# Polynômes

- Représentation d'un polynôme par un tableau.
- Opérations élémentaires : somme et produit.
- Application : calcul du polynôme $(1-X)\ (1-X^2)\ ...\ (1-X^n)$.
- Evaluation d'une fonction polynomiale.

## Représentation

La représentation la plus simple du polynôme :

$$P = a_0 + a_1\ X + a_2\ X^2 + ... + a_n\ X^n$$

est un tableau $P$ de taille $n+1$ avec, pour chaque indice $i$, $P[i] = a_i$ ; l'inconvénient de cette technique est le stockage d'un coefficient nul pour chaque monôme $X^i$ ne figurant pas dans $P$ ; en particulier, comme les tableaux sont de taille fixe (sauf en utilisant l'allocation dynamique de mémoire, qui sort 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.

On supposera dans tout ce chapitre que la constante `DEGRE_MAX` est définie, par exemple :

```
#define DEGRE_MAX 100
```

C possède une instruction qui permet d'attribuer un nom à un type, ce qui rend les programmes plus élégants et faciles à comprendre :

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

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];
```

obtenue en *substituant* *u* (ou *v*) à *polynome* dans la déclaration de ce type.

La technique de la *sentinelle*, qui permet par exemple de calculer la longueur d'une chaîne de caractères, n'est pas applicable aux polynômes, car il n'existe aucun coefficient conventionnel qui ne risque pas d'être confondu avec un véritable coefficient. En particulier, un coefficient nul n'annonce pas la fin du polynôme ; pour calculer le degré, il faut trouver le *dernier* coefficient non nul :

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

### Commentaires de programmation

- La définition de l'opérateur `&&` garantit que $p[i]$ n'est pas évalué si l'indice $i$ devient négatif, ce qui est le cas lorsque $p$ est le polynôme nul ;
- Dans ce cas, la fonction retourne la valeur -1, ce qui n'est pas une erreur : le polynôme nul n'a pas de degré (sinon le degré d'un produit de polynômes ne serait pas toujours égal à la somme des degrés), et dans un tel cas la solution informatique

est de retourner une valeur qui ne puisse pas être confondue avec un véritable degré.

- Comme les réels sont représentés en machine de façon *approximative*, les erreurs d'arrondi peuvent entraîner qu'un calcul fournisse un coefficient $a_i$ très petit au lieu d'un coefficient nul. Dans ce cas, le résultat de la fonction `degre` est faux.

  Le problème disparaît si l'on manipule des polynômes à coefficients entiers, mais il faut alors modifier la définition du type `polynome`, et recompiler le programme. Ceci illustre une lacune fondamentale de C.

## Opérations élémentaires

Voici une fonction qui remplace l'affectation, car avec les déclarations `polynome u, v;` l'instruction `u = v` n'a pas de sens (voir tableaux et adresses).

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

L'ordre des arguments est choisi de telle sorte que `copier (u, v)` remplace `u = v`.

Voici une fonction qui effectue la somme de deux polynômes ; l'ordre des arguments est choisi de telle sorte que `somme (p, u, v)` remplace `p = u + v`.

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

L'opération qui distingue les polynômes des vecteurs est le produit ; le degré du produit peut dépasser le maximum autorisé pour le stockage du résultat : dans ce cas on le tronque (que faire d'autre ?).

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

  for (i = 0; i <= DEGRE_MAX; i++)
    a[i] = 0;
  for (i = 0; i <= m; i++)
    for (j = 0; j <= n && i + j <= DEGRE_MAX; j++)
      a[i + j] = a[i + j] + b[i] * c[j];
}
```

### Commentaires de programmation

- Comme les produits `b[i]*c[j]` doivent être *accumulés* dans `a[i+j]`, pour $i+j$ constant, il ne faut pas oublier d'initialiser le polynôme $a$.

- Les instructions `for (i=0; i<=degre (b); i++)` et `for (j=0; j<=degre (c); j++)` seraient dangereuses, car il est vraisemblable que le degré du polynôme serait recalculé à chaque tour de boucle, ce qui entraînerait (surtout à cause de la boucle interne) un effondrement de l'efficacité de la fonction. C'est pourquoi il faut effectuer une seule fois les calculs de degrés, et les enregistrer dans les variables $m$ et $n$.

- C possède un opérateur `+=` (et les opérateurs analogues `-=, *=, /=`) pour abréger

une instruction comme :

```
a[i + j] = a[i + j] + b[i] * c[j];
```

en :

```
a[i + j] += b[i] * c[j];
```

## Application

Calculons le produit :

$$(1-X)\,(1-X^2)\,\ldots\,(1-X^n)$$

dont Euler a montré que les $n$ premiers termes sont (pour $n$ assez grand) :

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

```
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;
  }
}
```

### Commentaires de programmation

- Le polynôme $a$ vaut successivement $1-X$, $1-X^2$, ... $1-X^n$ ; à la fin du corps de boucle, ne pas oublier l'instruction `a[i]=0`, sinon les valeurs successives de $a$ seraient $1-X$, $1-X-X^2$, ... $1-X-X^2-\ldots-X^n$.
- Ecrire `produit (b, a, b)` serait une faute catastrophique, car la fonction *produit* commence par annuler le *contenu* de son premier paramètre. Dans ce cas $b$ serait donc détruit avant le 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.

## 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 le *schéma de Hörner* :

```
double evaluer (polynome p, double x) {
  int n = degre (p);
  double y = 0;

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

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