Approximation d'une fonction f par des polynômes pour le calcul approché des valeurs successives $f(0)$, $f(1)$, $f(2)$...

Vincent LEFÈVRE

LIP / INRIA

Groupe de travail Arénaire

14 mars 2008

Plan

- Contexte de la recherche de pires cas pour l'arrondi correct de fonctions élémentaires (en double précision).
- Algorithmes pour le calcul approché de valeurs successives d'une fonction (polynôme).
- Exemples et timings.

Le dilemme du fabricant de tables (TMD)

Données : arithmétique (virgule flottante ou fixe) à précision fixe, fonction f, nombre machine x, mode d'arrondi.

- Calcul d'une valeur approchée de $f(x)$ à ε près.
- Ce résultat peut être *a priori* très proche d'une frontière entre deux arrondis possibles : distance ε'.
- Si $\varepsilon' < \varepsilon$, impossible de garantir l'arrondi correct.

$\rightarrow$ À quelle précision m faut-il faire les calculs intermédiaires ?

Application d'un théorème de Nesterenko et Waldschmidt (1995)
$\rightarrow$ bornes de l'ordre de plusieurs millions ou milliards de chiffres.

Seule solution acceptable : **tests exhaustifs**.

Valeurs frontières

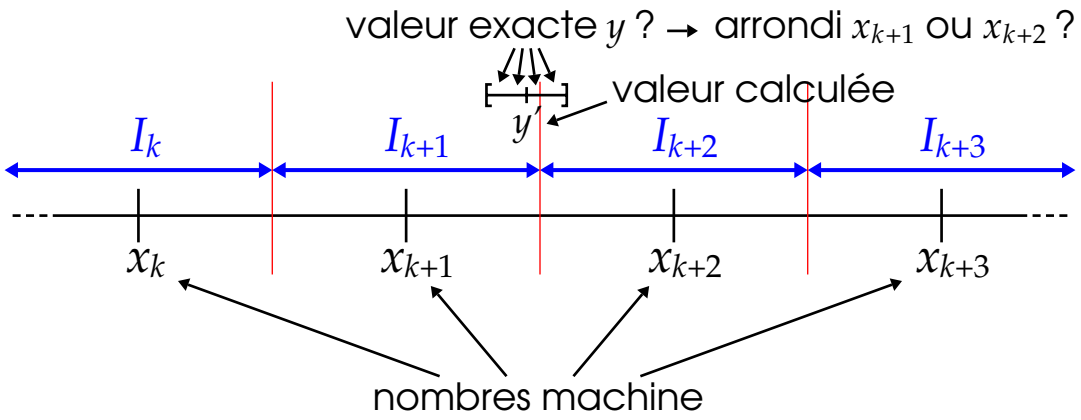
Valeurs frontières suivant le type du mode d'arrondi :

Au plus près : milieux de 2 nombres machine consécutifs.

Dirigé : nombres machine.

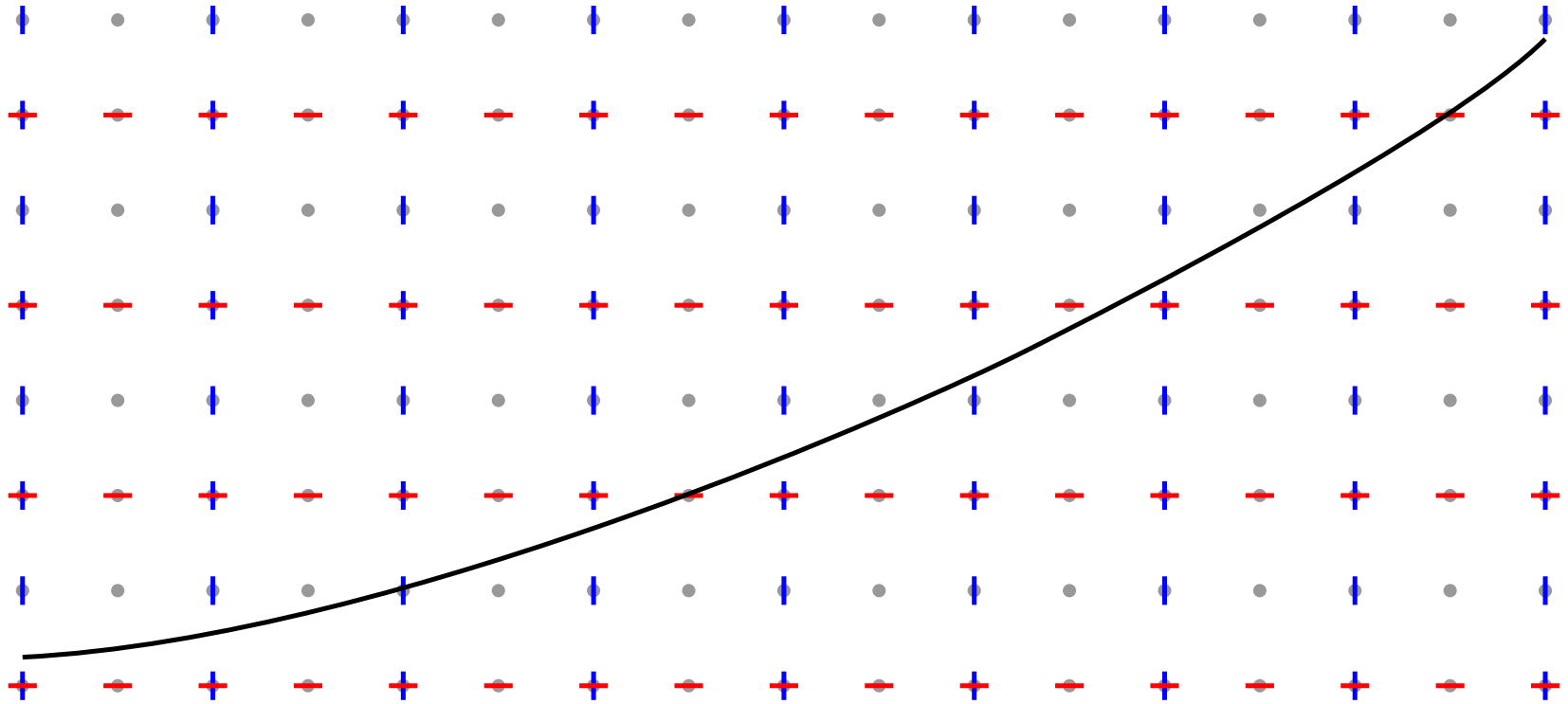
Arithmétique à virgule flottante ou fixe : valeurs frontières **en progression arithmétique**.

Exemple en arrondi au plus près :



Le TMD sur un petit domaine

Points gris : points dont les coordonnées sont des *valeurs frontières*.
Possibilité de tester f et f^{-1} en même temps.



Tests exhaustifs

À partir des formats IEEE754R 64 bits (binary64 / decimal64), un très grand nombre de valeurs à considérer (tester). Par exemple, en binaire double précision, 2^{53} par exposant.

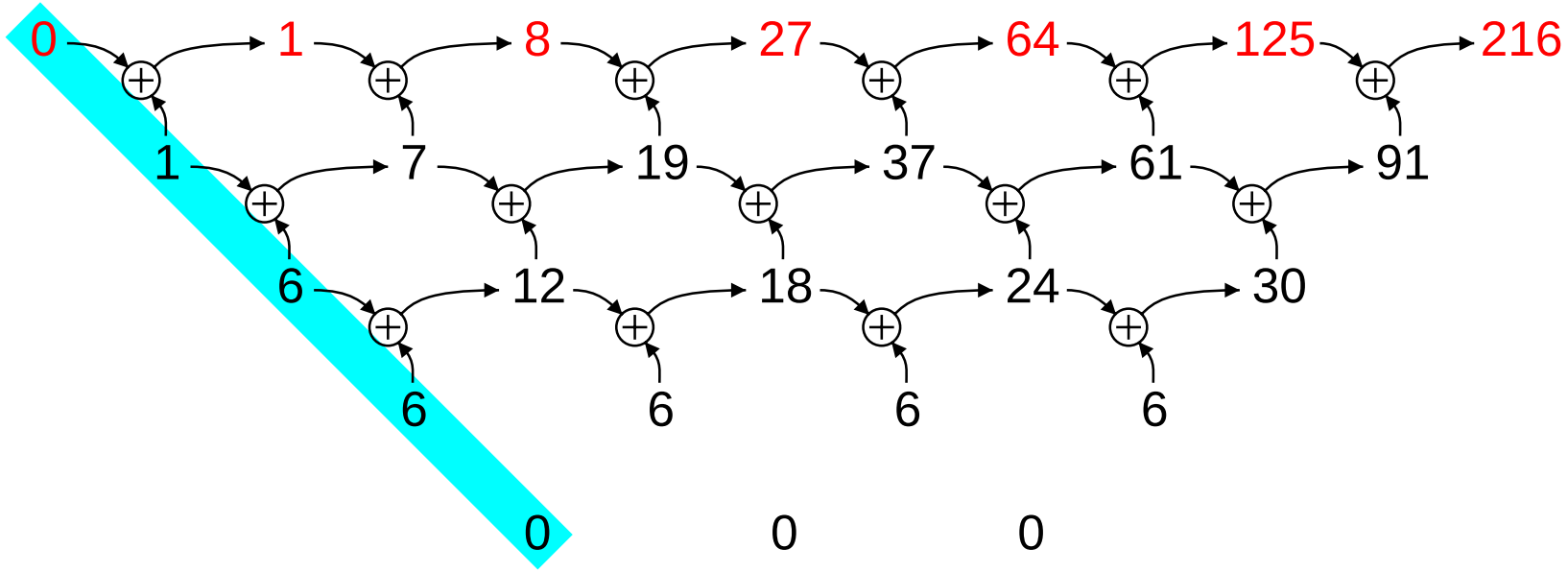
→ **Besoin d'algorithmes très rapides**.

1re idée : filtrage par calcul à faible précision (e.g., erreur de 2^{-32} ulp) pour éliminer la majorité des arguments, puis tester les arguments restants avec la méthode naïve sur une plus grande précision.

2e idée : approcher la fonction par un polynôme et exploiter le fait que les arguments successifs sont en progression arithmétique...

Calcul des valeurs successives d'un polynôme

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



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

Notes concernant cette méthode

Méthode appliquée après découpage du domaine en intervalles.

Avantages :

- Seulement d additions par valeur à calculer, où d est le degré du polynôme, pas de multiplication. $\rightarrow$ Très rapide par rapport à un calcul générique.
- Possibilité de calculs modulo l'ulp, ou même $\frac{1}{2}$ ulp (i.e. distance entre deux valeurs frontières consécutives).
- Nécessite peu de mémoire.

Inconvénient : nécessite de calculer dynamiquement une approximation polynomiale pour chaque intervalle.

Approximation d'une fonction par un polynôme

Implémentation actuelle : formule de Taylor.
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.

Exploiter le fait qu'on doit faire des approximations dans des intervalles consécutifs (et de même taille) ?
Pas possible sur une fonction quelconque (?), mais... cf plus loin.

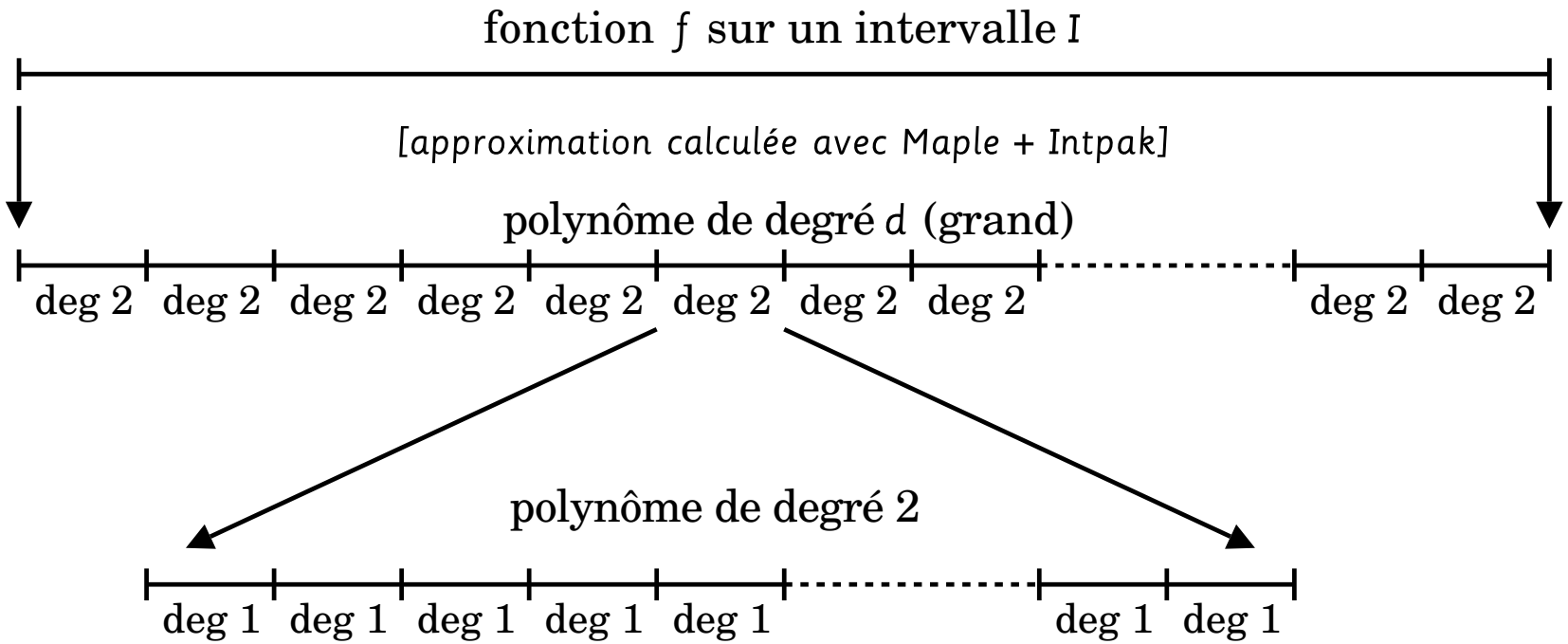
Petit vs grand degré

- Petit degré d :
  - Calcul rapide des valeurs successives (d additions) ;
  - mais l'approximation est valide dans un petit intervalle seulement : le calcul des coefficients des polynômes (pour chaque intervalle) prendrait beaucoup de temps.
- Grand degré d :
  - Le temps de calcul des approximations est négligeable ;
  - mais le calcul des valeurs successives prend plus de temps ;
  - et impossible d'appliquer des algo de filtrage spécifiques.

Avantages des deux à la fois ? $\rightarrow$ **Approximations hiérarchiques**.

Approximations hiérarchiques

Exemple d'implémentation (∼ implémentation actuelle) :



Généralisation : découpage par dichotomie ($d \to d'$ avec $d' \leqslant d$)...

Exemple (simplifié) d'instantiation

8192 intervalles I : 2^{40} valeurs, degré $d = 6$ (pour $\exp([1, 1 + 2^{-13}[)$).

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 de mon algo (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 de mon algo ;
- si échec, méthode naïve (table des différences) en degré 2 sur K.

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

Passer 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 de $P_{n+1}(X) = P_n(X + k)$, où k est la taille du sous-intervalle, puis à négliger les coefficients de degré $> d'$.

Avec la base des

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

(est-ce ici le meilleur choix ?), on simule la méthode à base de table des différences. En notant les coefficients initiaux a_i ($0 \leqslant i \leqslant d$), il suffit d'ajouter $\binom{k}{i} a_j$ à a_{j-i} pour $1 \leqslant i \leqslant j \leqslant d$.

S'ils sont connus, 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 (`LOGL` = $\log_2 \#K$ ci-dessous) :

```
a0 += a1 << LOGL;
a1 += a2 << LOGL;
a0 += (a2 << (2 * LOGL - 1)) - (a2 << (LOGL - 1));
```

- En cas d'échec, les coefficients ont été mis à jour par la méthode naïve (cf note sur les coefficients finaux, avec $d' = d$).

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

Méthode 2 : coefficients vus comme des polynômes

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

$$P_n(m) = P(kn+m) = \sum_{i=0}^{d} a_i(n) \cdot \binom{m}{i}, \text{ où } 0 \leqslant m < k.$$

- 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 (suite)

On note ΔP 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 : $a_i(n) = \Delta^i P_n(0) = \Delta^i P(kn)$.
$\rightarrow$ 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 : $a_i(n) = a_{i,0}(n)$.

C'est ce qu'on calcule avec $a_{i,j}(n+1) = a_{i,j}(n) + a_{i,j+1}(n)$, pour $0 \leqslant i \leqslant d'$ et $0 \leqslant j \leqslant d - i - 1$ ($a_{i,d-i}$ étant constant), lors du passage du sous-intervalle J_n au suivant J_{n+1}, afin d'obtenir les $a_i(n+1)$.

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

Récapitulatif et plan de ce qui suit

Avant de passer à l'application pratique de la méthode 2...

- Approximation de la fonction f par un polynôme P.
  Termes d'erreur : reste et arrondi de chaque coefficient.
  $\rightarrow$ Erreur ε_0 (calcul non abordé ici).
- Approximation du polynôme P de degré d par un polynôme P'_n de degré $d' \leqslant d$, en négligeant les coefficients de degré $> d'$.
  $\rightarrow$ Erreur ε_1 (transparent 19).
- Méthode à base de table de différences (y compris dans la méthode 2) : représentation approchée des coefficients.
  $\rightarrow$ Erreur ε_2 (transparent 21).

Analyse d'erreur : approximation de P par P'_n

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

$$|\varepsilon_1| = |P(kn+m) - P'_n(m)| = |P_n(m) - P'_n(m)|$$
$$= \left| \sum_{i=d'+1}^{d} a_i(n) \cdot \binom{m}{i} \right| \leqslant \sum_{i=d'+1}^{d} |a_i(n)| \cdot \binom{k-1}{i}$$

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

$$|\varepsilon_1| \leqslant \sum_{j=d'+1}^{d} |a_j(0)| \sum_{i=d'+1}^{j} \binom{K-k}{j-i} \binom{k-1}{i}$$

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

Représentation et calcul approché des coefficients

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

Les coefficients a_i seront représentés par des éléments $\hat{a}_i \in 2^{-n_i}\mathbb{Z}$ avec une erreur initiale inférieure à 2^{-n_i}, où (n_i) est croissante.

À chaque itération :

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

Coefficients tronqués $\rightarrow$ accumulation d'erreurs $< 2^{-n_i}$ sur a_i.

Analyse d'erreur : calcul approché des coefficients

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

Majorations aux limites : $\epsilon_i(0) = 2^{-n_i}$ pour tout $0 \leqslant i < \delta$,

et $\epsilon_\delta(m) = 0$ pour tout $m \geqslant 0$.

Relation de récurrence : $\epsilon_i(m) = \epsilon_i(m-1) + \epsilon_{i+1}(m-1) + 2^{-n_i}$.

On en déduit : $\epsilon_i(m) = \sum_{j=i}^{\delta-1} 2^{-n_j} \binom{m+1}{j-i+1}$

puis : $|\varepsilon_2| \leqslant \sum_{i=0}^{\delta-1} 2^{-n_i} \binom{k}{i+1}$ où $0 \leqslant m \leqslant k-1$.

Application de la méthode 2

Problème : pour chaque sous-intervalle, calcul des coefficients des polynômes de degré 2 approchant un polynôme de degré d.

Rappel pour $\exp([1, 1 + 2^{-13}[)$: degré $d = 6$.
3 coefficients : 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 n_i sont donc déterminés pour être des multiples de 32 : $n_0 = 4 \times 32$, $n_1 = 5 \times 32$, $n_2 = 6 \times 32$, $n_3 = 7 \times 32$, $n_4 = 9 \times 32$, $n_5 = 10 \times 32$, $n_6 = 10 \times 32$.

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

Coefficient a_0 (degré 6)

```
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,
                    0x000000000002B7E1, 0x0000000000000000,
                    0x0000000000000000 };
```

Mis sous forme plus lisible :

```
a0_0 = A6ABF7160809CF4F 3C762E7160F38B4E
a0_1 = 5458A4173B436123 9CA0E833FEB6CB85 ABFCA8C9
a0_2 = 00000 2B7E1516295 CCAFB049B66C0BEA 354AA25BAAB8404F
a0_3 = 000000000000000A DF85458A6CF1C94C 3BA51465E493E36F D8B90AB5
a0_4 = 0000000000000000 00000000 2B7E1516 2A0AC34F5D426FDA C4D9DF953D0EDFFB 16FE1543
a0_5 = 0000000000000000 0000000000000000 00ADF85458A986FD E62637A70A321BD8 4F1A4229E540A478
a0_6 = 0000000000000000 0000000000000000 000000000002B7E1 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.

Coefficient a_1 (degré 5)

```
uint64_t a1_0[] = { 0x225B715628DDCEBF, 0x5BF0A8B145769AA5 };
uint64_t a1_1[] = { 0x8A29425F00000000, 0xC520B9EFDA13A7F9,
                    0x00000000056FC2A2 };
uint64_t a1_2[] = { 0xD47E77DFB318E888, 0x0015BF0A8B14AE65,
                    0x0000000000000000 };
uint64_t a1_3[] = { 0xD9F0CD6100000000, 0x2A2C53678FA65285,
                    0x00000000000056FC, 0x0000000000000000 };
uint64_t a1_4[] = { 0xCD14C49700000000, 0x561FEAAC8725FFD3,
                    0x0000015BF0A8B150, 0x0000000000000000,
                    0x0000000000000000 };
uint64_t a1_5[] = { 0x7EE2B10139E9E78F, 0x6FC2A2C515DA54D5,
                    0x0000000000000005, 0x0000000000000000,
                    0x0000000000000000 };
```

```
a1_0 = 5BF0A8B145769AA5 225B715628DDCEBF
a1_1 = 00000000056FC2A2 C520B9EFDA13A7F9 8A29425F
a1_2 = 0000000000000000 0015BF0A8B14AE65 D47E77DFB318E888
a1_3 = 0000000000000000 00000000000056FC 2A2C53678FA65285 D9F0CD61
a1_4 = 0000000000000000 0000000000000000 0000015BF0A8B150 561FEAAC8725FFD3 CD14C497
a1_5 = 0000000000000000 0000000000000000 0000000000000005 6FC2A2C515DA54D5 7EE2B10139E9E78F
```

Coefficient a_2 (degré 4)

```
uint64_t a2_0[] = { 0x85458A2BB500A728, 0x0000000000000ADF };
uint64_t a2_1[] = { 0x05D02CC900000000, 0x0000002B7E151629,
                    0x0000000000000000 };
uint64_t a2_2[] = { 0xADF85458A573315C, 0x0000000000000000,
                    0x0000000000000000 };
uint64_t a2_3[] = { 0x629B3C8800000000, 0x0000000002B7E151,
                    0x0000000000000000, 0x0000000000000000 };
uint64_t a2_4[] = { 0xA9AAFDC500000000, 0x000ADF85458A2BB4,
                    0x0000000000000000, 0x0000000000000000,
                    0x0000000000000000 };
```

```
a2_0 = 0000000000000ADF 85458A2BB500A728
a2_1 = 0000000000000000 000000B7E151629 05D02CC9
a2_2 = 0000000000000000 0000000000000000 ADF85458A573315C
a2_3 = 0000000000000000 0000000000000000 00000000B7E151 629B3C88
a2_4 = 0000000000000000 0000000000000000 0000000000000000 000ADF85458A2BB4 A9AAFDC5
```

Note : encore plus de 0 à supprimer, et le mot de poids faible de `a2_4` (`0xA9AAFDC500000000`) est inutile.

```
  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);
```

Choix de chaque borne d'erreur

La borne d'erreur finale ε 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) : $\varepsilon_0 \leqslant \frac{1}{8}\varepsilon$.
$\rightarrow$ Détermination du degré d.

Borne pour les approximations suivantes : $\varepsilon_P = \varepsilon - \varepsilon_0$.

Borne d'erreur ε_1 due à l'approximation de $\hat{P}_n$ par un polynôme P'_n de degré 2 sur J_n (coefficients sur 64 bits) : $\varepsilon_1 \leqslant \frac{4}{7}\varepsilon_P$.
$\rightarrow$ Détermination de la longueur des sous-intervalles J_n.

Choix de chaque borne d'erreur (suite)

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

$\rightarrow$ Détermination des n_i ($n_d = n_{d-1}$ 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;
```

Note : marge d'erreur $\varepsilon_P - \varepsilon_1$ inutilisée ?

Timings

Test du code actuel, temps estimé de passage d'un sous-intervalle J au suivant. Idem en supprimant des mots nuls de poids fort (patch fait à la main, non garanti, mais donnant les mêmes résultats).

Fonctions et intervalles testés (double précision) :
e^x sur $[1, 1 + 2^{-13}[$ et x^{345} sur $[1 + 1234 \times 2^{-13}, 1 + 1235 \times 2^{-13}[$.

CPU	e^x		x^{345}	
	actuel	optimisé	actuel	optimisé
Opteron	11.9 (4.5)	12.1 (4.6)	218.7 (17.7)	214.1 (12.9)
Pentium D	18.2 (9.3)	15.5 (6.2)	307.7 (25.8)	300.3 (18.8)

Les diverses arithmétiques utilisées

Dans Maple (pour la génération de code C) :

- arithmétique entière (multiprécision) ;
- arithmétique rationnelle ;
- arithmétique d'intervalles (→ intpakX).

Dans le code C généré :

- arithmétique entière modulaire multiprécision (→ MPN (GMP)) ;
- arithmétique entière modulaire 64 bits (→ `uint64_t`) ;
- arithmétique entière 32 bits (→ `uint_fast32_t`).

Ailleurs :

- arithmétique entière multiprécision (→ GMP) ;
- arith. flottante multiprécision avec arrondi correct (→ MPFR).

Conclusion

Calcul approché des valeurs successives d'une fonction : rapide, mais on peut faire mieux.

Dans l'immédiat :

- Suppression des mots de poids fort inutiles, i.e. certains 0 pour les valeurs positives, certains -1 pour les valeurs négatives (vues en complément à 2).
  Note : certains mots doivent être gardés pour la propagation de la retenue éventuelle.
- Le code assembleur montre des `call __gmpn_add_n`.
  → Essayer de les *inliner*, voire de générer du code assembleur, en prenant en compte le fait que la taille est connue.

À long terme :

- Beaucoup de paramètres et de manières de sélectionner les algorithmes.
- Majorations d'erreur : la principale difficulté est de ne rien oublier. Mais aussi comment répartir les bornes d'erreur ?
- Besoin d'outils pour :
  - prouver (vérifier) automatiquement les majorations d'erreur,
  - générer du code portable,
  - faire des mesures.