# Cours du module I2

J.Bétréma

Automne 1996

## Table des matières

Ce cours est une introduction à la programmation, pour étudiants scientifiques. Il traite essentiellement de la façon d'exprimer, dans un langage de programmation classique, des *algorithmes* mathématiques simples.

Un algorithme est une procédure automatique de calcul : il existe par exemple des algorithmes pour décomposer un entier en facteurs premiers, pour calculer de bonnes approximations des fonctions trigonométriques, ou pour résoudre un système d'équations linéaires.

Il existe de "bons" et de "mauvais" algorithmes ; les mauvais sont ceux dont la *complexité* est excessive. La complexité a un sens précis en algorithmique : c'est une mesure du temps d'exécution d'un algorithme, en fonction de la taille des données (par exemple en fonction du nombre de chiffres de l'entier à décomposer en facteurs premiers, ou de la précision d'une approximation, ou du nombre d'équations à résoudre).

Il existe de nombreux problèmes pour lesquels il existe de bons algorithmes ; ce cours en donnera un échantillon. Il faut cependant savoir qu'il en existe d'autres pour lesquels il est démontré qu'il n'existe aucun algorithme (théorie de la calculabilité), ou aucun "bon" algorithme (théorie de la complexité).

Le langage de programmation choisi est *C*, simplement parce que c'est devenu le langage le plus courant sur les stations de travail scientifiques, qui utilisent presque toutes le système d'exploitation Unix. Il n'est pas difficile, une fois compris les principes de C, de passer à un autre langage de programmation classique, comme Pascal ou Fortran ; ces langages sont dits *impératifs*, ce qui signifie que les algorithmes y sont traduits pas à pas, en une suite d'instructions que l'ordinateur exécute fidèlement. Il existe d'autres types de langages de programmation, qui adoptent une approche plus globale : programmation fonctionnelle, programmation équationnelle, programmation logique, etc. . . ; nous n'en parlerons pas ici.

Un programme en C n'est pas directement exécutable par le processeur d'une machine ; il faut donc le traduire (le terme technique est *compiler*) ; toutes les stations de travail scientifiques possèdent un compilateur C.

Les travaux pratiques n'ont pas lieu sur des stations de travail Unix, mais sur des ordinateurs personnels, avec l'environnement de travail de la firme Borland ; l'apprentissage de cet environnement est indispensable pour entrer, compiler et exécuter des programmes, mais n'a pas plus d'intérêt que l'apprentissage du mode d'emploi d'un magnétoscope. Il n'en sera jamais question en cours ; des fiches aide-mémoire seront distribuées en TP.

# 1 Sommes et produits

## 1.1 Expressions

En C comme dans la plupart des langages de programmation, on peut manipuler des expressions algébriques simples à partir des quatre opérateurs de base : addition, soustraction, multiplication, division. On utilise les conventions mathématiques usuelles, sauf que le signe de multiplication est `*` et ne peut être omis. La division a même priorité que la multiplication ; à priorité égale, les opérations sont effectuées de gauche à droite. Exemples :

| Expression C | Formule mathématique |
|---|---|
| `a*b+c` | $ab + c$ |
| `a*(b+c)` | $a(b + c)$ |
| `a/b+c` | $\frac{a}{b} + c$ |
| `a/(b*c)` | $\frac{a}{bc}$ |
| `a*a+b*b` | $a^2 + b^2$ |

Cette section est consacrée au calcul des sommes et produits d'un nombre variable de termes, exprimés en mathématiques avec les symboles $\sum$ et $\prod$. Noter qu'en particulier la fonction *puissance*, qui est un produit de $n$ termes, n'est pas une fonction élémentaire du langage C (car ce n'est une fonction élémentaire pour aucun processeur).

## 1.2 Factorielle

```
long factorielle (int n) {
  int k;
  long f = 1;

  for (k = 2; k <= n; k++)
    f = f * k;
  return f;
}
```

FIG. 1 – Factorielle

La fonction

$$n! = 1.2.3 \dots (n-1)n = \prod_{k=1}^{n} k$$

se traduit en C comme sur la figure 1. Une *fonction C* se compose de deux parties :

- la *déclaration*, appelée aussi *prototype*, qui spécifie le domaine de départ et le domaine d'arrivée ;
- la *définition*, appelée aussi *corps*, qui spécifie l'algorithme de calcul de la fonction.

Lorsqu'on utilise une fonction, l'important, formellement, est sa déclaration ; la définition peut donc être reportée plus loin. Dans le cadre de ce cours, la définition suivra toujours immédiatement la déclaration ; noter que la définition est toujours isolée de la déclaration par des accolades.

La fonction *factorielle* est une fonction $\mathbf{N} \to \mathbf{N}$ ; la déclaration en est la traduction en langage C :

```
long factorielle (int n)
```

Une fonction C a un *nom* : `factorielle` et un ou plusieurs (ou zéro) *paramètres* – ici `n` –, toujours entre parenthèses. Chaque paramètre est précédé de son *type*, (en mathématiques on dit plutôt *domaine*), qui indique l'ensemble des valeurs possibles ; `int` désigne le type *nombre entier*. Le type qui précède la fonction elle-même indique le domaine d'arrivée : `long` est le type *nombre entier long*. La section suivante explique pourquoi il existe, bizarrement, plusieurs types d'entiers.

Noter que la convention qui consiste à placer le type *avant* le paramètre ou la fonction, est une spécialité, un peu déroutante au début, du langage C.

Etudions maintenant le corps (autrement dit la définition) de la fonction *factorielle* :

- $k$ et $f$ sont des variables *locales*, c'est à dire internes au calcul de la fonction, et qui ne servent que le temps de ce calcul. Chaque fonction possède ses propres variables locales, qui sont distinctes de celles des autres fonctions, même si elles portent le même nom ; on dit aussi qu'elles sont *privées*.
- Toutes les variables doivent être déclarées : il faut indiquer leur nom, précédé comme d'habitude de leur type. L'absence de lettres grecques, souvent utilisées comme noms de variables en mathématiques, est compensée par la possibilité d'utiliser des noms de plusieurs caractères, comme `alpha`. De même un nom comme $x'$ est interdit en C, et peut être remplacé par `xprime`, ou `x1`.
- La variable $f$ est *initialisée* en même temps qu'elle est déclarée : elle reçoit la valeur 1 ; tant qu'une variable n'a pas reçu de valeur, elle est indéfinie. L'une des fautes les plus courantes est d'oublier d'initialiser une variable, c'est à dire de lui donner une première valeur ; dans ce cas le résultat du calcul est imprévisible.
- La construction `for (k = 2; k <= n; k++)` indique que l'instruction suivante doit être répétée, d'abord avec $k = 2$, puis avec $k = 3$, etc..., tant que $k \leq n$. La forme générale de cette construction de *boucle* est :

  `for` *(initialisation ; condition ; incrémentation)*

  et l'instruction qui suit (appelée *corps de boucle*) est exécutée tant que la *condition* est vraie ; entre deux tours de boucle, l'*incrémentation* est exécutée : en général, une variable (ici $k$) est modifiée. La forme `k++` est une particularité du langage C : c'est une abréviation de $k = k + 1$ ; de même, `k--` abrège $k = k - 1$. Voir figure 2.
- L'instruction à répéter est `f = f * k`. Ce n'est pas une équation (on aurait alors $f = 0$), mais une affectation : on multiplie $f$ par $k$, puis on place le résultat dans $f$. Dans la plupart des langages de programmation, le signe d'affectation est `:=` , pour bien distinguer une affectation d'une égalité ; mais les créateurs de C ont préféré simplement `=` , par souci de concision.
- L'instruction `return f` spécifie quelle est la valeur calculée (on dit souvent *retournée*) par la fonction.

Les valeurs successives de $f$ sont donc :

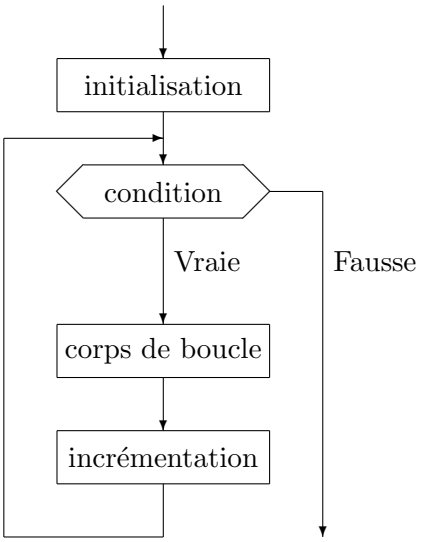


FIG. 2 – Organigramme d'une boucle *for*

$$
\begin{array}{lcl}
f & = & 1 \\
f & = & 1 \times 2 \quad = \quad 2 \\
f & = & 2 \times 3 \quad = \quad 6 \\
f & = & 6 \times 4 \quad = \quad 24 \\
\ldots &&
\end{array}
$$

La valeur finale de $f$ (pour $k = n$) est donc bien $n!$ . Lorsque $n < 2$, la boucle

```
for (k = 2; k <= n; k++)
```

n'est jamais exécutée, car la condition $k \leq n$ est fausse dès le début ; la valeur retournée est donc 1, ce qui est correct, puisque $0! = 1! = 1$.

Toute instruction et toute déclaration se terminent par un point-virgule ; à part cela, la mise en page est sans importance pour le compilateur, qui ne fait pas de différence entre la fonction précédente et celle-ci :

```
long factorielle(int n){int k;long
f=1;for(k=2;k<=n;k++)f=f*k;return f;}
```

Mais, pour faciliter la lecture du programme par un humain, il est vivement conseillé de respecter le style de mise en page employé dans ce cours, et adopté, à quelques détails près, par tous les programmeurs.

## 1.3 Types entiers

Un entier est codé en binaire, et un processeur n'est capable d'opérer directement que sur des entiers de longueur fixe $p$ ; typiquement $p = 16$, 32, ou 64 chiffres binaires. Donc lorsqu'on ajoute 1 au plus grand entier possible (celui dont les $p$ chiffres sont égaux à 1), on perd la retenue, et on trouve zéro comme résultat.

Les opérations arithmétiques du processeur sont donc *circulaires* ; autrement dit un processeur calcule modulo $2^p$. Noter que cette arithmétique *modulaire*, très classique en mathématiques, possède heureusement les propriétés d'associativité, commutativité et distributivité de l'arithmétique ordinaire.

D'autre part un entier modulaire est une classe d'équivalence d'entiers, et chaque classe contient une infinité d'entiers positifs et une infinité d'entiers négatifs : la notion de signe n'a pas de sens en arithmétique modulaire. Cependant, en informatique, on convient de choisir, dans chaque classe, l'unique représentant $n$ qui vérifie :

$$-2^{p-1} \leq n < 2^{p-1}.$$

Les entiers dont le chiffre binaire le plus à gauche vaut 1, sont donc considérés comme négatifs ; en particulier, celui dont les $p$ chiffres sont égaux à 1, représente -1. Avec cette convention, en ajoutant 1 à -1, on obtient, et c'est heureux, 0, mais on a une discontinuité en ajoutant 1 au plus grand entier positif, $2^{p-1} - 1$ : le résultat est $-2^{p-1}$. On dit que le calcul a débordé (*overflow*).

En informatique, le kilo vaut $1K = 2^{10} = 1024$, le méga vaut $1M = 2^{20} = 1\ 048\ 576$, et le giga vaut $1G = 2^{30} = 1\ 073\ 741\ 824$. Le compilateur de TURBO-C interprète le type `int` comme désignant les entiers relatifs sur 16 chiffres binaires (donc compris entre $-32\ 768 = -32K$ et 32 767) et le type `long` comme désignant les entiers relatifs sur 32 chiffres binaires (donc compris entre -2 gigas et 2 gigas -1). Noter que pour des entiers courts (16 bits), on a le paradoxe expliqué ci-dessus : $32\ 767 + 1 = -32\ 768$ !

On peut (mais c'est rare) spécifier qu'on préfère les entiers naturels aux entiers relatifs en ajoutant `unsigned` (traduire : sans signe) devant `int` ou `long`. Le plus grand entier de type `int` vaut alors $65\ 535 = 64K - 1$. Pour ce type d'entiers, la discontinuité apparaît en ajoutant 1 à ce nombre : le résultat est nul !

Note : il est possible de savoir, en examinant un registre spécial du processeur, si un débordement s'est produit lors d'une opération arithmétique. En exploitant, entre autres, cette propriété, certains logiciels, comme *Maple*, effectuent des calculs exacts sur des entiers possédant un nombre arbitraire de chiffres ; les opérations sont effectuées en utilisant des algorithmes voisins de ceux appris à l'école élémentaire, mais au lieu d'utiliser les tables d'opérations pour les dix chiffres du système décimal, on utilise le processeur pour les calculs intermédiaires, en notant les retenues éventuelles.

## 1.4 Séries

Le calcul approché de

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

peut se programmer comme sur la figure 3. La fonction *module* est une fonction auxiliaire, qui calcule $|x|$, et est utilisée par la fonction *exponentielle*. Ce programme utilise les constructions nouvelles suivantes du langage C :

– Les nombres réels sont représentés de façon approximative dans l'ordinateur par des *nombres en virgule flottante*, `float` ou `double` en C. La différence entre ces deux types est une différence de précision (voir section suivante) ; l'habitude est d'utiliser la *double* précision.

```
#define EPSILON 1e-7

double module (double x) {
  if (x >= 0) return x; else return -x;
}

double exponentielle (double x) {
  int n;
  double s = 1, u = 1;

  for (n = 1; module (u) > EPSILON; n++) {
    u = u * x / n;
    s = s + u;
  }
  return s;
}
```

FIG. 3 – Exponentielle

- La construction `if` *(condition) instruction* `else` *instruction* est la traduction C d'une alternative. Attention : *si* $a = b$ se traduit en C par `if (a == b)`, avec un double signe d'égalité.
- Les accolades après la boucle `for` permettent de regrouper les instructions à répéter.
- La directive `#define` permet de donner un nom à une constante ; ainsi le rôle de cette constante est plus clair, et on peut changer sa valeur en changeant simplement sa définition ; pour faciliter la compréhension des programmes, il est traditionnel de noter en majuscules les identificateurs définis de la sorte. La notation `1e-7` est la traduction C de $10^{-7}$.

Les valeurs successives de $u$, en fonction de $x$, sont :

$$1\ ,\ x\ ,\ x \times x/2 = \frac{x^2}{2}\ ,\ \frac{x^2}{2} \times x/3 = \frac{x^3}{6}\ ,\ \frac{x^3}{6} \times x/4 = \frac{x^4}{24}\ ,\ \ldots$$

Ce sont bien les termes de la série qui définit $e^x$. et les valeurs successives de $s$ sont les sommes partielles de cette série. On arrête le calcul dès que le terme général de la série devient, en module, inférieur à $\epsilon$ ; bien que cela ne soit pas correct du point de vue mathématique, on considère en pratique qu'on a obtenu une approximation de la somme de la série à $\epsilon$ près. Un autre test d'arrêt possible est $|u/s| < \epsilon$, car les approximations flottantes ont une précision *relative* fixe.

### 1.5 Représentation des réels en virgule flottante

Les nombres réels sont représentés de façon approximative en mémoire, avec une convention standardisée de la forme $m \times 2^e$, où $m$ est la mantisse, comprise entre 1 et 2, et $e$ l'exposant. On utilise $p$ chiffres binaires pour les "décimales" binaires de $m$, $q$ chiffres binaires pour l'exposant, et un dernier chiffre binaire pour le signe. La position de la virgule dépend donc de l'exposant, d'où le nom de *virgule flottante* (*floating point* chez les Anglo-saxons).

Pour le format *court* (32 chiffres binaires au total), on a : $p = 23, q = 8$, ce qui permet grosso modo de représenter des nombres compris, en valeur absolue, entre $2^{-128} \simeq 10^{-38}$ et $2^{128} \simeq 10^{38}$, avec 7 chiffres décimaux significatifs (car $2^{23} \simeq 10^7$). Pour le format *long* (64 chiffres binaires au total), on a : $p = 52, q = 11$, ce qui permet de représenter des nombres compris, en valeur absolue, entre $2^{-1024} \simeq 10^{-308}$ et $2^{1024} \simeq 10^{308}$, avec 15 chiffres décimaux significatifs (car $2^{52} \simeq 10^{15}$).

Le type `float` (resp. `double`) correspond, en Turbo-C, au format court (resp. long). Les calculs sur ces types sont effectués par une unité spéciale du processeur, la FPU, et sont beaucoup plus lents que les calculs sur les entiers ; on exprime la vitesse de cette FPU en Flops (Floating operations per second).

Attention aux **erreurs d'arrondi** : bien qu'on puisse représenter des nombres $\epsilon$ très petits en valeur absolue, on a : $1 + \epsilon = 1$ dès que le résultat exact exige trop de chiffres significatifs, ce qui arrive approximativement pour $\epsilon < 10^-7$ en format court, et $\epsilon < 10^-15$ en format long. En règle générale, tout calcul tel que l'ordre de grandeur du résultat soit beaucoup plus petit que celui des valeurs intermédiaires, fournit un résultat non fiable ; exemple :

```
exponentielle
Entrer un reel : -1
exp(-1.000000) = 0.367879
Valeur exacte =  0.367879

Entrer un reel : -5
exp(-5.000000) = 0.00673842
Valeur exacte =  0.00673795

Entrer un reel : -10
exp(-10.000000) = -6.25618e-05
Valeur exacte =    4.53999e-05
```

## 1.6 Entrées-sorties

Le reste du cours sera consacré à l'écriture de fonctions variées. Mais pour utiliser une fonction, il faut, à un moment ou un autre, entrer (le terme technique est *lire*) des valeurs au clavier, et afficher (le terme technique est *écrire*) des résultats sur l'écran ; ces activités sont regroupées sous le nom d'*entrées-sorties* (*input/output* en anglais, et *I/O* en abrégé).

La fonction de nom `main` est, par convention, celle exécutée en premier lorsqu'on demande l'exécution d'un programme ; dans le cadre de cet enseignement, c'est elle qui regroupe toutes les activités d'entrée-sortie, et elle ne fournit pas de résultat, ce qu'on indique par le mot `void` pour le type du résultat. La figure 4 donne sa forme type, et un exemple d'exécution :

- `printf` est la fonction qui imprime (f pour format) ; `scanf` est la fonction qui lit ce que l'utilisateur frappe au clavier.
- Les guillemets délimitent un texte ; `%d` est un format : d pour décimal ; il signifie que le nombre doit être lu, ou affiché, sous forme d'une suite de chiffres décimaux. `%ld` est un autre format : ld pour décimal long ; il signifie que le nombre à convertir est un entier long.

  Les autres formats importants sont `%f` et `%g` pour les nombres en virgule flottante (`float` ou `double`) : le premier format produit un nombre en virgule fixe, le second utilise la notation scientifique si nécessaire.

```
void main () {
  int n;

  printf ("Entrer un entier : ");
  scanf ("%d", &n);
  printf ("%d! = %ld\n", n, factorielle (n) );
}

Entrer un entier : 7
7! = 5040
```

FIG. 4 – Fonction principale

– L'instruction de lecture `scanf` attribue une valeur à la variable $n$, qu'il est absolument nécessaire de faire précéder du signe `&`, pour des raisons qui seront expliquées plus tard. Les formats importants pour `scanf` sont `%d` (entier de type `int`), `%ld` (entier de type `long`), `%f` (réel de type `float`), et `%lf` (réel de type `double`).

## 1.7 Exercices

1. Ecrire une fonction qui calcule un coefficient binomial, en utilisant la fonction `factorielle` et la formule :
$$C_n^k = \frac{n!}{k!(n-k)!}$$
Expliquer l'inconvénient majeur de cette formule, et transformer la fonction pour appliquer la formule :
$$C_n^k = \frac{n(n-1)\dots(n-k+1)}{1.2\dots k}$$
de façon à éviter tout débordement inutile.
2. Un dérangement est une permutation sans point fixe. Le nombre $d_n$ de dérangements de $n$ éléments vérifie l'étonnante récurrence suivante :
$$d_n = nd_{n-1} + (-1)^n$$
Ecrire la fonction C `derangement` qui calcule $d_n$.
3. Ecrire une fonction qui calcule $x^n$, pour $x$ entier, et $n$ entier positif. Modifier cette fonction pour l'étendre au cas où l'exposant est négatif.
TP : utiliser cette fonction pour afficher les entiers de type `int` compris entre $2^{15} - 10$ et $2^{15} + 10$ ; constater la discontinuité expliquée section 1.3. Adapter le programme aux entiers de type `long`, puis aux entiers naturels (types `unsigned int` ou `unsigned long`).
4. Comparer le comportement de la fonction `exponentielle`, pour un argument $x$ grand, selon que le test d'arrêt est $|u| < \epsilon$ ou $|u/s| < \epsilon$.
5. Ecrire des fonctions qui calculent $\sin x$ et $\cos x$ à partir de leurs développements en séries usuels.

6. Ecrire une fonction qui calcule $\ln x$ à partir de son développement en série :

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^n}{n}$$

   Quel problème mathématique rencontre-t-on ?

7. Ecrire une fonction qui calcule $(1+x)^\alpha$ (pour $\alpha$ réel) à partir du développement ;

$$\begin{aligned} (1+x)^\alpha &= 1 + \alpha x + \frac{\alpha(\alpha-1)}{2} x^2 + \frac{\alpha(\alpha-1)(\alpha-2)}{3!} x^3 + \ldots \\ &= \sum_{n=0}^{\infty} \frac{\alpha(\alpha-1)\ldots(\alpha-n+1)}{n!} x^n \end{aligned}$$

   Tester en TP avec $x = 1,\ \alpha = 1/2\,;\ x = 2,\ \alpha = -1/2$ .

# 2 Suites récurrentes et équations numériques

Diverses suites récurrentes permettent de trouver rapidement une solution approchée de l'équation $f(x) = 0$, dans un intervalle où cette équation admet une seule solution. On suppose toujours $f$ continue sur cet intervalle.

La première section présente d'abord le calcul des suites récurrentes dans le cas général.

## 2.1 Suites récurrentes

Soit une suite $u$ définie par la donnée de $u_0 = a$ et par la relation de récurrence $u_{n+1} = f(u_n)$. Le fragment de programme C suivant calcule $u_n$ :

```
u = a;
for (i = 1; i <= n; i++)
  u = f (u);
```

Remarquer que les indices ont disparu. Le calcul se complique un peu pour les récurrences à plusieurs termes.

Soit une suite $u$ définie par la donnée de $u_0 = a,\ u_1 = b$ et par la relation de récurrence $u_{n+1} = f(u_n, u_{n-1})$, pour $n > 0$. Le fragment de programme C suivant calcule $u_n$, pour $n > 0$ :

```
u = a; v = b;
for (i = 2; i <= n; i++)
  { w = f (u, v); u = v; v = w; }
```

Remarquer que trois variables (et non deux) sont nécessaires, et que l'ordre des 3 instructions à répéter est fondamental. On peut évidemment généraliser aux récurrences à $k$ termes, qui exigent k+1 variables.

## 2.2 Dichotomie

Soit $[a\ b]$ un intervalle tel que $f(a) < 0$ et $f(b) > 0$ ; comme $f$ est continue, l'équation $f(x) = 0$ admet au moins une solution sur $[a\ b]$ (si en outre $f$ est monotone, cette solution est unique). L'algorithme de dichotomie peut se formuler comme suit :

> soit $c$ le milieu de $[a\ b]$ ; si $f(c) < 0$ remplacer $[a\ b]$ par $[c\ b]$, et recommencer ;
> sinon remplacer $[a\ b]$ par $[a\ c]$, et recommencer. Arrêter lorsque $|b - a| < \epsilon$.

Avec des notations mathématiques, on dira que la double suite définie par :

$$a_0 = a\ ,\ b_0 = b\ ,\ c_n = \frac{a_n + b_n}{2}\ ,\ \begin{cases} a_{n+1} = c_n\ , & b_{n+1} = b_n & \text{si } f(c_n) < 0 \\ a_{n+1} = a_n\ , & b_{n+1} = c_n & \text{si } f(c_n) \geq 0 \end{cases}$$

converge vers une racine de l'équation. En effet, la suite $a_n$ est croissante, et majorée par $b$ ; de même la suite $b_n$ est décroissante, et minorée par $a$ ; enfin :

$$b_n - a_n = \frac{b - a}{2^n}$$

Donc ces deux suites convergent vers une même valeur $x$ ; d'autre part on a toujours $f(a_n) < 0$ et $f(b_n) > 0$, donc $f(x) = 0$ par continuité.

La traduction en langage C (voir figure 5) utilise la construction de boucle :

```
double dichotomie (double a, double b, double epsilon) {
  double c;
  while (b - a > epsilon) {
    c = (a + b) / 2;
    if (f(c) < 0)
      a = c;
    else
      b = c;
  }
  return c;
}
```

FIG. 5 – Résolution d'équation par dichotomie

`while` (*condition*) *instruction*;

qui est analogue, en plus simple, à la construction avec `for`. La condition est testée en début de boucle ; si l'on souhaite effectuer le test en fin de boucle (ce qui est assez rare) on emploie la construction :

`do` *instruction*; `while` (*condition*);

(voir un exemple section 2.4).

Complexité de l'algorithme : à chaque itération, on divise par deux la longueur de l'intervalle $[a\ b]$ ; en trois itérations, on gagne donc approximativement une décimale pour la précision du résultat ; la complexité de l'algorithme, en fonction du nombre $n$ de décimales correctes souhaitées, est donc de la forme $kn$, avec $k = \log_2 10 \simeq 3$ . On dit que l'algorithme est *linéaire*.

## 2.3 Méthode de Lagrange

Connaissant deux valeurs $a, b$ suffisamment proches de la racine cherchée, on remplace la courbe d'équation $y = f(x)$ par la droite qui joint les points $(a, f(a))$ et $(b, f(b))$ ; cette droite coupe l'axe des abscisses au point $c$ donné par :

$$c = \frac{af(b) - bf(a)}{f(b) - f(a)}$$

et on remplace $(a, b)$ par $(b, c)$. Rien ne garantit a priori que la suite des valeurs de $c$ converge, mais c'est le cas si les valeurs de départ sont choisies correctement.

Voir la figure 6 pour la traduction en C. Comme la convergence n'est pas sûre, on a ajouté une variable *itération* (qui compte le nombre d'itérations), et stoppé les calculs si cette variable devient trop grande. Le symbole `&&` se lit "et ensuite" ; de même, `||` signifie "ou alors". La variable *itération* a été déclarée en dehors de la fonction : c'est une variable *globale*, dont on peut examiner la valeur après appel de la fonction ; au contraire, une variable *locale*, déclarée à l'intérieur de la fonction, est inconnue en dehors de celle-ci.

Il existe une bibliothèque standard de fonctions mathématiques ; pour l'utiliser, il faut placer la directive de compilation :

```
#define MAX 50

int iteration;

double lagrange (double a, double b, double epsilon) {
  double c;

  for (iteration = 0;
       fabs (b - a) > epsilon && iteration < MAX;
       iteration++) {
    c = (a * f(b) - b * f(a)) / (f(b) - f(a));
    a = b;
    b = c;
  }
  return b;
}
```

FIG. 6 – Méthode de Lagrange

```
#include <math.h>
```

en tête du programme. Dans cette bibliothèque la fonction *fabs (x)* (abréviation de : floating point absolute value) calcule $|x|$ ; attention : l'argument et le résultat sont de type `double`, et non `float`, comme le nom pourrait le faire croire.

## 2.4 Méthode de Newton

Le mieux est évidemment de remplacer la courbe d'équation $y = f(x)$ par sa tangente en un point $a$ voisin de la racine cherchée ; cette tangente coupe l'axe des abscisses au point $b$ donné par :

$$b = a - \frac{f(a)}{f'(a)}$$

On itère, en remplaçant $a$ par $b$, ce qui revient à considérer la suite définie par récurrence :

$$a_0 = a \ , \ a_{n+1} = a_n - \frac{f(a_n)}{f'(a_n)}$$

Cette suite ne converge que si le point de départ $a$ est suffisamment proche d'une racine, ou si $f''$ est de même signe que $f$ entre la racine et $a$. Voir la figure 7 pour la traduction en C ; il faut évidemment savoir calculer $f'$ en plus de $f$.

On peut améliorer cette fonction en vérifiant, comme pour *lagrange*, que le nombre d'itérations ne devient pas excessif (ce qui est un signe presque certain de divergence de la suite).

Cet algorithme a une excellente complexité, car on prouve que le nombre de décimales exactes double approximativement à chaque itération ! Comme les conditions de convergence sont, par contre, délicates, on peut utiliser une méthode sûre, comme la dichotomie, pour obtenir les premières décimales, puis la méthode de Newton pour obtenir très rapidement les décimales suivantes.

```
double newton (double a, double epsilon) {
  double b = a;

  do {
    a = b;
    b = a - f (a) / fprime (a);
  } while (fabs (b - a) > epsilon);
  return b;
}
```

FIG. 7 – Méthode de Newton

## 2.5 Exercices

1. On considère la suite récurrente définie par :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ \dfrac{3u_n + 1}{2} & \text{si } u_n \text{ est impair} \end{cases}$$

   Ecrire les termes de la suite, pour une valeur $u_0$ arbitraire, jusqu'à obtenir $u_n = 1$. Vérifier expérimentalement que ce programme ne continue jamais indéfiniment.

   Note : la condition *si x pair* peut s'écrire `if (x % 2 == 0)...`

2. Calculer le terme $u_n$ de la suite de Fibonacci, définie par :

$$\begin{aligned} u_0 &= 0 \\ u_1 &= 1 \\ u_{n+1} &= u_n + u_{n-1} \end{aligned}$$

3. Montrer que la fonction de la figure 8 calcule $x^n$, en prouvant que le produit $yz^n$ reste constant. Compter le nombre de multiplications faites pour chaque $n$ compris entre 8 et 15 ; en déduire la formule générale pour la complexité de cette fonction.

   Remarque : l'expression `n % 2` désigne le reste de la division de $n$ par 2.

4. Comment modifier la fonction *dichotomie*, pour qu'elle reste correcte lorsque les conditions initiales sont : $f(a) > 0$ et $f(b) < 0$ ?

5. Comment modifier la fonction *lagrange* pour calculer une fois et une seule chaque valeur de $f$ ?

6. Comparer expérimentalement les complexités des fonctions *dichotomie, lagrange* et *newton*, pour résoudre l'équation $\sin x = 1/2$ avec 6, puis 12 décimales.

7. Vérifier que la suite définie par :

$$x_0 = 1 \ , \ x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

   converge vers $\sqrt{a}$, pour $a > 0$ ; écrire une fonction *racine_carrée* ; comparer avec la méthode de Newton appliquée à la fonction $f(x) = x^2 - a$.

```
long puissance (int x, int n) {
  long y = 1, z = x;

  while (n > 0)
    if (n % 2 == 1) {
      y = y * z;
      n = n - 1;
    }
    else {
      z = z * z;
      n = n / 2;
    }
  return y;
}
```

FIG. 8 – Fonction puissance

# 3 Nombres complexes

## 3.1 Structures

Un nombre complexe est constitué de deux réels, la partie réelle et le coefficient de $i$ ; la déclaration suivante en est la traduction C :

```
struct complexe {
  double reel, imaginaire;
};
```

Les composantes d'une structure (`reel` et `imaginaire` dans notre cas) sont appelées *champs*, et on utilise la notation *variable.champ* pour les désigner. On déclare deux variables complexes $u, v$ comme suit :

```
struct complexe u, v;
```

On peut les manipuler globalement, et écrire par exemple $v = u$, ou champ par champ ; une déclaration peut comporter une initialisation, en utilisant des accolades ; par exemple :

```
struct complexe u = {1, 0};
```

## 3.2 Exemples de fonctions

```
struct complexe somme (struct complexe u, struct complexe v) {
  struct complexe z;

  z.reel = u.reel + v.reel;
  z.imaginaire = u.imaginaire + v.imaginaire;
  return z;
}

struct complexe produit (struct complexe u, struct complexe v) {
  struct complexe z;

  z.reel = u.reel * v.reel - u.imaginaire * v.imaginaire;
  z.imaginaire = u.reel * v.imaginaire + u.imaginaire * v.reel;
  return z;
}

double module (struct complexe z) {
  return sqrt (z.reel * z.reel + z.imaginaire * z.imaginaire);
}
```

FIG. 9 – Opérations sur les complexes

Les fonctions d'addition et de multiplication se formulent comme sur la figure 9, en calculant séparément chaque champ. Noter qu'on ne peut pas, hélas, utiliser les opérateurs habituels `+, *` pour ces opérations sur les complexes ; par exemple $uv + w$ se traduit par :

```
somme (produit (u, v), w)
```

La figure 9 comporte aussi le calcul de $|u|$, en utilisant la fonction *sqrt* (racine carrée = square root), qui fait partie de la bibliothèque mathématique standard ; pour utiliser cette dernière, le programme doit commencer par :

```
#include <math.h>
```

```
struct complexe puissance (struct complexe u, int n) {
  struct complexe v = {1, 0};
  int i;

  for (i = 1; i <= n; i++)
    v = produit (u, v);
  return v;
}
```

FIG. 10 – Calcul élémentaire de $u^n$

La figure 10 illustre l'emploi global des structures pour calculer $u^n$, avec $n$ multiplications ; voir l'exercice 1 pour une méthode plus efficace.

### 3.3 Exercices

1. Adapter l'exercice 3 de la section 1 au calcul efficace de $u^n$, lorsque $u$ est complexe.
2. Ajouter une fonction qui convertisse un couple $(r, \theta)$ en un complexe de module $r$ et d'argument $\theta$ (en degrés) ; utiliser pour cela les fonctions trigonométriques de la bibliothèque standard, d'en-tête *math.h*. Tester en affichant les puissances du nombre complexe de module 2 et d'argument 30 degrés.

   Note : on pourrait aussi définir une nouvelle structure, composée d'un module et d'un argument, et définir le produit de telles structures, mais ce n'est pas le sujet de l'exercice.
3. Ecrire une fonction qui calcule l'exponentielle d'un nombre complexe $z$, comme somme de la série :

$$\sum_{n=0}^{\infty} \frac{z^n}{n!}$$

   Tester en vérifiant que, pour $\varphi$ réel, $e^{i\varphi} = \cos\varphi + i \sin\varphi$.

# 4 Numération

## 4.1 Tableaux

Un *tableau* est une collection d'éléments identiques ; on déclare un tableau, en C, en faisant suivre son nom de sa *taille*, entre crochets. Exemples :

```
int t [100];
double vecteur [5];
```

Noter que la taille ne peut pas être variable : elle doit être connue du compilateur pour que celui-ci réserve la place convenable en mémoire ; il existe une technique standard pour tourner cette restriction, mais elle sort du cadre de ce cours. Il est conseillé, en général, de définir plutôt la taille par une macro :

```
#define MAX 100
int t [MAX];
```

Souvent on utilise ensuite un nombre variable $n$ d'éléments du tableau, ce qui est autorisé tant que $n$ ne dépasse pas MAX ; il y a seulement gaspillage de place en mémoire lorsque $n$ est stictement inférieur à MAX.

Les éléments d'un tableau de taille $n$ sont numérotés, en C, de 0 à $n-1$ (et non de 1 à $n$) ; l'élément numéro $i$ du tableau $t$ est désigné par `t[i]`, et $i$ est appelé *indice* ; la notation mathématique analogue est $t_i$. Aucune opération n'existe, en C, pour les tableaux : on manipule toujours les éléments un à un, ou avec des fonctions définies par l'utilisateur.

Le langage C considère que si $t$ est un tableau, $t$ désigne aussi *l'adresse* de son premier élément (dite aussi *adresse de base*), tandis que $*t$ désigne cet élément. Deux conséquences importantes :

- L'affectation n'a pas de sens pour les tableaux, contrairement aux structures ; en effet une instruction comme $u = t$ ne signifierait pas de copier les valeurs du tableau $t$ dans le tableau $u$, mais de changer l'adresse de base de $u$ ; or celle-ci est fixée par le compilateur.
- Lorsqu'un paramètre d'une fonction $f$ est un tableau $t$ d'entiers – resp. réels –, on écrit la définition de $f$ sous la forme $f$ $(int * t)$, – resp. $f$ $(double * t)$ – qui signifie qu'un *élément* de $t$ est un entier – resp. un réel.

La figure 11 présente deux fonctions élémentaires standard, l'une pour calculer la somme des $n$ premiers éléments d'un tableau (de réels dans l'exemple), l'autre pour calculer un vecteur somme de deux autres. La seconde s'utilise par exemple comme suit :

```
double s [100], u [100], v [100];
....
somme (u, v, s, 50);
```

Ecrire, comme ce serait naturel, `s = somme (u, v, 50)`, n'a pas de sens, pour les raisons expliquées précédemment.

## 4.2 Changements de base

Pour obtenir, de droite à gauche, l'écriture d'un entier $n$ en base $b$, il suffit d'écrire le reste de la division de $n$ par $b$, de diviser $n$ par $b$, et de recommencer tant que $n > 0$. Comme on obtient les chiffres de droite à gauche, il faut les stocker dans un tableau, avant de les écrire

```
double sigma (double *t, int n) {
  int i;
  double s = 0;
  for (i = 0; i < n; i++)
     s = s + t [i];
  return s;
}

void somme (double *a, double *b, double *c, int n) {
  int i;
  for (i = 0; i < n; i++)
     c [i] = a [i] + b [i];
}
```

FIG. 11 – Fonctions élémentaires sur des tableaux

```
void ecrire (long n, int base) {
  int i, t [MAX];

  for (i = 0; n > 0; i++) {
    t [i] = n % base;
    n = n / base;
  }
  for (i--; i >= 0; i--)
    printf (" %d", t [i]);
}
```

FIG. 12 – Ecriture d'un entier dans une base donnée

à l'envers. La fonction *ecrire* de la figure 12 fait ce travail. Noter que si a et b sont entiers, $a/b$ désigne, en C, le quotient entier, et $a\%b$ le reste de la division.

Si la base est supérieure à 10, cette fonction affiche des "chiffres" tels que 10, 11, 12 ... ; on peut sans difficulté corriger ce comportement, et utiliser les chiffres a, b, c ... , comme en numération hexadécimale :voir exercice 2.

```
long lire (int *t, int base) {
  int i;
  long x = 0;

  for (i = 0; 0 <= t [i] && t [i] < base; i++)
    x = base * x + t [i];
  return x;
}
```

FIG. 13 – Lecture d'un entier dans une base donnée

La fonction *printf* fonctionne selon le principe précédent ; inversement, la fonction *scanf* utilise l'algorithme de la figure 13. La lecture cesse dès que $t$ contient un élément qui n'est pas un chiffre dans la base choisie ; en particulier, on peut indiquer la fin logique de $t$ par -1, qui n'est jamais un chiffre valide.

### 4.3 Développement décimal d'un rationnel

Soit un rationnel $a/b$ positif ; l'algorithme usuel de division, qui fournit le développement décimal de la fraction, s'écrit en C :

```
for (i = 0; i <= n; i++)
  {q = a / b; r = a % b; a = 10 * r; }
```

et les $n$ premières décimales apparaissent successivement dans la variable $q$ (on suppose $a < 10\,b$ pour que la partie entière de la fraction se réduise à un chiffre).

En fait ce développement est périodique, puisqu'il n'y a qu'un nombre fini possible de restes : la période est au plus $b$. Pour tenter de trouver cette période, lorsqu'elle n'est pas trop grande, on stocke chaque reste $r$ dans un tableau nommé *reste*, et on examine si ce reste figure déjà dans ce tableau : si oui, on a trouvé la période, sinon on continue. Il ne faut pas oublier qu'un tableau est de taille limitée, et stopper l'algorithme lorsqu'on atteint cette limite ; dans ce dernier cas, la période est supérieure à la taille du tableau. On obtient la fonction *decimales* de la figure 14.

On a choisi une version qui admet le tableau des décimales en paramètre, de telle sorte qu'on puisse ensuite utiliser librement ce tableau, sans se restreindre à son affichage brutal ; pour appeler cette fonction, il faut donc auparavant avoir déclaré un tableau de taille MAX ; par exemple :

```
int d, chiffre [MAX];
long numerateur, denominateur;
...
d = decimales (chiffre, numerateur, denominateur);
```

```
int periode;

int decimales (int *t, long a, long b) {
  int i;
  long r, reste [MAX];

  for (i = periode = 0; i < MAX && periode == 0; i++) {
    t [i] = a / b;
    r = a % b;
    a = 10 * r;
    reste [i] = r;
    periode =  i - position (reste, r);
  }
  return i;
}
```

FIG. 14 – Développement décimal périodique d'une fraction

```
int position (long *t, long x) {
  int i;

  for (i = 0; i < MAX && t [i] != x; i++);
  return i < MAX ? i : -1;
}
```

FIG. 15 – Recherche de l'indice d'un élément dans un tableau

La fonction *decimales* retourne le nombre de décimales écrites dans le tableau (ici *chiffre*) ; elle est correcte si l'on dispose d'une fonction *position* qui recherche une valeur $x$ dans un tableau, et retourne son indice ; dans le cas présent, $x$ désigne le reste $r$, qu'on vient de placer justement en fin de tableau, donc on est certain que $x$ figure dans le tableau ; mais il vaut mieux, et ce n'est pas difficile, écrire une fonction *position* générale, qui retourne $-1$ lorsque $x$ ne figure pas dans le tableau, comme sur la figure 15.

L'expression *condition ? x :y* vaut $x$ si la condition est vraie, $y$ sinon ; cette construction permet d'abéger une construction *if (condition)... else*.

### 4.4 Fractions continues

Le problème inverse du développement décimal d'un rationnel consiste à trouver des rationnels qui approchent le mieux possible un réel donné ; par exemple :

$$\frac{22}{7} = 3.1428\ldots \ , \ \frac{355}{113} = 3.1415929\ldots$$

sont de bonnes approximations (surtout la seconde) de $\pi = 3.1415926\ldots$. Pour découvrir de bonnes approximations rationnelles de $y > 0$, on développe $y$ en fraction continue, c'est à dire sous la forme :

$$y = x_0 + \cfrac{1}{x_1 + \cfrac{1}{x_2 + \cfrac{1}{\ldots}}}$$

où $x_0$ est la partie entière de $y$, $x_1$ la partie entière de $1/(y - x_0)$, et ainsi de suite. Par exemple :

$$\pi = 3 + \frac{1}{7.0625\ldots} = 3 + \cfrac{1}{7 + \cfrac{1}{15.996\ldots}} = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1.003\ldots}}}$$

On appelle développement partiel (ou *réduite*) de rang $n$ la fraction :

$$\frac{a_n}{b_n} = x_0 + \cfrac{1}{x_1 + \cfrac{1}{\ldots + \cfrac{1}{x_n}}}$$

Par exemple, les réduites de rang 1, 2, 3 du développement en fraction continue de $\pi$ sont :

$$3 + \frac{1}{7} = \frac{22}{7} \ , \ 3 + \cfrac{1}{7 + \cfrac{1}{15}} = \frac{333}{106} \ , \ 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1}}} = \frac{355}{113}$$

On démontre que :

1. Tout rationnel possède un développement en fraction continue fini (la réciproque est évidente).
2. Tout développement partiel $a_n/b_n$ de $y$ est la meilleure approximation rationnelle possible de $y$, parmi les nombres de dénominateur inférieur ou égal à $b_n$.

Le numérateur $a_n$ et le dénominateur $b_n$ se calculent facilement en lisant le développement de droite à gauche, et de bas en haut ; ce calcul fournit directement la fraction sous forme *irréductible*. Plus précisément, soit :

$$\frac{p_k}{q_k} = x_k + \cfrac{1}{x_{k+1} + \cfrac{1}{\ldots + \cfrac{1}{x_n}}}$$

On a :

$$\frac{p_k}{q_k} = x_k + \frac{1}{p_{k+1}/q_{k+1}}$$

d'où :

$$\begin{aligned} p_k &= p_{k+1}x_k + q_{k+1} \\ q_k &= p_{k+1} \end{aligned}$$

Avec les conditions initiales $p_n = x_n$ , $q_n = 1$, ces formules permettent de calculer la réduite $a_n/b_n = p_0/q_0$.

```
int fraction_continue (long *t, double x) {
  int i;
  double delta = 1, epsilon = 1e-5;

  for (i = 0; i < MAX && delta > epsilon; i++) {
    t [i] = (long) x;
    delta = x - t [i];
    x = 1 / delta;
  }
  return i;
}
```

FIG. 16 – Développement en fraction continue

Pour la traduction en C du développement en fraction continue d'un réel $x$, voir la figure 16 ; pour le calcul d'une réduite d'ordre $n$, voir la figure 17. L'expression *(long) x* réalise une conversion de type : $x$ est de type *double*, et le résultat de type *long*, avec troncature des décimales. Pendant un développement en fraction continue, les erreurs d'arrondi s'accumulent, et le test mathématique $\delta = 0$ doit être remplacé par le test $|\delta| < \epsilon$.

## 4.5 Exercices

1. Ecrire la fonction qui calcule le *produit scalaire* de deux vecteurs.
2. Un caractère (type `char`) est représenté, en C, par l'entier qui est son code ASCII ; on peut aussi désigner un caractère en le plaçant entre apostrophes (pour le distinguer d'une variable du même nom). Montrer que si $c$ est un nombre compris entre 10 et 15, le caractère hexadécimal correspondant est donné par la formule :

$$c + \text{'a'} - 10 \ .$$

```
struct fraction {
  long numerateur, denominateur;
};

struct fraction reduite (long *t, int n) {
  int i; long p, q, tmp;
  struct fraction f;

  p = t [n]; q = 1;
  for (i = n - 1; i >= 0; i--) {
    tmp = p;
    p = t [i] * p + q;
    q = tmp;
  }
  f.numerateur = p; f.denominateur = q;
  return f;
}
```

FIG. 17 – Calcul d'une réduite

3. Développer en fraction continue le nombre $5.43333\ldots$ , et calculer les réduites successives :
   (a) à la main,
   (b) en simulant l'exécution des fonctions de la figure 16 et de la figure 17.
4. Calculer, en utilisant un tableau (et un seul), la ligne numéro $n$ du triangle de Pascal, qui se déduit de la ligne $n-1$ par la formule :

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

# 5 Le vrai et le faux

## 5.1 Algèbre de Boole

Le mathématicien Boole a découvert qu'on peut définir des opérations *logiques*, qui portent sur deux valeurs, dites *booléennes* : 0 (le faux) et 1 (le vrai). Ces opérations sont définies par des tables ; les principales sont :

- la négation (notations : NON, NOT, $\neg$) qui échange 0 et 1 ;
- la conjonction (notations : ET, AND, $\wedge$, &) ;
- la disjonction (notations : OU, OR, $\vee$, |).

Voici les tables qui définissent la conjonction et la disjonction :

| $\wedge$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| $\vee$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Ces deux dernières opérations sont associatives, commutatives et distributives l'une par rapport à l'autre ; elles définissent donc une algèbre, dans laquelle les preuves sont très faciles : il suffit d'énumérer tous les cas possibles. Par exemple, la première des lois dites *de Morgan* :

$$\begin{aligned} \neg(a \vee b) &= (\neg a) \wedge (\neg b) \\ \neg(a \wedge b) &= (\neg a) \vee (\neg b) \end{aligned}$$

se démontre en constatant que les colonnes 4 et 7 de la table suivante sont identiques :

| $a$ | $b$ | $a \vee b$ | $\neg(a \vee b)$ | $\neg a$ | $\neg b$ | $(\neg a) \wedge (\neg b)$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

Les lois de Morgan montrent que par exemple la disjonction peut être définie à partir de la conjonction et de la négation :

$$a \vee b = \neg(\neg a \ \wedge \ \neg b) \ .$$

On peut définir 16 opérations distinctes sur 2 variables ; les plus utiles, outre la conjonction et la disjonction, sont la disjonction exclusive XOR et l'implication $\Rightarrow$, dont les tables sont les suivantes :

| XOR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| $\Rightarrow$ | 0 | 1 |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 1 |

La disjonction exclusive est aussi l'addition modulo 2, ou l'opérateur $\neq$ ; la table de l'implication n'est pas symétrique : il faut donc préciser que c'est la table de $a \Rightarrow b$, avec les valeurs de $a$ en lignes, et celles de $b$ en colonnes. On peut montrer facilement que $a \Rightarrow b$ équivaut à $\neg b \Rightarrow \neg a$ (contraposée, ou raisonnement par l'absurde).

## 5.2 Opérations logiques sur les entiers

Comme le processeur travaille sur des registres de 16, 32 ou 64 chiffres binaires (suivant le type du processeur), qui servent aussi à stocker des entiers, les opérations logiques sont effectuées chiffre par chiffre sur ces entiers ; par exemple, en se limitant à 8 chiffres :

$10001100 \wedge 10101010 = 10001000$

$10001100 \vee 10101010 = 10101110$

Si $m$ et $n$ sont des entiers, $m\&n$ désigne, en C, la conjonction de $m$ et $n$, et $m|n$ leur disjonction. **Attention :** il y a deux opérateurs de négation ; $\sim n$ désigne l'entier déduit de $n$ par échange des 0 et des 1, tandis que $!n$ vaut 1 si $n = 0$, 0 sinon ; c'est ce second opérateur qui est employé le plus souvent, car $\sim 1 \neq 0 : \sim \ldots0001 = \ldots1110$, qui représente -2 en numération binaire !

Test de parité : $n\&1$ vaut 1 si $n$ est impair, 0 sinon ; le test `if (n & 1)` signifie donc : *si $n$ est impair*, et est exécuté plus rapidement que le test `if (n % 2 == 1)`, qui implique une division (sauf si le compilateur C est intelligent, et reconnaît qu'une division par deux est un simple décalage).

## 5.3 Comparaisons

Chaque fois qu'une condition intervient dans une instruction *if, for, while*, cette condition est en fait un entier, avec la convention que 0 est faux, et que tout autre entier est vrai ; l'opérateur `!` inverse le vrai et le faux : il transforme 0 en 1, et tout entier non nul en 0. On peut donc écrire indifféremment :

- `if (a)` ou `if (a != 0)` ;
- `if (!a)` ou `if (a == 0)`.

Les principaux opérateurs de comparaison sont les suivants : `< <= > >= == !=` , et produisent les entiers 1 ou 0 suivant que la relation est vraie ou fausse. **Attention :** les deux constructions suivantes sont valides en C, mais ont des sens radicalement différents :

- `if (a==b)` signifie : si $a$ est égal à $b$ alors...
- `if (a=b)` signifie : $a$ prend la valeur de $b$, et si cette valeur est différente de zéro, alors...

La seconde construction est vivement déconseillée, car elle est trompeuse ; souvent le programmeur emploie cette construction par étourderie, au lieu de la première ; s'il a vraiment la seconde en tête, il est beaucoup plus clair qu'il écrive plutôt :

```
a = b;
if (a != 0) ...
```

## 5.4 Courts-circuits

Il existe une seconde forme de conjonction ou de disjonction, représentées par les symboles `&&` et `||`, qui se lisent "et alors", "ou sinon" ; cette forme garantit que le second opérande n'est évalué que si cela est nécessaire. L'usage typique est le suivant :

```
while (i < MAX && t [i] > 0)
```

On parcourt un tableau de taille MAX : si l'indice atteint cette valeur, l'élément $t[i]$ n'existe pas, et il ne faut pas l'évaluer, ce que garantit l'opérateur `&&`.

De même, lorsqu'on calcule la conjonction des éléments d'un tableau, il est inutile en général de parcourir tout le tableau (contrairement au calcul, par exemple, de la somme de ces éléments) : l'évaluation peut être court-circuitée dès la rencontre d'une valeur fausse :

```
int conjonction (int *t, int n){
  int i;
  for (i = 0; i < n; i++)
    if (! t[i]) return 0;
  return 1;
}
```

La fonction duale s'écrit :

```
int disjonction (int *t, int n){
  int i;
  for (i = 0; i < n; i++)
    if (t[i]) return 1;
  return 0;
}
```

## 5.5 Parties d'un ensemble

Soit $X$ un ensemble de $n$ éléments $x_1, \ldots, x_n$ ; à chaque partie $E \subset X$ on peut associer un tableau de $n$ booléens $b_1, \ldots, b_n$, avec la convention : $b_i = 1$ si et seulement si $x_i \in E$. La conjonction et la disjonction correspondent alors respectivement à l'intersection et à l'union des ensembles correspondants.

```
int successeur (int *t, int n) {
  int i;

  for (i = n - 1; i >= 0 && t [i]; i--)
    t [i] = 0;
  if (i < 0)
    return 0;
  t [i] = 1;
  return 1;
}

void enumeration (int n) {
  int i, t [MAX];

  for (i = 0; i < n; i++)
    t [i] = 0;
  do { /* traitement */ }
  while (successeur (t, n) );
}
```

FIG. 18 – Enumération des parties d'un ensemble

Enumérer les $2^n$ valeurs possibles d'un tableau de $n$ booléens revient à énumérer les parties de $X$ ; ce problème, qu'on rencontre fréquemment, peut être résolu en écrivant en binaire tous les entiers de 0 à $2^n - 1$ ; une solution plus élégante est décrite sur la figure 18 ; la fonction *successeur* transforme le tableau $t$ (de $n$ booléens) en son successeur dans l'ordre

lexicographique, et retourne 1, sauf si $t$ ne comporte que des 1, et est donc le dernier de la liste ; en d'autres termes, la fonction *successeur* retourne 1 ou 0 selon que $t$ a ou n'a pas de successeur.

Pour $n = 4$, la fonction *enumeration* traite successivement les 16 valeurs suivantes :

0000 , 0001 , 0010 , 0011 , 0100 , 0101 , 0110 , 0111 ,
1000 , 1001 , 1010 , 1011 , 1100 , 1101 , 1110 , 1111 .

## 5.6 Exercices

1. (a) Donner la représentation binaire des entiers 35 et 36.
   (b) Que valent les entiers 35&1, 36&1, 36&3, 36&7 ?
   (c) Quelle est la propriété arithmétique simple de $n$ exprimée par : $n\&7 = 0$ ?
   (d) Généraliser (simplement) le résultat précédent.
2. Ecrire la fonction *implique*, qui calcule $a \Rightarrow b$.
3. Ecrire une fonction *table_de_verite*, qui affiche la table de vérité d'une fonction $f$ de deux variables.
   TP : tester avec les fonctions &, | , et *implique*.
4. Ecrire une fonction *tautologie*, qui retourne la valeur 1 si $f$ est toujours vraie, 0 sinon ; on suppose que $f$ est une fonction de deux variables.
   TP : tester avec la formule $(a \Rightarrow b) \Longleftrightarrow (\neg b \Rightarrow \neg a)$.
5. Reprendre les deux exercices précédents en supposant que $f$ est une fonction de trois variables.
   TP : tester avec : $((a \Rightarrow b)\ \&\ (b \Rightarrow c)) \Rightarrow (a \Rightarrow c)$
6. Utiliser les fonctions de la figure 18 pour écrire des fonctions *table_de_verite* et *tautologie* indépendantes du nombre de variables booléennes qui interviennent dans la formule $f$.
7. Ecrire une fonction *liste*, qui énumère les parties à $k$ éléments d'un ensemble à $n$ éléments ; par exemple l'exécution de *liste (5, 3)* fournit le résultat suivant :
   1 2 3, 1 2 4, 1 2 5, 1 3 4, 1 3 5, 1 4 5, 2 3 4, 2 3 5, 2 4 5, 3 4 5.

# 6 Valeurs, adresses et paramètres

## 6.1 Valeurs

Si $x$ est un entier, un réel, ou une structure, $x$ désigne la *valeur* de l'objet, et l'affectation $x = y$ copie la valeur de $y$ dans $x$. Exemple :

```
struct complexe{
  double reel, imaginaire;
};

struct complexe u, v;
```

L'affectation $u = v$ copie le complexe $v$ dans $u$ ; comme un réel en double précision occupe deux mots de 32 bits, un complexe occupe 4 mots, et donc l'affectation provoque la recopie de 4 mots.

## 6.2 Adresses et tableaux

Si $t$ est un tableau, $t$ désigne *l'adresse* du premier élément, c'est à dire le numéro de sa case mémoire. L'affectation $t = u$ est interdite, car absurde : un tableau ne peut pas changer de position en mémoire.

Pour copier les éléments de $u$ dans $t$, il faut employer une boucle :

```
for (i = 0 ; i < n ; i++) t [i] = u [i] ;
```

La première raison de cette convention a priori surprenante, est qu'un tableau peut être de très grande taille : le programmeur doit indiquer explicitement sa volonté de copier tous les éléments d'un tableau, sans que cette opération coûteuse soit dissimulée par la notation anodine $t = u$. La seconde raison (la plus importante) est expliquée dans la section suivante.

## 6.3 Paramètres

```
void decimales (int *t, long a, long b) {
  int i; long r;

  for (i = periode = 0; i < MAX; i++) {
    t [i] = a / b;
    r = a % b;
    a = 10 * r;
  }
}
```

FIG. 19 – Développement décimal

Considérons à nouveau l'exemple du développement décimal d'un rationnel, vu section 4.3, et simplifié sur la figure 19. La première ligne est la *définition* de la fonction, et les paramètres $t, a, b$ sont dits *formels*.

Supposons qu'on utilise (le terme technique est *appelle*) cette fonction de la façon suivante :

```
int chiffre [MAX]; long p, q;
...
p = 22; q = 7; decimales (chiffre, p, q);
```

Les arguments *chiffre, p, q* sont aussi nommés paramètres *effectifs*. Si l'on y réfléchit, l'exécution de cette fonction est paradoxale :

1. on ne souhaite pas que $p$ soit différent de 22 après développement décimal de $p/q$, et effectivement $p$ reste inchangé, alors que le paramètre formel $a$ est modifié ;
2. par contre, on souhaite que le tableau *chiffre* contienne, après exécution de la fonction, les décimales de $p/q$, et c'est bien ce qui se passe ; les valeurs contenues dans le tableau sont donc modifiées.

La règle qui relie paramètres effectifs et paramètres formels est la suivante : l' exécution d'une fonction commence par la copie des valeurs des paramètres effectifs dans les paramètres formels, situés dans une partie réservée de la mémoire ; on appelle cette règle *passage des paramètres par valeur*, et c'est le seul mécanisme utilisé par le langage C. Il assure que les paramètres effectifs ne peuvent jamais être modifiés pendant l'exécution de la fonction.

En même temps, la section 6.2 entraîne que, lorsqu'un paramètre est un tableau, il s'agit d'une adresse : c'est donc l'adresse de *chiffre* qui est copiée dans $t$, ce qui présente un double avantage :

1. il n'y a pas de recopie subreptice des éléments du tableau, qui peuvent être très nombreux ;
2. l'instruction $t[i] = a/b$ est exécutée avec pour valeur de $t$ l'adresse de *chiffre*, donc elle modifie l'élément d'indice $i$ dans *chiffre*.

## 6.4 Références

```
int lire (int *t, int base, int *destination) {
  int i;
  long x = 0;

  for (i = 0; 0 <= t [i] && t [i] < base; i++)
    x = base * x + t [i];
  *destination = x;
  return i;
}
```

FIG. 20 – Développement décimal

Certains langages de programmation utilisent un second mécanisme de passage des paramètres, dit *par référence*, dans lequel c'est systématiquement l'adresse du paramètre effectif qui est copiée dans le paramètre formel. En C, on a vu que ce mécanisme s'applique automatiquement aux tableaux, grâce à la convention selon laquelle le nom d'un tableau désigne son adresse.

On dispose d'autre part, en C, d'une convention générale pour transmettre l'adresse d'un objet : $\&x$ désigne l'adresse de $x$, et inversement $*t$ désigne la valeur située à l'adresse $t$. Pour qu'une fonction modifie la valeur de $x$, il faut lui fournir en paramètre l'adresse de $x$ ; c'est le

cas de *scanf*. La figure 20 présente une modification de la fonction *lire*, vue section 4.2, pour qu'elle soit utilisée à la manière de *scanf*. Par exemple si le tableau *chiffre* contient 3 1 4 −1, l'appel :

```
lire (chiffre, 10, &n) ;
```

place la valeur 314 dans $n$ (et retourne accessoirement le nombre de chiffres lus). Noter que pour spécifier que le paramètre formel *destination* est l'adresse d'un entier, on écrit que sa valeur $*destination$ est un entier ! Cette notation déroute au début, mais on s'y habitue vite.

# 7 Polynômes

## 7.1 Représentation

La représentation la plus simple du polynôme $P = \sum a_i X^i$ est un tableau $P$, avec, pour chaque indice $i$, $P[i] = a_i$ ; l'inconvénient de cette technique est le stockage d'une valeur nulle pour chaque monôme $X^i$ ne figurant pas dans $P$ ; en particulier, comme les tableaux sont de taille fixe (sauf en utilisant des techniques avancées qui sortent du cadre de ce cours), le degré réel du polynôme est presque toujours inférieur à cette taille, et il faut compléter le tableau par des zéros.

```
typedef double polynome [DEGRE_MAX + 1];

int degre (polynome p) {
  int i;
  for (i = DEGRE_MAX; i >= 0 && p [i] == 0; i--);
  return i;
}
```

FIG. 21 – Degré d'un polynôme

La figure 21 présente le calcul du degré d'un polynôme, en fonction des remarques précédentes. La déclaration *typedef* a pour effet qu'une déclaration ultérieure de variables :

`polynome u, v ;`

est équivalente à la déclaration :

`double u [DEGRE_MAX + 1], v [DEGRE_MAX + 1] ;`

## 7.2 Fonctions élémentaires

La fonction de la figure 22 remplace l'affectation, qui n'a pas de sens pour les tableaux (voir section 6.2) ; l'ordre des arguments est choisi de telle sorte que :

`copier (u, v)` remplace : `u = v`.

La fonction de la figure 23 effectue la somme de deux polynômes ; l'ordre des arguments est choisi de telle sorte que :

`somme (p, u, v)` remplace : `p = u + v`.

La figure 24 décrit le produit de deux polynômes ; on applique la formule :

$$\left(\sum b_i X^i\right)\left(\sum c_j X^j\right) = \sum b_i c_j X^{i+j} \ .$$

La difficulté vient de ce que le degré du produit peut dépasser le maximum autorisé pour le stockage du résultat : dans ce cas on tronque le produit (que faire d'autre ?). Le test `if (b[i] != 0)` accélère considérablement le calcul du produit lorsque le polynôme $b$ est *creux*, c'est à dire contient beaucoup de zéros.

La fonction de la figure 25 montre comment utiliser les fonctions précédentes pour calculer :

$$\prod_{i=1}^{n}(1 - X^i)$$

```
void copier (polynome destination, polynome source) {
  int i;
  for (i = 0; i <= DEGRE_MAX; i++)
    destination [i] = source [i];
}
```

FIG. 22 – Copie de polynômes

```
  void somme (polynome a, polynome b, polynome c) {
    int i;
    for (i = 0; i <= DEGRE_MAX; i++)
      a [i] = b [i] + c [i];
  }
```

FIG. 23 – Somme de polynômes

```
void produit (polynome a, polynome b, polynome c) {
  int i, j, jmax, m = degre (b), n = degre (c);

  for (i = 0; i <= DEGRE_MAX; i++)
    a [i] = 0;
  for (i = 0; i <= m; i++)
    if (b[i] != 0) {
      jmax = DEGRE_MAX - i;
      if (jmax > n)
        jmax = n;
      for (j = 0; j <= jmax; j++)
        a [i + j] = a [i + j] + b [i] * c [j];
    }
}
```

FIG. 24 – Produit de polynômes

```
void euler (polynome p, int n) {
  int i;
  polynome a, b;

  a [0] = b [0] = 1;
  for (i = 1; i <= DEGRE_MAX; i++)
    a [i] = b [i] = 0;

  for (i = 1; i <= n; i++) {
    a [i] = -1;
    produit (p, a, b);
    copier (b, p);
    a [i] = 0;
  }
}
```

FIG. 25 – Calcul de $\prod(1 - X^i)$

dont Euler a montré que les $n$ premiers termes sont :

$$1 - X - X^2 + X^5 + X^7 - X^{12} - X^{15} + X^{22} + X^{26} - X^{35} - X^{40} \ldots$$

le terme général étant :

$$(-1)^n \left( X^{n(3n-1)/2} + X^{n(3n+1)/2} \right) .$$

Noter, dans le programme, que l'instruction :

```
produit (b, a, b) ;
```

serait une faute, car le polynôme $b$ ne doit pas être modifié avant la fin du calcul du produit de $a$ par $b$. La situation est différente pour les tableaux et pour les variables ordinaires : l'instruction `b = a * b` est correcte pour des entiers ou des réels, car l'opération se fait en une seule fois.

### 7.3 Evaluation d'un polynôme

Connaissant les coefficients du polynôme $P$ de degré $n$, le calcul de $y = P(x)$, pour un réel donné $x$, demande seulement $n$ multiplications – et non $n(n+1)/2$ –, en utilisant la *méthode de Hörner* :

```
y = p [n];
for (i = n - 1; i >= 0; i--)
  y = x * y + p [i];
```

Noter que c'est l'algorithme déjà vu, section 4.2, pour *lire* un entier connaissant son développement en base $b$.

## 7.4 Interpolation de Lagrange

Etant donnés $n+1$ points $(a_i, b_i),\ 0 \leq i \leq n$, il existe un polynôme $P$ de degré au plus $n$ et un seul, tel que pour tout $i$ : $P(a_i) = b_i$ ; $P$ est donné par la formule :

$$P = \sum_{i=0}^{n} b_i L_i$$

où les $L_i$ forment une *base de Lagrange*, c'est à dire vérifient :

$$L_i(a_j) = \begin{cases} 1 \text{ si } i = j \\ 0 \text{ sinon} \end{cases}$$

et sont donnés par les formules :

$$L_i(X) = \prod_{i \neq j} \frac{X - a_j}{a_i - a_j}$$

Ces formules se programment facilement, à condition d'introduire deux fonctions intermédiaires, pour calculer :

- le produit d'un polynôme par un monôme $X - a$
- un polynôme de base $L_i$

Voir le détail sur la figure 26. Note : il existe des algorithmes plus efficaces que celui-ci, pour calculer les coefficients du polynôme$P$.

```
void produit_monome (polynome p, double a, int n) {
  int i;

  p [n] = p [n - 1];
  for (i = n - 1; i > 0; i--)
    p [i] = p [i - 1] - a * p [i];
  p [0] = - a * p [0];
}

void base (polynome p, double *a, int n, int i) {
  int j, d = 0;
  double lambda = 1;

  p [0] = 1;
  for (j = 1; j <= n; j++)
    p [j] = 0;

  for (j = 0; j <= n; j++)
    if (j != i) {
      d++;
      produit_monome (p, a [j], d);
      lambda = lambda * (a [i] - a[j]);
    }
  lambda = 1 / lambda;
  for (j = 0; j <= n; j++)
    p [j] = lambda * p [j];
}

void lagrange (polynome p, double *a, double *b, int n) {
  int i, j;
  polynome l;

  for (i = 0; i <= DEGRE_MAX; i++)
    p [i] = 0;

  for (i = 0; i <= n; i++) {
    base (l, a, n, i);
    for (j = 0; j <= n; j++)
      p [j] = p [j] + b [i] * l [j];
  }
}
```

FIG. 26 – Interpolation de Lagrange

## 7.5 Exercices

1. En utilisant la fonction qui donne le produit de deux polynômes, écrire une fonction qui calcule $(1+X)^n$, et vérifier en TP qu'on retrouve bien les coefficients du binôme.
2. Ecrire une fonction qui calcule la dérivée d'un polynôme.
3. Les polynômes de Tchebycheff $T_n$ sont définis par :

   $$\cos nx = T_n(\cos x) \ ,$$

   et vérifient la récurrence :

   $$T_{n+1} = 2XT_n - T_{n-1} \ .$$

   Ecrire une fonction qui calcule $T_n$. Vérifier les résultats avec la table suivante :

   $$\begin{aligned} T_0(X) &= 1 \\ T_1(X) &= X \\ T_2(X) &= 2X^2 - 1 \\ T_3(X) &= 4X^3 - 3X \\ T_4(X) &= 8X^4 - 8X^2 + 1 \\ T_5(X) &= 16X^5 - 20X^3 + 5X \end{aligned}$$

4. On définit la division euclidienne du polynôme $A$ par le polynôme $B$ avec la formule :

   $$A = BQ + R$$

   et la condition : degré $(R)$ < degré $(B)$. Ecrire la fonction qui calcule le quotient $Q$ et le reste $R$.
5. La division selon les puissances croissantes à l'ordre $n$ est définie par :

   $$A = BQ + R$$

   où $R$ ne contient pas de terme de degré inférieur à $n$. Ecrire la fonction qui calcule le quotient $Q$ et le reste $R$.

# 8 Arithmétique

## 8.1 Algorithme d'Euclide

Le calcul du PGCD de deux entiers positifs $a$ et $b$ se fait par l'algorithme d'Euclide, remarquablement général – il fonctionne aussi pour les polynômes –, et efficace : si $b$ n'est pas nul, on effectue la division euclidienne de $a$ par $b$ :

$$a = bq + r \ , \ 0 \leq r < b \ ,$$

on remplace le couple $(a, b)$ par $(b, r)$, et on recommence ; lorsque $b$ est nul (c'est à dire en général lorsque le reste de la division précédente est nul), le PGCD $a \wedge b$ vaut $a$.

```
long pgcd (long a, long b) {
  long r;
  while (b > 0) {
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

FIG. 27 – Calcul du pgcd

La figure 27 donne la traduction en C de cet algorithme. Complexité : on prouve facilement que le cas le pire (c'est à dire celui où le nombre d'itérations à exécuter est le plus grand), est celui où $a$ et $b$ sont des termes consécutifs de la suite de Fibonacci, définie par :

$$F_1 = 1 \ ; \ F_2 = 1 \ ; \ F_n = F_{n-1} + F_{n-2} \ (n > 2)$$

Comme $F_n$ est une fonction exponentielle de $n$, la complexité de l'algorithme d'Euclide est proportionnelle au logarithme de $a$ (en supposant $a > b$), autrement dit proportionnelle au nombre de chiffres nécessaires pour écrire $a$ (par exemple en base 10) ; l'algorithme d'Euclide est donc très efficace.

## 8.2 Fonctions récursives

Une fonction peut, pendant son exécution, faire appel à une autre fonction, qui elle-même en appelle d'autres, etc...Le mécanisme qui garantit que ces appels en cascade fonctionnent correctement, autorise du même coup une fonction à s'appeler elle-même ; un tel appel, et la fonction ainsi définie, sont dits *récursifs*. Pour éviter les cercles vicieux, une fonction récursive doit toujours comporter un cas particulier où le résultat est calculé directement, c'est à dire sans appel récursif ; il faut aussi s'assurer que ce cas particulier finira toujours par se présenter.

La figure 28 donne la version récursive de l'algorithme d'Euclide, peut-être un peu plus facile à écrire que la version itérative. Les deux versions ont fondamentalement la même complexité, avec un avantage à la version itérative, car l'appel d'une fonction n'est pas gratuit.

```
long pgcd (long a, long b) {
  long r;

  r = a % b;
  if (r == 0)
    return b;
  else
    return pgcd (b, r);
}
```

FIG. 28 – Calcul du pgcd : version récursive

## 8.3 Théorème de Bezout

Le théorème de Bezout affirme que le PGCD de deux entiers $a, b$ est une combinaison linéaire (à coefficients entiers) de $a$ et $b$ :

$$a \wedge b = au + bv.$$

L'algorithme d'Euclide permet aussi de calculer ces coefficients $u$ et $v$ : soient $a_0$ et $b_0$ les valeurs initiales de $a$ et $b$ ; il suffit de remarquer que si l'on a, pendant l'exécution de l'algorithme :

$$a = a_0u + b_0v, \quad b = a_0s + b_0t, \quad a = bq + r$$

alors :

$$r = a_0(u - qs) + b_0(v - qt)$$

Donc chaque fois qu'on remplace, au cours de l'algorithme d'Euclide, le couple $(a, b)$ par $(b, r)$, il suffit de remplacer aussi le couple $(u, v)$ par $(s, t)$, et $(s, t)$ par $(u - qs, v - qt)$.

```
long pgcd (long a, long b, long *u, long *v) {
  long q, r, s, t, tmp;

  *u = 1; *v = 0; s = 0; t = 1;
  while (b > 0) {
    q = a / b; r = a % b; a = b; b = r;
    tmp = s; s = *u - q * s; *u = tmp;
    tmp = t; t = *v - q * t; *v = tmp;
  }
  return a;
}
```

FIG. 29 – Calcul du pgcd et des coefficients de Bezout

La figure 29 donne la traduction en C ; comme la fonction *pgcd* doit modifier les valeurs des paramètres $u, v$, il faut transmettre les *adresses* de ces paramètres, d'où les notations $*u$ et $*v$.

```
long pgcd (long a, long b, long *u, long *v) {
  long q, r, d, s, t;

  q = a / b;
  r = a % b;
  if (r == 0) {
    *u = 0;
    *v = 1;
    return b;
  }
  else {
    d = pgcd (b, r, &s, &t);
    *u = t;
    *v = s - q * t;
    return d;
  }
}
```

FIG. 30 – Calcul des coefficients de Bezout : version récursive

On peut aussi écrire une version récursive, en remarquant que si l'on a :

$$a = bq + r$$
$$d = a \wedge b = b \wedge r = bs + rt$$

alors :

$$d = bs + (a - bq)t = at + b(s - qt)$$

d'où la fonction de la figure 30.

## 8.4 Nombres premiers

```
int premier (long n) {
  long d;

  if (n % 2 == 0)
    return 0;
  for (d = 3; d * d <= n; d = d + 2)
    if (n % d == 0)
      return 0;
  return 1;
}
```

FIG. 31 – Test de primalité

La figure 31 donne la traduction en C de l'algorithme brutal qui teste si un nombre $n$ est premier : on examine si $n$ est pair, et si ce n'est pas le cas on cherche un diviseur impair $d$. Il

est important de cesser la recherche dès que $d^2 > n$, car si $n$ est composite : $n = d_1 d_2,\ d_1 \leq d_2$, et donc l'un des facteurs vérifie $d^2 \leq n$.

Pour des entiers qui ne dépassent pas 32 chiffres binaires, cet algorithme exige au plus $2^{15}$ divisions, qui peuvent être effectuées en quelques millisecondes.

Mais en général, si $n$ s'écrit avec $k$ chiffres en base $b$, la complexité de cet algorithme est proportionnelle à $\sqrt{b^k} = b^{k/2}$, et est donc exponentielle. Si l'on dispose de fonctions pour effectuer des opérations exactes sur des entiers de longueur quelconque (représentés par des tableaux), comme c'est le cas avec le langage Maple, il est hors de question d'appliquer cet algorithme à des nombres ayant beaucoup plus de 20 chiffres décimaux : $10^{10}$ divisions exigent au mieux 10 000 secondes, à raison d'une par microseconde, soit 3 heures, et si $n$ a 50 chiffres, le temps d'exécution dépasse largement l'âge de l'univers !

On peut accélérer l'algorithme en ne testant pas les diviseurs multiples de 3 ; il reste les nombres $d$ égaux à 1 ou 5 modulo 6. Mais le développement de cette idée ne mène à aucun progrès décisif, car il faut tester au moins les diviseurs $d$ premiers, et la densité de ceux-ci ne décroît que très lentement (voir exercice 2 ).

Il existe par contre des tests rapides de primalité, qui reposent sur des principes mathématiques sophistiqués, et permettent de déterminer en quelques secondes si un nombre de 200 chiffres est premier.

## 8.5 Décomposition en facteurs premiers

```
int decompose (long *t, long n) {
  long d;
  int i = 0;

  while (n % 2 == 0) {
    t [i++] = 2;
    n = n / 2;
  }
  for (d = 3; d * d <= n;)
    if (n % d == 0) {
      t [i++] = d;
      n = n / d;
    }
    else
      d = d + 2;
  if (n > 1)
    t [i++] = n;
  return i;
}
```

FIG. 32 – Décomposition en facteurs premiers

L'algorithme brutal de décomposition en facteurs premiers de $n$, consiste à chercher le plus petit diviseur $d$ de $n$, en arrêtant la recherche, comme précédemment, dès que $d^2 > n$ ; chaque diviseur ainsi trouvé est automatiquement premier (puisqu'il n'y en a pas de plus

petit) : on divise alors $n$ par $d$, et on continue *avec la même valeur de d*, car un facteur peut apparaître avec une multiplicité supérieure à 1.

La figure 32 donne la traduction en C, avec stockage des facteurs dans un tableau ; ne pas oublier le dernier facteur premier, qui est $n$, lorsque $d^2 > n$. Noter l'instruction `t [i++] = d;` qui est une abréviation de : `t [i] = d; i++;` pour abréger : `i++; t [i] = d;` on écrirait : `t [++i] = d;` .

L'algorithme est rapide si $n$ ne possède que de petits facteurs premiers ; on prouve hélas que ce n'est pas le cas statistiquement, et si $n = pq$, avec $p$ et $q$ premiers, $p < q$, l'algorithme a une complexité exponentielle en fonction du nombre de chiffres nécessaires pour écrire $p$. Contrairement au test de primalité, on ne connaît pas d'algorithme non exponentiel de décomposition en facteurs premiers ; la factorisation de :

$$F_9 = 2^{2^9} + 1$$

a occupé des centaines de machines pendant quelques semaines, en 1990, avec l'algorithme le plus sophistiqué connu, et pourtant $F_9$ n'a "que" 512 chiffres binaires.

L'absence d'algorithme rapide de factorisation conduit à utiliser les nombres premiers en cryptographie : voir section 8.7.

## 8.6 Test de Fermat

Le petit théorème de Fermat énonce que, si $p$ est premier, et si $a$ n'est pas un multiple de $p$ :

$$a^{p-1} \equiv 1 \ [p] \ .$$

Pour tester si $n$ est premier, on peut donc choisir $a < n$ (par exemple $a = 2$), et calculer $a^{n-1} \ [n]$ ; si le résultat est différent de 1, $n$ est certainement composite ; par contre, on ne peut rien conclure de façon sûre si le résultat vaut 1, car la réciproque du petit théorème de Fermat est fausse : il y a simplement une assez forte probabilité pour que $n$ soit premier. En recommençant le test pour d'autres valeurs de $a$, et en affinant les outils mathématiques d'analyse de la situation, on peut obtenir un test correct avec une excellente probabilité.

L'intérêt de cette approche réside dans l'existence d'un algorithme qui calcule $a^{n-1} \ [n]$ très rapidement : sa complexité est une fonction linéaire du nombre de chiffres de $n$.

```
long puissance (long a, long e, long n) {
  if (e == 0)
    return 1;
  if (e % 2 == 0)
    return puissance ( (a * a) % n, e / 2, n);
  else
    return (a * puissance (a, e - 1, n) ) % n;
}
```

FIG. 33 – Calcul rapide de $a^e \ [n]$

La figure 33 décrit une version récursive de cet algorithme, qui repose sur les identités évidentes :

$$\begin{array}{ll} a^e = (a \times a)^{e/2} & \text{si } e \text{ est pair,} \\ a^e = a \times a^{e-1} & \text{si } e \text{ est impair.} \end{array}$$

En remarquant que, si $e$ est impair, $e-1$ est pair, on voit que le nombre d'opérations à faire ne dépasse pas deux fois le nombre de chiffres de l'exposant $e$ en numération binaire.

Note : cette fonction donne des résultats incorrects si l'un des produits déborde avant la réduction modulo $n$ ; il faudrait employer des fonctions de calcul en précision arbitraire.

## 8.7 Cryptographie

Un message est assimilé à une suite de nombres entiers : par exemple chaque bloc de 8 caractères (octets) est assimilé à un entier représenté par 64 chiffres binaires.

- Le message est brouillé, pour en assurer la confidentialité, en remplaçant chaque nombre $a$ par $a^e$ $[n]$ (c'est possible rapidement comme on l'a vu dans la section précédente).
- Le message original (dit message *clair*) est reconstitué en remplaçant chaque nombre $a$ du message brouillé par $a^d$ $[n]$.

Bien entendu la fonction de décodage doit être l'inverse de celle de brouillage, soit :

$$\text{pour tout } a : \ a^{de} \equiv a \ [n].$$

Si $n$ est le produit $pq$ de deux nombres premiers distincts, le petit théorème de Fermat implique que cette condition est équivalente à :

$$de \equiv 1 \ [(p-1)(q-1)]$$

Soit $m = (p-1)(q-1)$ ; connaissant $e$, on peut calculer $d$ si et seulement si $e$ est premier avec $m$ ; si c'est le cas, en calculant le pgcd de $e$ et $m$ par la fonction de la figure 29, on obtient des coefficients $u$ et $v$ tels que : $1 = eu + mv$ (formule de Bezout), et il suffit donc de prendre $d = u$. Comme l'algorithme d'Euclide est linéaire, le calcul du coefficient $d$ de décodage est très rapide, à condition de connaître $m$, c'est à dire $p$ et $q$.

Cette méthode constitue l'algorithme cryptographique RSA, du nom de ses inventeurs, *Rivest, Shamir* et *Adelman*. Comme il est impossible de factoriser rapidement un grand nombre, on peut rendre publics $e$ et $n$, et donc la méthode de codage, sans qu'il soit possible de calculer $d$, et donc de décoder, sauf pour le destinataire du message qui est seul à connaître les facteurs premiers secrets $p$ et $q$ tels que $n = pq$.

## 8.8 Exercices

1. On démontre que :
$$F_m \wedge F_n = F_{m \wedge n}$$
où $F_n$ désigne le terme numéro $n$ de la suite de Fibonacci (avec $F_0 = 0, \ F_1 = 1$). Ecrire un programme qui range dans un tableau les 46 premiers termes de la suite de Fibonacci (le quarante-septième dépasse la limite des entiers positifs représentés sur 32 chiffres binaires), et fournisse le PGCD de $F_m$ et $F_n$, avec les coefficients de Bezout correspondants, lorsque l'utilisateur entre $m$ et $n$ au clavier. Que constate-t-on lorsque $n = m + 1$ ?

2. Soit $p_n$ le nombre d'entiers premiers compris entre 1 et $n$ ; on appelle densité des nombres premiers sur l'intervalle $[1 \,..\, n]$ le nombre $d_n = p_n/n$. Le "théorème fondamental de l'arithmétique" affirme que : $d_n \sim 1/\ln n$.
Ecrire une fonction qui calcule $p_n$ (en utilisant la fonction `premier` de la figure 31), et examiner les valeurs de la suite $p_n \times \ln n$, pour $n = 10\,000, 20\,000, \ldots 100\,000$.

3. La conjecture de Goldbach affirme que tout entier pair est somme de deux nombres premiers ; elle n'est toujours pas démontrée.

   Ecrire une fonction qui calcule une décomposition d'un entier pair $n$ en somme de deux nombres premiers (en utilisant la fonction `premier` de la figure 31).

   Ecrire un programme qui affiche les entiers pairs les plus "durs" à décomposer ; précisément, en notant $p_n$ le plus petit entier $p$ tel que :

   $$n = p + q, \quad \text{avec } p, q \text{ premiers}$$

   le programme doit lister les entiers pairs $n \leq m$ tels que $p_n$ soit supérieur à tous les $p_k$ , $k < n$ ; l'utilisateur entre la valeur maximale $m$, et le programme affiche aussi, pour chaque $n$ qui répond aux conditions ci-dessus, sa décomposition en somme de deux nombres premiers.

4. La fonction indicatrice d'Euler $\Phi(n)$ compte le nombre d'entiers inférieurs à $n$ et premiers avec $n$. Calculer cette fonction :

   (a) en appliquant brutalement la définition, et en utilisant la fonction `pgcd` de la figure 27 ;

   (b) en utilisant la fonction `decompose` de la figure 32, et en appliquant le théorème :

   $$n = \prod p_i^{\alpha_i} \implies \Phi(n) = \prod p_i^{\alpha_i - 1}(p_i - 1)$$

   où les $p_i$ sont les facteurs premiers *distincts* de $n$.

# 9 Algèbre linéaire

## 9.1 Matrices

Une matrice $a$ est représentée, en C, par un *tableau de tableaux* : $a[i]$ désigne la ligne numéro $i$, qui est elle-même un tableau, dont l'élément numéro $j$ est $a[i][j]$. Les notations sont ainsi analogues à celles des mathématiques : le premier indice désigne la ligne, le second la colonne ; il faut seulement prendre garde à la numérotation, qui commence à 0, au lieu de 1 en mathématiques.

Une bibliothèque de fonctions pour manipuler les matrices commence donc par des définitions analogues aux suivantes :

```
#define HAUTEUR_MAX 100
#define LARGEUR_MAX 200
typedef double matrice [HAUTEUR_MAX] [LARGEUR_MAX];
```

Les matrices ainsi définies sont de taille fixe ; il existe des techniques sophistiquées pour contourner cette limitation, mais dans le cadre de ce cours, on se contentera de fournir en paramètres des fonctions, le nombre $m$ (resp. $n$) de lignes (resp. colonnes) réellement utiles.

```
void ecrire (matrice a, int m, int n) {
  int i, j;
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
      printf ("%10.4lf", a [i][j]);
    printf ("\n");
  }
}
```

FIG. 34 – Ecrire une matrice

```
void copier (matrice a, matrice b, int m, int n) {
  int i, j;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      a [i][j] = b [i][j];
}
```

FIG. 35 – Copier une matrice

La figure 34 indique comment afficher une matrice sous sa forme rectangulaire traditionnelle, et la figure 35 comment copier une matrice dans une autre (puisque l'instruction `a = b` n'a pas de sens pour les tableaux). Noter l'usage des doubles boucles. Sur le même modèle, on peut écrire la somme de deux matrices, ou le produit d'une matrice par un scalaire.

La figure 36 est la traduction, en C, du produit $A$ (de taille $m \times p$) de la matrice $B$ (de taille $m \times n$) par la matrice $C$ (de taille $n \times p$) ; les trois instructions à l'intérieur de la double boucle traduisent la formule mathématique :

$$A_{ij} = \sum_k B_{ik} C_{kj} \ ,$$

```
void produit (matrice a, matrice b, matrice c,
              int m, int n, int p) {
  int i, j, k;
  double x;

  for (i = 0; i < m; i++)
    for (j = 0; j < p; j++) {
      x = 0;
      for (k = 0; k < n; k++)
        x = x + b [i][k] * c [k][j];
      a [i][j] = x;
    }
}
```

FIG. 36 – Produit de matrices

qui définit le produit de la ligne $i$ par la colonne $j$.

## 9.2 Algorithme du pivot de Gauss

L'algorithme du pivot de Gauss est la méthode fondamentale qui permet de résoudre des systèmes d'équations linéaires, et donc d'inverser des matrices ; c'est aussi une très bonne méthode pour calculer un déterminant.

Commençons par un exemple ; soit à résoudre le système :

$$\begin{array}{rcrcrcr} x & + & 2y & + & 3z & = & 14 \\ 2x & + & 3y & + & z & = & 11 \\ 3x & - & y & - & 2z & = & -5 \end{array}$$

En multipliant la première ligne par $-2$ et en l'ajoutant à la seconde, on peut éliminer $x$ de la seconde équation ; idem en multipliant la première ligne par $-3$ et en l'ajoutant à la troisième, soit :

$$\begin{array}{rcrcrcr} x & + & 2y & + & 3z & = & 14 \\ & - & y & - & 5z & = & -17 \\ & - & 7y & - & 11z & = & -47 \end{array}$$

Les deux dernières équations forment un système à seulement deux inconnues ; on peut éliminer $y$ de la troisième équation en lui ajoutant la seconde, multipliée par $-7$, soit :

$$\begin{array}{rcrcrcr} x & + & 2y & + & 3z & = & 14 \\ & - & y & - & 5z & = & -17 \\ & & & & 24z & = & 72 \end{array}$$

Le système est maintenant *triangulaire*, et donc immédiat à résoudre *de bas en haut* : $z = 3$, on reporte dans la seconde équation : $-y - 15 = -17$, soit $y = 2$ ; la première équation devient $x + 13 = 14$, soit $x = 1$.

L'algorithme général met sous forme triangulaire une matrice $A$ de taille $m \times n,\ m \leq n$, ici :

$$\begin{pmatrix} 1 & 2 & 3 & 14 \\ 2 & 3 & 1 & 11 \\ 3 & -1 & 2 & -5 \end{pmatrix}$$

Il existe plusieurs variantes de l'algorithme, selon qu'il opère sur les lignes, sur les colonnes, ou simultanément sur les lignes et les colonnes ; ici la seule opération employée consiste à multiplier une ligne par une constante, et à l'ajouter à une autre ; l'étape numéro $i$ peut se formuler ainsi :

pour chaque $j > i$, multiplier la ligne $i$ par $-A_{ij}/A_{ii}$, et l'ajouter à la ligne $j$.

On annule ainsi les coefficients $A_{ij}$ pour $j > i$, c'est à dire ceux qui se trouvent au-dessous de la diagonale dans la colonne $i$.

La seule difficulté est de savoir que faire lorsque $A_{ii}$, qu'on appelle le *pivot*, est nul ; la solution est d'échanger la ligne $i$ avec une ligne $j > i$ telle que $A_{ij} \neq 0$. S'il n'existe pas de telle ligne, c'est à dire si, à l'étape $i$ de l'algorithme, on a $A_{ij} = 0$ pour tout $j > i$, le rang de la matrice est inférieur au nombre $m$ de lignes ; **ce cas sera exclu dans ce chapitre**.

Une amélioration simple consiste à choisir pour pivot le coefficient $A_{ij}$ , $i \leq j$, le plus grand en valeur absolue : on montre que, si la matrice est de grande taille, les erreurs d'arrondi sont plus faibles avec cette stratégie.

```
void gauss (matrice a, int m, int n) {
  int i, j, k, pivot;
  double x;

  for (i = 0; i < m - 1; i++) {
    pivot = i;
    for (j = i + 1; j < m; j++)
      if ( fabs (a [j][i]) > fabs (a [pivot][i]) )
        pivot = j;

    if (i < pivot)
      for (k = i; k < n; k++) {
        x = a [pivot][k];
        a [pivot][k] = a [i][k];
        a [i][k] = x;
      }

    for (j = i + 1; j < m; j++) {
      x = a [j] [i] / a [i][i];
      for (k = i; k < n; k++)
        a [j][k] = a [j][k] - x * a [i][k];
    }
  }
}
```

FIG. 37 – Une version de l'algorithme de Gauss

La figure 37 donne l'algorithme complet en C, qui comporte $m-1$ étapes ; l'étape numéro $i$ est constituée de trois groupes d'instructions :

1. trouver le pivot ;
2. si nécessaire, échanger la ligne $i$ et celle du pivot ;

3. pour chaque $j > i$, modifier la ligne $j$ ; noter qu'il faut calculer le coefficient $A_{ij}/A_{ii}$ *avant* de modifier la ligne, puisque le but de la modification est d'annuler ce coefficient.

La figure 38 donne un exemple d'exécution, avec une matrice de 4 lignes et 5 colonnes.

```
Matrice initiale:
    1.0000    2.0000    3.0000   -4.0000   -2.0000
   -2.0000    3.0000    4.0000    1.0000   20.0000
    3.0000   -4.0000   -1.0000    2.0000    0.0000
    4.0000   -1.0000    2.0000   -3.0000   -4.0000

Etape 1:
    4.0000   -1.0000    2.0000   -3.0000   -4.0000
    0.0000    2.5000    5.0000   -0.5000   18.0000
    0.0000   -3.2500   -2.5000    4.2500    3.0000
    0.0000    2.2500    2.5000   -3.2500   -1.0000

Etape 2:
    4.0000   -1.0000    2.0000   -3.0000   -4.0000
    0.0000   -3.2500   -2.5000    4.2500    3.0000
    0.0000    0.0000    3.0769    2.7692   20.3077
    0.0000    0.0000    0.7692   -0.3077    1.0769

Etape 3:
    4.0000   -1.0000    2.0000   -3.0000   -4.0000
    0.0000   -3.2500   -2.5000    4.2500    3.0000
    0.0000    0.0000    3.0769    2.7692   20.3077
    0.0000    0.0000    0.0000   -1.0000   -4.0000
```

FIG. 38 – Exemple d'exécution de l'algorithme de Gauss

**Complexité :** les opérations les plus longues et les plus nombreuses sont celles de la partie suivante du programme :

```
for (j = i + 1; j < m; j++) {
  x = a [j] [i] / a [i][i];
  for (k = i; k < n; k++)
    a [j][k] = a [j][k] - x * a [i][k];
}
```

qui exécute $(m-i-1)(n-i+1)$ multiplications ou divisions. La complexité de l'algorithme est donc mesurée par :

$$\sum_{i=0}^{m-2}(m-i-1)(n-i+1)$$

En supposant pour simplifier la matrice carrée, c'est à dire $m = n$ (mais le cas général n'est

pas beaucoup plus difficile à traiter), et en posant $j = n - i - 1$, cette quantité vaut :

$$\sum_{j=1}^{n-1} j(j+2)$$

dont le terme dominant est :

$$\sum_{j=1}^{n-1} j^2 = \frac{n(n-1)(2n-1)}{6} \sim \frac{n^3}{3}$$

## 9.3 Systèmes d'équations linéaires

```
void systeme (matrice a, matrice b, matrice x, int m, int n) {
  int i, j, k;
  double y;

  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      a [i][m + j] = b [i][j];

  gauss (a, m, m + n);

  for (j = 0; j < n; j++)
    for (i = m - 1; i >= 0; i--) {
      y = a [i][m + j];
      for (k = i + 1; k < m; k++)
        y = y - a [i][k] * x [k][j];
      x [i][j] = y / a [i][i];
    }
}
```

FIG. 39 – Résolution du système linéaire $AX = B$

Pour résoudre un système linéaire $AX = B$, où $A$ est une matrice $m \times m$, $X$ et $B$ des vecteurs de longueur $m$ :

1. on copie le vecteur des seconds membres, $B$, à droite de la matrice $A$ ;
2. on applique l'algorithme de Gauss à la matrice ainsi obtenue ; l'étape précédente garantit que les transformations appliquées aux équations, le sont aussi aux seconds membres ;
3. on résout le système triangulaire ainsi obtenu, de bas en haut.

Il n'est guère plus difficile de résoudre un système $AX = B$, où $X$ et $B$ sont cette fois des matrices $m \times n$ ; l'essentiel est d'appliquer une fois et une seule l'algorithme de Gauss à la matrice $A$ complétée à gauche par $B$ ; on résout ensuite le système triangulaire comme précédemment, colonne par colonne. Comme le produit de la ligne $i$ de la matrice $A$ (triangularisée) par la colonne $j$ de $X$ vaut :

$$\sum_{k=i}^{m} A_{ik} X_{kj} = B_{ij}$$

on en déduit la formule :

$$X_{ij} = \left( B_{ij} - \sum_{k=i+1}^{m} A_{ik} X_{kj} \right) / A_{ii}$$

(les $X_{kj}$ , $k > i$ sont déjà connus, car on résout le système de bas en haut). La figure 39 donne les détails de la traduction en C.

La résolution du système tiangulaire demande approximativement

$$1 + 2 + \ldots + m = \frac{m(m+1)}{2} \sim \frac{m^2}{2}$$

opérations, pour chaque colonne de $X$.

## 9.4 Inversion de matrices

```
void inverse (matrice a, matrice b, int n) {
  int i, j;
  matrice id;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (i == j)
        id [i][j] = 1;
      else
        id [i][j] = 0;
  systeme (a, id, b, n, n);
}
```

FIG. 40 – Inversion de matrice

Trouver la matrice inverse de $A$ revient à résoudre le système $AX = I$, où $I$ désigne la matrice identité ; c'est donc une fonction très facile à écrire une fois qu'on dispose de la fonction *systeme* de la figure 39 : voir figure 40.

La complexité de l'algorithme de Gauss, appliqué à une matrice $n \times 2n$, a pour terme dominant $5n^3/6$ ; la résolution du système triangulaire, pour $n$ colonnes, demande approximativement $n^3/2$ opérations supplémentaires ; la complexité de l'inversion d'une matrice $n \times n$, comptée en multiplications, a donc pour terme dominant $4n^3/3$. Ce résultat est très satisfaisant, si l'on réalise qu'on calcule $n^2$ termes : le coût du calcul de chacun est linéaire.

## 9.5 Déterminants

L'algorithme de Gauss possède aussi l'avantage, lorsqu'il est appliqué à une matrice carrée, de ne pas modifier le déterminant de la matrice, au signe près. En effet, les seules opérations effectuées sont :

- remplacer la ligne $l_j$ par $l_j - x l_i$, ce qui ne change pas le déterminant ;
- échanger deux lignes (pour utiliser le meilleur pivot possible), ce qui change seulement le signe du déterminant.

```
double determinant (matrice a, int n) {
  int i;
  double x;

  x = gauss (a, n, n);
  for (i = 0; i < n; i++)
    x = x * a [i][i];
  return x;
}
```

FIG. 41 – Calcul de déterminant

D'autre part, le déterminant d'une matrice triangulaire est égal au produit des termes sur la diagonale, d'où la fonction de la figure 41, qui utilise une version légèrement modifiée de la fonction *gauss* (voir figure 42) : celle-ci retourne la valeur 1 ou $-1$ selon que le nombre d'échanges de lignes est pair ou impair.

La complexité du calcul du déterminant, par cette méthode, est approximativement $n^3/3$ ; en supposant que le processeur effectue un million de multiplications par seconde, on obtient les temps de calcul suivants, en fonction de la taille de la matrice :

| $n$ | Temps de calcul |
|---:|---:|
| 10 | $<$ 1 millisec. |
| 100 | $<$ 1 sec. |
| 1000 | $\simeq$ 6 min. |

Au contraire, l'application directe de la définition du déterminant, qui comporte $n!$ termes, fournit un algorithme terriblement inefficace :

- $25! \simeq 10^{25}$ ;
- une journée comporte environ $10^5$ secondes, une année $3 \times 10^7$ secondes, et l'âge de l'univers est estimé à 15 milliards d'années, soit moins de $10^{18}$ secondes, ou encore $10^{24} \mu s$.

Un processeur qui effectuerait un million de multiplications par seconde depuis la naissance de l'univers, n'aurait donc pas terminé le calcul du déterminant d'une matrice $25 \times 25$, en appliquant directement la définition ! Cet exemple est typique de la différence entre les algorithmes de complexité polynomiale, utiles en pratique, et ceux de complexité exponentielle, inutilisables en dehors de cas très simples.

```
int gauss (matrice a, int m, int n) {
  int i, j, k, pivot, echange = 1;
  double x;

  for (i = 0; i < m - 1; i++) {
    pivot = i;
    for (j = i + 1; j < m; j++)
      if ( fabs (a [j][i]) > fabs (a [pivot][i]) )
        pivot = j;

    if (i < pivot) {
      for (k = i; k < n; k++) {
        x = a [pivot][k];
        a [pivot][k] = a [i][k];
        a [i][k] = x;
      }
      echange = -echange;
    }

    for (j = i + 1; j < m; j++) {
      x = a [j] [i] / a [i][i];
      for (k = i; k < n; k++)
        a [j][k] = a [j][k] - x * a [i][k];
    }
  }
  return echange;
}
```

FIG. 42 – Version modifiée de l'algorithme de Gauss

# 10 Annales

## 10.1 Janvier 1995

UNIVERSITÉ BORDEAUX I — Département premier cycle Sciences
Janvier 1995

**Module I2**
(Durée : 1h30)

Notes de cours et de TD autorisées. Autres documents interdits.

Les **3** exercices sont indépendants ; donnez des réponses simples et précises : les fonctions C demandées sont très courtes (quelques lignes). Ecrire des généralités, ou copier des fonctions vues en cours ou en TD, vous fera perdre du temps sans vous rapporter aucun point.

1. Pour calculer une valeur approchée de l'intégrale :

$$\int_a^b f(x)dx$$

on peut diviser l'intervalle $[a\ b]$ en $n$ segments *égaux*, délimités par les points :

$$x_0 = a,\ x_1,\ x_2,\ \ldots,\ x_n = b$$

et remplacer $f$, sur chaque segment $[x_i\ x_{i+1}]$, par la fonction constante égale à $f(x_i)$ ; en d'autres termes on approche $f$ par une fonction en escalier.

Ecrire, en langage C, la fonction `integrale` , qui effectue ce calcul approché ; on suppose la fonction $f$ écrite par ailleurs. Veiller à spécifier au mieux les paramètres.

Une meilleure approximation est obtenue en remplaçant $f$, sur $[x_i\ x_{i+1}]$, par la fonction constante égale à :

$$\frac{f(x_i) + f(x_{i+1})}{2}\ ;$$

écrire la version modifiée correspondante de la fonction `integrale`, en veillant à ne pas calculer deux fois la même valeur $f(x)$.

2. On considère la suite $u_n$ définie par la récurrence suivante :

$$\begin{cases} u_0 & = & 1 \\ u_{n+1} & = & u_0u_n + u_1u_{n-1} + \ldots + u_nu_0 \quad = \quad \displaystyle\sum_{k=0}^{n} u_ku_{n-k}\ ,\ n \geq 0 \end{cases}$$

Calculer les cinq termes $u_1, u_2, \ldots, u_5$ de la suite. Ecrire, en langage C, la fonction `suite` , qui remplit un tableau avec les $n+1$ premiers termes $u_0, \ldots, u_n$ de la suite. Veiller à spécifier au mieux les paramètres.

3. On considère la fonction suivante :

```
#define MAX 20;

void f (int *t, int *u, int n) {
  int i, j;

  for (j = 0; j <= MAX; j++)
    u [j] = 0;
  for (i = 0; i < n; i++) {
    j = t [i];
    u [j]++;
  }
}
```

On suppose donné un tableau `notes`, contenant les dix valeurs suivantes :

$$12\ 14\ 10\ 8\ 12\ 12\ 15\ 8\ 13\ 10\ ,$$

et déclaré un tableau : `int a [MAX + 1] ;` quel est le résultat de l'appel :

`f (notes, a, 10) ?`

Que calcule, de façon générale, la fonction $f$ ?
Cette fonction comporte une instruction dangereuse : laquelle ? Que faudrait-il ajouter pour rendre cette fonction sûre ?

## 10.2 Corrigé

1. En posant :
$$\delta = \frac{b-a}{n}$$
la méthode de calcul proposée consiste à remplacer :
$$\int_a^b f(x)dx \text{ par : } \sum_{i=0}^{n-1} \delta f(x_i)$$
La fonction C n'utilise pas de tableau ; chaque $x_i$ se déduit du précédent en ajoutant $\delta$, qu'on calcule une seule fois, en début de fonction.

```
double integrale (double a, double b, int n) {
  int i;
  double x = a, delta = (b - a) / n, s = 0;

  for (i = 0; i < n; i++) {
    s = s + f (x);
    x = x + delta;
  }
  return s * delta;
}
```

De nombreuses variantes sont possibles, par exemple, sans utiliser de variable $x$ (mais au prix de multiplications supplémentaires) :

```
  for (i = 0; i < n; i++)
    s = s + f (a + i * delta);
```

Noter qu'il est inutile de traiter à part le cas $b < a$ : la formule, et la fonction ci-dessus, sont correctes dans ce cas.
La seconde méthode de calcul (plus précise, et appelée *méthode du trapèze*) consiste à remplacer :

$$\int_a^b f(x)dx \text{ par : } \sum_{i=0}^{n-1} \delta \frac{f(x_i) + f(x_{i+1})}{2} = \delta \left( \frac{f(a) + f(b)}{2} + \sum_{i=1}^{n-1} f(x_i) \right)$$

d'où la fonction modifiée, très voisine de la précédente :

```
double integrale (double a, double b, int n) {
  int i;
  double
    x = a,
    delta = (b - a) / n,
    s = ( f(a) + f(b) ) / 2;

  for (i = 1; i < n; i++) {
    x = x + delta;
    s = s + f (x);
  }
  return s * delta;
}
```

2. Les premières valeurs de la suite sont :

$$u_1 = 1 \ , \ u_2 = 2 \ , \ u_3 = 5 \ , \ u_4 = 14 \ , \ u_5 = 42 \ .$$

Contrairement aux récurrences à deux ou trois termes, celle-ci exige un tableau pour sa traduction en C :

```
void suite (long *u, int n) {
  int i, j;

  u [0] = 1;
  for (i = 1; i <= n; i++) {
    u [i] = 0;
    for (j = 0; j < i; j++)
      u [i] = u [i] + u [j] * u [i - j - 1];
  }
}
```

3. L'appel : `f (notes, a, 10)` remplit le tableau $a$ avec les valeurs suivantes :

| $i$ | 0 | ... | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | ... | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $a[i]$ | 0 | 0 | 0 | 2 | 0 | 2 | 0 | 3 | 1 | 1 | 1 | 0 | 0 | 0 |

La fonction $f$ enregistre, dans le tableau $u$, le nombre d'occurrences de chaque note dans le tableau $t$ (par exemple pour préparer un histogramme).

L'instruction dangereuse est `j = t [i] ; u [j]++` , car elle suppose que le tableau $t$ ne contient que des entiers entre 0 et 20 ; des fautes de saisie peuvent alors entraîner un comportement dangereux (ou une terminaison anormale) du programme. Il est conseillé d'écrire plutôt :

```
j = t [i];
if (j >= 0 && j <= MAX)
  u [j]++;
```

## 10.3 Juin 1995

UNIVERSITÉ BORDEAUX I — Département premier cycle Sciences
Juin 1995

### Module I2
(Durée : 1h30)

Polycopié, notes de cours et de TD autorisés. Autres documents interdits.

Les **3** exercices sont indépendants ; donner des réponses simples et précises : les fonctions C demandées sont très courtes (quelques lignes). Ecrire des généralités, ou copier des fonctions vues en cours ou en TD, vous fera perdre du temps sans vous rapporter aucun point.

1. On représente un nombre entier positif $x$ par le tableau $a$ de ses chiffres, en base $b$, de telle sorte que :
$$x = a_0 + a_1 b + a_2 b^2 \ldots + a_n b^n$$
On dit que $a_k$ est le chiffre de poids $k$ ; noter que l'ordre des chiffres dans le tableau (poids croissants) est l'inverse de celui utilisé pour écrire habituellement $x$ (poids décroissants). On appellera MAX le nombre maximal de chiffres autorisés ; on pourra (mais ce n'est pas obligatoire) utiliser une déclaration :
```
typedef int nombre [MAX];
```
Veiller, dans tous les cas, à la qualité des en-têtes de fonctions.
   (a) Ecrire, en langage C, une fonction qui affiche les chiffres de $x$ dans l'ordre habituel, c'est à dire en commençant par le premier chiffre non nul de poids le plus fort.
   Note : ici, comme dans le reste de l'exercice, on ne demande aucune conversion d'une base dans une autre.
   (b) Ecrire, en langage C, une fonction qui calcule la somme $x + y$ ; pourquoi est-elle différente de la fonction qui calcule la somme de deux polynômes ?
   Comment proposez-vous de traiter le cas où le nombre de chiffres de $x + y$ dépasse MAX ?
   (c) On souhaite manipuler les nombres les plus grands possibles, pour une valeur donnée de MAX ; quel est alors, à votre avis, le meilleur choix pour la base $b$ ? Justifier soigneusement votre réponse, en vérifiant en particulier que la fonction précédente, pour le calcul de la somme, reste correcte.

2. On représente, comme en cours, un nombre complexe par la structure :
```
struct complexe {double reel, imaginaire; };
```
et on dispose des fonctions :
```
struct complexe somme (struct complexe u, struct complexe v);
struct complexe produit (struct complexe u, struct complexe v);
double module (struct complexe z);
```

Soit $f$ la fonction définie par $f(z) = z^2 + c$, où $c$ désigne une constante complexe, et soit $z_n$ la suite définie par la récurrence :

$$z_0 = z \ , \ z_n = f(z_{n-1}) \text{ pour } n > 0.$$

(a) Ecrire, en langage C, la fonction $f$.

(b) Ecrire, en langage C, la fonction booléenne qui indique si $z_n$ se trouve dans le disque, centré en l'origine, de rayon $r$.

Note : veiller à bien spécifier les paramètres de cette fonction.

3. On considère la fonction suivante, vue en cours, qui affiche la décomposition en facteurs premiers d'un entier :

```
void decompose (long n) {
  long d;
  for (d = 2; d * d <= n;)
    if (n % d == 0) {
      printf (" %d", d);
      n = n / d;
    }
    else d++;
  printf (" %d\n", n);
}
```

Par exemple, si $n = 200$, cette fonction affiche : 2 2 2 5 5.

Modifier cette fonction pour qu'elle affiche chaque facteur avec sa multiplicité ; par exemple, si $n = 200$, la fonction modifiée affiche : 2 puissance 3, 5 puissance 2.

## 10.4 Corrigé

1. (a) *(5 points)* Il faut parcourir les chiffres de droite à gauche (dans le tableau), jusqu'à trouver un chiffre non nul (comme pour calculer le degré d'un polynôme), puis afficher tous les suivants :

```
void afficher (nombre x) {
  int i, j;

  for (i = MAX - 1; i > 0 && x [i] == 0; i--);
  for (j = i; j >= 0; j--)
    printf ("%d", x [j]);
  printf ("\n");
}
```

Variantes et remarques :

– On peut écrire l'en-tête sous la forme :
`void afficher (int *x)` ou : `void afficher (int x [])`

– L'oubli de `i > 0` dans la condition `i > 0 && x [i] == 0` est une faute (mineure dans le contexte de l'examen), car $x$ peut représenter le nombre nul.

– On peut ne pas utiliser de variable $j$, et écrire la seconde boucle :
`for (; i >= 0; i--) printf ("%d", x [i]);`

(b) *(5 points)* Pour faire la somme de deux nombres, il faut modifier la fonction qui effectue la somme de deux polynômes, en tenant compte des retenues, pour que les chiffres ne dépassent jamais la base :

```
void somme (nombre z, nombre x, nombre y, int base) {
  int i, retenue = 0;

  for (i = 0; i < MAX; i++) {
    z [i] = x [i] + y [i] + retenue;
    if (z [i] >= base) {
      retenue = 1;
      z [i] = z [i] - base;
    }
    else
      retenue = 0;
  }
}
```

Noter que la base est un *paramètre* de cette fonction, car il est impossible d'additionner $x$ et $y$ sans connaître la base utilisée.

Le nombre de chiffres de $x+y$ dépasse MAX si et seulement si la fonction se termine avec `retenue = 1`. Le moyen le plus simple de le signaler à l'utilisateur (qui en fera ce qu'il voudra), est de transformer cette fonction en une fonction *booléenne* : remplacer `void` par `int` dans l'en-tête, et ajouter

```
return retenue;
```

à la fin *(2 points)*.

Remarque : certains étudiants n'ont pas utilisé de variable `retenue`, et ont écrit :

```
for (i = 0; i < MAX; i++) {
  z [i] = x [i] + y [i] + z [i];
  if (z [i] >= base) {
    z [i + 1] = 1;
    z [i] = z [i] - base;
  }
  else
    z [i + 1] = 0;
}
```

Mais cette solution est fausse pour la première et la dernière valeur de $i$ : `z [0]` n'est pas initialisé, et surtout `z [MAX]` n'existe pas.

(c) *(3 points)* Il faut choisir la base $b$ aussi grande que possible, en veillant à ce que la somme de deux chiffres reste inférieure au plus grand entier positif représentable en machine ; si les entiers sont représentés avec 16 chiffres binaires, la plus grande valeur de $b$ possible est donc $2^{15} - 1$ (en fait il faut alors remplacer le type `int` par `unsigned int`, ce qui sort du cadre de I2).

2. (a) *(4 points)*

```
struct complexe f (struct complexe z, struct complexe c) {
  return somme ( produit (z, z), c );
}
```

(b) *(5 points)* Il faut calculer $z_n$ par récurrence, puis comparer son module au rayon du disque ; cette fonction comporte *quatre* paramètres : le point de départ $z$, la valeur de $c$, l'indice $n$ (un entier), et le rayon $r$ (un réel).

```
int disque (struct complexe z, struct complexe c,
            int n, double r) {
  int i;
  for (i = 1; i <= n; i++)
    z = f (z, c);
  return ( module (z) <= r );
}
```

La variante suivante est évidemment correcte, mais peu élégante :

```
  if ( module (z) <= r )
    return 1;
  else
    return 0;
```

3. *(6 points)* La solution la plus simple est de remplacer la condition `if` par une boucle `while` :

```
void decompose (long n) {
  long d;
  int multiplicite;
  for (d = 2; d * d <= n; d++) {
    multiplicite = 0;
    while (n % d == 0) {
      multiplicite++;
      n = n / d;
    }
    if (multiplicite > 0)
      printf ("%d puissance %d, ", d, multiplicite);
  }
  if (n > 1)
    printf ("%d", n);
  printf ("\n");
}
```

Ne pas oublier `if (multiplicite > 0)` avant `printf`. La solution voisine :

```
int multiplicite = 0;

for (d = 2; d * d <= n;) {
  if (n % d == 0) {
    multiplicite++;
    n = n / d;
  }
  else {
    if (multiplicite > 0)
      printf ("%d puissance %d, ", d, multiplicite);
```

```
      multiplicite = 0;
      d++;
    }
  }
```

a été considérée comme correcte, mais en fait elle pose des problèmes délicats quand on sort de la boucle : le dernier diviseur $d$ a-t-il bien été affiché, et sinon est-il égal à $n$ ?

## 10.5 Janvier 1996

UNIVERSITÉ BORDEAUX I — Département premier cycle Sciences
Janvier 1996

### Module I2
(Durée : 1h30)

Polycopié, notes de cours et de TD autorisées. Autres documents interdits.

Le sujet comporte une page recto-verso. Les **3** exercices sont indépendants ; donnez des réponses simples et précises : les fonctions C demandées sont très courtes (quelques lignes). Ecrire des généralités, ou copier des fonctions vues en cours ou en TD, vous fera perdre du temps sans vous rapporter aucun point.

1. (a) Ecrire en langage C la fonction qui, à deux tableaux d'entiers positifs $p, \alpha$ associe :

$$\prod_{i=0}^{n-1} p_i^{\alpha_i}$$

où $n$ désigne le plus petit indice tel que $p_n = 0$.

Note : le calcul devra être fait directement, sans recourir à une fonction *puissance* écrite par ailleurs.

(b) On suppose disponible une fonction booléenne `int premier (long x)` qui indique si l'entier $x$ est premier ; en utilisant cette fonction, écrire en langage C une fonction booléenne qui indique si les $n$ premiers éléments du tableau $p$ sont des nombres premiers *distincts*.

Notes : comme ci-dessus, $n$ désigne le plus petit indice tel que $p_n = 0$. Une fonction booléenne est une fonction dont le résultat est vrai (1) ou faux (0).

2. On suppose qu'un polynôme est représenté, comme en cours, par le tableau de ses coefficients, et on pourra utiliser librement le type *polynome* et la fonction *degre* définis en cours.

(a) Ecrire en langage C la fonction qui, à un polynôme $A$, associe :

$$\int_0^1 A(t)dt$$

(b) Ecrire en langage C la fonction qui, à un polynôme $A$, associe le polynôme $B$ défini par :

$$\begin{cases} B' = A \\ \displaystyle\int_0^1 B(t)dt = 0 \end{cases}$$

3. On suppose disponible une fonction `int hasard (int n)` qui fournit aléatoirement un entier compris entre 0 et $n-1$. On utilise `hasard (2)` pour simuler une marche aléatoire sur un axe : selon le résultat, on fait un pas en avant ou un pas en arrière ; de même, on utilise `hasard (4)` pour simuler une marche aléatoire sur un quadrillage plan : selon le résultat, on fait un pas vers le Nord, ou vers l'Est, ou vers le Sud, ou vers l'Ouest. Dans tous les cas, on part de l'origine, et on appelle *retour à l'origine* tout passage par le point de départ après $k$ pas ($k > 0$).

(a) Ecrire, en langage C, une fonction qui simule une marche aléatoire de $n$ pas sur un axe ; cette fonction devra calculer l'abscisse du point d'arrivée, l'abscisse du point le plus à droite atteint au cours de la marche, et le nombre de retours à l'origine.

Note : on tentera de bien spécifier, soit en définissant une structure, soit en utilisant pour paramètres des adresses, la façon dont cette fonction retourne simultanément les trois résultats demandés.

(b) Ecrire, en langage C, une fonction qui simule une marche aléatoire d'au plus $n$ pas, sur un quadrillage plan, jusqu'au premier retour à l'origine, et calcule le nombre de pas effectués.

## 10.6 Corrigé

1. (a) Une double boucle suffit :

```
long produit (int *p, int *alpha) {
  long x = 1;
  int i, j;

  for (i = 0; p [i] > 0; i++)
    for (j = 0; j < alpha [i]; j++)
      x = x * p [i];
  return x;
}
```

Noter que cette solution est correcte même pour un exposant $\alpha_i$ nul, ou un vecteur vide. L'énoncé n'indique pas de taille `MAX` pour les vecteurs $p$ et $\alpha$ ; celle-ci n'est donc pas connue, et il n'est pas nécessaire d'écrire, dans ces conditions :

```
for (i = 0 ; i < MAX && p [i] > 0 ; i++)
```

(b) On termine le calcul dès qu'un $p_i$ n'est pas premier, ou dès qu'un $p_i$ figure déjà dans le tableau ; si par contre tous les éléments de $p$ franchissent positivement les tests, le résultat est vrai (1).

```
int premiers_distincts (int *p) {
  int i, j;

  for (i = 0; p [i] > 0; i++) {
    if (! premier (p[i]) )
      return 0;
    for (j = 0; j < i; j++)
      if (p [j] == p [i])
        return 0;
  }
  return 1;
}
```

2. (a) Si

$$A = \sum_{i=0}^{n} a_i X^i$$

l'intégrale définie vaut :

$$\int_0^1 A(t)dt = \left[\sum_{i=0}^{n} a_i X^{i+1}/(i+1)\right]_0^1 = \sum_{i=0}^{n} \frac{a_i}{i+1}$$

d'où la fonction C :

```
double integrale (polynome a) {
  double s = 0;
  int i, n = degre (a);

  for (i = 0; i <= n; i++)
    s = s + a[i] / (i+1);
  return s;
}
```

(b) La primitive $B$, avec terme constant nul, vaut :

$$B = \sum_{i=0}^{n} a_i X^{i+1}/(i+1)$$

Il suffit ensuite de calculer le terme constant de telle sorte que :

$$\int_0^1 B(t)dt = 0$$

```
void primitive  (polynome b, polynome a) {
  int i, n = degre (a);

  b [0] = 0;
  for (i = 0; i <= n; i++)
    b [i+1] = a[i] / (i+1);
  for (i = n + 2; i <= DEGRE_MAX; i++)
    b [i] = 0;
  b [0] = - integrale (b);
}
```

Noter qu'il faut prendre garde à bien annuler les termes de degré supérieur à $n+1$. Il est bon de signaler, sur la copie, que le calcul d'une primitive $B$ n'a de sens que si $A$ n'est pas de degré maximal.

3. (a) En définissant une structure, on a la solution suivante :

```
struct marche {
  int position, droite, retour;
};

struct marche simul1 (int n) {
  struct marche m = {0, 0, 0};
  int i, x = 0;
```

```
  for (i = 0; i < n; i++) {
    if ( hasard (2) )
      { x++; if (m.droite < x) m.droite = x; }
    else x--;
    if (x == 0) m.retour++;
  }
  m.position = x;
  return m;
}
```

On peut évidemment éviter la variable $x$, en la remplaçant par `m.position`. En utilisant des adresses, la solution devient :

```
void simul1 (int n, int *position, int *droite, int *retour) {
  int i, x = 0;

  for (i = 0; i < n; i++) {
    if ( hasard (2) )
      { x++; if (*droite < x) *droite = x; }
    else x--;
    if (x == 0) (*retour)++;
  }
  *position = x;
}
```

On peut, comme précédemment, éviter la variable $x$, en la remplaçant par `*position`.

(b) En deux dimensions, on retourne le nombre de pas dès qu'on retourne à l'origine ; sinon, on convient de retourner 0.

```
int simul2 (int n) {
  int i, h, x = 0, y = 0;

  for (i = 0; i < n; i++) {
    h = hasard (4);
    if (h == 0) y++;
    else if (h == 1) x++;
    else if (h == 2) y--;
    else x--;
    if (x == 0 && y == 0)
      return i + 1;
  }
  return 0;
}
```

Noter que l'instruction *switch* est mieux adaptée à des choix multiples, comme ici, mais elle n'a pas été présentée en cours.

## 10.7 Juin 1996

UNIVERSITÉ BORDEAUX I — Département premier cycle Sciences
Juin 1996

### Module I2
(Durée : 1h30)

Polycopié, notes de cours et de TD autorisées. Autres documents interdits.

Le sujet comporte une page recto-verso. Les **3** exercices sont indépendants ; donnez des réponses simples et précises : les fonctions C demandées sont très courtes (quelques lignes). Ecrire des généralités, ou copier des fonctions vues en cours ou en TD, vous fera perdre du temps sans vous rapporter aucun point.

1. Pour tout réel $x$ et tout entier positif $n$, on définit :

$$(x)_n = x(x-1)(x-2)\dots(x-n+1)$$

(noter que ce produit comporte $n$ facteurs).

Ecrire en langage C la fonction $f$ définie par :

$$f(x,n) = \sum_{k=1}^{n} \frac{(x)_k}{x^k}$$

2. On appelle *inversion*, dans un tableau d'entiers $t$, tout couple $(i,j)$ d'indices tel que :

$$i < j \text{ et } t_i > t_j$$

Ecrire une fonction C qui calcule le nombre d'inversions d'un tableau $t$ de taille $n$.

3. Le *crible d'Eratosthène* permet d'obtenir rapidement une (petite) liste de nombres premiers ; par exemple, pour calculer la liste des nombres premiers inférieurs à 100, on part d'une liste de tous les entiers inférieurs à 100, puis on *barre* tous les multiples de 2 (sauf 2), tous les multiples de 3 (sauf 3), tous les multiples de 5 (sauf 5), et tous les multiples de 7 (sauf 7) ; les "survivants" forment la liste souhaitée. Si cet algorithme ne vous est pas familier, essayez-le sur un petit exemple (par exemple pour obtenir la liste des nombres premiers inférieurs à 20) !

Voici une traduction en langage C :

```
#define MAX 100 int barre[MAX];

void crible (int n) {
  ...
}

void liste_premiers () {
  int n;

  for (n = 0; n < MAX; n++) barre[n] = 0;
  crible (2); crible (3); crible (5); crible (7);
  for (n = 2; n < MAX; n++)
    if (! barre[n]) printf (" %d", n);
  printf ("\n");
}
```

(a) Compléter la fonction `crible`.

(b) Si on remplace 100 par 200, dans la définition de `MAX`, comment faut-il modifier la fonction `liste_premiers` ?

(c) Ecrire une version de la fonction `liste_premiers`, qui soit correcte pour toute valeur de MAX.

Note : cette question est probablement plus difficile que les autres, bien que la réponse soit aussi courte que d'habitude.

## 10.8 Corrigé

1.
```
double f (double x, int n) {
  int k;
  double s = 0, t = 1;

  for (k = 0; k < n; k++) {
    t = t * (x - k) / x;
    s = s + t;
  }
  return s;
}
```

2. La solution la plus simple est la suivante :

```
int inversions (int *t, int n) {
  int i, j, compteur = 0;

  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if ( t[i] > t[j]) compteur++;
  return compteur;
}
```

Note : on peut aussi bien écrire la première boucle :

`for (i = 0 ; i < n - 1 ; i++)` .

La solution suivante n'est pas fausse, mais maladroite :

```
int inversions (int *t, int n) {
  int i, j, compteur = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if ( i < j && t[i] > t[j]) compteur++;
  return compteur;
}
```

3. (a) La solution la plus simple est la suivante :

```
void crible (int n) {
  int i;

  for (i = 2 * n; i < MAX; i = i + n)
    barre[i] = 1;
}
```

Note : on peut en fait commencer le crible avec `i=n*n`, car les multiples de $n$ précédents ont déjà été barrés.

Cette solution est de loin préférable à la suivante, qui effectue de nombreux calculs inutiles :

```
void crible (int n) {
  int i;

  for (i = n + 1; i < MAX; i++)
    if (i % n == 0)
      barre[i] = 1;
}
```

Attention : avec cette solution, ajouter

```
else
  barre[i] = 0;
```

est une grave erreur.

(b) Si on remplace 100 par 200, dans la définition de `MAX`, il faut ajouter :

`crible (11) ; crible (13) ;`

dans la fonction `liste_premiers`.

(c) De façon générale, il faut appeler la fonction `crible` pour tout nombre premier inférieur à la racine carrée de MAX ; soit :

```
for (n = 2; n * n < MAX; n++)
  if (! barre[n]) crible (n);
```