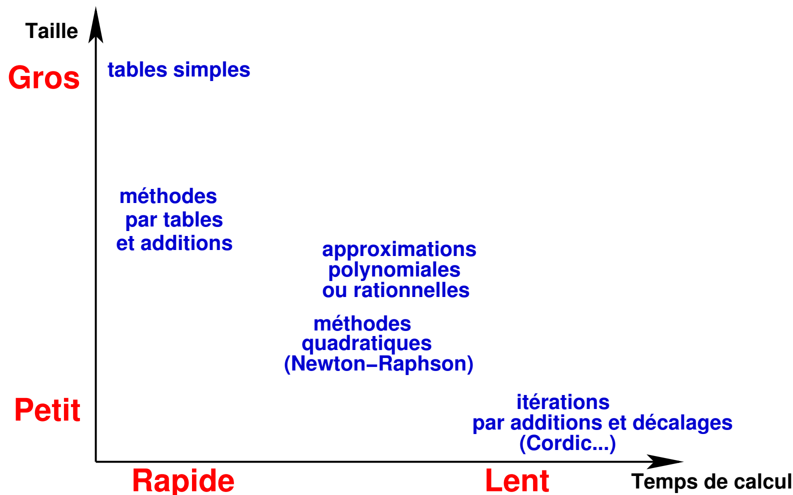
Évaluation des fonctions élémentaires

Florent de Dinechin
CITI/Socrates
(sur des travaux dans le projet **AriC**)

En guise de plan



Trois bouquins

J.-M. Muller.
Elementary Functions, Algorithms and Implementation.
Birkhäuser Boston, MA, 2nd edition, 2006.

P. Markstein.
IA-64 and Elementary Functions : Speed and Precision.
Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.

M. Cornea, J. Harrison, and P. T. P. Tang.
Scientific Computing on Itanium®-based Systems.
Intel Press, Hillsboro, OR, 2002.

Seul le Muller couvre aussi bien hard que soft
(en restant au niveau algorithmique)

Fonctions élémentaires, ou pas

Fonctions élémentaires, ou pas

Méthodes à base de table

Approximation polynomiale en virgule fixe

Sinus et cosinus en virgule fixe

Une fonction flottante en détail : le logarithme

Fonction ?

Fonction continue et dérivable (par morceaux) d'une variable réelle

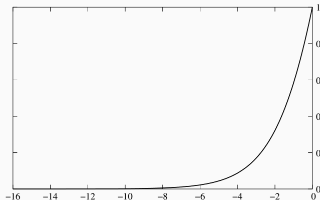
- Fonction algébrique : polynôme, $\sqrt[3]{x}$, $\frac{1}{\sqrt{x}}$, ...
- Fonction élémentaire : sinus, exponentielle, logarithme
- Autre fonction ayant sa place dans une bibliothèque mathématique : erf, Gamma, ...
- Composée, toute fonction utile

Fonction ?

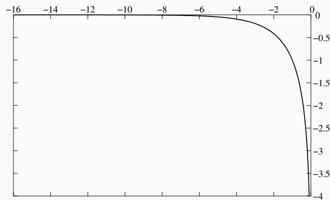
Fonction continue et dérivable (par morceaux) d'une variable réelle

- Fonction algébrique : polynôme, $\sqrt[3]{x}$, $\frac{1}{\sqrt{x}}$, ...
- Fonction élémentaire : sinus, exponentielle, logarithme
- Autre fonction ayant sa place dans une bibliothèque mathématique : erf, Gamma, ...
- Composée, toute fonction utile

Exemple : les fonctions de l'arithmétique logarithmique



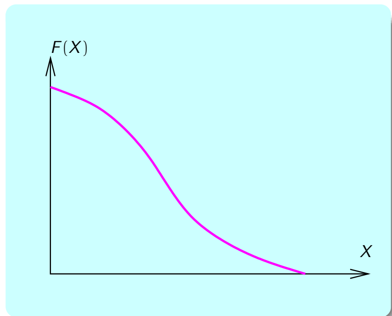
$f_{\oplus}(r) = \log_2(1 + 2^r)$

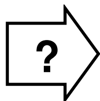


$f_{\ominus}(r) = \log_2(1 - 2^r)$

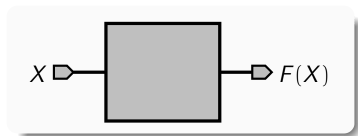
D'une fonction à un opérateur

Une fonction



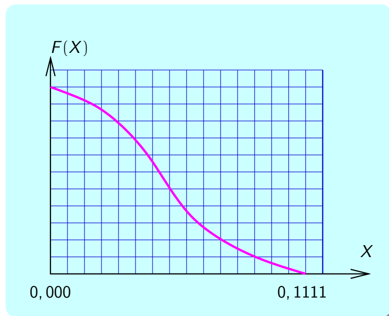


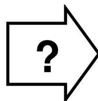
Un opérateur qui l'évalue



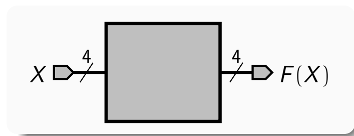
D'une fonction à un opérateur

Une fonction





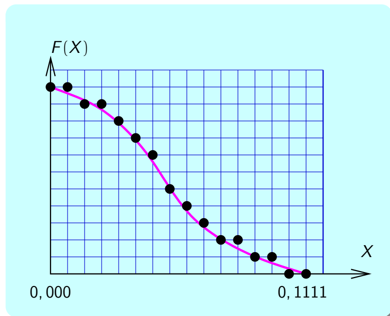
Un opérateur qui l'évalue

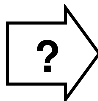


Représentation finie des nombres en entrée et en sortie

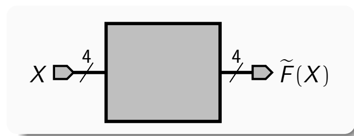
D'une fonction à un opérateur

Une fonction





Un opérateur qui l'évalue



Représentation finie des nombres en entrée et en sortie

$\Longrightarrow$ le plus souvent, fonction approchée

Accuracy versus precision, en logiciel et en matériel

En logiciel

- Le résultat sera un mot de 32 ou 64 bits *(precision)*
- On peut estimer que c'est trop précis pour l'application, et implémenter une *accuracy* inférieure

Accuracy versus precision, en logiciel et en matériel

En logiciel

- Le résultat sera un mot de 32 ou 64 bits *(precision)*
- On peut estimer que c'est trop précis pour l'application, et implémenter une *accuracy* inférieure

En matériel, l'*accuracy* doit correspondre à la *precision*

- Si vous calculez plus précis que votre format de sortie, personne ne le remarque, mais vous le payez.
- Si vous calculez moins précis, vous avez en sortie des fils qui portent du bruit, et vous les payez...

$\longrightarrow$ target accuracy : poids du bit de poids faible de la sortie

Que faut-il pour implémenter une fonction ?

Que faut-il pour implémenter une fonction ?

- Continuité, dérivabilité à un certain ordre
  - Théorème de Taylor
  - Plus généralement, approximation polynomiale

Que faut-il pour implémenter une fonction ?

- Continuité, dérivabilité à un certain ordre
  - Théorème de Taylor
  - Plus généralement, approximation polynomiale

Briques de base de l'arithmétique des ordinateurs :
opérateurs $+$, $\times$, et mémoire

Que faut-il pour implémenter une fonction ?

- Continuité, dérivabilité à un certain ordre
  - Théorème de Taylor
  - Plus généralement, approximation polynomiale

Briques de base de l'arithmétique des ordinateurs :
opérateurs $+$, $\times$, et mémoire

- Identités mathématiques spécifiques
  - parité/imparité,
  - périodicité,
  - propriétés algébriques comme $\log(ab) = \log(a) + \log(b)$...

Que faut-il pour implémenter une fonction ?

- Continuité, dérivabilité à un certain ordre
  - Théorème de Taylor
  - Plus généralement, approximation polynomiale

Briques de base de l'arithmétique des ordinateurs :
opérateurs $+$, $\times$, et mémoire

- Identités mathématiques spécifiques
  - parité/imparité,
  - périodicité,
  - propriétés algébriques comme $\log(ab) = \log(a) + \log(b)$...

Exploitées pour

- des réductions d'argument
- des itérations spécifiques (exemple : CORDIC)

Qualité numérique du résultat

Deux sources d'erreurs qui cumulent leurs effets :

Qualité numérique du résultat

Deux sources d'erreurs qui cumulent leurs effets :

- erreurs d'approximation
  - approximation d'une fonction par un polynôme
  - suite qui converge vers une valeur
  - ...

Qualité numérique du résultat

Deux sources d'erreurs qui cumulent leurs effets :

- erreurs d'approximation
  - approximation d'une fonction par un polynôme
  - suite qui converge vers une valeur
  - ...
- erreurs d'arrondi
  - dans la plupart des opérations
  - (discrétisation des résultats intermédiaires)

Qualité numérique du résultat

Deux sources d'erreurs qui cumulent leurs effets :

- erreurs d'approximation
  - approximation d'une fonction par un polynôme
  - suite qui converge vers une valeur
  - ...
- erreurs d'arrondi
  - dans la plupart des opérations
  - (discrétisation des résultats intermédiaires)

On peut toujours réduire ces erreurs en calculant pour plus cher.

Qualité numérique du résultat

Deux sources d'erreurs qui cumulent leurs effets :

- erreurs d'approximation
  - approximation d'une fonction par un polynôme
  - suite qui converge vers une valeur
  - ...
- erreurs d'arrondi
  - dans la plupart des opérations
  - (discrétisation des résultats intermédiaires)

On peut toujours réduire ces erreurs en calculant pour plus cher.

Compromis précision / performance

Objectif : calculer au plus juste

Remarque importante

Dans toute la suite, je considère les entrées exactes.

- cela rend l'analyse plus simple (pas de $x \pm \varepsilon$)
- Dans une application, elles ne sont pas exactes, mais on gère par *diviser pour régner*

Et si vous savez que vos entrées sont franchement inexactes

- vous transportez des bits qui ne contiennent pas d'information
- vous calculez dessus

... il faut peut-être se poser des questions

Virgule fixe ou virgule flottante ?

Le format virgule flottante est un format logarithmique

Virgule fixe ou virgule flottante ?

Le format virgule flottante est un format logarithmique

- une exponentielle transforme un nombre fixe en un nombre flottant
  - See next slide
  - Considérons $e^{\sum_{i=0}^{N} a_i}$

Virgule fixe ou virgule flottante ?

Le format virgule flottante est un format logarithmique

- une exponentielle transforme un nombre fixe en un nombre flottant
  - See next slide
  - Considérons $e^{\sum_{i=0}^{N} a_i}$
- un logarithme, c'est le contraire

Virgule fixe ou virgule flottante ?

Le format virgule flottante est un format logarithmique

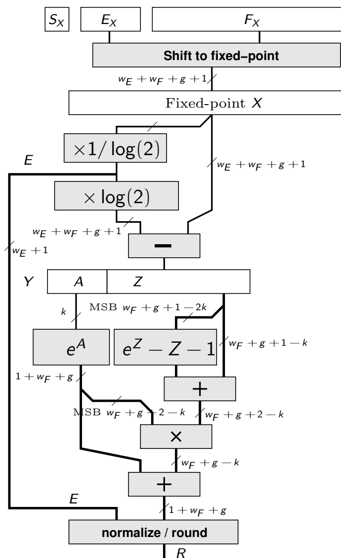
- une exponentielle transforme un nombre fixe en un nombre flottant
  - See next slide
  - Considérons $e^{\sum_{i=0}^{N} a_i}$
- un logarithme, c'est le contraire
- sin/cos, sur une période, sont des fonctions naturellement en virgule fixe

Virgule fixe ou virgule flottante ?

Le format virgule flottante est un format logarithmique

- une exponentielle transforme un nombre fixe en un nombre flottant
  - See next slide
  - Considérons $e^{\sum_{i=0}^{N} a_i}$
- un logarithme, c'est le contraire
- sin/cos, sur une période, sont des fonctions naturellement en virgule fixe
- atan(x/y) ?

Une architecture pour l'exponentielle flottante

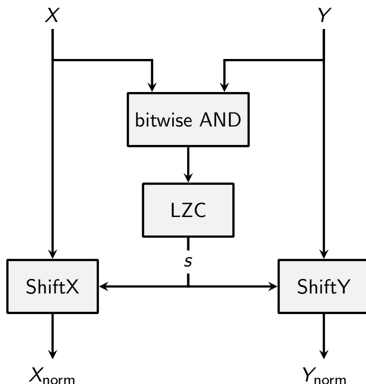


Réduction d'argument pour atan(x/y)

Ici, il faut que je lance des gnuplot.

Réduction d'argument pour atan(x/y)

Ici, il faut que je lance des gnuplot.



Matériel ou logiciel ?

G. Paul and M. W. Wilson.
Should the elementary functions be incorporated into computer instruction sets ?
ACM Transactions on Mathematical Software, 2(2) :132–142, June 1976.

- Sa réponse est : oui

Matériel ou logiciel ?

G. Paul and M. W. Wilson.
Should the elementary functions be incorporated into computer instruction sets ?
ACM Transactions on Mathematical Software, 2(2) :132–142, June 1976.

- Sa réponse est : oui
- L'histoire lui a donné tort

Matériel ou logiciel ?

G. Paul and M. W. Wilson.
Should the elementary functions be incorporated into computer instruction sets ?
ACM Transactions on Mathematical Software, 2(2) :132–142, June 1976.

- Sa réponse est : oui
- L'histoire lui a donné tort
  - Votre Pentium offre des instructions pour exp, log, sin, ...
  - Elles sont cablés en dur en microcode
  - Je sais en écrire de plus rapides et plus précises en soft
  - C'est vraiment du gâchis de silicium...
- Mais à l'époque
  - cela permettait un OS plus compact
  - cela garantissait une certaine qualité numérique

Comment est-ce possible d'aller plus vite en soft ?

C'est parce que le PC a profondément changé depuis 1976 ou 1986 (introduction du 8087)

- La mémoire est passée du Ko au Go
  - on se permet de nos jours de précalculer de grosses tables
- Le processeur est devenu superscalaire
  - Il peut faire plusieurs $+$ et $\times$ en parallèle,
  - parce que c'est généralement utile
  - (plus que des sinus ou des exp)

Comment est-ce possible d'aller plus vite en soft ?

C'est parce que le PC a profondément changé depuis 1976 ou 1986 (introduction du 8087)

- La mémoire est passée du Ko au Go
  - on se permet de nos jours de précalculer de grosses tables
- Le processeur est devenu superscalaire
  - Il peut faire plusieurs $+$ et $\times$ en parallèle,
  - parce que c'est généralement utile
  - (plus que des sinus ou des exp)

Remarque : Retour de bâton à partir des années 2000 :

- grand écart de performance entre mémoire et CPU
- et grand écart de consommation

Mémoriser est toujours bon marché, mais pas forcément pertinent.

Méthodes à base de table

Fonctions élémentaires, ou pas

Méthodes à base de table

Approximation polynomiale en virgule fixe

Sinus et cosinus en virgule fixe

Une fonction flottante en détail : le logarithme

Introduction : La leçon (E. Ionesco)

LE PROFESSEUR

Écoutez-moi, Mademoiselle, si vous n'arrivez pas à comprendre profondément ces principes, ces archétypes arithmétiques, vous n'arriverez jamais à faire correctement un travail de polytechnicien. (...) à calculer mentalement combien font(...), par exemple, trois milliards sept cent cinquante-cinq millions neuf cent quatre-vingt-dix-huit mille deux cent cinquante et un, multiplié par cinq milliards cent soixante-deux millions trois cent trois mille cinq cent huit ?

L'ÉLÈVE, *très vite.*

Ça fait dix-neuf quintillions trois cent quatre-vingt-dix quadrillions deux trillions huit cent quarante-quatre milliards deux cent dix-neuf millions cent soixante-quatre mille cinq cent huit...

(...)

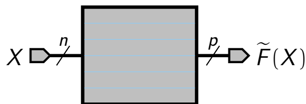
LE PROFESSEUR, *stupéfait.*

Mais comment le savez-vous, si vous ne connaissez pas les principes du raisonnement arithmétique ?

L'ÉLÈVE

C'est simple. Ne pouvant me fier à mon raisonnement, j'ai appris par cœur tous les résultats possibles de toutes les multiplications possibles.

Tabulation simple



Coût en matériel

- surface : $O(p \cdot 2^n)$
  (2^n entrées de p bits)
- délai : $O(n)$

Coût en logiciel

- taille mémoire $p \cdot 2^n$ bits
- délai : cela dépend (caches etc)

Coût en FPGA (e.g avec des LUT5)

- ressources : $p \cdot 2^{n-5}$ LUT5, ou des blockRAM...
- délai : proportionnel à $n - 5$, élémentaire si $n = 5$

Envisageable uniquement si n est petit.

Approximation bipartite

- technique inventée en 1985 pour sinus/cosinus par Sunderland *et al* (des traiteurs de signaux)

Approximation bipartite

- technique inventée en 1985 pour sinus/cosinus par Sunderland *et al* (des traiteurs de signaux)
- redécouverte en 1995 pour $1/x$ par Matula et al (des informaticiens)

Approximation bipartite

- technique inventée en 1985 pour sinus/cosinus par Sunderland *et al* (des traiteurs de signaux)
- redécouverte en 1995 pour $1/x$ par Matula et al (des informaticiens)
  Il faut dire que le titre de l'article de Sunderland est :
  CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications

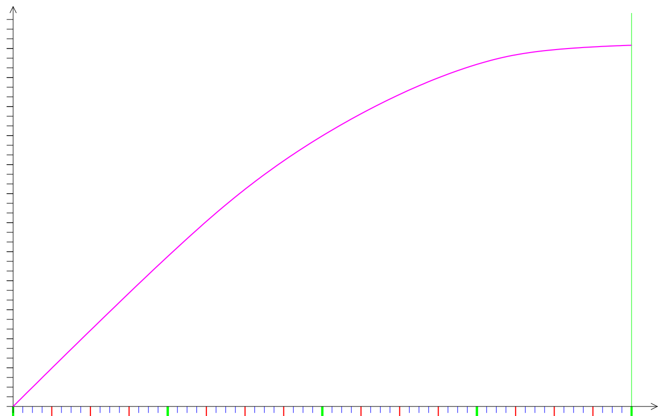
Approximation bipartite

- technique inventée en 1985 pour sinus/cosinus par Sunderland *et al* (des traiteurs de signaux)
- redécouverte en 1995 pour $1/x$ par Matula et al (des informaticiens)
  Il faut dire que le titre de l'article de Sunderland est :
  CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications
- Schulte et Stine remarquent en 1998 que la méthode est générale

Approximation bipartite

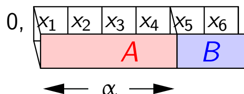
$$X = \sum_{i=1}^{n} 2^{-i} x_i$$

0, | x_1 | x_2 | x_3 | x_4 | x_5 | x_6 |
|---|---|---|---|---|---|



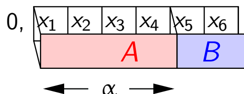
Approximation bipartite

$$X = A + 2^{-\alpha} B$$



Approximation bipartite

$$X = A + 2^{-\alpha} B$$

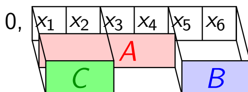


Taylor à l'ordre 1 en les points + :

$$f(X) \approx f(A) + f'(A) \times 2^{-\alpha} B$$

Approximation bipartite

$$X \;=\; A + 2^{-\alpha} B$$

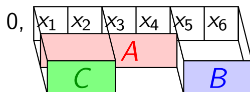


Taylor à l'ordre 1 en les points + :

$f(X) \;\approx\; f(A) \;+\; f'(C) \times 2^{-\alpha} B$

Approximation bipartite

$$X = A + 2^{-\alpha} B$$

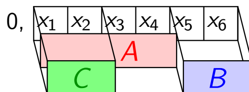


Taylor à l'ordre 1 en les points + :

$$f(X) \approx TVI(A) + s(C) \times 2^{-\alpha} B$$

Approximation bipartite

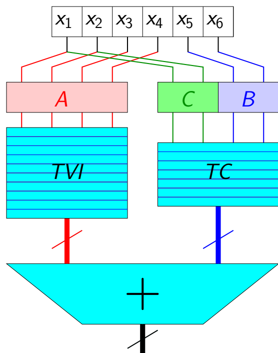
$$X \;=\; A + 2^{-\alpha} B$$



Taylor à l'ordre 1 en les points + :

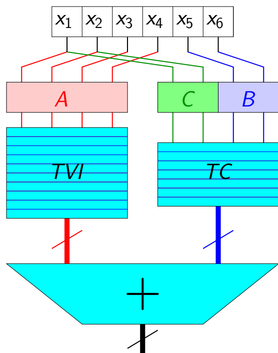
$f(X) \;\approx\; TVI(A) \;+\; TC(C, B)$

Architecture bipartite



$$f(X) \approx TVI(A) + TC(C, B)$$

Architecture bipartite



$n = 3k$

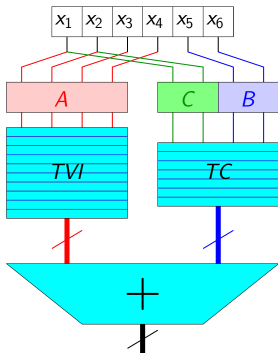
Architecture bipartite

- deux tables, chacune adressée par $2k$ bits
- une addition finale

(Table simple adressée par $3k$ bits)

$$f(X) \approx TVI(A) + TC(C, B)$$

Architecture bipartite



$f(X) \approx TVI(A) + TC(C, B)$

$n = 3k$

Architecture bipartite

- deux tables, chacune adressée par $2k$ bits
- une addition finale

(Table simple adressée par $3k$ bits)

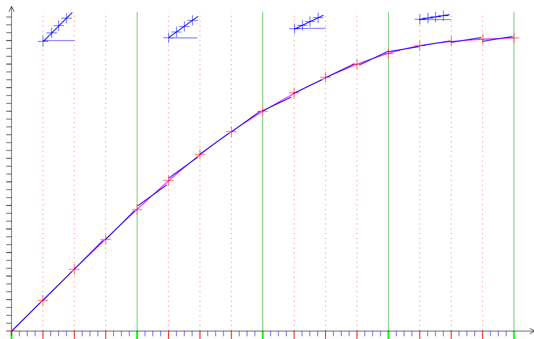
2×2^{2k} entrées au lieu de 2^{3k}

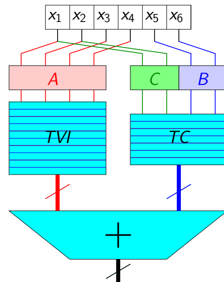
Gain d'un facteur 2^{k-1} :

- $n = 6$ bits : facteur 2
- $n = 12$ bits : facteur 8
- $n = 24$ bits : facteur 128

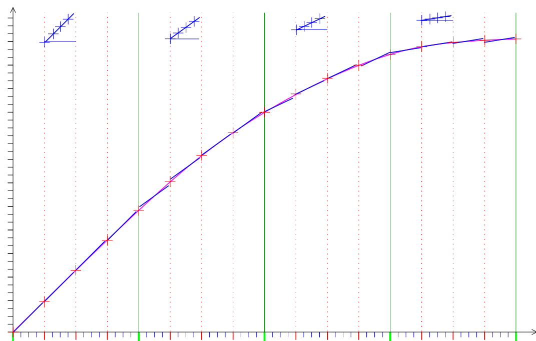
Cela vaut bien une addition.

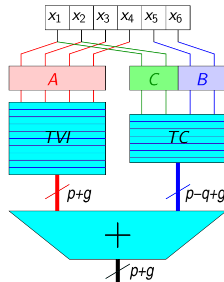
Qualité numérique du résultat





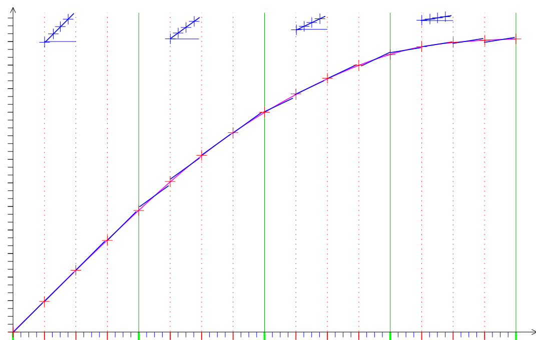
Qualité numérique du résultat

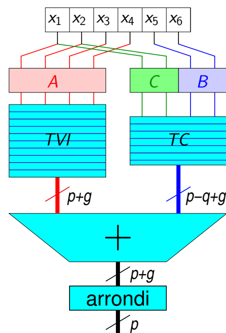




- Précision intermédiaire $p + g$
  - erreurs d'arrondi au remplissage des tables
  - rendues arbitrairement petite en augmentant g
  - (et la TC a besoin de moins de bits de sortie)

Qualité numérique du résultat





- Précision intermédiaire $p+g$
  - erreurs d'arrondi au remplissage des tables
  - rendues arbitrairement petite en augmentant g
  - (et la TC a besoin de moins de bits de sortie)
- Arrondi final

Décomposition multipartite généralisée

(à partir de deux articles simultanés par Schulte et Stine, et Muller)

$$f(X) \approx TVI(A) + TC(C, B)$$

Décomposition multipartite généralisée

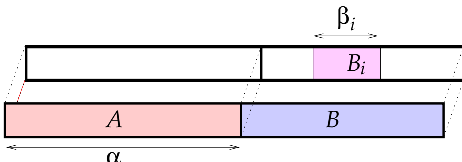
(à partir de deux articles simultanés par Schulte et Stine, et Muller)

$$f(X) \approx TVI(A) + s(C) \times 2^{-\alpha} B$$

Décomposition multipartite généralisée

(à partir de deux articles simultanés par Schulte et Stine, et Muller)
Redécoupage de B

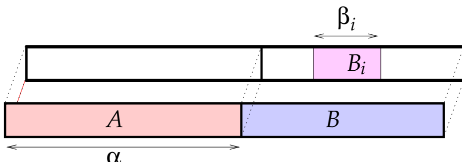
$$f(X) \approx TVI(A) + s(C) \times \sum_{i=0}^{m-1} 2^{-\alpha_i} B_i$$



Décomposition multipartite généralisée

(à partir de deux articles simultanés par Schulte et Stine, et Muller)
Redécoupage de B, distribution de la multiplication

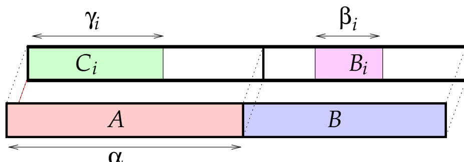
$$f(X) \approx TVI(A) + \sum_{i=0}^{m-1} s(C) \times 2^{-\alpha_i} B_i$$



Décomposition multipartite généralisée

(à partir de deux articles simultanés par Schulte et Stine, et Muller)
Redécoupage de B, distribution de la multiplication et équilibrage des précisions

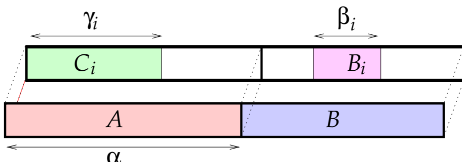
$$f(X) \approx TVI(A) + \sum_{i=0}^{m-1} s(C_i) \times 2^{-\alpha_i} B_i$$



Décomposition multipartite généralisée

(à partir de deux articles simultanés par Schulte et Stine, et Muller)
Redécoupage de B, distribution de la multiplication et équilibrage des précisions

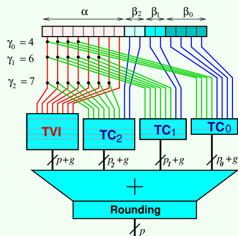
$$f(X) \approx TVI(A) + \sum_{i=0}^{m-1} TC_i(C_i, B_i)$$

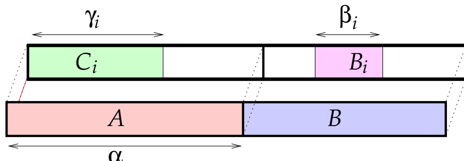


Décomposition multipartite généralisée

(à partir de deux articles simultanés par Schulte et Stine, et Muller)
Redécoupage de B, distribution de la multiplication et équilibrage des précisions

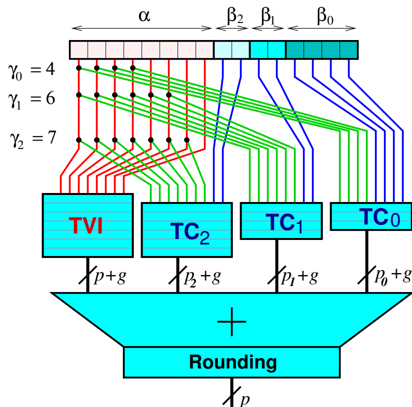
$$f(X) \approx TVI(A) + \sum_{i=0}^{m-1} TC_i(C_i, B_i)$$





Architectures multipartites

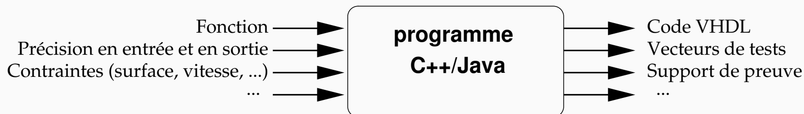
$\alpha, \beta, m, (\gamma_i, \beta_i)_{i=0 \ldots m-1}$



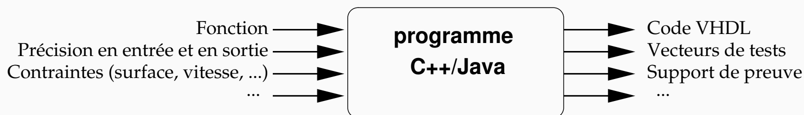
Comment choisir les paramètres pour une architecture

- qui satisfait aux contraintes de précision,
- qui minimise les coûts ?

Un générateur d'opérateurs



Un générateur d'opérateurs



- définir un espace des paramètres,
- construire les fonctions de coût
- explorer l'espace des paramètre
- produire le VHDL pour les meilleurs candidats

Quelques résultats d'implantation des tables multipartites sur FPGA

Pour $p = 16$ bits :

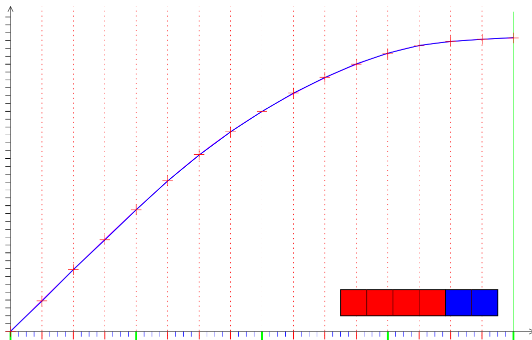
f	m	α	β	α_i	β_i	tables	size	ref size
sin	2	8	8	7,4	3,5	$20.2^8 + 11.2^9 + 9.2^8$	13056	20480
	3	6	10	6,6,4	3,3,4	$20.2^6 + 13.2^8 + 10.2^8 + 8.2^7$	8192	17920
	4	6	10	6,6,5,4	3,2,3,2	$20.2^6 + 13.2^8 + 10.2^7 + 8.2^7 + 5.2^5$	7072	na
2^x	2	8	8	7,5	3,5	$20.2^8 + 12.2^9 + 9.2^9$	15872	14592
	3	6	10	6,6,4	4,3,3	$19.2^6 + 13.2^9 + 9.2^8 + 6.2^6$	10560	13568
	4	6	10	6,6,6,5	3,2,3,2	$21.2^6 + 15.2^8 + 12.2^7 + 10.2^8 + 7.2^6$	9728	na

Tabulation simple : $16 \times 2^{16} = 1\ 048\ 576$ bits

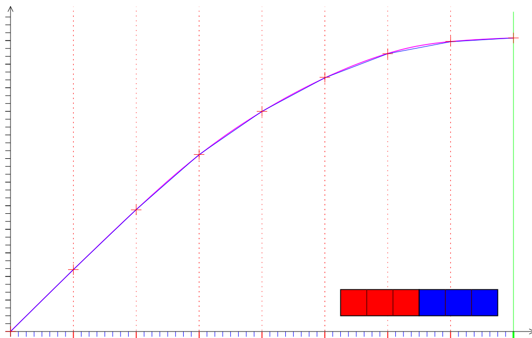
La limite pratique c'est environ 24 bits :

f	m	α	β	α_i	β_i	tables	size	ref size
sin	2	13	11	10,7	4,7	$8.2^{13} + 15.2^{13} + 11.2^{13}$	442368	753664
	3	9	15	9,9,6	5,4,6	$29.2^{9} + 20.2^{13} + 15.2^{12} + 11.2^{11}$	262656	610304
	4	10	14	10,10,8,6	3,3,4,4	$29.2^{10} + 18.2^{12} + 15.2^{12} + 13.2^{11} + 9.2^{9}$	196096	507904
2^x	2	13	11	11,7	4,7	$28.2^{13} + 15.2^{14} + 11.2^{13}$	565248	581632
	3	10	14	10,10,6	4,4,6	$29.2^{10} + 19.2^{13} + 15.2^{13} + 11.2^{11}$	330752	425984
	4	10	14	10,10,9,7	4,2,3,5	$28.2^{10} + 18.2^{13} + 14.2^{11} + 12.2^{11} + 9.2^{11}$	247808	360448

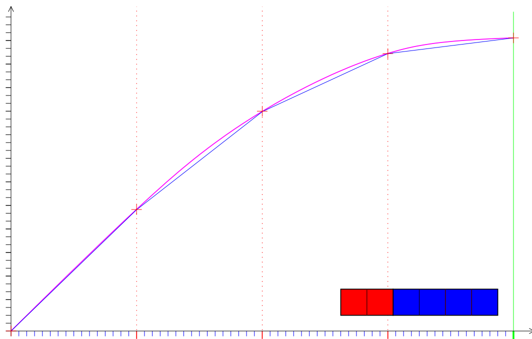
Une méthode sans compromis



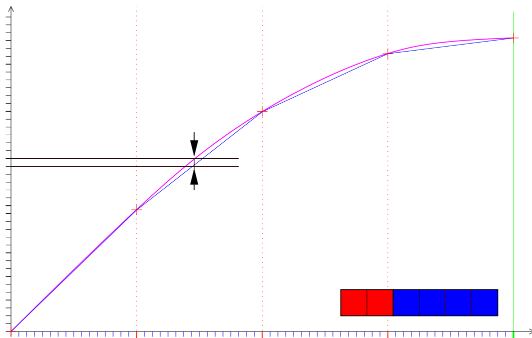
Une méthode sans compromis



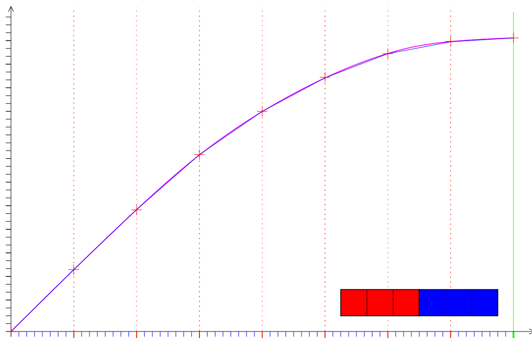
Une méthode sans compromis



Une méthode sans compromis

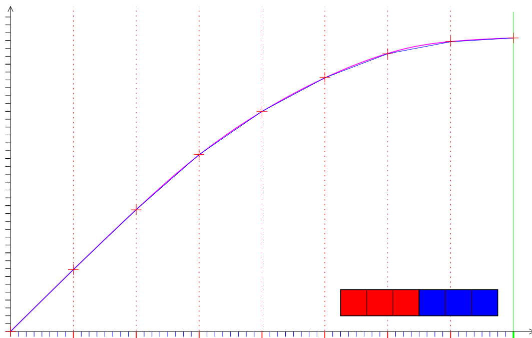


Une méthode sans compromis



Avec un multiplieur on n'aurait pas des tables plus petites

Une méthode sans compromis



Avec un multiplieur on n'aurait pas des tables plus petites

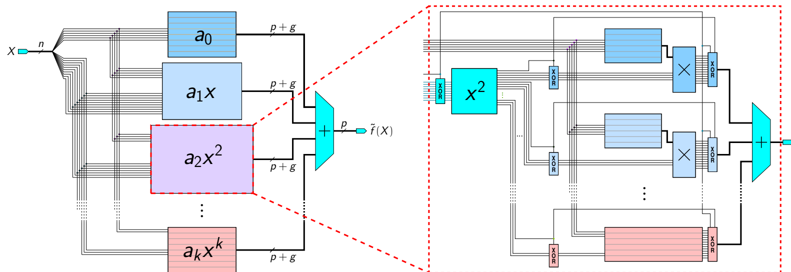
Mais où est le compromis espace/temps ?

Une architecture plus petite est aussi plus rapide
(au moins sur FPGA)

Passage à des ordres supérieurs (J. Detrey)

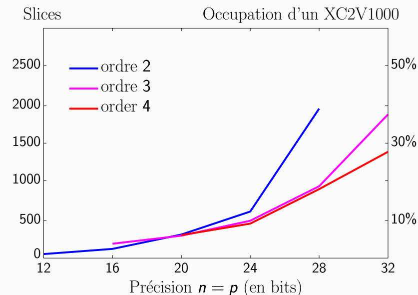
Même philosophie, mais c'est plus compliqué

- Approximation polynomiale par morceaux, de degré 2 à 5
  - Plus de briques de bases : tables, multiplieurs, unités puissance
- Évaluation parallèle de la forme développée
  - Décomposer les mots et développer les produits
  - Tabuler quand c'est moins cher
  - Ne jamais calculer plus précis que nécessaire
- Un générateur-optimiseur d'opérateurs : HOTBM

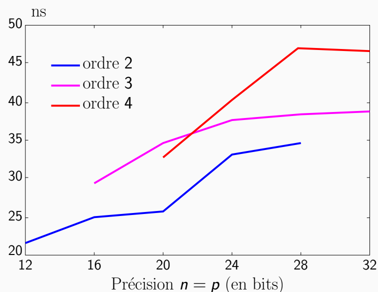


Quelques résultats (sinus)

Surface



Délai



On obtient bien un compromis

Approximation polynomiale en virgule fixe

Fonctions élémentaires, ou pas

Méthodes à base de table

Approximation polynomiale en virgule fixe

Sinus et cosinus en virgule fixe

Une fonction flottante en détail : le logarithme

Introduction

- Idée : approcher une fonction f gentille (suffisamment continue et dérivable) sur un intervalle I par un polynôme p.
  - Erreur d'approximation : $\|f - p\|_I$ (norme infinie, norme sup)
  - LCDE : Plus le degré du polynôme est élevé, plus l'approximation est bonne
- Théorie sur les réels : Taylor, Chebyshev, Remez. Voir le livre de Muller.
- Passage en virgule fixe non trivial :
  - comment arrondir les coefficients ?
  - combien de bits pour les coefficients ?

  Je vais parler surtout de cela.

Exemple : approximation de l'exponentielle $f(x) = e^x$ sur $[0, 1]$.

La loi de conservation des emmerdements

Approximation de e^x par le meilleur polynôme possible :

degree	2	3	4	5
$[0, 1]$	1.41e-2	8.76e-4	5.07e-5	4.86e-6
$[0, 1/2]$	1.08e-3	3.40e-5	8.33e-7	5.27e-8
$[0, 1/4]$	1.07e-4	1.90e-6	4.57e-8	1.94e-9

$\|e^x - p(x)\|$ versus degree and interval size

Ne pas extrapoler ce tableau à d'autres fonctions !

La loi de conservation des emmerdements

Approximation de e^x par le meilleur polynôme possible :

degree	2	3	4	5
$[0, 1]$	1.41e-2	8.76e-4	5.07e-5	4.86e-6
$[0, 1/2]$	1.08e-3	3.40e-5	8.33e-7	5.27e-8
$[0, 1/4]$	1.07e-4	1.90e-6	4.57e-8	1.94e-9

$||e^x - p(x)||$ versus degree and interval size

Ne pas extrapoler ce tableau à d'autres fonctions !

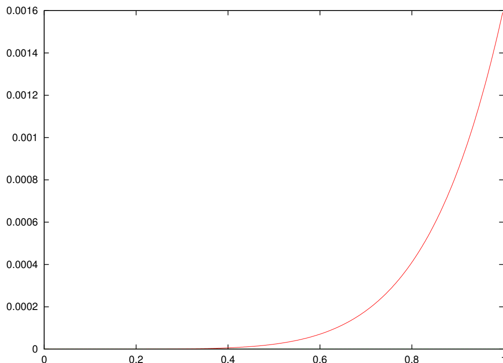
Moralité : pour diminuer le degré (donc le coût) il faut
réduire la taille de l'intervalle.

- Techniques générales, par exemple découper l'intervalle en n sous-intervalles avec autant de polynômes
- Techniques spécifiques, par exemple $e^{a+b} = e^a \times e^b$

Quand on n'a pas d'ordinateur, on a toujours Taylor

Première idée : approximation de Taylor en 0 de $f(x) = e^x$, par exemple à l'ordre 5 :

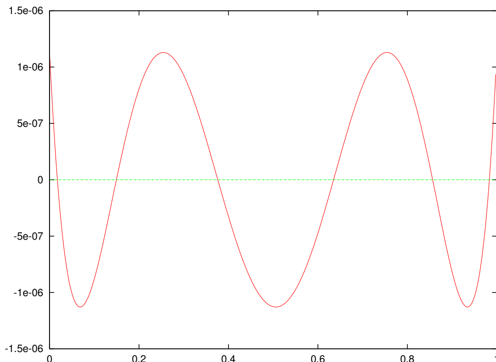
$$f(x) \approx p(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120}$$



Plot de l'erreur d'approximation $f(x) - p(x)$ sur $[0, 1]$

Approximation minimax

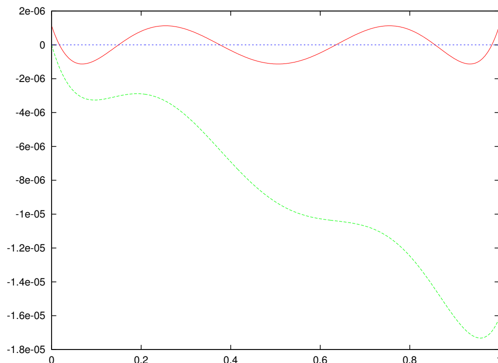
Taylor est une approximation locale.
Remez a décrit un algorithme numérique qui fait osciller p autour de f :



Plot de l'erreur d'approximation $f(x) - p(x)$ sur $[0, 1]$,
toujours pour p de degré 5.

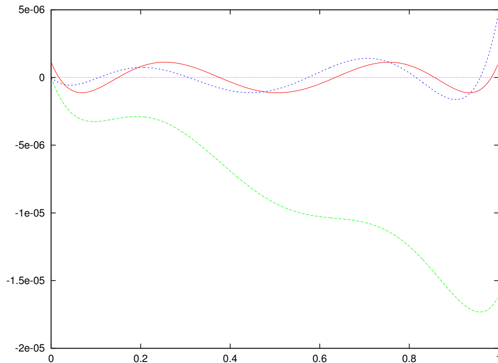
Passage en virgule fixe

Mais si l'on arrondit les coefficients (réels) du Remez à des nombres en virgule fixe :



`fpminimax` de Sollya

Chercher le polynôme qui minimise $\|f - p\|$, parmi les polynômes à coefficients machine :



Donc pub pour Sollya

The Swiss Army Knife for elementary function development
Authors : Lauter, Chevillard, Joldes
Open-source, LGPL, `http://sollya.gforge.inria.fr/`

Killer feature 1

apologizes each time it rounds something

```
fdedinec@krupnik: sollya
> 1+1;
2
> 1/3;
Warning: rounding has happened.
The value displayed is a faithful rounding
of the true result.
0.33333333333333333333333333333333333333333333
```

The Patriot bug

In 1991, a Patriot missile failed to intercept a Scud, and 28 people were killed.

- The code worked with time increments of 0.1 s.
- But 0.1 is not representable in binary.
- In the 24-bit format used, the number stored was 0.099999904632568359375
- The error was 0.0000000953.
- After 100 hours = 360,000 seconds, time is wrong by 0.34s.
- In 0.34s, a Scud moves 500m

Test : which of the following increments should you use ?

10 5 3 1 0.5 0.25 0.2 0.125 0.1

Sollya (2)

Killer feature 2

multiple-precision, last-bit accurate evaluation of arbitrary expressions

```
fdedinec@krupnik: sollya
> e=exp(x) - (1+x+x^2/2+x^3/6);
> e(0.125);
Warning: rounding has happened.
The value displayed is a faithful rounding
of the true result.
1.04322334929834956738944784605392321 7e-5
>
```

All these digits are meaningful! This is better than Maple.

Sollya (3)

Killer feature 3

guaranteed infinite norm $||f(x)||_\infty$ even in degenerate cases

- $||f(x) - P(x)||_\infty$ is a degenerate case...

Sollya (4)

Killer feature 4

Machine-efficient polynomial approximation

- Remez' minimax algorithm finds the best polynomial approximation over the reals
- But we need polynomials with machine coefficients
  - `float`, `double`, fixed-point, ...
- Rounding Remez coefficients does not provide the best polynomial among polynomial with machine coefficients.

Sollya (4)

Killer feature 4

Machine-efficient polynomial approximation

- Remez' minimax algorithm finds the best polynomial approximation over the reals
- But we need polynomials with machine coefficients
  - `float`, `double`, fixed-point, ...
- Rounding Remez coefficients does not provide the best polynomial among polynomial with machine coefficients.
- Sollya does.

Sollya (4)

Killer feature 4

Machine-efficient polynomial approximation

- Remez' minimax algorithm finds the best polynomial approximation over the reals
- But we need polynomials with machine coefficients
  - `float`, `double`, fixed-point, ...
- Rounding Remez coefficients does not provide the best polynomial among polynomial with machine coefficients.
- Sollya does.

Nice number theory behind. And needs all the previous.

Example : 6 approximations to log(x) on one slide

```
f= log(1+y);
I=[-0.25;.5];
filename="/tmp/polynomials";
print("") > filename;
for deg from 2 to 8 do begin
  p = fpminimax(f, deg,[|0,23...|],I, floating, absolute);
  display=decimal;
  acc=floor(-log2(sup(supnorm(p, f, I, absolute, 2^(-40)))));
  print( "   // degree = ", deg,
         "  => absolute accuracy is ",  acc, "bits" ) >> filenam
  print("#if ( DEGREE ==", deg, ")") >> filename;
  display=hexadecimal;
  print("   float p = ", horner(p) , ";") >> filename;
  print("#endif") >> filename;
end;
```

Polynômes en virgule fixe

Soit le polynôme $a_0 + a_1 x + a_2 x^2 + a_3 x^3$, avec $a_0 \approx a_1 \approx a_2 \approx a_3$.
Évaluons-le pour $x \in [-1, 1]$:
Tous les termes de la somme sont alignés
(on peut utiliser le même format virgule fixe)

```
 p(x) =  xxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
   a0      xxxxxxxxxxxxxxxxxxxxxxxxxxx...
 +a1x      xxxxxxxxxxxxxxxxxxxxxxxxxxx...
+a2x^2     xxxxxxxxxxxxxxxxxxxxxxxxxxx...
+a3x^3     xxxxxxxxxxxxxxxxxxxxxxxxxxx...
```

Polynômes en virgule fixe

Soit le polynôme $a_0 + a_1x + a_2x^2 + a_3x^3$, avec $a_0 \approx a_1 \approx a_2 \approx a_3$.
Évaluons-le pour $x \in [-1, 1]$:
Tous les termes de la somme sont alignés
(on peut utiliser le même format virgule fixe)

$p(x) =$	`xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...`
a_0	`  xxxxxxxxxxxxxxxxxxxxxxxxxxxx...`
$+a_1x$	`  xxxxxxxxxxxxxxxxxxxxxxxxxxxx...`
$+a_2x^2$	`  xxxxxxxxxxxxxxxxxxxxxxxxxxxx...`
$+a_3x^3$	`  xxxxxxxxxxxxxxxxxxxxxxxxxxxx...`

Remarque capitale

Evaluation par Horner :

$$a_0 + x(a_1 + x(a_2 + xa_3))$$

toutes les additions sont toujours alignées

Polynômes en virgule fixe

Soit toujours le polynôme $a_0 + a_1 x + a_2 x^2 + a_3 x^3$,
avec toujours $a_0 \approx a_1 \approx a_2 \approx a_3$.
Mais évaluons-le à présent sur un petit intervalle : $x \in [-2^{-k}, 2^{-k}]$.
Du coup $|x^2| < 2^{-2k}$ et $|x^3| < 2^{-3k}$:

```
p(x) =    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
a0         xxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
+a1x             xxxxxxxxxxxxxxxxxxxxxxx...
+a2x^2                 xxxxxxxxxxxxxxxx...
+a3x^3     <--k--><--k--><--k-->xxxxxxxxx...
```

Polynômes en virgule fixe

Soit toujours le polynôme $a_0 + a_1 x + a_2 x^2 + a_3 x^3$,
avec toujours $a_0 \approx a_1 \approx a_2 \approx a_3$.
Mais évaluons-le à présent sur un petit intervalle : $x \in [-2^{-k}, 2^{-k}]$.
Du coup $|x^2| < 2^{-2k}$ et $|x^3| < 2^{-3k}$:

```
p(x) =    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
   a0      xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
 +a1x            xxxxxxxxxxxxxxxxxxxxxxxxx...
 +a2x^2                xxxxxxxxxxxxxxxxxxx...
 +a3x^3   <---k---><---k---><---k--->xxxxxxxxx...
```

Remarque capitale

Cet alignement reste valable pour une évaluation par Horner :

$$a_0 + x(a_1 + x(a_2 + x a_3))$$

Polynômes en virgule fixe

Pour obtenir une précision à 2^{-p},

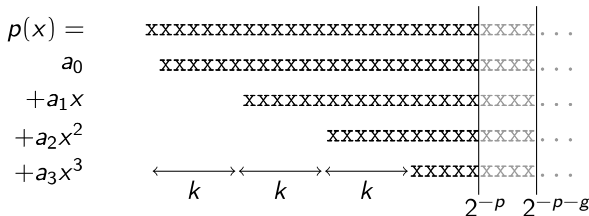
il suffit que chaque $a_i x^i$ soit précis à 2^{-p-g}.

```
p(x) =   xxxxxxxxxxxxxxxxxxxxxxxx|xxxx|...
a0        xxxxxxxxxxxxxxxxxxxxxxx|xxxx|...
+a1x            xxxxxxxxxxxxxxxxx|xxxx|...
+a2x^2                xxxxxxxxxxx|xxxx|...
+a3x^3   <--k--><--k--><--k-->xxxxx|xxxx|...
                                 2^-p  2^-p-g
```

Polynômes en virgule fixe

Pour obtenir une précision à 2^{-p},

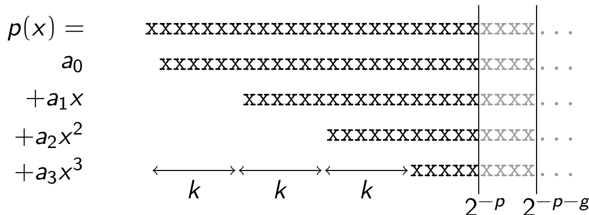
il suffit que chaque $a_i x^i$ soit précis à 2^{-p-g}.



(g bits de garde pour absorber l'accumulation des erreurs d'arrondi.)

Format des a_i

- Le poids fort des a_i est celui du coefficient obtenu par Remez.
- Pour que chaque $a_i x^i$ soit précis à 2^{-p-g}
  il suffit que chaque a_i ait la même taille que $a_i x^i$ sur le schéma



Sollya permet cela aussi

... en demandant le poids du LSB de chaque coefficient.

Calcul au plus juste des formats intermédiaires pour Horner

$$a_0 + x(a_1 + x(a_2 + xa_3))$$

$$x \in [-2^{-k}, 2^{-k}]$$

- Le format de $a_2 + xa_3$ est celui de a_2 (à un bit près),
- et on peut tronquer x au même format que a_3 pour cette multiplication
- Le format de $a_1 + x(a_2 + xa_3)$ est celui de a_1 (à un bit près)
- et on peut tronquer x au même format que a_2 pour cette multiplication
- ...

$p(x) =$	xxxxxxxxxxxxxxxxxxxxxxxxxx|xxxx|...
a_0	xxxxxxxxxxxxxxxxxxxxxxxxx|xxxx|...
$+a_1x$	xxxxxxxxxxxxxxxxxx|xxxx|...

Approximation polynomiale par morceaux

- On découpe l'intervalle de départ en 2^k intervalles
- On cherche des polynômes sur ces petits intervalles
- Par un changement de variable, c'est le cas précédent.

Approximation polynomiale par morceaux

- On découpe l'intervalle de départ en 2^k intervalles
- On cherche des polynômes sur ces petits intervalles
- Par un changement de variable, c'est le cas précédent.

Moulinette implémentée dans FloPoCo (démo ?).

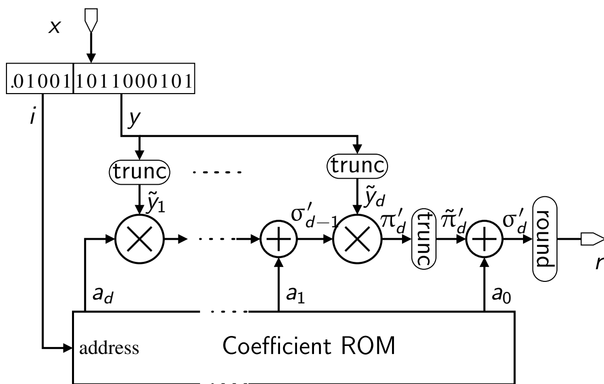
Exemple

e^x, précision cible 10^{-7}, en utilisant des polynômes de degré 4 :

- Un découpage en 4 sous-intervalles suffit
- LSB = -26
- Formats virgule fixe valables pour les les 4 polynômes :

```
> PiecewisePolyApprox: Final report:
> PiecewisePolyApprox: Degree=4 maxApproxErrorBound=4.17201e-08
> PiecewisePolyApprox: MSB[0] = 2 size=29
> PiecewisePolyApprox: MSB[1] = 0 size=27
> PiecewisePolyApprox: MSB[2] = -3 size=24
> PiecewisePolyApprox: MSB[3] = -7 size=20
> PiecewisePolyApprox: MSB[4] = -11 size=16
> PiecewisePolyApprox: Total size of the table is 4 x 116 bits
```

Pour obtenir l'architecture que voici :



Pour une évaluation en virgule flottante

... une réduction d'argument nous ramène dans un petit domaine.
Dès que l'exposant est constant sur le domaine,

on est en virgule fixe.

Sinus et cosinus en virgule fixe

Fonctions élémentaires, ou pas

Méthodes à base de table

Approximation polynomiale en virgule fixe

Sinus et cosinus en virgule fixe

Une fonction flottante en détail : le logarithme

Par recyclage des slides de HEART 2013

Une fonction flottante en détail : le logarithme

Fonctions élémentaires, ou pas

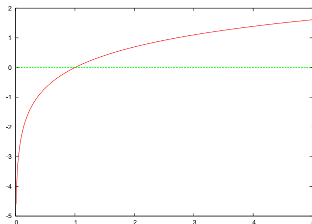
Méthodes à base de table

Approximation polynomiale en virgule fixe

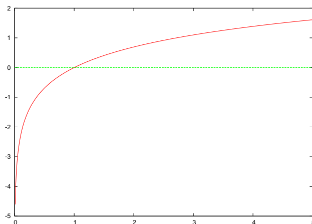
Sinus et cosinus en virgule fixe

Une fonction flottante en détail : le logarithme

J'ai choisi la plus gentille des fonctions

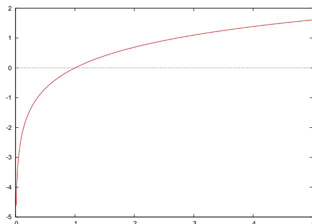


J'ai choisi la plus gentille des fonctions



Pourquoi elle est gentille ?

J'ai choisi la plus gentille des fonctions



Pourquoi elle est gentille ?

- Parce que $\ln(ab) = \ln(a) + \ln(b)$
- Parce que $\ln(1+\varepsilon) = \varepsilon - \frac{\varepsilon^2}{2} + \frac{\varepsilon^3}{3} \ldots$
- Parce qu'elle est strictement croissante
- Parce qu'elle ne fait pas de dépassement de capacité
  - Exemple : double-précision,
    maxFloat $\approx 1.798 \cdot 10^{308}$ $\ln(\text{maxFloat}) < 710$
    minFloat $\approx 4.941 \cdot 10^{-324}$ $\ln(\text{minFloat}) > -745$

Démontage du format flottant

(Dans ce transparent et ceux qui suivent, je décris ce qu'il faut faire, pas encore comment on peut le faire)

- Test des cas exceptionnels (voir la norme IEEE-754-2008)

$$\begin{array}{llll} X = \mathrm{NaN} & \longrightarrow & \ln(X) = \mathrm{NaN}, & \text{signaler Invalid} \\ X < 0 & \longrightarrow & \ln(X) = \mathrm{NaN}, & \text{signaler Invalid} \\ X = \pm 0 & \longrightarrow & \ln(X) = -\infty, & \text{signaler DivisionByZero} \end{array}$$

Démontage du format flottant

(Dans ce transparent et ceux qui suivent, je décris ce qu'il faut faire, pas encore comment on peut le faire)

- Test des cas exceptionnels (voir la norme IEEE-754-2008)

$$\begin{array}{llll} X = \mathrm{NaN} & \longrightarrow & \ln(X) = \mathrm{NaN}, & \text{signaler Invalid} \\ X < 0 & \longrightarrow & \ln(X) = \mathrm{NaN}, & \text{signaler Invalid} \\ X = \pm 0 & \longrightarrow & \ln(X) = -\infty, & \text{signaler DivisionByZero} \end{array}$$

- Première réduction d'argument
  - $X = 2^E \cdot 1,F$
  - Donc $\ln(X) = E \cdot \ln(2) + \ln(1,F)$
  - Pour recentrer la mantisse sur 1 et éviter un problème de précision autour de $X = 2$, on préfère

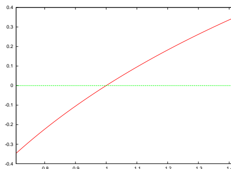
$$\begin{cases} E' = E & \text{et } Y = 1,F \quad \text{si} \quad m \leqslant \sqrt{2} \\ E' = E + 1 & \text{et } Y = \frac{1,F}{2} \quad \text{if} \quad m > \sqrt{2} \end{cases}$$

  - alors $\ln(X) = E' \cdot \ln(2) + \ln(Y)$
    - avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$
    - (choisi pour que $\log(Y)$ soit centré autour de 0)

Évaluation

On va donc calculer $\ln(X)$ comme $E' \cdot \ln(2) + \ln(Y)$

- $E' \cdot \ln(2)$ c'est un petit entier par une constante réelle
  - fastoche...
- Reste à calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

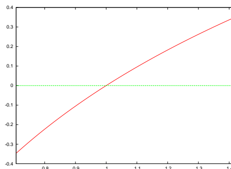


- Elle a l'air encore plus gentille comme cela.
  Approchons-là

Évaluation

On va donc calculer $\ln(X)$ comme $E' \cdot \ln(2) + \ln(Y)$

- $E' \cdot \ln(2)$ c'est un petit entier par une constante réelle
  - fastoche...
- Reste à calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$



- Elle a l'air encore plus gentille comme cela.
  Approchons-là par un polynôme.
- Le couteau suisse à polynômes c'est Sollya
  (`http://sollya.gforge.inria.fr/`).
  Quel degré faut-il pour une approximation à 2^{-55} :

  ```
  > guessdegree(log(x), [1/sqrt(2);sqrt(2)], 1b-55);
  ```

  Sollya répond : degré 20.

On le tient, notre algorithme ?

Un polynôme $p(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$ de degré n s'évalue en $2n$ opérations par le schéma de Horner :

$$p(Y) = a_0 + Y \times (a_1 + Y \times (a_2 + Y \times (... + Y \times a_n))..)$$

En comptant 4 cycles l'opération flottante, cela nous fait dans les 160 cycles pour le polynôme, et 300 cycles en tout.

On le tient, notre algorithme ?

Un polynôme $p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$ de degré n s'évalue en $2n$ opérations par le schéma de Horner :

$$p(Y) = a_0 + Y \times (a_1 + Y \times (a_2 + Y \times (... + Y \times a_n))..)$$

En comptant 4 cycles l'opération flottante, cela nous fait dans les 160 cycles pour le polynôme, et 300 cycles en tout.

C'est vendable. Peut-on mieux faire ?

On le tient, notre algorithme ?

Un polynôme $p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$ de degré n s'évalue en $2n$ opérations par le schéma de Horner :

$$p(Y) = a_0 + Y \times (a_1 + Y \times (a_2 + Y \times (... + Y \times a_n))..)$$

En comptant 4 cycles l'opération flottante, cela nous fait dans les 160 cycles pour le polynôme, et 300 cycles en tout.

C'est vendable. Peut-on mieux faire ?

Oui, en

- parallélisant l'évaluation du polynôme (pub pour CGPE), ou
- en brûlant de la mémoire. Je vais montrer cela.

Seconde réduction d'argument à base de table

Donc on veut calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

- Soit i l'entier composé des k bits de poids fort de Y.

Seconde réduction d'argument à base de table

Donc on veut calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

- Soit i l'entier composé des k bits de poids fort de Y.
- Lisons, dans une table indicée par i,

$$R[i] \approx \frac{1}{Y}$$

- Calculons

$$Z = Y \times R[i] - 1$$

Seconde réduction d'argument à base de table

Donc on veut calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

- Soit i l'entier composé des k bits de poids fort de Y.
- Lisons, dans une table indicée par i,

$$R[i] \approx \frac{1}{Y}$$

- Calculons

$$Z = Y \times R[i] - 1$$

- Z est petit (on peut montrer que $|Z| < 2^{-k+1}$) et

$$\ln(Y) = \ln(1+Z) - \ln(R[i])$$

Seconde réduction d'argument à base de table

Donc on veut calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

- Soit i l'entier composé des k bits de poids fort de Y.
- Lisons, dans une table indicée par i,

$$R[i] \approx \frac{1}{Y}$$

- Calculons

$$Z = Y \times R[i] - 1$$

- Z est petit (on peut montrer que $|Z| < 2^{-k+1}$) et

$$\ln(Y) = \ln(1 + Z) - \ln(R[i])$$

- Précalculons et tabulons aussi, pour chaque valeur de $R[i]$,

$$L[i] = \ln(R[i])$$

- Et notre calcul devient

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1 + Z)$$

Le log selon Tang

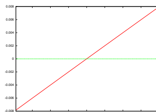
$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1 + Z)$$

- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$

Le log selon Tang

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1 + Z)$$

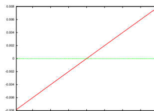
- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$
- La fonction est de plus en plus gentille.



Le log selon Tang

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1+Z)$$

- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$
- La fonction est de plus en plus gentille.

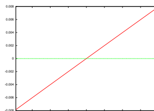


- Un polynôme de degré 6 fera l'affaire

Le log selon Tang

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1+Z)$$

- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$
- La fonction est de plus en plus gentille.

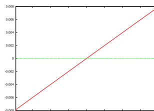


- Un polynôme de degré 6 fera l'affaire
- On a gagné 14× et 14+, au prix de
  - l'extraction de i
  - deux lectures de tables dans des tables à 256 entrées
  - le calcul de $Z = Y \times R[i] - 1$

Le log selon Tang

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1+Z)$$

- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$
- La fonction est de plus en plus gentille.



- Un polynôme de degré 6 fera l'affaire
- On a gagné $14\times$ et $14+$, au prix de
  - l'extraction de i
  - deux lectures de tables dans des tables à 256 entrées
  - le calcul de $Z = Y \times R[i] - 1$
- Remarque : on choisit $R[i]$:
  - on peut le choisir tenant sur peu de bits (environ k bits)
  - alors (TSVP) on peut calculer Z exactement (sans arrondi)
  - Les deux réductions d'argument sont exactes

Un peu d'adéquation tartine – confiture

Deux manières de calculer Z exactement, suivant votre processeur :

- Vous voulez calculer 24 bits de mantisse (flottant binary32) sur un processeur entier 32 bits
  - vous avez donc 8 bits de marge
  - si $R[i]$ tient sur 8 bits le produit $YR[i]$ tient sur 32 bits
- Vous voulez calculer 53 bits de mantisse sur un Pentium
  - vous pouvez utiliser du double-étendu (64 bits de mantisse)
  - vous avez donc 11 bits de marge

Et en matériel...

On peut itérer la seconde réduction d'argument.

- sans table d'inverse : $\frac{1}{1+Z}$ peut s'approcher par $1 - Z$
- Et notre calcul devient

$$\ln(X) = E' \cdot \ln(2) - L_0 - L_1 ... - L_k + \ln(1 + Z_k)$$

- on s'arrête par exemple dès que $\ln(1 + Z_k) \approx Z_k$ est assez précis.