[TRIG]

GL01

Janvier 2022

1 Introduction

Liste des fonctions implémentées

Fonctions	Arguments	Description
float ulp[1]	float f	Retourne en résultat *Unit in the Last Place* du flottant f
float sin	float f	Retourne en résultat $sin(f)$
float cos	float f	Retourne en résultat $cos(f)$
float tan	float f	Retourne en résultat $tan(f)$
float asin	float f	Retourne en résultat $asin(f)$
float atan	float f	Retourne en résultat $atan(f)$
int puiss	int base, int exponent	Retourne en résultat $base^{exponent}$
float abs	float f	Retourne en résultat $abs(f) = \|f\|$
int eed	float f	Retourne en résultat l'exponent $e - 127$ de l'écriture en Deca du float f
int mant	float f	Retourne en résultat la mantisse m de l'écriture en Deca du float f

2 Spécifications détaillées des fonctions

2.1 Représentation des flottants en langage Deca

Pour comprendre le fonctionnement de toutes ces fonctions y compris *ulp*, il est nécessaire de comprendre le fonctionnement des flottants. Les flottants

1. Lien vers Ulp - Wikipedia

Deca sont stockés en mémoire d'une manière similaire qu'en Java mais avec une différence au niveau de la mantisse :

$$(-1)^s 2^{e-127}(1 + m.2^{-23})$$

Cette représentation des flottants nous permet de les stocker sur 32 bits :

$$b_{31}|b_{30}...b_{23}|b_{22}...b_0$$

- bit 31 : Contient le bit de signe s
- bits 30 à 23 : Contiennent l'exponent e codé sur 8 bits (Avec un minimum de 0 et maximum de 255)
- bits 22 à 0 : La mantisse m codée sur 23 bits (Minimum de 0 et maximum de 8388607)

Ne pouvant pas représenter tous les nombres réels présents dans notre intervalle, il est nécessaire d'étudier ce qu'on appelle *Unit in the Last Place* ou *ulp* qui est définie comme étant la distance au prochain nombre réel représentable en float. On peut en déduire une expression, en ayant un x donné et y le prochain réel représentable :

$|y - x| = ulp(x)$ or si $x = (-1)^s 2^{e-127}(1 + m.2^{-23})$ on sait que $y = (-1)^s 2^{e-127}(1 + (m + 1).2^{-23})$ (En supposant que $m \neq m_{max}$) d'où :

$$ulp(x) = |y - x| = 2^{e-127-23}$$

Pour donner un exemple, le flottant 1 est codé comme suit : $0b0011111110...0$, puisque $e = 127$ on aura $ulp(1) = 2^{-23}$ mais le flottant en dessous de 1 est :

$$x = 2^{126-127}(1 + m_{max}2^{-23}) = 2^{-1}(1 + 1 - 2^{-23}) = 1 - 2^{-24}$$

$$x_{max} = 2^{255-127}(1 + (2^{23} - 1)2^{-23}) = 2^{128}(2 - 2^{-23}) = 2^{129} - 2^{105}$$

Donc

$$ulp(x_{max}) = 2^{128-23} = 2^{105}$$

2.2 Algorithmes des fonctions trigonométriques

Nous avons choisi de baser notre algorithme principalement sur les séries de Taylor en 0 et leur troncature à un ordre donné pour approximer les fonctions l'écriture formelle est la suivante :

$$\sum_{k=0}^{n} \frac{f^{(k)}(0)}{k!} x^k$$

Cette méthode a des avantages et inconvénients :

- **Polynomiale** : Elle ne nécessite que l'utilisation de sommes et multiplications (Les coefficients factoriels ne sont pas recalculés à chaque fois et sont stockés en dur).

— **Convergence rapide avec majoration de l'erreur** : Avec la formule de $Taylor - Lagrange$[1] donnant une majoration du reste :

$$|f(x) - \sum_{k=0}^{n} \frac{f^{(k)}(0)}{k!} x^k| \leq M \frac{x^{(n+1)}}{(n+1)!}$$

— **Instabilité lorsqu'on est loin de 0** : Comme il est montré dans la formule précédente, la majoration de l'erreur devient de plus en plus faible lorsque x s'éloigne de 0, ceci rend de cette méthode inefficace pour des valeurs au-delà de 1 par exemple (Puisque cela nécessiterai un ordre plus grand donc une complexité plus importante). Prenons l'exemple de $\frac{\pi}{4}$ pour la fonction sin à l'ordre 13 (itération 6 car le développement de la fonction sinus ne contient pas de puissances paires) : Un calcul rapide donnera un écart de la valeur réelle majoré par $3.8980732e - 13$, en fonction de nos besoins ceci est susceptible d'être satisfaisant comme il peut ne pas l'être. Ceci nous oblige à pousser le développement lorsqu'on s'éloigne de 0 pour rétrécir cet écart.

Ceci nous invite à considérer les symétries des différentes fonctions pour réduire l'argument à un intervalle où l'on sait que la série de Taylor donnera un résultat approximatif correcte. Ces symétries viennent principalement de la périodicité et parité des fonctions trigonométriques, ce qui nous permet de ne travailler que sur l'intervalle $[0, \frac{\pi}{4}]$ pour les fonctions sin et cos par exemple, et d'utiliser les formules trigonométriques suivantes pour réduire l'argument à cet intervalle :

Si $x < 0$:

$$sin(-x) = -sin(x)$$
$$cos(-x) = cos(x)$$

Si $\frac{\pi}{4} \leq x \leq \frac{\pi}{2}$

$$sin(\frac{\pi}{2} - x) = cos(x)$$
$$cos(\frac{\pi}{2} - x) = sin(x)$$

nous remet en $0 \leq x \leq \frac{\pi}{4}$

Si $\frac{\pi}{2} \leq x \leq \pi$

$$sin(\pi - x) = sin(x)$$
$$cos(\pi - x) = -cos(x)$$

1. Lien vers Théorème de Taylor - Wikipedia

nous remet en $0 \leq x \leq \frac{\pi}{2}$

Si $x > \pi$ on utilisera la réduction de cet argument avec la formule :

$$sin(x) = sin(x_r + k\frac{\pi}{2}) = sign(k)sin(x_r)$$

avec $0 \leq x_r \leq \frac{\pi}{2}$.

On voit aussi apparaitre la fonction *sign* qui est définie comme suit pour la fonction sinus seulement (Elle est définie autrement pour *cos* et *tan*) :

$$sign(k) = 1 \text{ si } k\%4 \in \{0, 1\} \text{ et } 0 \text{ sinon}$$

En principe, *sign* peut être conçue comme le signe que prend le sinus en fonction de quel **quart du cercle trigonométrique** l'argument x se trouve. On choisit le quart pour réduire les appels consécutifs aux fonctions.
On note aussi que l'utilisation de **Lookup table**, qui est un tableau (ou un fichier) contenant des valeurs distribuées uniformément sur un intervalle donné et leur *sin* et *cos* exact calculé au préalable et stocké, a été considérée au début de la conception. Ceci donne naissance à toute une branche de possibilités mais occupe de la mémoire physique. Cependant, un choix de ne pas continuer sur ce chemin a été fait à cause des faibles améliorations en performance (ou eventuellement manque) en échange du coût en complexité spatiale qui est d'en moyenne 300kB pour remarquer une hausse de performance (Un tableau de $3 * 10^4$ valeurs par exemple). Néanmoins le principe étant intéressant si l'on juge que la complexité spatiale de notre système importe peu par rapport à la performance/précision.

3 A propos des problemes techniques sur Deca

L'implémentation de l'extension ne s'étant pas très bien passé, il est malheureusement impossible d'éxécuter un appel "récursif" après la réduction de l'argument par exemple, ceci est probablement dû à un bug de la partie "Codegen" : L'argument passé à la fonction suivante n'est pas l'argument réduit mais son adresse.