

# Génération de coefficients de polynômes d'approximation sur des sous-intervalles, avec bornes d'erreur garanties

Vincent LEFÈVRE

Arénaire, INRIA Grenoble – Rhône-Alpes / LIP, ENS-Lyon

Journées TaMaDi, Sophia Antipolis, 2011-02-22

# Plan

- Introduction
- Approximation d'une fonction par un polynôme
- Vers des polynômes de plus petits degrés
- Passage d'un sous-intervalle au suivant
- Analyse d'erreur
- Futur

# Introduction

Contexte/but : résoudre le Dilemme du Fabricant de Tables dans un système à virgule flottante en base  $\beta$  (2 ou 10) à précision  $p$ .

## Données :

- Fonction mathématique  $f$  sur un intervalle  $I$ , e.g. de la forme  $[\beta^k, \beta^{k+1}[$  ou un sous-découpage (e.g.  $2^{13}$  sous-intervalles).
- Entrées : valeurs  $x_i$  en progression arithmétique sur  $I$  (des nombres machine du système VF, ou des entiers après changement de variable).
- Sorties : valeurs approchées de  $f(x_i)$ , avec une borne d'erreur  $\varepsilon_0$  garantie (imposée en entrée), e.g.  $\varepsilon_0 \approx 2^{-30}$  ulp.

## Notes :

- La fonction  $f$  devra être « numériquement régulière » (pas  $\sin x$  avec  $x \gg \pi$ ).
- On n'aura pas besoin de calculer toutes les valeurs  $f(x_i)$ , mais le problème reviendra au même (du point de vue de la précision).
- On n'aura besoin que des valeurs modulo leur ulp.

## Plage d'exposants de sortie

- L'intervalle  $I$  sera choisi de manière à ce que l'exposant des valeurs  $f(x_i)$  de la fonction **change peu**, et dans l'idéal il ne change pas du tout.
- On se ramènera ainsi à du calcul en **virgule fixe**, beaucoup plus efficace que la virgule flottante, surtout dans ce contexte où...
- On pourra se ramener à du **calcul modulo l'ulp**, enfin presque, car l'ulp peut changer (on peut considérer le maximum pour le calcul, et le minimum pour le test interne, cf discussion dans ma thèse).

Exposant minimum:                    mmmmmm...mmmmmmmtttt...ttttx

Exposant maximum:    mmmmmm...mmmmmmxxxxxtttt...ttttx

Question pour plus tard : compromis entre le choix de la taille de  $I$  et les variations de l'exposant de sortie ?

Le point essentiel : **virgule fixe modulaire**.

# Approximation d'une fonction par un polynôme

On va approcher la fonction  $f$  par des polynômes.

- Je montrerai que cela permet d'avoir des algorithmes rapides.
- En plus, c'est compatible avec la virgule fixe modulaire.

**Problème initial : approcher la fonction  $f$  par un polynôme  $P$  sur  $I$ ,** avec une borne d'erreur  $\varepsilon_f$  garantie, typiquement une fraction de la borne d'erreur  $\varepsilon_0$  finale.

Ce n'est pas le sujet de cet exposé  $\rightarrow$  peu de détails dans ce qui suit. On suppose que l'on dispose d'outils fournissant les coefficients de l'approximation en question (et éventuellement une meilleure borne d'erreur  $\varepsilon'_f < \varepsilon_f$ ).

On pourra faire d'autres approximations, avec une erreur bornée par  $\varepsilon_1 = \varepsilon_0 - \varepsilon'_f$  (ou  $\varepsilon_0 - \varepsilon_f$ ), par exemple :

- approcher  $P$  par des polynômes de plus petit degré,
- approcher les coefficients par des valeurs à plus faible précision,
- effectuer les opérations de manière approchée.

# Approximation d'une fonction par un polynôme : implémentation actuelle

Implémentation actuelle : formule de Taylor.

Ce n'est pas la meilleure approximation, mais :

- facile et rapide (pour une approximation générique) à calculer ;
- l'erreur peut être facilement majorée ;
- bon compromis entre précision et rapidité de calcul (?) ;
- on peut déterminer le degré du polynôme dynamiquement (rappel : la borne d'erreur est fournie en entrée).

Conséquence par rapport à une approximation minimax : le degré est plus grand. Mais un découpage en sous-intervalles permettrait de faire diminuer le degré, cf approximations hiérarchiques (plus loin).

À faire : tests pour comparer les deux types de méthodes (ou autres).

# Approximation d'une fonction par un polynôme : détermination de l'erreur

- **Erreur d'approximation.** Par exemple : troncature de la série de Taylor.  
En posant  $t = x - x_0$  :

$$f(x) = f(x_0) + \frac{f^{(1)}(x_0)}{1!}t + \frac{f^{(2)}(x_0)}{2!}t^2 + \dots + \frac{f^{(d)}(x_0)}{d!}t^d + R(x)$$

avec  $R(x) = \int_{x_0}^x \frac{f^{(d+1)}(u)}{d!}(x-u)^d du$ , donc  $|R(x)| \leq \frac{M \cdot T^{d+1}}{(d+1)!}$ , où  
 $|f^{(d+1)}(u)| \leq M$  et  $|t| \leq T$ .

- **Arrondi des coefficients.** On peut cependant considérer les coefficients comme étant exacts,
  - ▶ soit parce que leur représentation à une précision fixée a été prise en compte dans l'algorithme d'approximation,
  - ▶ soit parce qu'on les considère initialement comme étant des réels. La raison : on va faire d'autres calculs approchés sur ces coefficients, et les majorations des erreurs d'arrondi pourront tenir compte des arrondis initiaux à ce moment-là.

# Vers des polynômes de plus petits degrés

Polynôme de degré  $d$  sur  $I \rightarrow$  se ramener à des **polynômes de petit degré  $d'$**  sur intervalles  $J_n$ , pour efficacité ou par limitation (degré 1 pour le L-algorithme).

- Approximations valides sur de petits intervalles. Plus  $d'$  est petit, plus les intervalles  $J_n$  seront petits, et plus il y aura d'approximations à faire.
- Plus  $d$  est grand, plus les approximations demanderont de calculs (en supposant que le coefficient de degré  $d$  ne puisse pas être ignoré).

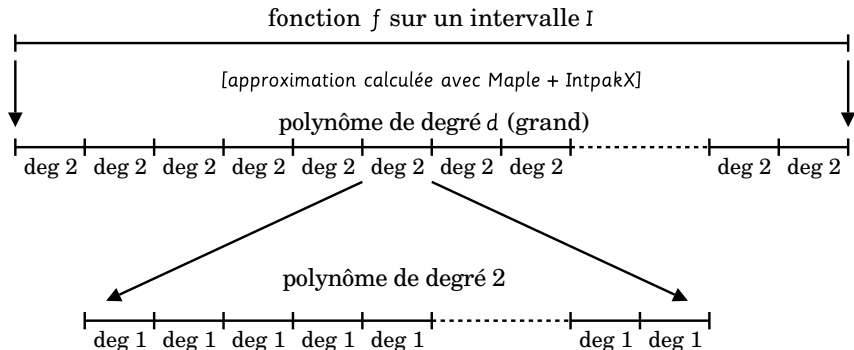
Pour déterminer rapidement les approximations :

- **Approximations hiérarchiques**, i.e. découper récursivement en sous-intervalles, et diminuer le degré  $d$  à chaque découpage.
  - ▶ Avantage : la contribution des plus grands degrés est déterminée moins souvent (pour moins de sous-intervalles).
  - ▶ Inconvénient : on introduit *a priori* des erreurs supplémentaires.
- **Passer d'un intervalle  $J_n$  au suivant**, i.e. utiliser le fait qu'on connaît une approximation dans l'intervalle précédent.



# Approximations hiérarchiques

Exemple d'implémentation ( $\sim$  implémentation actuelle) :



Généralisation : découpage en un plus grand nombre de niveaux.

# Approximations hiérarchiques : exemple d'instantiation

Exemple simplifié pour donner des ordres de grandeurs...

- Base 2, double précision ( $p = 53$ , mais nombres machine de 54 bits pour pouvoir déterminer les pires cas de la réciproque).
- $2^{13} = 8192$  intervalles  $I$  par exposant :  $2^{40}$  valeurs, degré  $d = 6$  pour  $\exp([1, 1 + 2^{-13}])$ .
- $2^{25}$  sous-intervalles  $J$  :  $2^{15} = 32768$  valeurs, degré 2.
- Degré 2  $\rightarrow$  degré 1 sur  $J$  (en négligeant le coefficient de degré 2) pour application du L-algorithme (non décrit ici).

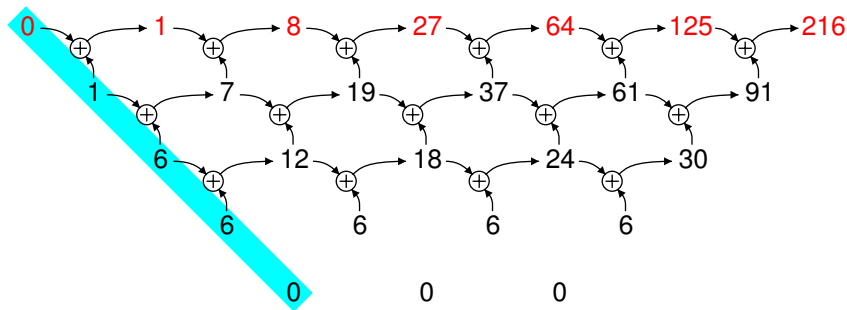
Si échec :

- ▶ découpage de  $J$  en 8 sous-intervalles  $K$  de taille 4096, degré 2 ;
- ▶ degré 2  $\rightarrow$  degré 1 sur  $K$  pour application du L-algorithme ;
- ▶ si échec, différences tabulées (cf transparent suivant) en degré 2 sur  $K$ .

Dans la réalité : certaines étapes peuvent être sautées, variantes.

# Calcul des valeurs successives d'un polynôme

Exemple:  $P(X) = X^3$ . Table des différences:



Coefficients dans la base  $\left\{ 1, X, \frac{X(X-1)}{2}, \frac{X(X-1)(X-2)}{3!}, \dots \right\}$ .

Avantages: uniquement quelques additions par itération, calcul modulo l'ulp.

# Représentation et calcul approché des coefficients

Donnée : polynôme  $Q(X) = \sum_{i=0}^{\delta} a_i \cdot \binom{X}{i}$  de degré  $\delta$ .

Les coefficients  $a_i$  seront représentés par des éléments  $\hat{a}_i \in u_i \mathbb{Z}$  où  $u_i = \beta^{-p_i}$ , avec une erreur initiale inférieure à  $u_i$ , et où la suite  $(p_i)$  des précisions est croissante.

À chaque itération :

$$\begin{cases} \hat{a}_0 & = & \hat{a}_0 & + & o(\hat{a}_1) \\ \hat{a}_1 & = & \hat{a}_1 & + & o(\hat{a}_2) \\ & & \vdots & & \\ \hat{a}_{\delta-1} & = & \hat{a}_{\delta-1} & + & o(a_{\delta}) \end{cases}$$

Coefficients tronqués  $\rightarrow$  accumulation d'erreurs  $< u_i$  sur  $a_i$ .

Note : dans la pratique, on peut considérer que les coefficients  $a_i$  initiaux sont exacts (cf transparent 7). Donc on a une borne d'erreur initiale  $\epsilon_i(0)$  sur  $a_i$  :

$$\begin{cases} \epsilon_i(0) & = & u_i & \text{pour tout } 0 \leq i < \delta, \\ \epsilon_{\delta}(0) & = & 0 & \text{(arrondi pris en compte lors des additions).} \end{cases}$$

# Passage d'un sous-intervalle au suivant

**Problème :** passer très rapidement d'un sous-intervalle au suivant, i.e. mettre à jour l'approximation polynomiale.

Deux méthodes (idées de départ décrites dans ma thèse) :

- 1 Mise à jour des coefficients en tenant compte des calculs qui n'ont pas été effectués.
- 2 Utiliser le fait que les intervalles sont de même longueur, i.e. leur origine est en progression arithmétique.  
→ Problème similaire au calcul des valeurs successives d'un polynôme, les coefficients étant vus comme des polynômes.

# Méthode 1 : mise à jour des coefficients

De manière générale, cela revient à déterminer les coefficients des polynômes  $P_{n+1}(X) = P_n(X + k)$ , où  $k$  est la taille du sous-intervalle, puis à négliger les coefficients de degré  $> d'$ .

Avec la base des

$$\binom{X}{i} = \frac{X(X-1)(X-2)\dots(X-i+1)}{i!}$$

(est-ce ici le meilleur choix ?), on simule la méthode à base de table des différences.

En notant les coefficients initiaux  $a_i$  ( $0 \leq i \leq d$ ), il suffit d'ajouter  $\binom{k}{i} a_j$  à  $a_{j-i}$  pour  $1 \leq i \leq j \leq d$ .

S'ils sont connus et exacts, utiliser les coefficients finaux.  $\rightarrow j > d'$ .

Ici,  $k$  est constant.  $\rightarrow$  Multiplications par des constantes entières.

# Application de la méthode 1

Cette méthode n'est utilisée qu'à un seul endroit dans mon code : pour passer d'un intervalle  $K$  au suivant (ajout du 2003-02-13).

- Lorsque l'algo rapide a réussi ( $\text{LOGL} = \log_2 \#K$  ci-dessous) :  
     $a0 += a1 \ll \text{LOGL};$   
     $a1 += a2 \ll \text{LOGL};$   
     $a0 += (a2 \ll (2 * \text{LOGL} - 1)) - (a2 \ll (\text{LOGL} - 1));$
- En cas d'échec, les coefficients ont été mis à jour par la méthode naïve (cf note sur les coefficients finaux, avec  $d' = d$ ).

Dans mon code, les trois coefficients ont la même précision (64 bits), donc le calcul est exact (aucune erreur d'arrondi).

Pour passer d'un intervalle  $J$  au suivant, j'utilise la méthode 2...

## Méthode 2 : coefficients vus comme des polynômes

**Idee générale :** Les coefficients (initiaux) des polynômes de degré  $d'$  sont vus comme les valeurs successives de polynômes.

→ Chaque coefficient s'obtient avec la méthode à base de table de différences.

### Notations et définitions :

- Polynôme  $P$  de degré  $d$ .
- Dans chaque sous-intervalle  $J_n$  : polynôme  $P_n$  de degré  $d$ .

$$P_n(m) = P(kn + m) = \sum_{i=0}^d a_i(n) \cdot \binom{m}{i}$$

où  $0 \leq m < k$ .

- $P_n$  est approché par  $P'_n$  de degré  $d'$  :

$$P'_n(X) = \sum_{i=0}^{d'} a_i(n) \cdot \binom{X}{i}.$$



## Méthode 2 : coefficients vus comme des polynômes [2]

On note  $\Delta P$  le polynôme tel que  $\Delta P(X) = P(X+1) - P(X)$ .  
 $\Delta^i P$  est un polynôme de degré  $d - i$ .

On cherche à calculer les coefficients:  $a_i(n) = \Delta^i P_n(0) = \Delta^i P(kn)$ .  
→ Les  $a_i$  sont des polynômes en  $n$  de degré  $d - i$ .

Les coefficients de ces polynômes  $a_i(n+X)$  à l'itération  $n$  dans la base des  $\binom{X}{j}$  sont:  $a_{i,j}(n) = \Delta^j a_i(n)$ . On cherche:  $a_i(n) = a_{i,0}(n)$ .

On les calcule lors du passage du sous-intervalle  $J_n$  au suivant  $J_{n+1}$  avec :

$$a_{i,j}(n+1) = a_{i,j}(n) + a_{i,j+1}(n) \quad \text{pour} \quad \begin{cases} 0 \leq i \leq d' \\ 0 \leq j \leq d - i - 1 \end{cases}$$

( $a_{i,d-i}$  étant constant). On obtient ainsi les  $a_i(n+1)$ .

Initialisation : calcul des  $a_{i,j}(0)$ .

## Application de la méthode 2

**Problème :** pour chaque sous-intervalle, calcul des coefficients des polynômes de degré 2 approchant un polynôme de degré  $d = 6$  pour  $\exp([1, 1 + 2^{-13}])$ .

3 coefficients pour  $P'_n$ :  $a_0$  (degré 6),  $a_1$  (degré 5),  $a_2$  (degré 4).

Le code généré est prévu pour tourner sur machines 32 bits et 64 bits (grâce au préprocesseur). Les  $p_i$  sont donc déterminés pour être des multiples de 32 :

$$\left\{ \begin{array}{l} p_0 = 4 \times 32, \\ p_1 = 5 \times 32, \\ p_2 = 6 \times 32, \\ p_3 = 7 \times 32, \\ p_4 = 9 \times 32, \\ p_5 = 10 \times 32, \\ p_6 = 10 \times 32. \end{array} \right.$$

Les transparents suivants donnent les valeurs initiales (pour machines 64 bits, mots stockés en *little endian*, modulo 1).

# Application de la méthode 2 : coefficient $a_0$ (degré 6)

Code :

```
uint64_t a0_0[] = { 0x3C762E7160F38B4E, 0xA6ABF7160809CF4F };
uint64_t a0_1[] = { 0xABFCA8C900000000, 0x9CA0E833FEB6CB85, 0x5458A4173B436123 };
uint64_t a0_2[] = { 0x354AA25BAAB8404F, 0xCCAFB049B66C0BEA, 0x000002B7E1516295 };
uint64_t a0_3[] = { 0xD8B90AB500000000, 0x3BA51465E493E36F, 0xDF85458A6CF1C94C, 0x000000000000000A };
uint64_t a0_4[] = { 0x16FE154300000000, 0xC4D9DF953D0EDFFB, 0x2A0AC34F5D426FDA, 0x000000002B7E1516, 0x0000000000000000 };
uint64_t a0_5[] = { 0x4F1A4229E540A478, 0xE62637A70A321BD8, 0x00ADF85458A986FD, 0x0000000000000000, 0x0000000000000000 };
uint64_t a0_6[] = { 0x58809CF4F3C762E7, 0x51628AED2A6ABF71, 0x0000000000002B7E1, 0x0000000000000000, 0x0000000000000000 };
```

Mis sous forme plus lisible :

```
a0_0 = A6ABF7160809CF4F 3C762E7160F38B4E
a0_1 = 5458A4173B436123 9CA0E833FEB6CB85 ABFCA8C9
a0_2 = 000002B7E1516295 CCAF B049B66C0BEA 354AA25BAAB8404F
a0_3 = 000000000000000A DF85458A6CF1C94C 3BA51465E493E36F D8B90AB5
a0_4 = 0000000000000000 00000002B7E1516 2A0AC34F5D426FDA C4D9DF953D0EDFFB 16FE1543
a0_5 = 0000000000000000 0000000000000000 00ADF85458A986FD E62637A70A321BD8 4F1A4229E540A478
a0_6 = 0000000000000000 0000000000000000 00000000002B7E1 51628AED2A6ABF71 58809CF4F3C762E7
```

Optimisation possible : suppression de certains mots 0 de poids fort (et  $-1$  dans le cas de petits coefficients négatifs). Cela demande de déterminer (au moment de générer le code) des plages des valeurs possibles.

# Application de la méthode 2 : coefficient $a_1$ (degré 5)

Code :

```
uint64_t a1_0[] = { 0x225B715628DDCEBF, 0x5BFOA8B145769AA5 };
uint64_t a1_1[] = { 0x8A29425F00000000, 0xC520B9EFDA13A7F9,
                    0x00000000056FC2A2 };
uint64_t a1_2[] = { 0xD47E77DFB318E888, 0x0015BFOA8B14AE65,
                    0x0000000000000000 };
uint64_t a1_3[] = { 0xD9F0CD6100000000, 0x2A2C53678FA65285,
                    0x000000000000056FC, 0x0000000000000000 };
uint64_t a1_4[] = { 0xCD14C49700000000, 0x561FEAAC8725FFD3,
                    0x0000015BFOA8B150, 0x0000000000000000,
                    0x0000000000000000 };
uint64_t a1_5[] = { 0x7EE2B10139E9E78F, 0x6FC2A2C515DA54D5,
                    0x0000000000000005, 0x0000000000000000,
                    0x0000000000000000 };
```

Mis sous forme plus lisible :

```
a1_0 = 5BFOA8B145769AA5 225B715628DDCEBF
a1_1 = 00000000056FC2A2 C520B9EFDA13A7F9 8A29425F
a1_2 = 0000000000000000 0015BFOA8B14AE65 D47E77DFB318E888
a1_3 = 0000000000000000 0000000000056FC 2A2C53678FA65285 D9F0CD61
a1_4 = 0000000000000000 0000000000000000 0000015BFOA8B150 561FEAAC8725FFD3 CD14C497
a1_5 = 0000000000000000 0000000000000000 0000000000000005 6FC2A2C515DA54D5 7EE2B10139E9E78F
```

# Application de la méthode 2 : coefficient $a_2$ (degré 4)

Code :

```
uint64_t a2_0[] = { 0x85458A2BB500A728, 0x000000000000ADF };
uint64_t a2_1[] = { 0x05D02CC900000000, 0x0000002B7E151629,
                    0x0000000000000000 };
uint64_t a2_2[] = { 0xADF85458A573315C, 0x0000000000000000,
                    0x0000000000000000 };
uint64_t a2_3[] = { 0x629B3C8800000000, 0x000000002B7E151,
                    0x0000000000000000, 0x0000000000000000 };
uint64_t a2_4[] = { 0xA9AAFDC500000000, 0x00ADF85458A2BB4,
                    0x0000000000000000, 0x0000000000000000,
                    0x0000000000000000 };
```

Mis sous forme plus lisible :

```
a2_0 = 000000000000ADF 85458A2BB500A728
a2_1 = 0000000000000000 000002B7E151629 05D02CC9
a2_2 = 0000000000000000 0000000000000000 ADF85458A573315C
a2_3 = 0000000000000000 0000000000000000 000000002B7E151 629B3C88
a2_4 = 0000000000000000 0000000000000000 0000000000000000 00ADF85458A2BB4 A9AAFDC5
```

Note : encore plus de 0 à supprimer, et le mot de poids faible de  $a_{2\_4}$  (0xA9AAFDC500000000) est inutile.

## Application de la méthode 2: additions (code)

```
do
{
[...]
```

```
    next:
#if GMP_LIMB_BITS == 32
[...]
```

```
    #else
        mpn_add_n((mp_limb_t *) a0_0, (mp_limb_t *) a0_0, (mp_limb_t *) a0_1 + 1, 2);
        mpn_add_n((mp_limb_t *) a0_1, (mp_limb_t *) a0_1, (mp_limb_t *) a0_2 + 0, 3);
        mpn_add_n((mp_limb_t *) a0_2, (mp_limb_t *) a0_2, (mp_limb_t *) a0_3 + 1, 3);
        mpn_add_n((mp_limb_t *) a0_3, (mp_limb_t *) a0_3, (mp_limb_t *) a0_4 + 1, 4);
        mpn_add_n((mp_limb_t *) a0_4, (mp_limb_t *) a0_4, (mp_limb_t *) a0_5 + 0, 5);
        mpn_add_n((mp_limb_t *) a0_5, (mp_limb_t *) a0_5, (mp_limb_t *) a0_6 + 0, 5);
        mpn_add_n((mp_limb_t *) a1_0, (mp_limb_t *) a1_0, (mp_limb_t *) a1_1 + 1, 2);
        mpn_add_n((mp_limb_t *) a1_1, (mp_limb_t *) a1_1, (mp_limb_t *) a1_2 + 0, 3);
        mpn_add_n((mp_limb_t *) a1_2, (mp_limb_t *) a1_2, (mp_limb_t *) a1_3 + 1, 3);
        mpn_add_n((mp_limb_t *) a1_3, (mp_limb_t *) a1_3, (mp_limb_t *) a1_4 + 1, 4);
        mpn_add_n((mp_limb_t *) a1_4, (mp_limb_t *) a1_4, (mp_limb_t *) a1_5 + 0, 5);
        mpn_add_n((mp_limb_t *) a2_0, (mp_limb_t *) a2_0, (mp_limb_t *) a2_1 + 1, 2);
        mpn_add_n((mp_limb_t *) a2_1, (mp_limb_t *) a2_1, (mp_limb_t *) a2_2 + 0, 3);
        mpn_add_n((mp_limb_t *) a2_2, (mp_limb_t *) a2_2, (mp_limb_t *) a2_3 + 1, 3);
        mpn_add_n((mp_limb_t *) a2_3, (mp_limb_t *) a2_3, (mp_limb_t *) a2_4 + 1, 4);
    #endif
}
while (i -= K);
```

# Analyse d'erreur

On note essentiellement 3 types d'erreur :

- Approximation de la fonction  $f$  par un polynôme  $P$ .  
Termes d'erreur: reste et arrondi éventuel de chaque coefficient.  
→ Borne d'erreur  $\varepsilon_f$  ou  $\varepsilon'_f$  (calcul non abordé ici).
- Approximation du polynôme  $P$  de degré  $d$  par un polynôme  $P'_n$  de degré  $d' \leq d$ , en négligeant les coefficients de degré  $> d'$ .  
→ Erreur  $\varepsilon_p$  (transparent 25).
- Méthode à base de table de différences (y compris dans la méthode 2) :  
représentation approchée des coefficients.  
→ Erreur  $\varepsilon_c$  (transparent 26).

Le calcul de bornes sur  $\varepsilon_p$  et  $\varepsilon_c$  se fait en considérant la table des différences...

# Table des différences et bornes d'erreur

Polynôme  $Q(X) = \sum_{i=0}^{\delta} a_i \cdot \binom{X}{i}$  de degré  $\delta$ .

On considère le calcul approché des valeurs successives de ce polynôme par table des différences.

La contribution d'un coefficient  $a_i$  sur une valeur  $Q(m)$  est :  $a_i \cdot \binom{m}{i}$ .

- Si un degré  $i$  est ignoré (cas de l'approximation de  $P$  par  $P'_n$ ), l'erreur correspondante sera de  $a_i \cdot \binom{m}{i}$ .
- Si  $a_i$  est pris en compte dans le calcul, une erreur (ou borne d'erreur)  $\epsilon_i$  sur le coefficient  $a_i$  donnera une erreur (ou borne d'erreur) de la forme  $\epsilon_i \cdot \binom{m}{i}$  sur la valeur à l'itération  $m$ .

Note : ignorer un coefficient  $a_i$  revient à dire que l'erreur correspondante sur ce coefficient est de  $a_i$  (le premier point ci-dessus est donc un cas particulier du second).



# Approximation de $P$ par $P'_n$ : analyse d'erreur

Notations:  $K = \#I$  et  $k = \#J_n$  avec  $k|K$ , e.g.  $K = 2^{40}$  et  $k = 2^{15}$ .

On a :

$$\begin{aligned} |\varepsilon_p| &= |P(kn + m) - P'_n(m)| = |P_n(m) - P'_n(m)| \\ &= \left| \sum_{i=d'+1}^d a_i(n) \cdot \binom{m}{i} \right| \leq \sum_{i=d'+1}^d |a_i(n)| \cdot \binom{k-1}{i} \end{aligned}$$

avec  $a_i(n) = \Delta^i P(kn) = \sum_{j=i}^d a_j(0) \cdot \binom{kn}{j-i}$ , ce qui donne :

$$|\varepsilon_p| \leq \sum_{j=d'+1}^d |a_j(0)| \sum_{i=d'+1}^j \binom{K-k}{j-i} \binom{k-1}{i}$$

Note : majoration optimale si  $a_j(0) \geq 0$  pour tout  $j \geq d' + 1$ .

# Calcul approché des coefficients : analyse d'erreur

Soit  $\epsilon_i(m)$  une majoration de l'erreur sur  $a_i$  à l'itération  $m$ .

Majorations aux limites : 
$$\begin{cases} \epsilon_i(0) = u_i & \text{pour tout } 0 \leq i < \delta, \\ \epsilon_\delta(m) = 0 & \text{pour tout } m \geq 0. \end{cases}$$

Relation de récurrence :  $\epsilon_i(m) = \epsilon_i(m-1) + \epsilon_{i+1}(m-1) + u_i$ .

On en déduit : 
$$\epsilon_i(m) = \sum_{j=i}^{\delta-1} u_j \cdot \binom{m+1}{j-i+1}.$$

En effet, c'est vrai pour  $m=0$  et pour  $i=\delta$ , et  $\epsilon_i(m-1) + \epsilon_{i+1}(m-1) + u_i =$

$$\begin{aligned} u_i + \sum_{j=i+1}^{\delta-1} u_j \cdot \binom{m}{j-i} + \sum_{j=i}^{\delta-1} u_j \cdot \binom{m}{j-i+1} \\ = \sum_{j=i}^{\delta-1} u_j \cdot \left[ \binom{m}{j-i} + \binom{m}{j-i+1} \right] = \sum_{j=i}^{\delta-1} u_j \cdot \binom{m+1}{j-i+1} = \epsilon_i(m) \end{aligned}$$

Par conséquent :  $|\epsilon_c| \leq \max \epsilon_0(m) \leq \sum_{i=0}^{\delta-1} u_i \cdot \binom{k}{i+1}$  où  $0 \leq m \leq k-1$ .

# Choix de chaque borne d'erreur (mon implémentation)

La borne d'erreur finale  $\varepsilon_0$  est déterminée suivant :

- la plage des exposants des valeurs de  $f$  sur l'intervalle ;
- le nombre de bits testés (en paramètre, 32 par défaut).

Borne d'erreur maximale acceptable pour  $|f(x) - P(x)|$ , i.e. reste pour  $P$  à coefficients réels (arithmétique d'intervalles) :  $\varepsilon_f \leq \frac{1}{8}\varepsilon_0$ .

→ Détermination du degré  $d$ .

Borne pour les approximations suivantes :  $\varepsilon_1 = \varepsilon_0 - \varepsilon_f$ .

Borne d'erreur  $\varepsilon_p$  due à l'approximation de  $P_n$  par un polynôme  $P'_n$  de degré 2 sur  $J_n$  (coefficients sur 64 bits) :  $\varepsilon_p \leq \frac{4}{7}\varepsilon_1$ .

→ Détermination de la longueur  $k$  des sous-intervalles  $J_n$  (cf transparent 25).

## Choix de chaque borne d'erreur (mon implémentation) [2]

Bornes d'erreur dues à l'approximation des coefficients dans  $2^{-p_i}\mathbb{Z}$  telles que les coefficients  $a_0$ ,  $a_1$  et  $a_2$  soient calculés à 64 bits près...

→ Détermination des  $p_i$  ( $p_d = p_{d-1}$  car on stocke  $\circ(a_d)$  directement):

```
maple_wr "errmax := 2^(-64): err0 := 0:";
my $nj = 2;
my @n;
for ($j = 0; $j < $d; $j++)
  { maple_wr "b := binomial(np,$j+1):";
    $nj++ while (&maple_getint("errnj := b * 2^(-$nj*32):\n".
      "r := errmax &- ((3/2) * errnj): signum(r[1])") < 0);
    maple_wr "errmax := errmax &- errnj:";
    $n[$j] = $nj; warn "n$j = $nj * 32\n"; }
$n[$d] = $nj; warn "n$d = $nj * 32\n";
my @n64 = map { ($_ + 1) >> 1 } @n;
```

# Futur

- Généraliser.
- Formaliser.
- Recentrer les polynômes pour diminuer l'erreur ou doubler la taille des intervalles.
- Supprimer les calculs sur les mots de poids fort qui valent toujours 0 (pour les valeurs positives) ou  $-1$  (pour les valeurs négatives).