$ car $(2\phi - \phi_1)$ appartient à $[0, \pi/2]$. Il suffit donc de démontrer, par exemple, la deuxième et la troisième équation. Pour cela nous observons que leurs membres gauches sont, respectivement, la partie réelle et la partie imaginaire du nombre complexe $a_1\, e^{i(2\phi-\phi_1)} + c_1\, e^{-i\phi_1}$. Or ce nombre complexe coïncide avec Δ ; en effet

$$\begin{aligned}
a_1\, e^{i(2\phi-\phi_1)} + c_1\, e^{-i\phi_i} &= e^{i(\phi-\phi_1)} \left\{a_1\, e^{i\phi} + c_1\, e^{-i\phi}\right\} \\
&= e^{i(\phi-\phi_1)} \left\{(a_1 + c_1) \cos\phi + i(a_1 - c_1) \sin\phi\right\} \\
&= e^{i(\phi-\phi_1)} \left\{a \cos\phi + ib \sin\phi\right\} \\
&= e^{i(\phi-\phi_1)}\, \Delta\, e^{i(\phi_1-\phi)} \\
&= \Delta
\end{aligned}$$

ce qui termine la preuve de la première partie de la proposition.

Pour démontrer la seconde partie nous différentions la troisième équation de (4.9) ; il vient, après regroupement des termes en $d\phi$ et $d\phi_1$,

$$2a_1 \cos(2\phi - \phi_1)\, d\phi = (c_1 \cos\phi_1 + a_1 \cos(2\phi - \phi_1))\, d\phi_1$$

qui s'écrit encore, toujours en utilisant (4.9), $2\Delta_1\, d\phi = \Delta\, d\phi_1$, ce qui termine la preuve de la proposition. ■

Preuve du théorème 12.4.2. En intégrant l'équation différentielle (4.10) pour $\phi \in [0, \pi/2]$ (et par suite $\phi_1 \in [0, \pi]$) nous obtenons la première identité

$$I(a, b) = \int_0^{\pi/2} \frac{d\phi}{\Delta(a, b, \phi)} = \frac{1}{2} \int_0^{\pi} \frac{d\phi_1}{\Delta(a_1, b_1, \phi_1)} = \int_0^{\pi/2} \frac{d\phi_1}{\Delta(a_1, b_1, \phi_1)} = I(a_1, b_1)$$

La seconde identité s'obtient de manière similaire ; posons, pour simplifier les notations, $\Delta = \Delta(a, b, \phi)$ et $\Delta_1 = \Delta(a_1, b_1, \phi_1)$. L'équation différentielle $2\Delta_1\, d\phi = \Delta\, d\phi_1$ s'écrit encore, en utilisant $2\Delta_1 = \Delta + ab/\Delta$ et $\Delta = \Delta_1 + c_1 \cos\phi_1$ (cf. 4.9),

$$(\Delta + ab/\Delta)\, d\phi = (\Delta_1 + c_1 \cos\phi_1)\, d\phi_1$$

que nous intégrons pour $\phi \in [0, \pi/2]$ (et par suite $\phi_1 \in [0, \pi]$) pour obtenir la seconde identité

$$J(a, b) + abI(a, b) = 2J(a_1, b_1)$$

■

Preuve du théorème 12.4.3. Par itération de la relation (4.6) nous écrivons

$$I(a, b) = I(a_1, b_1) = \cdots = I(a_n, b_n)$$

La fonction $I(a,b)$ étant continue par rapport à ses paramètres, il vient

$$I(a,b) = \lim_{n\to\infty} I(a_n,b_n) = I(\mu(a,b),\mu(a,b)) = \frac{\pi}{2\mu(a,b)}$$

Pour la seconde égalité nous transformons (4.7), en utilisant $a^2 - 2a_1^2 = 1/2c_0^2 - ab$, en

$$\left\{J(a,b) - a^2 I(a,b)\right\} - 2\left\{J(a_1,b_1) - a_1^2 I(a_1,b_1)\right\} = -\frac{1}{2}c_0^2 I(a,b)$$

La même égalité vaut si l'on substitue a_n, b_n à a, b, et a_{n+1}, b_{n+1} à a_1, b_1. En combinant les égalités ainsi obtenues pour n variant de 0 à l'infini il vient

$$\begin{aligned}\left\{J(a,b) - a^2 I(a,b)\right\} - A &= -\frac{1}{2}\sum_{n=0}^{\infty} 2^n c_n^2 I(a_n,b_n)\\ &= \left\{-\frac{1}{2}\sum_{n=0}^{\infty} 2^n c_n^2\right\} I(a,b)\end{aligned}$$

où

$$\begin{aligned}A &= \lim_{n\to\infty} 2^n \left\{J(a_n,b_n) - a_n^2 I(a_n,b_n)\right\}\\ &= \lim_{n\to\infty}\int_0^{\pi/2} \frac{2^n c_n^2 \sin^2\phi}{(a_n^2\cos^2\phi + b_n^2\sin^2\phi)^{1/2}}\,d\phi\\ &= \lim_{n\to\infty} O(2^n c_n^2) = 0\end{aligned}$$

car la convergence de c_n vers 0 est quadratique. Il suffit alors de résoudre en $J(a,b)$ pour terminer la démonstration. ∎

La troisième et dernière formule nécessaire pour obtenir celle de Brent-Salamin est la formule de Legendre sur les intégrales elliptiques.

Théorème 12.4.5. *Pour $|x| < 1$, posons*

$$K(x) = \int_0^{\pi/2} (1 - x^2\sin^2\phi)^{-1/2} d\phi \quad \text{et} \quad E(x) = \int_0^{\pi/2} (1 - x^2\sin^2\phi)^{1/2} d\phi$$

et $x' = \sqrt{1-x^2}$. Alors

$$K(x)E(x') + K(x')E(x) - K(x)K(x') = \frac{\pi}{2} \tag{4.11}$$

Preuve. Soit $c = x^2$ et $c' = 1 - c$ et posons $E = E(x)$, $K = K(x)$. Un simple calcul donne

$$\begin{aligned}\frac{d}{dc}(E-K) &= -\frac{d}{dc}\int_0^{\pi/2} \frac{c\sin^2\phi}{(1 - c\sin^2\phi)^{1/2}} d\phi\\ &= \frac{E}{2c} - \frac{1}{2c}\int_0^{\pi/2} \frac{1}{(1 - c\sin^2\phi)^{3/2}} d\phi\end{aligned}$$

De
$$\frac{d}{d\phi}\left(\frac{\sin\phi\cos\phi}{(1-c\sin^2\phi)^{1/2}}\right)=\frac{1}{c}(1-c\sin^2\phi)^{1/2}-\frac{c'}{c}(1-c\sin^2\phi)^{-3/2}$$
nous déduisons que
$$\frac{d}{dc}(E-K)=\frac{E}{2c}-\frac{E}{2cc'}+\frac{1}{2c'}\int_0^{\pi/2}\frac{d}{d\phi}\left(\frac{\sin\phi\cos\phi}{(1-c\sin^2\phi)^{1/2}}\right)d\phi \tag{4.12}$$
$$=\frac{E}{2c}\left(1-\frac{1}{c'}\right)=-\frac{E}{2c'} \tag{4.13}$$
Pour simplifier les notations, posons $K'=K(x')$ et $E'=E(x')$. De $c'=1-c$, il suit que
$$\frac{d}{dc}(E'-K')=\frac{E'}{2c} \tag{4.14}$$
Enfin, des calculs simples conduisent à
$$\frac{dE}{dc}=\frac{E-K}{2c}\quad\text{et}\quad\frac{dE'}{dc}=-\frac{E'-K'}{2c'} \tag{4.15}$$
Si G désigne le membre gauche de (4.11), nous pouvons écrire G sous la forme
$$G=EE'-(E-K)(E'-K')$$
En usant de (4.12-4.15), nous trouvons que
$$\frac{dG}{dc}=\frac{(E-K)E'}{2c}-\frac{E(E'-K')}{2c'}+\frac{E(E'-K')}{2c'}-\frac{(E-K)E'}{2c}=0$$
Ainsi, G est une constante et nous déterminons sa valeur en faisant tendre c vers 0. Premièrement,
$$E-K=-c\int_0^{\pi/2}\frac{\sin^2\phi}{(1-c\sin^2\phi)^{1/2}}d\phi=O(c)$$
lorsque c tend vers 0. Puis,
$$K'=\int_0^{\pi/2}(1-c'\sin^2\phi)^{-1/2}d\phi\le\int_0^{\pi/2}(1-c')^{-1/2}d\phi=O(c^{-1/2})$$
lorsque c tend vers 0. Enfin,
$$\lim_{c\to0}G=\lim_{c\to0}\{(E-K)K'+E'K\}$$
$$=\lim_{c\to0}\left\{O(c^{1/2})+1\cdot\frac{\pi}{2}\right\}=\frac{\pi}{2}$$
et la preuve est complète. ∎

Nous sommes maintenant en mesure de démontrer la formule de Brent-Salamin.

Théorème 12.4.6.

$$\pi = \frac{4\mu^2(1, 1/\sqrt{2})}{1 - \sum_{n=1}^{\infty} 2^{n+1} c_n^2} \tag{4.16}$$

où (c_n) est définie par (4.2).

Preuve. Prenons $x = x' = 1/\sqrt{2}$ dans le théorème 12.4.5, nous trouvons que

$$2K(\frac{1}{\sqrt{2}})E(\frac{1}{\sqrt{2}}) - K^2(\frac{1}{\sqrt{2}}) = \frac{\pi}{2} \tag{4.17}$$

D'autre part, en posant $a = 1$ et $b = 1/\sqrt{2}$ dans le théorème 12.4.3, on voit que

$$E\left(\frac{1}{\sqrt{2}}\right) = \left(1 - \frac{1}{2}\sum_{n=0}^{\infty} 2^n c_n^2\right) K\left(\frac{1}{\sqrt{2}}\right) \tag{4.18}$$

car $I(1, \sqrt{2}) = K(1/\sqrt{2})$ et $J(1, 1/\sqrt{2}) = E(1/\sqrt{2})$. Enfin, en usant du théorème 12.4.3,

$$\mu(1, 1/\sqrt{2}) = \frac{\pi}{2K(1/\sqrt{2})} \tag{4.19}$$

On complète la preuve en substituant (4.18) dans (4.17), en employant (4.19), en notant que $c_0^2 = 1/2$ et en résolvant pour π. ∎

Pour terminer, nous donnons sans démonstration la majoration de l'erreur commise sur le calcul de π en prenant la valeur approchée suivante

$$\pi_n = \frac{4a_{n+1}^2}{1 - \sum_{i=1}^{n} 2^{i+1} c_i^2} \tag{4.20}$$

Théorème 12.4.7. *La suite π_n est croissante, converge vers π et satisfait les inégalités*

$$\pi - \pi_n \leq \frac{\pi^2 2^{n+4}}{\mu^2(1, 1/\sqrt{2})} \exp(-\pi 2^{n+1})$$

et

$$\pi - \pi_{n+1} \leq \frac{2^{-(n+1)}}{\pi^2}(\pi - \pi_n)^2$$

La deuxième inégalité montre en particulier que la convergence de π_n vers π est quadratique.

12.4.3 Programme : la formule de Brent-Salamin

Pour le calcul de π par la formule de Brent-Salamin, on évalue l'approximation

$$\pi_n = \frac{4a_{n+1}^2}{1 - \displaystyle\sum_{j=1}^{n} 2^{j+1} c_j^2}$$

où $a_0 = 1$, $b_0 = 1/\sqrt{2}$ et pour $n \geq 1$

$$\begin{aligned} a_n &= (a_{n-1} + b_{n-1})/2 \\ b_n &= \sqrt{a_{n-1} b_{n-1}} \\ c_n &= a_{n-1} - a_n \end{aligned}$$

En fait, il suffit de disposer de a_{n+1}^2 et de c_n^2, et il apparaît que l'on peut calculer par récurrence ces valeurs. Si l'on pose, en effet, $\alpha_n = a_n^2$, $\beta_n = b_n^2$ et $\gamma_n = c_n^2$, on obtient $\alpha_0 = 1$, $\beta_0 = 1/2$ et, pour $n \geq 0$,

$$\begin{aligned} \gamma_n &= \alpha_n - \beta_n \\ \beta_{n+1} &= \sqrt{\alpha_n \beta_n} \\ 4\alpha_{n+1} &= \alpha_n + \beta_n + 2\beta_{n+1} \end{aligned}$$

Dans la procédure ci-dessous, on calcule α_{n+1}, β_{n+1} et γ_n à partir des valeurs correspondantes pour $n-1$. Il est commode de garder aussi 2^{n+1} (dans une variable δ_n) ainsi que le dénominateur de π_n. Enfin, pour calculer β_{n+1}, on calcule d'abord la racine de $1/\alpha_n\beta_n$ et, comme les valeurs changent peu, mieux vaut partir pour l'évaluation de la valeur obtenue à l'itération précédente de π.

```
PROCEDURE IterationPi (VAR alpha, beta, gamma, delta, D, UnSurD, UnSurRBeta,
   piapproche: Flottant);
   Calcule l'approximation suivante de π. D est le dénominateur, UnSurD est son inverse,
   UnSurRBeta est l'inverse de la racine de β.
   VAR
      N: Flottant;                                         Numérateur de π_n.
      alphaPlusbeta, deltagamma: Flottant;
   BEGIN
   Calcul de γ
      FlottantMoinsFlottant(alpha, beta, gamma);           γ_n = α_n − β_n.
      FlottantPlusFlottant(alpha, beta, alphaPlusbeta);    α_n + β_n.
   Calcul de β
      FlottantParFlottant(alpha, beta, beta);
      FlottantUnSurRacineIteration(beta, UnSurRBeta);      1/√(α_n β_n).
      FlottantParFlottant(beta, UnSurRBeta, beta);         β_{n+1}.
   Calcul du numérateur
      N := beta;
```

```
      FlottantparChiffre(N, 2);
      FlottantPlusFlottant(alphaPlusbeta, N, N);          N_n = α_n + β_n + 2β_{n+1}.
   Calcul de α
      alpha := N;
      FlottantSurChiffre(alpha, 4);                       α_{n+1} = N/4.
   Calcul du dénominateur
      FlottantParChiffre(delta, 2);
      FlottantParFlottant(delta, gamma, deltagamma);
      FlottantMoinsFlottant(D, deltagamma, D);            D_n = D_{n-1} − 2^{n+1} γ_n.
   Calcul de π
      FlottantInverseIteration(D, UnSurD);
      FlottantParFlottant(N, UnSurD, piapproche)          π_n = N_n/D_n.
   END; { de "IterationPi" }
```

Cette procédure est employée par :

```
PROCEDURE CalculPiParSalamin (VAR piapproche: Flottant);
   VAR
      vieuxpi: Flottant;
      UnSurD, D: Flottant;
      alpha, beta, gamma, UnSurRBeta, delta: Flottant;
   BEGIN                                           Initialisation pour n = −1.
      ChiffreEnFlottant(D, 2);                            D_{-1} = 2.
      ChiffreEnFlottant(UnSurD, 1);
      ChiffreEnFlottant(alpha, 1);                        α_0 = 1.
      ChiffreEnFlottant(beta, 1);
      FlottantSurChiffre(beta, 2);                        β_0 = 1/2.
      ChiffreEnFlottant(UnSurRBeta, 1);
      ChiffreEnFlottant(delta, 1);                        δ_0 = 1.
      ChiffreEnFlottant(piapproche, 0);                   π_{-1} = 0, p.ex.
      REPEAT
         vieuxpi := piapproche;
         IterationPi(alpha, beta, gamma, delta, D, UnSurD, UnSurRBeta,
            piapproche);
      UNTIL (FlottantProches(vieuxpi, piapproche)) OR (LeSigne(gamma) = -1);
   END; { "CalculPiParSalamin" }
```

Deux remarques : pour le calcul de l'inverse du dénominateur, on prend comme valeur approchée de départ la valeur précédente de ce nombre; en effet, la suite converge et même très rapidement.

Notons le test d'arrêt dans la procédure. Les erreurs d'arrondi jouent à plein lors du calcul de γ_n comme différence de α_n et β_n. Comme ces suites convergent vers la même valeur, la valeur approchée de γ_n ne porte que sur les derniers chiffres de ces nombres. Il se peut que cette valeur devienne très petite, mais négative. On arrête dans ce cas puisque les résultats ne sont plus significatifs.

```
n = 0
gamma = 0. 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E 0
```

```
beta  = 0. 70 71 06 78 11 86 54 75 24 40 08 44 36 21 04 84 90 39 28 50 E 0
numer = 0. 02 91 42 13 56 23 73 09 50 48 80 16 88 72 42 09 69 80 78 57 E 1
alpha = 0. 72 85 53 39 05 93 27 37 62 20 04 22 18 10 52 42 45 19 64 00 E 0
denom = 0. 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E 1
piapp = 0. 02 91 42 13 56 23 73 09 50 48 80 16 88 72 42 09 69 80 78 57 E 1

n = 1
gamma = 0. 02 14 46 60 94 06 72 62 37 79 95 77 81 89 47 57 54 80 35 50 E 0
beta  = 0. 71 77 49 98 63 77 53 75 38 19 49 37 73 07 56 72 64 26 34 52 E 0
numer = 0. 02 87 11 60 14 45 34 89 63 62 99 11 42 00 46 70 72 64 11 61 E 1
alpha = 0. 71 77 90 03 61 33 72 40 90 74 77 85 50 11 67 68 16 02 90 00 E 0
denom = 0. 91 42 13 56 23 73 09 50 48 80 16 88 72 42 09 69 80 78 58 00 E 0
piapp = 0. 03 14 05 79 25 05 22 16 82 48 31 13 31 26 89 75 82 33 11 75 E 1

n = 2
gamma = 0. 40 04 97 56 18 65 52 55 28 47 77 04 10 95 51 76 55 48 00 00 E -2
beta  = 0. 71 77 70 01 09 76 29 63 92 27 45 34 23 43 16 74 13 62 48 80 E 0
numer = 0. 02 87 10 80 04 44 63 85 44 13 49 17 91 70 05 57 89 07 54 21 E 1
alpha = 0. 71 77 70 01 11 15 96 36 03 37 29 47 92 51 39 47 26 88 55 00 E 0
denom = 0. 91 38 93 16 43 23 60 26 28 37 89 06 56 09 22 05 66 66 14 16 E 0
piapp = 0. 03 14 15 92 64 62 13 54 22 82 14 93 44 43 19 82 69 57 74 14 E 1

n = 3
gamma = 0. 01 39 66 72 11 09 84 13 69 08 22 73 13 26 06 20 00 00 00 00 E -4
beta  = 0. 71 77 70 01 10 46 12 99 97 82 03 43 93 83 11 67 02 27 03 27 E 0
numer = 0. 02 87 10 80 04 41 84 51 99 91 28 81 70 03 60 79 55 45 05 09 E 1
alpha = 0. 71 77 70 01 10 46 12 99 97 82 20 42 50 90 19 88 86 26 27 00 E 0
denom = 0. 91 38 93 16 20 88 92 72 50 80 42 87 50 77 58 35 54 49 14 96 E 0
piapp = 0. 03 14 15 92 65 35 89 79 32 38 27 95 12 77 48 01 86 39 73 88 E 1

n = 4
gamma = 0. 16 98 57 07 08 21 83 99 23 73 00 00 00 00 00 00 00 00 00 00 E -10
beta  = 0. 71 77 70 01 10 46 12 99 97 82 11 93 22 36 65 77 94 26 65 19 E 0
numer = 0. 02 87 10 80 04 41 84 51 99 91 28 47 72 89 46 63 11 77 06 60 E 1
alpha = 0. 71 77 70 01 10 46 12 99 97 82 11 93 22 36 65 77 94 26 65 00 E 0
denom = 0. 91 38 93 16 20 88 92 72 50 74 99 33 24 50 95 36 66 73 55 60 E 0
piapp = 0. 03 14 15 92 65 35 89 79 32 38 46 26 43 38 32 79 50 28 83 04 E 1

n = 5
gamma =-0. 19 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E -19
beta  = 0. 71 77 70 01 10 46 12 99 97 82 11 93 22 36 65 77 94 26 65 11 E 0
numer = 0. 02 87 10 80 04 41 84 51 99 91 28 47 72 89 46 63 11 77 06 60 E 1
alpha = 0. 71 77 70 01 10 46 12 99 97 82 11 93 22 36 65 77 94 26 65 00 E 0
denom = 0. 91 38 93 16 20 88 92 72 50 74 99 33 24 50 95 36 66 73 67 76 E 0
piapp = 0. 03 14 15 92 65 35 89 79 32 38 46 26 43 38 32 79 50 28 82 61 E 1

pi = 0. 03 14 15 92 65 35 89 79 32 38 46 26 43 38 32 79 50 28 82 61 E 1
```

La valeur exacte de π se termine par les chiffres 84 20. Comme annoncé, la valeur approchée de γ_n devient négative.

Notes bibliographiques

Les algorithmes pour l'arithmétique en multiprécision se trouvent notamment dans :

D. E. Knuth, *The Art of Computer Programming, Vol. II Seminumerical Algorithms*, Reading, Addison-Wesley, 1968. Ce livre contient aussi une discussion détaillée de la façon de minimiser les erreurs d'arrondi.

L'interprétation combinatoire des formules d'arctangentes est exposée dans :

B. Gréco, Papier quadrillé et calcul de pi, *Bulletin de l'APMEP* **354** (1986), 347–361.

Pour la détermination de formules «à la Machin» on peut aussi consulter :

J.P. Friedelmeyer, Arcs de cercle à tangente rationnelle et entiers imaginaires premiers, *Bulletin de l'APMEP* **358** (1987), 145–159.

J. Todd, A problem on arc tangent relations, *American Mathematical Monthly* **56** (1949), 517–528.

La formule de Brent-Salamin a été donnée indépendamment par :

R. P. Brent, Fast multiple-precision evaluation of elementary functions, *J. Assoc. Comput. Mach.* **23** (1976), 242–251.

E. Salamin, Computation of π using arithmetic-geometric mean, *Math. Computation* **30** (1976), 565–570.

La moyenne arithmético-géométrique s'applique aussi au calcul numérique des fonctions élémentaires comme $\log x, e^x, \sin x$ et $\cos x$. On consultera à ce sujet l'article de Brent. Un exposé synthétique est :

G. Almkvist, B. Berndt, Gauss, Landen, Ramanujan, The arithmetic-geometric mean, ellipses, π, and the Ladies Diary, *American Math. Monthly* **95** (1988), 585–608.

Enfin, un exposé approfondi est :

J. M. Borwein, P. B. Borwein, *Pi and the AGM*, New York, Wiley and Sons, 1987.

Annexes

Annexes

Annexe A

Un environnement

A.1 Conseils de programmation

Un programme, ou un ensemble de procédures, n'est pas seulement un moyen de communication avec l'ordinateur. C'est aussi une façon de présenter un algorithme, et donc un support d'échanges entre personnes. Dans cette optique, un programme doit être rédigé et présenté de façon lisible. Bien évidemment, ceci est passablement subjectif. Dans l'idéal, un programme bien écrit se lit à la même vitesse qu'un texte mathématique bien rédigé. Cela va en général moins vite que la lecture d'un roman policier, mais il n'y a pas de raison que cela aille beaucoup moins vite. Pour arriver à cette facilité de lecture, il y a quelques règles à observer.

Tout d'abord, une procédure doit être courte. Un programme qui s'étale sur plusieurs pages, où l'on est constamment obligé de faire des allées et venues entre le début, contenant les définitions (déclarations) et le texte lui-même, ne facilite pas la lecture. Pour arriver à des procédures courtes, il faut structurer le programme, et donc aussi l'algorithme sous-jacent, en mettant en évidence des groupements organiques de calculs élémentaires.

On procède en fait comme dans une démonstration mathématique un peu longue : on met en évidence un certain nombre de lemmes, ou de propositions intermédiaires, et la démonstration du théorème lui-même se réduit alors à un enchaînement logique judicieux de propriétés établies auparavant. En ce sens, l'en-tête d'une procédure peut se comparer à l'énoncé d'une proposition, et le corps de la procédure à la démonstration. Remarquons que cette démarche, qui en mathématique conduit parfois à des lemmes «qui ont un intérêt intrinsèque», conduit en programmation à des algorithmes dont la portée dépasse le cadre initial.

Un programme, bien plus encore qu'une démonstration, doit être accompagnée de *commentaires.* Ils paraphrasent le cheminement algorithmique, résument les groupements d'instructions, mettent en garde contre des oublis, justifient les hypothèses; en début de procédure, le commentaire explique la nature des paramètres et résume les restrictions (variable non nulle par exemple). On entend parfois dire que les commentaires doivent avoir la même longueur que le programme, tellement ils contribuent à la compréhension du programme. Sans aller jusque-là, on ne peut qu'encourager la multiplication des commentaires.

Nommer les objets

Le choix des notations, en mathématiques, est un autre élément qui facilite la lecture. En général, une variable réelle est notée x et non pas i, l'indice d'un vecteur est noté i et rarement z. On procédera de la même manière pour la rédaction des procédures. Voici un exemple à éviter :

```
FUNCTION a (b: integer): integer;
   VAR
      c, d: integer;
      e: boolean;
   BEGIN
      a := b;
      e := false;
      c := 1;
      REPEAT
         d := f[i];
         e := b MOD d = 0;
         IF e THEN
            a := d
         ELSE
            c := c + 1
      UNTIL (d * d > b) OR e;
   END; { de "a" }
```

Voici la même procédure, qui cherche dans une table `NombresPremiers` de nombres premiers le plus petit facteur premier de l'entier n donné en argument :

```
FUNCTION FacteurPremier (n: integer): integer;
   VAR
      i, p: integer;
      EstCompose: boolean;
   BEGIN
      FacteurPremier := n;
      EstCompose := false;
      i := 1;
      REPEAT
         p := NombresPremiers[i];
         EstCompose := n MOD p = 0;
         IF EstCompose THEN
            FacteurPremier := p
```

```
         ELSE
            i := i + 1
      UNTIL (p * p > n) OR EstCompose;
   END; { de "FacteurPremier" }
```

Comment choisir les noms des variables et des procédures ? Ce choix est à la fois plus simple et plus compliqué qu'en mathématiques. Plus compliqué, parce que l'on ne dispose que du seul alphabet latin et d'aucun signe diacritique (flèches, indices, points, accents) ; plus simple parce que, contrairement à l'usage en mathématiques, des noms composés de plusieurs caractères (comme Ker, dét, etc.) non seulement sont fréquents, mais en fait souhaités. Il n'y a pratiquement pas de limite à la longueur des noms des identificateurs : bien que la norme originale de Pascal Standard précise que seuls les huit premiers caractères d'un identificateur sont significatifs, la plupart des compilateurs récents repoussent cette limite à trente-deux ou au-delà, et il faut faire amplement usage de cette possibilité. Ainsi, `PlusPetitFacteurPremier` aurait été un titre bien plus instructif pour la fonction ci-dessus (lisez à voix haute : `PlusPetitFacteurPremier` *de* n). L'emploi d'identificateurs courts raccourcissait le temps de frappe et les erreurs de frappe lorsque l'on disposait seulement d'éditeurs de textes rudimentaires (sans parler des temps préhistoriques et des cartes perforées). Maintenant, les éditeurs sont suffisamment développés pour permettre des copies instantanées. En revanche, il est inutile de charger les identificateurs, lorsque l'usage mathématique ou informatique courant rend leur signification claire : p désigne en général un nombre premier et i est un indice. On peut donc raisonnablement éviter l'excès contraire, consistant à écrire par exemple :

```
iteration := 1;
REPEAT
   PremierCourant := NombresPremiers[iteration];
   EstCompose := n MOD PremierCourant = 0;
   IF EstCompose THEN
      FacteurPremier := PremierCourant
   ELSE
      iteration := iteration + 1
UNTIL (PremierCourant * PremierCourant > n) OR EstCompose;
```

Le plus souvent, les compteurs de boucles, qui reviennent fréquemment, sont identifiés par une seule lettre : i, j, k, n, etc. En revanche, nous avons choisi des noms longs, parfois presque excessivement longs, pour les noms des procédures, pour que l'effet de la procédure soit le plus complètement possible décrit dans le nom même. Ainsi `MatricePlusMatrice`, ou `EchangerColonnes` ou encore `EcrireMatriceHermitienne`. Cette façon de faire a l'avantage de permettre la transposition instantanée des fonctions à d'autres contextes : par exemple, `FlottantPlusFlottant` est manifestement l'en-tête d'une procédure d'addition de deux «flottants», quelle que soit la signification de «flottant» : il s'agit en fait de transposer, dans un contexte de programmation, la convention qui veut que + désigne l'addition, dans n'importe quelle structure algébrique (dans un langage de programmation plus évolué, ce genre de convention est d'ailleurs réalisable). Le choix du nom convenable pour une procédure devient alors presque enfantin : l'addition de deux nombres rationnels s'appelle `RatPlusRat`, et pour des vecteurs à coefficients complexes `VecteurCPlusVecteurC`. Une autre solution, fréquemment employée, consiste à séparer les constituants d'un identificateur par le caractère de soulignement «_», à défaut du tiret `-` qui est plus naturel, mais interdit en Pascal. On peut ainsi écrire `rat_plus_rat` ou `vecteur_c_plus_vecteur_c`. On gagne peut-être en lisibilité ; en échange, les en-têtes de procédure se rallongent encore plus. (En Fortran, où le blanc n'est pas significatif, on peut écrire très commodément `matrice plus matrice` ; en Lisp, le tiret est utilisable et on dispose d'autres caractères précieux : ainsi, on peut écrire `Nul?` au lieu de notre `EstNul`.)

Ces règles ne vont pas sans exception. Ainsi, quelques types reviennent constamment dans les programmes et, pour ne pas trop allonger le texte, nous appelons `vec` et `mat` les types vecteur et matrice, et `pol` le type polynôme.

Organiser les structures

Un autre problème qui se pose est l'accès, la modification et la construction d'objets complexes. Considérons par exemple les nombres complexes. Un nombre complexe est représenté par un couple de réels, ses parties réelle et imaginaire. Il faut pouvoir y accéder, les modifier et pouvoir construire un complexe à partir de deux réels. Ces procédures sont faciles à écrire, mais dépendent évidemment de la représentation choisie : nous définissons un complexe par

```
TYPE
   complexe = ARRAY[0..1] OF real;
```

mais un lecteur qui connaît les enregistrements préfère peut-être la définition :

```
TYPE
   complexe = RECORD
                 PartieReelle,PartieImaginaire : real
              END;
```

Pour rendre un programme indépendant de la représentation choisie, on convient d'accéder aux structures complexes seulement à travers des procédures convenues. Ainsi, la partie réelle d'un nombre complexe `z` ne sera pas appelée par `z[0]`, mais à travers une procédure `Re(z)`. De la même manière, on peut convenir de ranger le degré d'un polynôme dans l'emplacement d'indice -1 du tableau de ses coefficients (ou dans celui d'indice maximal) ou ailleurs encore, mais l'appel se fera uniquement par une fonction bien précise.

Cette façon de procéder a non seulement l'avantage de rendre l'implémentation plus souple, mais aussi de rendre la lecture plus facile. En effet, avec l'une des conventions ci-dessus pour le degré, on écrit `d := round(p[-1])` ou encore `d := round(p[degremax])` là où `d := degre(p)` est bien plus parlant.

Ces opérations se font en fait par paires : on veut récupérer la partie réelle d'un complexe, et la modifier, de même, obtenir et fixer le degré d'un polynôme. En anglais, il existe une convention de notation commode, fondée sur le couple «get» et «set» (ainsi on peut écrire `GetDegree` et `SetDegree`), mais en français, on ne dispose pas de vocables aussi brefs et aussi parlants. Nous avons choisi le préfixe `Fixer` chaque fois que l'on veut changer un attribut, et l'absence de préfixe pour l'obtenir : ainsi `FixerRe` et `FixerDegre`; ou encore `FixerNumerateur` et `Numerateur`. Là encore, le choix fait se transpose (fonctoriellement!) à travers les divers exemples traités.

Passer les paramètres

Lors de l'appel d'une procédure ou fonction, les paramètres formels sont remplacés par les paramètres dit «actuels». La substitution se fait différemment selon que le passage du paramètre est par *valeur* ou par *référence* (ce mode de passage étant spécifié par le préfixe `VAR` dans la déclaration).

Le passage par valeur s'effectue comme suit : l'expression substituée au paramètre formel est évaluée et le résultat est transféré dans une variable locale à la procédure. Toute référence au paramètre formel interne au corps de la procédure est remplacé par la variable locale à la procédure. A la fin de l'exécution de la procédure, la variable locale disparaît. Considérons par exemple la procédure :

```
PROCEDURE somme (a, b, c:integer);
   BEGIN
      c := a + b
   END ;
```

L'exécution de `somme(5,x+3,z)` équivaut à

```
alocal := 5; blocal := x+3 ; clocal := z ; clocal := alocal + blocal
```

et en particulier, la valeur de `z` est inchangée après l'exécution.

Le passage par référence s'effectue différemment : ce n'est pas un contenu qui est transmis, mais l'*adresse* d'une variable (la localisation de l'emplacement). Plus précisément, l'expression substituée au paramètre formel est évaluée et doit désigner une adresse. L'expression est donc très particulière : c'est une variable ou une variable indicée (ou toute autre expression dont le résultat est une adresse). Puis, l'adresse ainsi obtenue remplace le paramètre formel en toutes ses occurrences. Considérons par exemple :

```
PROCEDURE produit (VAR a, b, c:integer);
   BEGIN
      c := a * b
   END ;
```

Les trois paramètres sont passés par référence. On ne peut donc pas appeler `produit(x, y+3, z)` parce que le résultat de l'évaluation de `y+3` n'est pas une adresse. En revanche, `produit(u[i], v[i], w[i])` est un appel licite.

Si l'on veut que le résultat d'un calcul soit disponible à la sortie d'une procédure, le paramètre correspondant doit nécessairement être passé par référence. En revanche, une quantité qui intervient dans un calcul sans être modifiée est passée en valeur. Toutefois, on peut parfois gagner du temps en choisissant un passage par référence plutôt que par valeur ; ainsi, si le paramètre est une matrice, le passage par référence ne demande que l'évaluation du nom, alors que le passage par valeur provoque la création d'une matrice locale à la procédure et la recopie de la matrice avant toute exécution. Le temps supplémentaire pris par la recopie peut être très important en pourcentage, car cette recopie concerne la taille maximale du tableau. Ainsi, si `OrdreMax` est fixé à 20, ce sont 400 éléments qui sont recopiés, même si les matrices considérées dans un programme particulier ne sont que d'ordre 2. Le surcoût, pris par rapport à l'addition de deux matrices, peut aller de quelques pour cents à plus de 700 % pour les grandeurs données ci-dessus. Or, la recopie est inutile lorsque l'on est certain de ne jamais modifier le contenu de la matrice, comme par exemple dans une procédure d'impression. Dans ces cas, on pourra déclarer le paramètre en référence. Dans les autres cas en revanche, le passage par valeur est recommandé.

Dans le même ordre d'idées, on peut discuter notre choix de la représentation de la matrice unité comme objet séparé qui est employé dans une instruction d'affectation comme

```
a := MatriceUnite
```

On pourrait, à la place, faire appel à une procédure `Unite(n,a)` qui crée, dans a, la matrice unité d'ordre n. Lorsque n est nettement plus petit que l'ordre maximal, on peut s'attendre à un gain de temps en utilisant la deuxième méthode ; en contrepartie, la gestion de l'appel de procédure prend aussi du temps. On constate que si l'ordre maximal est 20, la durée des deux méthodes s'équilibre autour de $n = 7$. En deçà, l'appel de procédure est plus rapide. Notons tout de même qu'avec les matériels informatiques modernes, on peut faire plusieurs centaines d'affectations de matrices en une seconde. Ceci relativise notre discussion sur la perte de temps éventuelle due à une programmation un peu laxiste.

Faciliter le dialogue

Les programmes sont tous interactifs, c'est-à-dire demandent à l'utilisateur d'intervenir. Dans les programmes qui nous concernent, l'interaction est rudimentaire : elle se borne à demander des données et à afficher les résultats correspondants. Dans d'autres programmes, l'interaction est très sophistiquée (jeux, éditeurs, compilateurs) et la programmation de cette partie constitue fréquemment plus de la moitié de l'effort de programmation total.

Il est utile de prendre en considération quelques principes simples, fondés sur le fait que l'interlocuteur est un être intelligent : éviter un bavardage inutile, encombrant l'écran de messages multiples, surtout en cas d'erreur sur les données ; souvent, un simple bip sonore est suffisant. Eviter aussi les trop nombreuses demandes de confirmation, du genre : «Etes-vous sûr de vos données (o/n) ?». Une indication claire sur la façon d'arrêter le programme suffit.

En revanche, l'*invite* doit être claire : ne pas commencer l'exécution d'un programme par un point d'interrogation, sans annoncer au moins quel type de données on attend. Un deuxième principe à observer est physique : toute action provoque une réaction ; le programme doit réagir à l'entrée de données par un *accusé de réception*, surtout si les calculs subséquents sont longs. La meilleure façon de faire, lorsque les données sont un peu complexes (lecture d'une matrice ou d'un polynôme) est de réafficher, dans votre format, les données lues. Enfin, les résultats doivent être présentés : il convient de dire quels sont les résultats affichés.

Dans un programme à dialogue simple, les options possibles se présentent sous la forme d'un menu (voir ci-dessous une façon de faire) ; une option du menu, accessible par exemple par `?`, peut contenir un ou plusieurs écrans d'explications.

A.2 Variations en Pascal

En théorie, il n'y a qu'un seul langage de programmation Pascal, celui défini par la norme ISO. En pratique, il y a autant de langages Pascal que de compilateurs et ceci pour plusieurs raisons. D'abord, le langage Pascal officiel est plutôt pauvre en structures de contrôle et, de plus, ces structures sont parfois assez lourdes à utiliser. Il est donc normal qu'un compilateur qui se veut «intéressant» apporte des possibilités supplémentaires, et c'est une trop grande tentation pour ne pas en profiter. Ensuite, Pascal (toujours la norme) ne fixe pas de règles commodes pour la manipulation de chaînes de caractères et est assez difficile sur les entrées-sorties interactives. Enfin, rien n'est prévu pour l'affichage graphique et sur d'autres aspects qui, s'ils dépendent certes du matériel utilisé, pourraient néanmoins être codifiés d'une manière agréable. Là aussi, chaque compilateur vient avec ses solutions. Enfin, il n'y a pas de syntaxe pour l'usage des bibliothèques. (Il ne s'agit pas ici de critiquer Pascal : ce langage a contribué à rendre la programmation raisonnable et, à ce titre, est très important ; les restrictions énumérées sont bien connues et Wirth, le père de Pascal, les a levées pour la plupart dans son langage suivant, Modula.)

Les programmes présentés dans ce livre ont tous été testés soit sur Macintosh, sous Think Pascal, soit sur compatible PC, sous Turbo Pascal, soit les deux. Les modifications à apporter pour les transférer de l'un des compilateurs à l'autre sont essentiellement les deux suivantes :

• Dans une instruction `CASE`, le champ optionnel qui décrit l'action à entreprendre lorsque aucun cas ne s'applique, est introduit en Turbo Pascal par `ELSE` (qui n'est pas précédé d'un point-virgule) et en Think Pascal par `OTHERWISE` (qui *est* précédé d'un point-virgule). Nous ne nous en servons que dans les menus.

• Bien plus importante est l'écriture des *opérateurs booléens séquentiels*, aussi appelés opérateurs courts ou opérateurs minimaux. Considérons par exemple l'instruction :

```
WHILE (i >= 1) AND (a[i] = 0) DO i := i - 1;
```

Dans Pascal standard, les opérandes du test sont tous les deux évalués avant d'en faire la conjonction logique. Ainsi, même si $i = 0$, l'élément $a[i]$ est comparé à 0, ce qui provoque un message d'erreur si les indices du tableau commencent à 1. Lorsque l'opérateur `AND` est *séquentiel*, la première condition est évaluée d'abord et la deuxième n'est évaluée que si la première est vraie (*mutatis mutandis* pour `OR`) ; dans notre exemple, il n'y a pas de dépassement des bornes de tableau.

Ce mode d'évaluation est extrêmement commode et a été utilisé systématiquement dans le livre. En Turbo Pascal, c'est le fonctionnement par défaut (mais il peut être modifié en changeant une option du compilateur) ; en Think Pascal, les opérateurs `AND` et `OR` ne sont pas séquentiels, mais ce compilateur fournit deux variantes séquentielles, notées `&` et `|`. Nous encourageons explicitement les utilisateurs de nos programmes en Think Pascal à remplacer systématiquement `AND` par `&` et `OR` par `|`. Les endroits où le changement est obligatoire sont signalés en commentaires dans nos programmes.

D'autres différences entre Turbo Pascal et Think Pascal sont moins importantes et sont plutôt des différences liées au matériel et au système d'exploitation. Nous en parlons lors de la présentation de la bibliothèque générale, ou dans la présentation des programmes géométriques.

Pour une description détaillée du langage Pascal et de nombreux exemples, voir :

P. Cousot, *Introduction à l'algorithmique numérique et à la programmation en Pascal*, Paris, McGraw-Hill, 1988.

A.3 Bibliothèques

Les bibliothèques de procédures sont une façon commode de grouper les déclarations, procédures et initialisations relatives à un ensemble de calculs. Les compilateurs Pascal (Turbo Pascal 4 ou Think Pascal par exemple) offrent, sous une forme qui varie légèrement d'un compilateur à l'autre, un moyen de réaliser ces groupements : ce sont les *unités* (« unit » en anglais).Une unité peut être définie et compilée indépendamment de tout programme. L'emploi d'une unité dans un programme particulier est indiqué par une instruction spécifique.

En Think Pascal et en Turbo Pascal, la syntaxe d'une unité est la suivante :

```
UNIT nom;

   INTERFACE
      ...
   IMPLEMENTATION
      ...
   END.
```

En Turbo Pascal, une variante est possible, dont on expliquera l'usage :

```
UNIT nom;

   INTERFACE
```

```
   ...
IMPLEMENTATION
   ...
BEGIN
   ...
END.
```

L'*interface* contient les informations publiques, c'est-à-dire utilisables par le programme qui inclut l'unité. Il peut s'agir de constantes, de déclarations de types, de variables et d'en-têtes de procédures. La partie *implémentation* contient la réalisation des diverses procédures. Enfin en Turbo Pascal, le corps de l'unité (la partie *initialisation*) délimité par le dernier couple `BEGIN`, `END` peut contenir des instructions qui sont exécutées avant chaque exécution du programme qui utilise l'unité.

Un programme qui utilise une unité le spécifie par l'instruction :

```
USES
   nom;
```

Considérons un exemple. La bibliothèque des nombres rationnels, décrite en détail dans le paragraphe suivant, se déclare comme unité de la manière que voici :

```
UNIT rationnels;

INTERFACE
USES
   general;                                 Utilise, à son tour, une unité.
TYPE
   rat = ARRAY[0..1] OF integer;            Déclaration de type.
VAR
   RatZero, RatUn: rat;                     Déclaration de variables.

PROCEDURE InitRationnels;                   En-têtes de procédures.
FUNCTION Numerateur (VAR r: rat): integer;

                                   ...

PROCEDURE RatPlusRat (u, v: rat; VAR w: rat);

                                   ...

IMPLEMENTATION                              Réalisation des procédures.

PROCEDURE RatPlusRat (u, v: rat; VAR w: rat);
   VAR
      n, d: integer;
   BEGIN
      n := Numerateur(u) * Denominateur(v) + Denominateur(u) * Numerateur(v);
      d := Denominateur(u) * Denominateur(v);
      FaireRat(n, d, w);
   END; { de "RatPlusRat" }
```

```
PROCEDURE InitRationnels;
   BEGIN
      EntierEnRat(RatZero, 0);
      EntierEnRat(RatUn, 1)
   END; { de "InitRationnels" }

BEGIN
   InitRationnels;                          Appel (en Turbo Pascal).
END.
```

En Think Pascal par exemple, la procédure `InitRationnels` doit être appelée au début du programme principal.

A.3.1 Généralités

Un certain nombre d'opérations arithmétiques reviennent constamment dans les programmes. Il est utile de disposer d'un ensemble de procédures qui les réalisent. C'est l'objet de cette bibliothèque. Nous en profitons pour fixer des valeurs par défaut de certaines variables. La constante π existe déjà en TURBO Pascal, donc la définition suivante n'est pas utile dans ce cas.

```
CONST
   pi = 3.1415926536;              Prédéfinie dans certains compilateurs.
```

Le type suivant sert à transmettre des titres aux procédures de lecture et d'écriture :

```
TYPE
   texte = string[80];
```

Enfin, voici trois variables utiles : la première permet de paramétrer la précision d'affichage (non pas de calcul), la dernière est employée dans le calcul «à ε près», enfin la variable `test` permet de basculer en mode test et ainsi d'activer, resp. de désactiver, des impressions intermédiaires.

```
VAR
   precision: integer;
   test: boolean;
   epsilon: real;
```

Ces variables sont initialisées dans Initialisations :

```
PROCEDURE InitGeneral;
   BEGIN
      precision := 3;              Précision d'affichage par défaut.
      epsilon := 0.001;            Nombre petit par défaut.
      test := false                Pas d'impression intermédiaire par défaut.
   END; { de "InitGeneral" }
```

Les fonctions arithmétiques sur les entiers que nous avons mentionnées sont :

```
FUNCTION min (i, j: integer): integer;
FUNCTION max (i, j: integer): integer;
PROCEDURE EchangerE (VAR u, v: integer);
FUNCTION PuissanceE (x: integer; n: integer): integer;
FUNCTION Binomial (n, k: integer): integer;
FUNCTION pgcd (a, b: integer): integer;
```

Elles se réalisent comme suit :

```
FUNCTION min (i, j: integer): integer;
   BEGIN
      IF i <= j THEN min := i ELSE min := j
   END; { de "min" }
FUNCTION max (i, j: integer): integer;
   BEGIN
      IF i >= j THEN max := i ELSE max := j
   END; { de "max" }
PROCEDURE EchangerE (VAR u, v: integer);
   VAR
      w: integer;
   BEGIN
      w := u; u := v; v := w
   END; { de "EchangerE" }
```

Les fonctions suivantes sont plus intéressantes :

```
FUNCTION PuissanceE (x: integer; n: integer): integer;         Calcule x^n.
   VAR
      i, p: integer;
   BEGIN
      p := 1;
      FOR i := 1 TO n DO       Méthode élémentaire.
         p := p * x;
      PuissanceE := p
   END; { de "PuissanceE" }
```

On peut gagner du temps en utilisant l'algorithme «dichotomique» fondé sur le développement binaire de n. Le calcul du coefficient binomial ne doit pas être mené par l'évaluation des factorielles. La méthode ci-dessous est basée sur la formule :

$$\binom{n}{k} = \frac{n-k+1}{k}\binom{n-1}{k-1}$$

et permet d'évaluer les coefficients binomiaux pour $n \leq 14$.

```
FUNCTION Binomial (n, k: integer): integer;
   VAR
      b, i: integer;
   BEGIN
```

```
      IF n < k THEN
         Binomial := 0
      ELSE BEGIN
         k := min(k, n - k);
         b := 1;
         FOR i := 1 TO k DO
            b := (b * (n - i + 1)) DIV i;
         Binomial := b
      END
   END; { de "Binomial" }
```

Enfin, le calcul du pgcd se fait par l'ancêtre des algorithmes :

```
FUNCTION pgcd (a, b: integer): integer;
   VAR
      r: integer;
   BEGIN
      a := abs(a); b := abs(b);
      WHILE b <> 0 DO BEGIN
         r := a MOD b;
         a := b;
         b := r
      END;
      pgcd := a
   END; { de "pgcd" }
```

Le résultat est toujours positif ou nul. Voici des fonctions sur les réels :

```
FUNCTION rmin (x, y: real): real;
FUNCTION rmax (x, y: real): real;
PROCEDURE Echanger (VAR u, v: real);
FUNCTION log10 (x: real): real;
FUNCTION PuissanceR (x, y: real): real;            Calcule x^y.
FUNCTION Puissance (x: real; n: integer): real;    Calcule x^n.
FUNCTION signe (r: real): integer;
FUNCTION EstNul (r: real): boolean;
FUNCTION EstPositif (r: real): boolean;
```

Les trois premières sont évidentes. Le logarithme en base 10 se calcule par :

```
FUNCTION log10 (x: real): real;
   BEGIN
      log10 := ln(x) / ln(10)
   END; { de "log10" }
```

Deux façons de calculer la puissance d'un réel :

```
FUNCTION PuissanceR (x, y: real): real;            Calcule x^y.
   BEGIN
      IF EstNul(x) THEN
```

```
         PuissanceR := 0
      ELSE
         PuissanceR := exp(y * ln(x))
   END; { de "PuissanceR" }
```

```
FUNCTION Puissance (x: real; n: integer): real;
   VAR
      i, p: integer;
   BEGIN
      p := 1;
      FOR i := 1 TO n DO
         p := p * x;
      Puissance := p
   END; { de "Puissance" }
```
Calcule x^n.

Le test si la représentation d'un nombre réel est nulle ne peut se faire qu'à ε près. On est donc naturellement conduit aux fonctions :

```
FUNCTION EstNul (r: real): boolean;
   BEGIN
      EstNul := abs(r) < epsilon
   END; { de "EstNul" }
```
Vrai si r est nul à ε près.

```
FUNCTION EstPositif (r: real): boolean;
   BEGIN
      EstPositif := r >= epsilon
   END; { de "EstPositif" }
```
Vrai si r est positif à ε près.

Voici, en première application, la fonction signe :

```
FUNCTION signe (r: real): integer;
   BEGIN
      IF EstNul(r) THEN
         signe := 0
      ELSE IF r > 0 THEN
         signe := 1
      ELSE
         signe := -1
   END; { de "signe" }
```

Les trois procédures suivantes, à savoir :

```
PROCEDURE bip;
PROCEDURE TopChrono;
FUNCTION TempsEcoule: real;
```

dépendent du matériel utilisé. En THINK Pascal sur MacIntosh, elles s'écrivent :

```
PROCEDURE bip;
   BEGIN
      SysBeep(1)
   END; { de "bip" }
```

```
PROCEDURE TopChrono;
   BEGIN
      DebutChrono := TickCount
   END; { de "TopChrono" }
FUNCTION TempsEcoule: real;
   BEGIN
      TempsEcoule := (TickCount - DebutChrono) / 60
   END; { de "TempsEcoule" }
```

La variable `DebutChrono`, de type `longint`, est locale à la bibliothèque. En Turbo Pascal, sur PC, ces mêmes procédures s'écrivent :

```
PROCEDURE bip;
   BEGIN
      write(chr(7))
   END; { de "bip" }
PROCEDURE TopChrono;
   BEGIN
      DebutChrono := heure
   END; { de "TopChrono" }
FUNCTION TempsEcoule: real;
   BEGIN
      TempsEcoule := heure - DebutChrono
   END; { de "TempsEcoule" }
```

où `heure` est une fonction qui transforme la représentation de l'heure :

```
FUNCTION heure: real;
   VAR
      h, m, s, c : word;
   BEGIN
      GetTime(h, m, s, c);
      heure := c / 100 + s + 60 * m + 3600 * h
   END; { de "heure" }
```

Notons que l'appel à `GetTime` demande d'inclure la bibliothèque prédéfinie `Dos`.

A.3.2 Polynômes

Un polynôme à coefficients réels est représenté par la suite de ses coefficients. Une façon naturelle de représenter un polynôme est de lui associer un `RECORD` qui contient, d'une part, le degré et, d'autre part, les coefficients. Comme nous ne voulons pas utiliser cette structure, nous allons représenter un polynôme par un tableau. Il y a deux façons de faire, chacune avec ses avantages et inconvénients. Fixons d'abord le degré maximal des polynômes représentés, disons N. Un polynôme $p(X) = a_0 + a_1X + \cdots + a_nX^n$, avec $a_n \neq 0$, peut être identifié à la suite finie $(a_0, \ldots, a_N)$, avec $a_{n+1} = \cdots = a_N = 0$. Avec cette convention, un polynôme se représente par un tableau de nombres réels. On peut aussi identifier p au couple formé de son degré n, et de la suite $(a_0, \ldots, a_n)$ de ses coefficients. On range alors le degré de p dans un emplacement réservé du tableau contenant ses coefficients.

L'avantage de la première représentation, celle que nous adoptons ci-dessous, est sa simplicité : les opérations arithmétiques notamment se réalisent très facilement. L'inconvénient majeur est la lenteur du calcul du degré d'un polynôme, puisque l'on doit parcourir la suite des coefficients ; paradoxalement, le temps de calcul du degré décroît avec le degré.

L'avantage de la deuxième représentation (qui est utilisée par exemple dans le programme de factorisation de polynômes à coefficients dans $\mathbb{Z}/2\mathbb{Z}$) est donc la rapidité, mais la programmation est un peu plus délicate, dans la mesure où il faut réajuster le degré à chaque opération.

Voici les déclarations :

```
CONST
   DegreMax = 20;                       Le degré maximal des polynômes.
   DegrePolNul = -1;                    Le degré du polynôme nul.
TYPE
   pol = ARRAY[0..DegreMax] OF real;
VAR
   PolynomeUnite: pol;                  Polynôme prédéfini.
   PolynomeNul: pol;                    Polynôme prédéfini.
```

Quelques fonctions et procédures simples :

```
FUNCTION Degre (VAR p: pol): integer;                       Donne le degré de p.
FUNCTION EstPolNul (VAR p: pol): boolean;                   Teste si p = 0.
PROCEDURE ReelEnMonome (s: real; n: integer; VAR p: pol);   Donne p(X) = sX^n.
```

La plus longue de ces procédures est celle qui calcule le degré :

```
FUNCTION Degre (VAR p: pol): integer;
   Donne le degré du polynôme p.
   VAR
      n: integer;
   BEGIN
      n := DegreMax;
      WHILE (n >= 0) AND (EstNul(p[n])) DO      { AND séquentiel }
         n := n - 1;
      Degre := n
   END; { de "Degre" }
```

On voit l'avantage d'avoir choisi de représenter le degré du polynôme nul par -1. Les deux autres procédures s'écrivent comme suit :

```
FUNCTION EstPolNul (VAR p: pol): boolean;
   Teste si p = 0.
```

```
    BEGIN
       EstPolNul := Degre(p) = DegrePolNul
    END; { de "EstPolNul" }
 PROCEDURE ReelEnMonome (s: real; n: integer; VAR p: pol);
```
Donne le polynôme $p(X) = sX^n$.
```
    BEGIN
       p := PolynomeNul; p[n] := s
    END; { de "ReelEnMonome" }
```

La deuxième procédure fait appel au polynôme nul prédéfini. Celui-ci ainsi que le polynôme unité sont initialisés dans la procédure :

```
 PROCEDURE InitPolynomes;
```
Définition du polynôme nul et du polynôme unité.
```
    VAR
       n: integer;
    BEGIN
       FOR n := 0 TO DegreMax DO PolynomeNul[n] := 0;
       PolynomeUnite := PolynomeNul;
       PolynomeUnite[0] := 1;
    END; { de "InitPolynomes" }
```

Cette procédure doit être appelée avant toute manipulation. Venons-en aux procédures de lecture et d'écriture. La lecture est simple :

```
 PROCEDURE EntrerPolynome (n: integer; VAR p: pol; titre: texte);
```
Affichage du titre, puis lecture du polynôme p de degré n.
```
    VAR
       i: integer;
    BEGIN
       p := PolynomeNul;
       writeln;
       writeln(titre);
       FOR i := 0 TO n DO BEGIN
          write(i : 2, '  :  ');
          readln(p[i]);
       END
    END; { de "EntrerPolynome" }
```

Notons que l'on doit connaître le degré n avant d'utiliser la procédure. L'affichage doit respecter les usages : ne pas afficher un terme nul (sauf si le polynôme est nul), ne pas afficher un coefficient de valeur absolue égale à 1 (sauf si c'est le terme constant). Ces conventions allongent quelque peu l'écriture, mais il est très important de les respecter, pour le confort du lecteur.

```
 PROCEDURE EcrireMonome (a: real; VAR n: integer);
```
Est appelée avec $a > 0$.
```
    BEGIN
       IF (a <> 1) OR (n = 0) THEN
```

```
            write(a : 1 : precision, ' ');
         IF n > 1 THEN
            write('X^', n : 1)
         ELSE IF n = 1 THEN
            write('X')
      END; { de "EcrireMonome" }
   PROCEDURE EcrirePolynome (VAR p: pol; titre: texte);
      Affichage du titre, puis du polynôme p.
      VAR
         i, n: integer;
      BEGIN
         writeln;
         write(titre);  { La suite sur la même ligne. }
         n := Degre(p);
         IF n = DegrePolNul THEN                Le polynôme nul.
            write(0 : 1)
         ELSE BEGIN
            IF p[n] < 0 THEN write('- ');
            EcrireMonome(abs(p[n]), n);         Le premier coefficient.
            FOR i := n - 1 DOWNTO 0 DO          Les autres coefficients non nuls.
               IF NOT EstNul(p[i]) THEN BEGIN
                  IF p[i] > 0 THEN write(' + ') ELSE write(' - ');
                  EcrireMonome(abs(p[i]), i)
               END
         END;
         writeln
      END; { de "EcrirePolynome" }
```

Les opérations arithmétiques sur les polynômes sont les suivantes :

```
PROCEDURE PolynomePlusPolynome (p, q: pol; VAR r: pol);
PROCEDURE PolynomeMoinsPolynome (p, q: pol; VAR r: pol);
PROCEDURE PolynomeOppose (p: pol; VAR mp: pol);
PROCEDURE PolynomeParPolynome (p, q: pol; VAR r: pol);
PROCEDURE PolynomeParXn (p: pol; n: integer; VAR q: pol);
PROCEDURE PolynomeParMonome (p: pol; s: real; n: integer; VAR q: pol);
PROCEDURE PolynomeParConstante (p: pol; s: real; VAR q: pol);
PROCEDURE PolynomeDivEuclPolynome (a, b: pol; VAR q, r: pol);
PROCEDURE PolynomeModPolynome (a, b: pol; VAR r: pol);
PROCEDURE PolynomeDivPolynome (a, b: pol; VAR q: pol);
PROCEDURE PolynomePgcd (a, b: pol; VAR pgcd: pol);
PROCEDURE PolynomeDerive (p: pol; VAR dp: pol);
```

Enfin, l'évaluation de la fonction polynôme se fait par :

```
FUNCTION Valeur (VAR p: pol; x: real): real;
```

Une variante de cette procédure est utile, lorsque l'on connaît par ailleurs le degré de p; elle permet un gain de temps considérable :

```
FUNCTION Valeurd (d: integer; VAR p: pol; x: real): real;
```

Voici donc une réalisation de ces procédures :

```
PROCEDURE PolynomePlusPolynome (p, q: pol; VAR r: pol);          r = p + q
   VAR
      i: integer;
   BEGIN
      FOR i := 0 TO DegreMax DO r[i] := p[i] + q[i];
   END; { de "PolynomePlusPolynome" }
PROCEDURE PolynomeMoinsPolynome (p, q: pol; VAR r: pol);         r = p − q
   VAR
      i: integer;
   BEGIN
      FOR i := 0 TO DegreMax DO r[i] := p[i] - q[i];
   END; { de "PolynomeMoinsPolynome" }
PROCEDURE PolynomeOppose (p: pol; VAR mp: pol);                  mp = −p
   VAR
      i: integer;
   BEGIN
      FOR i := 0 TO DegreMax DO mp[i] := -p[i];
   END; { de "PolynomeOppose" }
PROCEDURE PolynomeParPolynome (p, q: pol; VAR r: pol);           r = pq
   VAR
      i, j: integer;
   BEGIN
      r := PolynomeNul;
      FOR i := 0 TO Degre(p) DO
         FOR j := 0 TO Degre(q) DO
            r[i + j] := r[i + j] + p[i] * q[j]
   END; { de "PolynomeParPolynome" }
PROCEDURE PolynomeParXn (p: pol; n: integer; VAR q: pol);        q = pX^n
   VAR
      i: integer;
   BEGIN
      q := PolynomeNul;
      FOR i := n TO DegreMax DO q[i] := p[i - n]
   END; { de "PolynomeParXn" }
PROCEDURE PolynomeParMonome (p: pol; s: real; n: integer; VAR q: pol);
   q = psX^n
   VAR
      i: integer;
   BEGIN
      q := PolynomeNul;
      FOR i := n TO n + Degre(p) DO q[i] := s * p[i - n]
   END; { de "PolynomeParMonome" }
PROCEDURE PolynomeParConstante (p: pol; s: real; VAR q: pol);    q = ps
```

```
   VAR
      i: integer;
   BEGIN
      q := PolynomeNul;
      FOR i := 0 TO Degre(p) DO q[i] := s * p[i]
   END; { de "PolynomeParConstante" }
PROCEDURE PolynomeDivEuclPolynome (a, b: pol; VAR q, r: pol);       $a = bq + r$
   VAR
      m, n: integer;
      s: real;
      p: pol;
   BEGIN
      q := PolynomeNul; r := a;
      n := Degre(r); m := Degre(b);
      WHILE n >= m DO BEGIN
         s := r[n] / b[m];
         q[n - m] := s;
         PolynomeParMonome(b, s, n - m, p);     $p(X) = sX^{n-m}b(X)$.
         PolynomeMoinsPolynome(r, p, r);        $r(X) := r(X) - sX^{n-m}b(X)$.
         n := Degre(r)
      END;
   END; { de "PolynomeDivEuclPolynome" }
PROCEDURE PolynomeModPolynome (a, b: pol; VAR r: pol);           $r = a \bmod b$
   VAR
      q: pol;
   BEGIN
      PolynomeDivEuclPolynome(a, b, q, r)
   END; { de "PolynomeModPolynome" }
PROCEDURE PolynomeDivPolynome (a, b: pol; VAR q: pol);           $q = a \operatorname{div} b$
   VAR
      r: pol;
   BEGIN
      PolynomeDivEuclPolynome(a, b, q, r)
   END; { de "PolynomeDivPolynome" }
PROCEDURE PolynomePgcd (a, b: pol; VAR pgcd: pol);               $\mathrm{pgcd}(a, b)$
   VAR
      r: pol;
   BEGIN
      WHILE NOT EstPolNul(b) DO BEGIN
         PolynomeModPolynome(a, b, r);
         a := b; b := r
      END;
      PolynomeParConstante(a, 1 / a[Degre(a)], a);
      pgcd := a
   END; { de "PolynomePgcd" }
PROCEDURE PolynomeDerive (p: pol; VAR dp: pol);                  $dp = d'$
   VAR
```

```
      i: integer;
   BEGIN
      dp := PolynomeNul;
      FOR i := 1 TO Degre(p) DO dp[i - 1] := i * p[i]
   END; { de "PolynomeDerive" }
```

Bien entendu, l'évaluation d'un polynôme se fait par le schéma de Horner :

```
FUNCTION Valeurd (d: integer; VAR p: pol; x: real): real;
   Calcule la valeur p(x) du polynôme p de degré d au point x.
   VAR
      i: integer;
      r: real;
   BEGIN
      r := 0;
      FOR i := d DOWNTO 0 DO r := r * x + p[i];
      Valeurd := r
   END; { de "Valeurd" }
FUNCTION Valeur (VAR p: pol; x: real): real;
   Calcule la valeur p(x) de p au point x.
   BEGIN
      Valeur := Valeurd(Degre(p), p, x)
   END; { de "Valeur" }
```

A.3.3 Les complexes

Nous présentons dans cette section une bibliothèque destinée à la manipulation des nombres complexes. On déclare

```
TYPE
   complexe = ARRAY[0..1] OF real;
```

Pour accéder aux complexes et pour les construire, on utilise les fonctions et procédures suivantes :

```
FUNCTION Re (z: complexe): real;
FUNCTION Im (z: complexe): real;
FUNCTION ModuleAuCarre (z: complexe): real;
FUNCTION Module (z: complexe): real;
FUNCTION Arg (z: complexe): real;
PROCEDURE FixerRe (VAR z: complexe; r: real);
PROCEDURE FixerIm (VAR z: complexe; r: real);
PROCEDURE ReelEnComplexe (VAR z: complexe; r: real);
PROCEDURE CartesienEnComplexe (a, b: real; VAR z: complexe);
PROCEDURE PolaireEnComplexe (rho, theta: real; VAR z: complexe);
```

Voici une implémentation.

Donne $\Re(z)$.

```
FUNCTION Re (z: complexe): real;
   BEGIN
      Re := z[0]
   END; { de "Re" }
```

Donne $\Im(z)$.

```
FUNCTION Im (z: complexe): real;
   BEGIN
      Im := z[1]
   END; { de "Im" }
```

Donne $|z|^2$.

```
FUNCTION ModuleAuCarre (z: complexe): real;
   BEGIN
      ModuleAuCarre := Sqr(Re(z)) + Sqr(Im(z))
   END; { de "ModuleAuCarre"}
```

Donne $|z|$.

```
FUNCTION Module (z: complexe): real;
   BEGIN
      Module := Sqrt(ModuleAuCarre(z))
   END; { de "Module"}
```

Donne l'argument de z.

```
FUNCTION Arg (z: complexe): real;
  VAR
   a, b, x: real;
   VAR
      a, b, x: real;
   BEGIN
      a := Re(z);
      b := Im(z);
      IF a <> 0 THEN
         Arg := ArcTan(b / a)
      ELSE IF b > 0 THEN
         Arg := Pi/2
      ELSE IF b < 0 THEN
         Arg := -Pi/2
      ELSE
         Arg := 0
   END;
```

$\Re(z) = r$

```
PROCEDURE FixerRe (VAR z: complexe; r: real);
   BEGIN
      z[0] := r
   END; { de "FixerRe" }
```

$\Im(z) = r$

```
PROCEDURE FixerIm (VAR z: complexe; r: real);
   BEGIN
      z[1] := r
   END; { de "FixerIm" }
```

$z = r$

```
PROCEDURE ReelEnComplexe (VAR z: complexe; r: real);
   BEGIN
      z[0] := r; z[1] := 0
   END; { de "ReelEnComplexe" }
```

$z = a + ib$

```
PROCEDURE CartesienEnComplexe (a, b: real; VAR z: complexe);
```

```
      BEGIN
         z[0] := a; z[1] := b
      END;
   PROCEDURE PolaireEnComplexe (rho, theta: real; VAR z: complexe); z = ρe^{iθ}
      BEGIN
         z[0] := rho * cos(theta); z[1] := rho * sin(theta)
      END;
```

Les opérations arithmétiques de base sont les suivantes :

```
   PROCEDURE ComplexePlusComplexe (u, v: complexe; VAR w: complexe);
   PROCEDURE ComplexeMoinsComplexe (u, v: complexe; VAR w: complexe);
   PROCEDURE ComplexeParComplexe (u, v: complexe; VAR w: complexe);
   PROCEDURE ComplexeSurComplexe (u, v: complexe; VAR w: complexe);
   PROCEDURE ComplexeParReel (z: complexe; r: real; VAR w: complexe);
   PROCEDURE ComplexeOppose (z: complexe; VAR moinsz: complexe);
   PROCEDURE Conjugue (z: complexe; VAR zbarre: complexe);
```

Elles sont réalisées par :

```
   PROCEDURE ComplexePlusComplexe (u, v: complexe; VAR w: complexe);w = u + v
      VAR
         a, b: real;
      BEGIN
         a := Re(u) + Re(v); b := Im(u) + Im(v);
         CartesienEnComplexe(a, b, w);
      END; { de "ComplexePlusComplexe" }
   PROCEDURE ComplexeMoinsComplexe (u, v: complexe; VAR w: complexe);w = u − v
      VAR
         a, b: real;
      BEGIN
         a := Re(u) - Re(v); b := Im(u) - Im(v);
         CartesienEnComplexe(a, b, w);
      END; { de "ComplexeMoinsComplexe" }
   PROCEDURE ComplexeParComplexe (u, v: complexe; VAR w: complexe); w = uv
      VAR
         a, b: real;
      BEGIN
         a := Re(u) * Re(v) - Im(u) * Im(v);
         b := Re(u) * Im(v) + Im(u) * Re(v);
         CartesienEnComplexe(a, b, w);
      END; { de "ComplexeParComplexe" }
   PROCEDURE ComplexeOppose (z: complexe; VAR moinsz: complexe);    Donne −z
      BEGIN
         CartesienEnComplexe(-Re(z), -Im(z), moinsz);
      END; { de "ComplexeOppose" }
   PROCEDURE Conjugue (z: complexe; VAR zbarre: complexe);          Donne z̄
      BEGIN
```

```
      zbarre := z;
      FixerIm(zbarre, -Im(z));
   END; { de "Conjugue" }
PROCEDURE ComplexeSurComplexe (u, v: complexe; VAR w: complexe); w = u/v
   On suppose v ≠ 0.
   VAR
      a, b, r: real;
   BEGIN
      r := ModuleAuCarre(v);
      a := (Re(u) * Re(v) + Im(u) * Im(v)) / r;
      b := (Im(u) * Re(v) - Re(u) * Im(v)) / r;
      CartesienEnComplexe(a, b, w)
   END; { de "ComplexeSurComplexe" }
PROCEDURE ComplexeParReel (z: complexe; r: real; VAR w: complexe);w = rz
   BEGIN
      CartesienEnComplexe(r * Re(z), r * Im(z), w)
   END; { de "ComplexeParReel" }
```

Il est commode de pouvoir disposer des complexes 0 et 1. On les déclare comme variables

```
VAR
   ComplexeZero, ComplexeUn: complexe;
```

et leur valeurs sont définies dans une procédure d'initialisation :

```
PROCEDURE InitComplexes;
   BEGIN
      ReelEnComplexe(ComplexeZero, 0);
      ReelEnComplexe(ComplexeUn, 1)
   END; { de "InitComplexes" }
```

Une procédure permet d'échanger les valeurs de deux paramètres :

```
PROCEDURE EchangerC (VAR u, v: complexe);
   VAR
      w: complexe;
   BEGIN
      w := u; u := v; v := w
   END; { de "EchangerC" }
```

Comme dans le cas des réels, on utilise un test de nullité «à epsilon près» :

```
FUNCTION EstCNul (z: complexe): boolean;
   Vrai si |z| est nul à ε près.
   BEGIN
      EstCNul := Module(z) < epsilon
   END; { de "EstCNul" }
```

Enfin, la lecture et l'affichage des complexes se font par deux procédures :

```
PROCEDURE EntrerComplexe (VAR z: complexe; titre: texte);
PROCEDURE EcrireComplexe (z: complexe; titre: texte);
```

La procédure de lecture ne présente pas de difficulté :

```
PROCEDURE EntrerComplexe (VAR z: complexe; titre: texte);
   VAR
      a, b: real;
   BEGIN
      writeln; writeln(titre);
      write('Partie reelle : '); readln(a);
      write('Partie imaginaire : '); readln(b);
      CartesienEnComplexe(a, b, z)
   END; { de "EntrerComplexe" }
```

La procédure d'affichage est compliquée par les problèmes de mise en page. Il faut en effet éviter des affichages tels que $2+1i$, $1-1i$, $0+3i$, ou $1+\ -2i$. Pour pouvoir utiliser la procédure pour l'affichage des matrices, il faut aussi que la taille totale d'un affichage soit indépendante du nombre complexe. On définit d'abord des constantes :

```
CONST
   AligneAGauche = -1;
   Centre = 0;
   AligneADroite = 1;
```

La procédure suivante permet d'afficher un nombre complexe en mode «aligné à gauche», «aligné à droite» ou «centré».

```
PROCEDURE FormaterComplexe (z: complexe; Mode: integer);
   VAR
      a, b, r: real;
      n: integer;
   FUNCTION LargeurReel (a: real): integer;
```

Donne le nombre de positions requises pour l'affichage d'un réel. Composé de trois parties : le signe (0 ou 2 positions), la partie entière, et la mantisse (1 + précision).

```
      VAR
         LargeurPartieEntiere, LargeurSigne: integer;
      BEGIN
         LargeurPartieEntiere := 1 + max(0, trunc(log10(abs(a))));
         LargeurSigne := 1 - signe(a);
         LargeurReel := LargeurSigne + LargeurPartieEntiere + 1 + precision
      END; { de "LargeurReel" }
   FUNCTION LargeurComplexe (a, b: real): integer;
```

Donne le nombre de positions requises pour afficher le complexe $a + ib$. Composé des largeurs de a et de b, et d'un éventuel groupe de liaison.

```
      VAR
         Largeura, Largeurb, Liaison: integer;
      BEGIN
```

Largeur de a est nulle si et seulement si $a = 0$.

```
      IF EstNul(a) THEN
         Largeura := 0
      ELSE
         Largeura := LargeurReel(a);
```
Il y a un signe entre a et b si et seulement si $ab \neq 0$. Mais si $b < 0$, le signe de b est compris dans la largeur de b.
```
      IF EstNul(a * b) THEN
         Liaison := 0
      ELSE
         Liaison := 2 + signe(b);
```
Largeur de b est nulle si et seulement si $b = 0$. Si $|b| = 1$, on n'écrit pas 1.
```
      IF EstNul(b) THEN
         Largeurb := 0
      ELSE IF EstNul(abs(b) - 1) THEN
         Largeurb := 2 - signe(b)
      ELSE
         Largeurb := 2 + LargeurReel(b);
```
Cas traité séparément : le complexe nul.
```
      IF EstNul(a) AND EstNul(b) THEN
         LargeurComplexe := 1
      ELSE
         LargeurComplexe := Largeura + Liaison + Largeurb
   END; { de "LargeurComplexe" }

PROCEDURE DebutMiseEnPage (Mode, NombreDeBlancs: integer);
   BEGIN
      IF Mode = AligneADroite THEN
         write(' ' : NombreDeBlancs);
      IF Mode = Centre THEN
         write(' ' : NombreDeBlancs DIV 2)
   END; { de "DebutMiseEnPage" }

PROCEDURE FinMiseEnPage (Mode, NombreDeBlancs: integer);
   BEGIN
      IF Mode = AligneAGauche THEN
         write(' ' : NombreDeBlancs);
      IF Mode = Centre THEN
         write(' ' : NombreDeBlancs - NombreDeBlancs DIV 2)
   END; { de "FinMiseEnPage" }

PROCEDURE EcrireReel (a: real);
   BEGIN
      CASE signe(a) OF
         1: write(a : (precision + 2) : precision);
         0: ;
        -1: write('- ', -a : (precision + 2) : precision)
      END
   END; { de "EcrireReel" }

BEGIN { de "FormaterComplexe" }
   a := Re(z);
```

```
      b := Im(z);
      n := 3 + 2 * FormatDunComplexe - LargeurComplexe(a, b);
      DebutMiseEnPage(Mode, n);
      IF EstNul(a) AND EstNul(b) THEN write('0')
      ELSE BEGIN
         EcrireReel(a);
         IF NOT EstNul(a * b) THEN
            IF (b >= 0) AND NOT EstNul(b) THEN
               write(' + ')
            ELSE
               write(' ');
         IF NOT EstNul(b) THEN BEGIN
            IF EstNul(b - 1) THEN write('i')
            ELSE IF EstNul(b + 1) THEN
               write('- i')
            ELSE BEGIN
               EcrireReel(b);
               write(' i')
            END
         END
      END;
      FinMiseEnPage(Mode, n);
   END; { de "FormaterComplexe" }
```

Voici quelques exemples de sortie dans les trois modes.

```
0                      |          0            |                   0|
1.00                   |         1.00          |                1.00|
244.12                 |        244.12         |              244.12|
- 1.00                 |        - 1.00         |              - 1.00|
- 245.12               |       - 245.12        |            - 245.12|
i                      |          i            |                   i|
2.00 i                 |        2.00 i         |              2.00 i|
- i                    |         - i           |                 - i|
- 51.00 i              |       - 51.00 i       |           - 51.00 i|
1.00 + i               |       1.00 + i        |            1.00 + i|
- 1.00 + i             |      - 1.00 + i       |          - 1.00 + i|
1.00 - i               |       1.00 - i        |            1.00 - i|
- 1.00 - i             |      - 1.00 - i       |          - 1.00 - i|
- 245.12 + 2.00 i      |   - 245.12 + 2.00 i   |     - 245.12 + 2.00 i|
- 245.12 - i           |     - 245.12 - i      |        - 245.12 - i|
- 245.12 - 51.00 i     |  - 245.12 - 51.00 i   |    - 245.12 - 51.00 i|
0.09 + 0.40 i          |     0.09 + 0.40 i     |         0.09 + 0.40 i|
2.00 - 153.00 i        |    2.00 - 153.00 i    |       2.00 - 153.00 i|
```

Le plus souvent, on utilisera la procédure suivante :

```
   PROCEDURE EcrireComplexe (z: complexe; titre: texte);
   BEGIN
```

```
      write(titre);
      FormaterComplexe(z, AligneAGauche)
   END;
```

A.3.4 Les rationnels

Les nombres rationnels sont représentés par des couples d'entiers, normalisés de telle sorte que numérateur et dénominateur soient premiers entre eux et que le dénominateur soit positif.

On déclare

```
TYPE
   rat = ARRAY[0..1] OF integer;
```

Pour accéder aux rationnels et pour les construire, on utilise les fonctions et procédures suivantes :

```
FUNCTION Numerateur (VAR r: rat): integer;
FUNCTION Denominateur (VAR r: rat): integer;
PROCEDURE FixerNumerateur (VAR r: rat; n: integer);
PROCEDURE FixerDenominateur (VAR r: rat; d: integer);
PROCEDURE EntierEnRat (VAR r: rat; e: integer);
PROCEDURE FaireRat (n, d: integer; VAR r: rat);
```

Ces procédures font évidemment appel à une procédure de calcul du pgcd. Voici une implémentation. La procédure la plus importante crée un rationnel à partir de deux entiers :

```
PROCEDURE FaireRat (n, d: integer; VAR r: rat);
   VAR
      p: integer;
   BEGIN
      n := signe(d) * n;
      d := abs(d);                Le dénominateur est positif.
      p := pgcd(n, d);
      r[0] := n DIV p;            Numérateur et dénominateur ...
      r[1] := d DIV p;            ... sont premiers entre eux.
   END; { de "FaireRat" }
```

Les autres procédures s'écrivent comme suit :

```
FUNCTION Numerateur (VAR r: rat): integer;
   BEGIN
      Numerateur := r[0]
   END; { de "Numerateur" }

FUNCTION Denominateur (VAR r: rat): integer;
   BEGIN
      Denominateur := r[1]
```

```
   END; { de "Denominateur" }

PROCEDURE FixerNumerateur (VAR r: rat; n: integer);
   BEGIN
      r[0] := n
   END; { de "FixerNumerateur" }

PROCEDURE FixerDenominateur (VAR r: rat; d: integer);
   BEGIN
      r[1] := d
   END; { de "FixerDenominateur" }

PROCEDURE EntierEnRat (VAR r: rat; e: integer);
   BEGIN
      FaireRat(e, 1, r)
   END; { de "EntierEnRat" }
```

Les opérations arithmétiques sont les suivantes :

```
PROCEDURE RatPlusRat (u, v: rat; VAR w: rat);
PROCEDURE RatMoinsRat (u, v: rat; VAR w: rat);
PROCEDURE RatOppose (u: rat; VAR v: rat);
PROCEDURE RatParRat (u, v: rat; VAR w: rat);
PROCEDURE RatParEntier (u: rat; x: integer; VAR w: rat);
PROCEDURE RatSurRat (u, v: rat; VAR w: rat);
PROCEDURE RatSurEntier (u: rat; x: integer; VAR w: rat);
```

Elles sont réalisées par :

```
PROCEDURE RatPlusRat (u, v: rat; VAR w: rat);              w = u + v
   VAR
      n, d: integer;
   BEGIN
      n := Numerateur(u) * Denominateur(v) + Denominateur(u) * Numerateur(v);
      d := Denominateur(u) * Denominateur(v);
      FaireRat(n, d, w);
   END; { de "RatPlusRat" }

PROCEDURE RatMoinsRat (u, v: rat; VAR w: rat);             w = u − v
   VAR
      n, d: integer;
   BEGIN
      n := Numerateur(u) * Denominateur(v) - Denominateur(u) * Numerateur(v);
      d := Denominateur(u) * Denominateur(v);
      FaireRat(n, d, w);
   END; { de "RatMoinsRat" }

PROCEDURE RatOppose (u: rat; VAR v: rat);                  v = −u
   BEGIN
      v := u;
      FixerNumerateur(v, -Numerateur(v));
```

```
   END; { de "RatOppose" }

PROCEDURE RatParRat (u, v: rat; VAR w: rat);                w = u v
   VAR
      n, d: integer;
   BEGIN
      n := Numerateur(u) * Numerateur(v);
      d := Denominateur(u) * Denominateur(v);
      FaireRat(n, d, w)
   END; { de "RatParRat" }

PROCEDURE RatParEntier (u: rat; x: integer; VAR w: rat);    w = u x
   BEGIN
      FaireRat(x * Numerateur(u), Denominateur(u), w);
   END; { de "RatParEntier" }

PROCEDURE RatSurRat (u, v: rat; VAR w: rat);                w = u/v
   VAR
      n, d: integer;
   BEGIN
      n := Numerateur(u) * Denominateur(v);
      d := Denominateur(u) * Numerateur(v);
      FaireRat(n, d, w)
   END; { de "RatSurRat" }

PROCEDURE RatSurEntier (u: rat; x: integer; VAR w: rat);    w = u/x
   BEGIN
      FaireRat(Numerateur(u), x * Denominateur(u), w)
   END; { de "RatSurEntier" }
```

Il est commode de pouvoir disposer des entiers 0 et 1 comme rationnels. On les déclare comme variables (on pourrait aussi avoir des procédures spécifiques) :

```
VAR
   RatZero, RatUn: rat;
```

Les valeurs sont définies dans une procédure d'initialisation :

```
PROCEDURE InitRationnels;
   BEGIN
      EntierEnRat(RatZero, 0);
      EntierEnRat(RatUn, 1)
   END; { de "InitRationnels" }
```

Enfin, la lecture et l'affichage des rationnels se font par deux procédures sur le modèle déjà rencontré plusieurs fois :

```
PROCEDURE EntrerRat (VAR a: rat; titre: texte);
   VAR
      n, d: integer;
   BEGIN
```

```
      writeln; write(titre);
      readln(n, d);
      FaireRat(n, d, a)
   END; { de "EntrerRat" }

PROCEDURE EcrireRat (VAR a: rat; titre: texte);
   BEGIN
      write(titre); write(Numerateur(a) : 1);
      IF Denominateur(a) <> 1 THEN
         write('/', Denominateur(a) : 1)
   END; { de "EcrireRat" }
```

A.4 Menus

Comme nous l'avons déjà dit, nos programmes sont tous interactifs. Ils donnent parfois le choix entre plusieurs calculs, et demandent toujours des données. Il n'existe pas de recette universelle pour organiser le dialogue. Dans nos programmes, les choix se présentent sous la forme d'un menu; une option du menu, accessible par exemple par `?`, peut contenir un ou plusieurs écrans d'explications. Un programme a donc la forme suivante :

```
PROGRAM Nom;
USES
   Les bibliothèques;
VAR
   Variables globales ;
PROCEDURE Menu;
   ...
   END { de "Menu" };
BEGIN
   Initialisations ;
   En-tête ;
   REPEAT
      Menu
   UNTIL false;
END.
```

Les variables globales sont les variables qui doivent conserver leur valeur entre deux appels du menu. Les initialisations comprennent notamment les procédures d'initialisation des diverses bibliothèques utilisées. L'en-tête décrit brièvement l'objet du programme. Le menu lit les choix, et l'une des options fait arrêter le programme par la commande `halt`. La procédure a donc approximativement la forme suivante :

```
PROCEDURE Menu;
   VAR
      choix: char;
```

```
   Autres variables, contingentes
BEGIN
   writeln;
   write('Donner votre choix :  ');
   readln(choix);
   CASE choix OF
      '?':           Les choix offerts.
         BEGIN
         END;
      Autre choix :
         BEGIN
         END;
      Autre choix :
         BEGIN
         END;
      Autre choix :
         BEGIN
         END;
      '.':           Fin.
         BEGIN
         writeln;
         writeln('J''arrête.');
         halt
      END;
      OTHERWISE      Choix inexistant; ELSE en Turbo Pascal.
         bip
   END { case }
END; { de "Menu" }
```

Comme nous l'avons déjà dit, la syntaxe est légèrement différente en Turbo Pascal, où `OTHERWISE` est remplacé par `ELSE` sans point-virgule. La procédure `bip`, elle aussi, s'écrit différemment en Think Pascal (`SysBeep(1)`) et en Turbo Pascal (`write(chr(7))`). Il nous a paru commode de paramétrer quelques données; le menu offre la possibilité de modifier les valeurs de ces paramètres. La procédure du menu contient donc les choix suivants :

```
'!':       Bascule test;
   BEGIN
      IF test THEN BEGIN
         test := false;
         writeln('Maintenant hors test')
      END
      ELSE BEGIN
         test := true;
         writeln('Maintenant en test')
      END
   END;
'#':       Modification de la précision d'affichage.
```

```
      BEGIN
         writeln('Précision d''affichage actuelle :  ', precision : 1);
         write('Donner la nouvelle valeur :  ');
         readln(precision);
      END;
   '&':        Modification de ε.
      BEGIN
         writeln('Actuellement, epsilon vaut :  ', epsilon : 10 : 8);
         write('Donner la nouvelle valeur :  ');
         readln(epsilon);
      END;
```

Le premier paramètre, initialisé à faux, permet de basculer entre deux modes lors du test d'un programme. En effet, un programme ne marche parfois pas du premier coup. Il est utile de prévoir, à certains endroits que l'on estime appropriés, des impressions intermédiaires. Ces impressions peuvent d'ailleurs aussi être utiles pour se faire une idée du déroulement du programme, de la vitesse de convergence, etc. On active et désactive les impressions intermédiaires au moyen de la variable `test`, en faisant précéder chaque impression intermédiaire par la condition `IF test THEN`. On verra un exemple plus bas. Il existe des directives de compilation spécifiques aux compilateurs qui permettent de ne pas compiler ces tests, une fois le programme au point.

Les deux autres paramètres concernent la précision d'affichage, c'est-à-dire le nombre de décimales souhaitées, et la valeur de ε, qui est utilisé pour tester qu'un réel est «nul».

Voici un exemple complet d'une procédure de menu. Il s'agit du menu d'un programme qui teste les diverses opérations offertes dans la bibliothèque concernant les polynômes à coefficients réels. Le programme lui-même a l'allure suivante :

```
PROGRAM LibPolynomes;
USES
   general, polynomes;
VAR
   n: integer;
   p, q, r: pol;
PROCEDURE Menu;
 ...
   END; { de "menu" }
BEGIN
   showtext;
   InitGeneral;
   InitPolynomes;
   writeln;
   writeln('Manipulation de polynômes à coefficients reels.');
   writeln('Taper ''?'' pour informations.');
   REPEAT
      Menu
   UNTIL false;
```

```
END.
```

La procédure `Menu` est un peu longue, parce que le nombre de choix offerts est élevé.

```
PROCEDURE Menu;
   VAR
      choix: char;
      u: pol;
      x, t: real;
   BEGIN
      writeln;
      write('Donner votre choix :  ');
      readln(choix);
      CASE choix OF
      '?':
         BEGIN
            writeln('p, q, r : Lecture du polynôme p, q, r');
            writeln('- : r:=p-q');
            writeln('+ : r:=p+q');
            writeln('* : r:=p*q');
            writeln('/ : p:=qa+r');
            writeln('n : p:=p*X^n');
            writeln('c : r:=pgcd(p,q)');
            writeln('d : r:=dérivé(p)');
            writeln('v : Valeur p(x)');
            writeln('x : p <-- q, q <-- r, r <-- p');
            writeln('. : Fin');
         END;
      'p':   Lecture polynôme p
         BEGIN
            write('Donner le degré de p :   ');
            readln(n);
            EntrerPolynome(n, p, 'Entrer les coefficients de p');
            write('Voici le polynôme lu :');
            EcrirePolynome(p, 'p(X) = ');
         END;
      'q':   Lecture polynôme q
         BEGIN
            write('Donner le degré de q :   ');
            readln(n);
            EntrerPolynome(n, q, 'Entrer les coefficients de q');
            write('Voici le polynôme lu :');
            EcrirePolynome(q, 'q(X) = ');
         END;
      'r':   Lecture polynôme r
         BEGIN
            write('Donner le degré de r :   ');
            readln(n);
            EntrerPolynome(n, r, 'Entrer les coefficients de r');
```

```
      write('Voici le polynôme lu :');
      EcrirePolynome(r, 'r(X) = ');
   END;
'+':   Somme
   BEGIN
      EcrirePolynome(p, 'p(X) = ');
      EcrirePolynome(q, 'q(X) = ');
      PolynomePlusPolynome(p, q, r);
      EcrirePolynome(r, 'r(X) = p(X) + q(X) = ');
   END;
'-':   Différence
   BEGIN
      EcrirePolynome(p, 'p(X) = ');
      EcrirePolynome(q, 'q(X) = ');
      PolynomeMoinsPolynome(p, q, r);
      EcrirePolynome(r, 'r(X) = p(X) - q(X) = ');
   END;
'*':   Produit
   BEGIN
      EcrirePolynome(p, 'p(X) = ');
      EcrirePolynome(q, 'q(X) = ');
      PolynomeParPolynome(p, q, r);
      EcrirePolynome(r, 'r(X) = p(X) * q(X) = ');
   END;
'/':   Division euclidienne
   BEGIN
      EcrirePolynome(p, 'p(X) = ');
      EcrirePolynome(q, 'q(X) = ');
      TopChrono;
      PolynomeDivEuclPolynome(p, q, a, r);
      t := TempsEcoule;
      EcrirePolynome(a, 'p/q  = ');
      EcrirePolynome(r, 'r(X) = ');
   END;
'c':   Pgcd
   BEGIN
      EcrirePolynome(p, 'p(X) = ');
      EcrirePolynome(q, 'q(X) = ');
      TopChrono;
      PolynomePgcd(p, q, r);
      EcrirePolynome(r, 'pgcd = ');
      t := TempsEcoule;
      writeln;
      writeln('Le calcul a pris ', t : 1 : 8, '  secondes');
   END;
'd':   Dérivée
   BEGIN
      EcrirePolynome(p, 'p(X) = ');
```

```
      TopChrono;
      PolynomeDerive(p, r);
      t := TempsEcoule;
      EcrirePolynome(r, 'p'' ( X ) = ');
   END;
'n':   Somme
   BEGIN
      write('Donner le degré de X^n :   ');
      readln(n);
      EcrirePolynome(p, 'p(X) = ');
      PolynomeParXn(p, n, p);
      EcrirePolynome(p, 'p.X^n= ');
   END;
'v':   Valeur
   BEGIN
      write('Donner la valeur de x :   ');
      readln(x);
      EcrirePolynome(p, 'p(X) = ');
      writeln('p(', x : 1 : precision, ') = ',
         Valeurd(Degre(p), p, x) : 1 : precision);
   END;
'x':
   BEGIN
      u := p;
      p := q;
      q := r;
      r := u;
      EcrirePolynome(p, 'p(X) = ');
      EcrirePolynome(q, 'q(X) = ');
      EcrirePolynome(r, 'r(X) = ')
   END;
'!':      Bascule test
   BEGIN
      IF test THEN BEGIN
         test := false;
         writeln('Maintenant hors test')
      END
      ELSE BEGIN
         test := true;
         writeln('Maintenant en test')
      END
   END;
'#':      Modification de la précision d'affichage
   BEGIN
      writeln('Précision d''affichage actuelle :  ', precision : 1);
      write('Donner la nouvelle valeur :  ');
      readln(precision);
   END;
```

```
        '&':      Modification de ε
           BEGIN
              writeln('Actuellement, epsilon vaut :  ', epsilon : 10 : 8);
              write('Donner la nouvelle valeur :  ');
              readln(epsilon);
           END;
        '.':      Fin
           BEGIN
              writeln;
              writeln('J''arrête.');
              halt
           END;
           OTHERWISE   Choix inexistant
              bip
        END { case }
     END; { de "Menu" }
```

Considérons la procédure de calcul du pgcd de deux polynômes :

```
PROCEDURE PolynomePgcd (a, b: pol; VAR pgcd: pol);
   VAR
      r: pol;
   BEGIN
      WHILE NOT EstPolNul(b) DO BEGIN
         PolynomeModPolynome(a, b, r);
         IF test THEN
            EcrirePolynome(r, '--> ');
         a := b;
         b := r
      END;
      PolynomeParConstante(a, 1 / a[Degre(a)], a);
      pgcd := a
   END; { de "PolynomePgcd" }
```

La variable de test, si on lui a affecté la valeur vrai, provoque l'impression intermédiaire :

```
Donner votre choix :  c

p(X) = X^5 + 3.0 X^4 + 3.0 X^3 + X^2
q(X) = - X^4 + X^2
--> 4.0 X^3 + 4.0 X^2
--> 0
pgcd = X^3 + X^2
Le calcul a pris 1.61666667  secondes
```

Les procédures `TopChrono` et `TempsEcoule` sont expliquées dans la présentation de la bibliothèque générale. Elles dépendent du matériel utilisé.

Annexe B

Les bibliothèques

Dans cette annexe sont regroupées les déclarations de types et les en-têtes de procédures des principales bibliothèques.

B.1 Généralités

```
CONST
   pi = 3.1415926536;              Prédéfinie en Turbo Pascal.
TYPE
   texte = string[80];
VAR
   precision: integer;
   test: boolean;
   epsilon: real;
```

Initialisation :

```
PROCEDURE InitGeneral;
```

Gestion du temps :

```
PROCEDURE TopChrono;
FUNCTION TempsEcoule: real;
```

Fonctions arithmétiques sur des entiers :

```
FUNCTION min (i, j: integer): integer;
FUNCTION max (i, j: integer): integer;
PROCEDURE EchangerE (VAR u, v: integer);
FUNCTION PuissanceE (x: integer; n: integer): integer;
FUNCTION Binomial (n, k: integer): integer;
```

```
FUNCTION pgcd (a, b: integer): integer;
```

Fonctions arithmétiques sur des réels :

```
FUNCTION rmin (x, y: real): real;
FUNCTION rmax (x, y: real): real;
PROCEDURE Echanger (VAR u, v: real);
FUNCTION log10 (x: real): real;
FUNCTION PuissanceR (x, y: real): real;          Calcule x^y.
FUNCTION Puissance (x: real; n: integer): real;  Calcule x^n.
FUNCTION signe (r: real): integer;
FUNCTION EstNul (r: real): boolean;
FUNCTION EstPositif (r: real): boolean;
```

Divers :

```
PROCEDURE bip;
```

B.2 Calcul matriciel

```
CONST
   OrdreMax = 18;
TYPE
   vec = ARRAY[1..OrdreMax] OF real;
   mat = ARRAY[1..OrdreMax] OF vec;
   vecE = ARRAY[1..OrdreMax] OF integer;

VAR
   MatriceUnite: mat;
   MatriceNulle: mat;
```

Initialisation :

```
PROCEDURE InitMatrices;
```

Lecture-écriture :

```
PROCEDURE EntrerMatrice (n: integer; VAR a: mat; titre: texte);
PROCEDURE EntrerMatriceRect (m, n: integer; VAR a: mat; titre: texte);
PROCEDURE EntrerVecteur (n: integer; VAR a: vec; titre: texte);
PROCEDURE EntrerMatriceSymetrique (n: integer; VAR a: mat; titre: texte);
PROCEDURE EcrireMatrice (n: integer; VAR a: mat; titre: texte);
PROCEDURE EcrireMatriceRect (m, n: integer; VAR a: mat; titre: texte);
PROCEDURE EcrireVecteur (n: integer; VAR a: vec; titre: texte);
PROCEDURE EcrireVecteurE (n: integer; VAR a: vecE; titre: texte);
```

Opérations :

```
PROCEDURE MatriceParMatrice (n: integer; a, b: mat; VAR ab: mat);
PROCEDURE MatricePlusMatrice (n: integer; a, b: mat; VAR ab: mat);
PROCEDURE MatriceParMatriceRect (m, n, p: integer; a, b: mat; VAR ab: mat);
PROCEDURE MatriceParVecteur (n: integer; a: mat; x: vec; VAR ax: vec);
PROCEDURE VecteurParMatrice (n: integer; x: vec; a: mat; VAR xa: vec);
PROCEDURE VecteurPlusVecteur (n: integer; a, b: vec; VAR ab: vec);
PROCEDURE VecteurParScalaire (n: integer; x: vec; s: real; VAR sx: vec);
PROCEDURE MatriceRectParVecteur (m, n: integer; a: mat; x: vec; VAR ax: vec);
PROCEDURE Transposer (n: integer; a: mat; VAR ta: mat);
PROCEDURE TransposerRect (m, n: integer; a: mat; VAR ta: mat);
FUNCTION ProduitScalaire (n: integer; VAR a, b: vec): real;
FUNCTION Norme (n: integer; VAR a: vec): real;
FUNCTION NormeInfinie (n: integer; VAR a: vec): real;
```

Systèmes :

```
PROCEDURE SystemeTriangulaireSuperieur (n: integer; a: mat; b: vec;
   VAR x: vec);
PROCEDURE SystemeParGauss (n: integer; a: mat; b: vec; VAR x: vec;
   VAR inversible: boolean);
PROCEDURE InverseParGauss (n: integer; a: mat; VAR ia: mat;
   VAR inversible: boolean);
PROCEDURE SystemeParJordan (n: integer; a: mat; b: vec; VAR x: vec;
   VAR inversible: boolean);
PROCEDURE InverseParJordan (n: integer; a: mat; VAR ia: mat;
   VAR inversible: boolean);
FUNCTION Determinant (n: integer; a: mat): real;
```

B.3 Polynômes

```
CONST
   DegreMax = 20;
   DegrePolNul = -1;
TYPE
   pol = ARRAY[0..DegreMax] OF real;
VAR
   PolynomeUnite: pol;
   PolynomeNul: pol;
```

Initialisation :

```
PROCEDURE InitPolynomes;
```

Lecture-écriture :

```
PROCEDURE EntrerPolynome (n: integer; VAR p: pol; titre: texte);
PROCEDURE EcrirePolynome (VAR p: pol; titre: texte);
```

Constructeurs :

```
FUNCTION Degre (VAR p: pol): integer;
FUNCTION EstPolNul (VAR p: pol): boolean;
PROCEDURE ReelEnMonome (s: real; n: integer; VAR p: pol);
```

Opérations arithmétiques :

```
PROCEDURE PolynomePlusPolynome (p, q: pol; VAR r: pol);
PROCEDURE PolynomeMoinsPolynome (p, q: pol; VAR r: pol);
PROCEDURE PolynomeOppose (p: pol; VAR mp: pol);
PROCEDURE PolynomeParPolynome (p, q: pol; VAR r: pol);
PROCEDURE PolynomeParXn (p: pol; n: integer; VAR q: pol);
PROCEDURE PolynomeParMonome (p: pol; s: real; n: integer; VAR q: pol);
PROCEDURE PolynomeParConstante (p: pol; s: real; VAR q: pol);
PROCEDURE PolynomeDivEuclPolynome (a, b: pol; VAR q, r: pol);
PROCEDURE PolynomeModPolynome (a, b: pol; VAR r: pol);
PROCEDURE PolynomeDivPolynome (a, b: pol; VAR q: pol);
PROCEDURE PolynomePgcd (a, b: pol; VAR pgcd: pol);
PROCEDURE PolynomeDerive (p: pol; VAR dp: pol);
```

Evaluation de la fonction polynôme :

```
FUNCTION Valeur (VAR p: pol; x: real): real;
FUNCTION Valeurd (d: integer; VAR p: pol; x: real): real;
```

B.4 Nombres complexes

```
TYPE
   complexe = ARRAY[0..1] OF real;
```

Lecture-écriture :

```
PROCEDURE EntrerComplexe (VAR z: complexe; titre: texte);
PROCEDURE EcrireComplexe (z: complexe; titre: texte);
PROCEDURE FormaterComplexe (z: complexe; Mode: integer);
```

Constructeurs :

```
FUNCTION Re (z: complexe): real;
FUNCTION Im (z: complexe): real;
FUNCTION ModuleAuCarre (z: complexe): real;
FUNCTION Module (z: complexe): real;
FUNCTION Arg (z: complexe): real;
```

```
PROCEDURE FixerRe (VAR z: complexe; r: real);
PROCEDURE FixerIm (VAR z: complexe; r: real);
PROCEDURE ReelEnComplexe (VAR z: complexe; r: real);
PROCEDURE CartesienEnComplexe (a, b: real; VAR z: complexe);
PROCEDURE PolaireEnComplexe (rho, theta: real; VAR z: complexe);
```

Opérations :

```
PROCEDURE ComplexePlusComplexe (u, v: complexe; VAR w: complexe);
PROCEDURE ComplexeMoinsComplexe (u, v: complexe; VAR w: complexe);
PROCEDURE ComplexeParComplexe (u, v: complexe; VAR w: complexe);
PROCEDURE ComplexeSurComplexe (u, v: complexe; VAR w: complexe);
PROCEDURE ComplexeParReel (z: complexe; r: real; VAR w: complexe);
PROCEDURE ComplexeOppose (z: complexe; VAR moinsz: complexe);
PROCEDURE Conjugue (z: complexe; VAR zbarre: complexe);
```

B.5 Nombres rationnels

```
TYPE
   rat = ARRAY[0..1] OF integer;

VAR
   RatZero, RatUn: rat;
```

Initialisation :

```
PROCEDURE InitRationnels;
```

Lecture-écriture :

```
PROCEDURE EntrerRat (VAR a: rat; titre: texte);
PROCEDURE EcrireRat (VAR a: rat; titre: texte);
```

Constructeurs :

```
FUNCTION Numerateur (VAR r: rat): integer;
FUNCTION Denominateur (VAR r: rat): integer;
PROCEDURE FixerNumerateur (VAR r: rat; n: integer);
PROCEDURE FixerDenominateur (VAR r: rat; d: integer);
PROCEDURE EntierEnRat (VAR r: rat; e: integer);
PROCEDURE FaireRat (n, d: integer; VAR r: rat);
```

Opérations :

```
PROCEDURE RatPlusRat (u, v: rat; VAR w: rat);
PROCEDURE RatMoinsRat (u, v: rat; VAR w: rat);
PROCEDURE RatParRat (u, v: rat; VAR w: rat);
PROCEDURE RatSurRat (u, v: rat; VAR w: rat);
PROCEDURE RatParEntier (u: rat; x: integer; VAR w: rat);
PROCEDURE RatSurEntier (u: rat; x: integer; VAR w: rat);
PROCEDURE RatOppose (u: rat; VAR v: rat);
```

B.6 Mots

```
CONST
   LongueurMot = 25;
TYPE
   mot = ARRAY[0..LongueurMot] OF integer;
```

Lecture-écriture :

```
PROCEDURE EntrerMot (VAR u: mot; titre: texte);
PROCEDURE EcrireMot (VAR u: mot; titre: texte);
```

Constructeurs :

```
FUNCTION Longueur (VAR u: mot): integer;
PROCEDURE FixerLongueur (VAR u: mot; n: integer);
FUNCTION EstMotVide (VAR u: mot): boolean;
PROCEDURE LettreEnMot (VAR u: mot; x: integer);
PROCEDURE LeSuffixe (u: mot; i: integer; VAR v: mot);
PROCEDURE LePrefixe (u: mot; i: integer; VAR v: mot);
PROCEDURE LeFacteur (u: mot; i, j: integer; VAR v: mot);
PROCEDURE LeConjugue (u: mot; i: integer; VAR v: mot);
```

Opération :

```
PROCEDURE Concatener (u, v: mot; VAR w: mot);
```

B.7 Entiers en multiprécision

```
CONST
   base = 100;
   digitparchiffre = 2;
   TailleMax = 20;
   TailleEntierNul = -1;
TYPE
   chiffre = integer;
   entier = ARRAY[-2..TailleMax] OF chiffre;
```

Lecture-écriture :

```
PROCEDURE EntrerEntier (VAR u: entier; titre: texte);
PROCEDURE EcrireEntier (VAR u: entier; titre: texte);
```

Constructeurs :

```
FUNCTION EstEntierNul (VAR u: entier): boolean;
PROCEDURE EntierNul (VAR u: entier);
FUNCTION Taille (VAR u: entier): integer;
PROCEDURE FixerTaille (VAR u: entier; p: integer);
FUNCTION LeSigne (VAR u: entier): integer;
PROCEDURE FixerSigne (VAR u: entier; s: integer);
PROCEDURE ChiffreEnEntier (x: chiffre; VAR u: entier);
```

Opérations arithmétiques :

```
PROCEDURE EntierPlusEntier (u, v: entier; VAR w: entier);
PROCEDURE EntierMoinsEntier (u, v: entier; VAR w: entier);
PROCEDURE EntierParChiffre (u: entier; x: chiffre; VAR w: entier);
PROCEDURE EntierSurChiffre (u: entier; x: chiffre; VAR w: entier);
PROCEDURE EntierParEntier (u, v: entier; VAR w: entier);
PROCEDURE EntierDivEuclEntier (u, v: entier; VAR q, r: entier);
PROCEDURE EntierModEntier (u, v: entier; VAR w: entier);
PROCEDURE EntierDivEntier (u, v: entier; VAR w: entier);
```

Opérations sur entiers naturels :

```
PROCEDURE ChiffreEnNaturel (x: chiffre; VAR u: entier);
PROCEDURE NaturelPlusNaturel (VAR u, v, w: entier);
PROCEDURE NaturelMoinsNaturel (VAR u, v, w: entier);
PROCEDURE NaturelParNaturel (u, v: entier; VAR w: entier);
PROCEDURE NaturelDivEuclNaturel (u, v: entier; VAR q, r: entier);
FUNCTION CompareNaturel (VAR u, v: entier): integer;
PROCEDURE NaturelParMonome (u: entier; x:chiffre; n:integer; VAR w: entier);
PROCEDURE NaturelParChiffre (u: entier; x: chiffre; VAR w: entier);
PROCEDURE NaturelPlusChiffre (u: entier; x: chiffre; VAR w: entier);
PROCEDURE NaturelSurChiffre (u: entier; x: chiffre; VAR w: entier);
```

B.8 Arithmétique flottante

```
CONST
   base = 100;
   digitparChiffre = 2;
   TailleMax = 16;
TYPE
   chiffre = integer;
   Flottant = ARRAY[-1..LaTailleMax] OF chiffre;
VAR
   FlottantNul: Flottant;
```

Initialisation :

```
PROCEDURE InitFlottants;
```

Lecture-écriture :

```
PROCEDURE EntrerFlottant (VAR u: Flottant; titre: texte);
PROCEDURE EcrireFlottant (VAR u: Flottant; titre: texte);
```

Constructeurs :

```
FUNCTION EstFlottantNul (VAR u: Flottant): boolean;
FUNCTION Exposant (VAR u: Flottant): chiffre;
PROCEDURE FixerExposant (VAR u: Flottant; p: chiffre);
FUNCTION LeSigne (VAR u: Flottant): chiffre;
PROCEDURE FixerSigne (VAR u: Flottant; s: chiffre);
PROCEDURE ChiffreEnFlottant (VAR u: Flottant; x: chiffre);
FUNCTION FlottantProches (VAR u, v: Flottant): boolean;
```

Opérations arithmétiques :

```
PROCEDURE FlottantPlusFlottant (u, v: Flottant; VAR w: Flottant);
PROCEDURE FlottantMoinsFlottant (u, v: Flottant; VAR w: Flottant);
PROCEDURE FlottantParFlottant (u, v: Flottant; VAR w: Flottant);
PROCEDURE FlottantSurFlottant (u, v: Flottant; VAR w: Flottant);
PROCEDURE FlottantParChiffre (VAR u: Flottant; x: chiffre);
PROCEDURE FlottantSurChiffre (VAR u: Flottant; x: chiffre);
PROCEDURE InverseFlottant (a: Flottant; VAR b: Flottant);
PROCEDURE RacineFlottant (a: Flottant; VAR b: Flottant);
PROCEDURE UnSurRacineFlottant (a: Flottant; VAR b: Flottant);
```

B.9 Géométrie

```
TYPE
        Point = ARRAY[0..1] OF real;
```

Initialisation :

```
PROCEDURE InitGeometrie;
```

Lecture-écriture :

```
PROCEDURE SaisirPoint (VAR q: Point);
PROCEDURE TracerSegment (p, q: Point);
PROCEDURE TracerDemiDroite (p, d: Point);
PROCEDURE MarquerPointCarre (p: Point; couleur: integer);
PROCEDURE MarquerPointRond (p: Point; couleur: integer);
PROCEDURE NumeroterPoint (p: Point; n: integer);
```

Fonctions d'accès et constructeur :

```
FUNCTION Abscisse (VAR q: Point): real;
FUNCTION Ordonnee (VAR q: Point): real;
PROCEDURE FairePoint (x, y: real; VAR q: Point);
```

Manipulations géométriques de base :

```
FUNCTION det (VAR p, q, r: Point): real;
FUNCTION EstSitue (VAR p, q, r: Point): integer;
FUNCTION EstAGauche (VAR p, q, r: Point): boolean;
FUNCTION EstADroite (VAR p, q, r: Point): boolean;
FUNCTION CarreNorme (VAR p: Point): real;
FUNCTION CarreDistance (VAR p, q: Point): real;
```

Conversions :

```
PROCEDURE PointEnPixel (p: Point; VAR a: Pixel);
PROCEDURE PixelEnPoint (c: Pixel; VAR q: Point);
```

Graphique

```
TYPE
        Pixel = ARRAY[0..1] OF integer;
VAR
        HauteurEcran: integer;
        LargeurEcran: integer;
```

Initialisation :

```
PROCEDURE InitGraphik;
```

Opérations :

```
PROCEDURE TracerCarre (a: Pixel; r: integer);
PROCEDURE TracerDisque (a: Pixel; r: integer);
PROCEDURE LireCurseur (VAR curseur: Pixel);
PROCEDURE RelierPixels (a, b: Pixel);
PROCEDURE EcrireGraphique (a: Pixel; n: integer);
```

Index

V