Documentation de l'Extension TRIGO

Architecture et Implémentation

GL56

14 janvier 2026

Table des matières

1 Architecture et Implémentation de l'Extension TRIGO **3**
- 1.1 Architecture Globale du Système . . . 3
- 1.2 Gestion des Registres . . . 3

2 Intégration avec le Compilateur Deca **3**
- 2.1 Détection et Analyse Lexicale . . . 3
- 2.2 Interception et Substitution des Appels . . . 4
- 2.3 Gestion de la Pile . . . 4

3 Fichier Math.decah : Interface Utilisateur **4**
- 3.1 Utilisation . . . 4

4 Aspects théoriques **4**
- 4.1 Interprétation des Équations Itératives . . . 4
- 4.2 Modes de Fonctionnement : Rotation et Vectorisation . . . 5
  - 4.2.1 Mode Rotation : Calcul de sin et cos . . . 5
  - 4.2.2 Mode Vectorisation : Calcul de arctan et arcsin . . . 5
  - 4.2.3 Comparaison Conceptuelle . . . 6
- 4.3 Précision Théorique et Convergence . . . 6

5 Optimisations et Adaptations pour IMA **6**
- 5.1 Représentation des Données . . . 6
- 5.2 Réduction d'Angle . . . 7
- 5.3 Absence d'Instruction FMA . . . 7

6 Spécifications des algorithmes : **7**
- 6.1 Fonction rapprochePi : Réduction d'Angle (première approche) . . . 7
  - 6.1.1 Contexte Mathématique . . . 7
  - 6.1.2 Fonctionnement . . . 7
  - 6.1.3 Exemple . . . 8
- 6.2 Gestion précise des erreurs d'arrondi . . . 8
  - 6.2.1 Annulation numérique lors des réductions . . . 8
  - 6.2.2 Accumulation des erreurs . . . 8
  - 6.2.3 Décomposition haute et basse des constantes . . . 9
  - 6.2.4 Impact sur la précision finale . . . 9
- 6.3 Calcul du sinus . . . 9
  - 6.3.1 Traitement des cas particuliers . . . 9
  - 6.3.2 Réduction modulaire de l'angle . . . 9
  - 6.3.3 Normalisation dans $[-\pi, \pi]$ . . . 10
  - 6.3.4 Réduction dans l'intervalle $[-\pi/2, \pi/2]$ . . . 10

6.3.5 Calcul final du sinus . . . 10
6.4 Calcul du cosinus . . . 10
6.4.1 Traitement des petits angles . . . 11
6.4.2 Réduction finale et gestion du signe . . . 11
6.4.3 Calcul itératif du cosinus . . . 11
6.5 Calcul de l'arctangente . . . 12
6.5.1 Gestion du signe . . . 12
6.5.2 Traitement des grandes valeurs . . . 12
6.5.3 Initialisation du calcul itératif . . . 12
6.5.4 Boucle de convergence . . . 13
6.5.5 Correction finale . . . 13
6.6 Calcul de l'arcsinus . . . 13
6.6.1 Initialisation . . . 13
6.6.2 Gestion du signe et direction de rotation . . . 14
6.6.3 Rotations successives . . . 14
6.6.4 Correction d'échelle . . . 14
6.6.5 Résultat final . . . 14
6.6.6 Différences principales par rapport à sin et cos . . . 15
6.7 Calcul de l'ULP (Unit in the Last Place) . . . 15
6.7.1 Explication de l'algorithme . . . 15
6.7.2 Lien avec les fonctions trigonométriques . . . 16

7 Gestion des Cas Particuliers et Erreurs **16**
7.1 Vérification des Domaines d'Entrée . . . 16
7.2 Réduction des Grands Angles . . . 16
7.3 Contrôle de la Précision Numérique . . . 16

8 Performance et Utilisation des Ressources **16**
8.1 Analyse des Temps d'Exécution . . . 16
8.2 Structure d'une Itération CORDIC . . . 17

9 Compatibilité et Évolutivité **17**

10 Tests de Validation **17**
10.1 Tests pour des Valeurs Usuelles . . . 17
10.2 Test de Cohérence Fonctionnelle . . . 18
10.2.1 Code du Test . . . 18
10.2.2 Analyse des Résultats . . . 18
10.3 Tests sur des Grandes Valeurs . . . 18
10.3.1 Code du Test . . . 19
10.3.2 Analyse des Résultats . . . 19
10.4 Limitation : Accumulation d'Erreur avec des Multiples de π . . . 19
10.4.1 Code du Test . . . 19
10.4.2 Analyse des Résultats . . . 21

11 Conclusion **21**

12 Bibliographie **21**

1 Architecture et Implémentation de l'Extension TRIGO

1.1 Architecture Globale du Système

L'extension TRIGO suit une architecture modulaire qui s'intègre harmonieusement dans l'écosystème existant du compilateur Deca. Le système repose sur trois piliers fondamentaux :

— La détection automatique de l'extension via le mécanisme d'inclusion de fichiers

— La génération de code assembleur optimisé pour la machine IMA

— La fourniture d'une interface de programmation cohérente pour les utilisateurs

Le flux de compilation commence lorsque l'utilisateur inclut le fichier `Math.decah` dans son programme source. Cette inclusion déclenche une cascade d'événements dans le compilateur. Le lexer, en analysant la directive `#include`, reconnaît spécifiquement le fichier `Math.decah` dans la bibliothèque resources, évitant ainsi de surcharger inutilement la compilation de programmes qui n'utilisent pas les fonctions trigonométriques.

Le cœur de l'implémentation réside dans le générateur de code CORDIC, responsable de la production du code assembleur IMA hautement optimisé. Ce générateur s'appuie sur des tables pré-calculées stockées en mémoire, contenant les angles arctangent nécessaires aux itérations CORDIC. Ces tables permettent de réaliser toutes les rotations élémentaires de manière très rapide, en utilisant uniquement des additions, soustractions et décalages binaires, ce qui correspond parfaitement aux capacités de la machine IMA.

Les fonctions trigonométriques directes (`sin` et `cos`) sont implémentées en mode rotation, où l'angle d'entrée est connu et les coordonnées du vecteur initial sont progressivement tournées jusqu'à obtenir la valeur désirée. Les fonctions inverses (`atan` et `asin`) utilisent le mode vectoring, où l'on part d'un vecteur dont l'angle est inconnu et que l'on aligne progressivement sur l'axe horizontal, accumulant l'angle total. Pour `asin`, l'algorithme commence par calculer une racine via 5 à 6 itérations de Newton-Raphson, garantissant une précision proche de la limite de la simple précision, puis applique le CORDIC en mode vectoring pour obtenir l'angle.

1.2 Gestion des Registres

L'architecture est soigneusement optimisée pour l'utilisation des registres de la machine IMA :

— **Registres R2 à R9** : réservés aux calculs intermédiaires du CORDIC (coordonnées du vecteur, angle résiduel, compteurs d'itérations)

— **Registres R0 et R1** : utilisés respectivement pour passer les paramètres et retourner les résultats, conformément aux conventions d'appel standard de l'IMA

Cette organisation garantit non seulement une exécution rapide, mais aussi une intégrité complète des registres pour le code appelant, évitant tout effet de bord.

2 Intégration avec le Compilateur Deca

L'intégration de l'extension TRIGO dans le compilateur Deca se fait à plusieurs niveaux et repose sur une coordination fine entre le front-end (analyse lexicale et syntaxique) et le back-end (génération de code assembleur optimisé).

2.1 Détection et Analyse Lexicale

Au niveau du lexer, le compilateur a été modifié pour reconnaître la directive `#include` et identifier spécifiquement l'inclusion du fichier `Math.decah`. Lorsqu'un programme source inclut ce fichier, le compilateur active un état interne signalant que les fonctions trigonométriques de l'extension TRIGO devront être disponibles dans le code assembleur final.

2.2 Interception et Substitution des Appels

Lors de la phase de génération de code, le compilateur intercepte tous les appels aux méthodes de la classe `Math` (telles que `sin`, `cos`, `atan` ou `asin`) et les remplace automatiquement par des appels aux routines assembleur correspondantes optimisées pour la machine IMA.

Cette approche présente plusieurs avantages :

1. **Performance maximale** : le code natif CORDIC est implémenté en assembleur
2. **Précision garantie** : grâce aux itérations CORDIC pré-calculées et aux tables d'angles arctangents
3. **Sécurité et robustesse** : conversion automatique des paramètres et des valeurs de retour

2.3 Gestion de la Pile

Pour préserver l'état du processeur et assurer la compatibilité avec le reste du code Deca, chaque routine CORDIC sauvegarde et restaure soigneusement les registres qu'elle utilise. La pile est également gérée avec précision : les sauvegardes temporaires lors des rotations CORDIC (push/pop) garantissent que l'espace mémoire alloué reste constant et que les instructions de retour (RTS) fonctionnent correctement.

3 Fichier Math.decah : Interface Utilisateur

Le fichier `Math.decah` constitue l'interface publique de l'extension TRIGO. Conforme aux conventions de nommage de la bibliothèque standard Deca, ce fichier définit une classe `Math` offrant cinq méthodes principales :

— `sin(float)` : calcul du sinus

— `cos(float)` : calcul du cosinus

— `asin(float)` : calcul de l'arc sinus

— `atan(float)` : calcul de l'arc tangente

— `ulp(float)` : distance entre deux nombres flottants consécutifs autour d'une valeur donnée

3.1 Utilisation

Pour utiliser l'extension TRIGO, il suffit d'inclure le fichier dans votre programme Deca :

```
#include Math.decah
```

4 Aspects théoriques

4.1 Interprétation des Équations Itératives

À chaque itération i, l'algorithme CORDIC applique la transformation suivante :

$$x_{i+1} = x_i - d_i\, y_i\, 2^{-i} \tag{1}$$

$$y_{i+1} = y_i + d_i\, x_i\, 2^{-i} \tag{2}$$

$$z_{i+1} = z_i - d_i\, \sigma_i \tag{3}$$

Ces équations traduisent une rotation élémentaire du vecteur (x_i, y_i) d'un angle $\pm\sigma_i$, où chaque terme joue un rôle précis :

— x_i et y_i représentent les coordonnées courantes du vecteur dans le plan. Géométriquement, ce vecteur est progressivement rapproché de l'axe des abscisses par une suite de rotations successives.

— Le terme 2^{-i} correspond à une puissance de deux, ce qui permet d'implémenter les multiplications sous forme de simples décalages binaires. La taille de la rotation diminue exponentiellement avec i, assurant une correction de plus en plus fine.

— Le facteur $d_i \in \{-1, +1\}$ détermine le sens de la rotation. Il est choisi comme le signe de l'angle résiduel z_i, de sorte que chaque rotation réduit la valeur absolue de cet angle. Ce choix garantit la convergence de l'algorithme.

— Les équations sur x_{i+1} et y_{i+1} proviennent de l'approximation d'une rotation plane, où les fonctions trigonométriques sont remplacées par des termes proportionnels à 2^{-i}. L'erreur introduite par cette approximation est compensée par l'accumulation contrôlée des rotations suivantes.

— La variable z_i représente l'angle résiduel à annuler. À chaque itération, on soustrait ou ajoute l'angle élémentaire $\sigma_i = \arctan(2^{-i})$, ce qui permet de suivre précisément l'angle total de rotation appliqué au vecteur.

Ainsi, le triplet (x_i, y_i, z_i) évolue de manière cohérente : tandis que le vecteur (x_i, y_i) se rapproche progressivement de l'orientation souhaitée, la variable z_i tend vers zéro, signalant que l'angle cible a été atteint avec la précision permise par la représentation numérique.

4.2 Modes de Fonctionnement : Rotation et Vectorisation

L'algorithme CORDIC peut être utilisé selon deux modes fondamentaux, en fonction de la nature de la fonction à calculer : le *mode rotation* et le *mode vectorisation*. Cette distinction explique les différences structurelles entre les algorithmes de calcul de sin et cos, et ceux de arctan et arcsin.

4.2.1 Mode Rotation : Calcul de sin et cos

Le calcul des fonctions sin et cos s'effectue en *mode rotation*. L'objectif est de faire tourner un vecteur initial connu d'un angle donné z_0, jusqu'à ce que l'angle résiduel tende vers zéro.

On initialise généralement le vecteur par :

$$(x_0, y_0) = (K, 0), \qquad z_0 = \theta$$

où K est le facteur de gain compensant les rotations successives. À chaque itération, le sens de rotation d_i est choisi de manière à réduire l'angle résiduel :

$$d_i = \text{sign}(z_i)$$

À la fin des itérations, la composante x_n converge vers $\cos(\theta)$ et la composante y_n vers $\sin(\theta)$. Le rôle de l'algorithme est donc ici de *transférer l'information angulaire vers les coordonnées du vecteur*, par une suite de rotations de plus en plus fines.

Dans le cas de cos, des ajustements supplémentaires sont nécessaires lors de la réduction d'angle afin de tenir compte des symétries de la fonction, notamment le changement de signe induit par les identités :

$$\cos(\pi - x) = -\cos(x), \qquad \cos(-x) = \cos(x)$$

Ces transformations n'affectent pas la structure du cœur CORDIC, mais uniquement la phase de normalisation de l'angle.

4.2.2 Mode Vectorisation : Calcul de arctan et arcsin

Le calcul des fonctions inverses, telles que arctan et arcsin, s'effectue en *mode vectorisation*. Dans ce cas, l'objectif n'est plus d'annuler un angle donné, mais de faire converger une composante du vecteur vers une valeur cible, tout en accumulant l'angle correspondant.

Pour $\arctan(x)$, on initialise le vecteur par :

$$(x_0, y_0) = (1, x), \qquad z_0 = 0$$

et le sens de rotation est choisi de manière à réduire la composante verticale y_i :

$$d_i = -\text{sign}(y_i)$$

À la convergence, la composante y_n tend vers zéro et l'angle accumulé z_n converge vers $\arctan(x)$.

Le calcul de $\arcsin(x)$ est plus délicat. Il ne correspond pas directement à une simple vectorisation, mais repose sur une variante du schéma CORDIC visant à faire converger la composante verticale du vecteur vers la valeur cible x, tout en ajustant simultanément l'angle accumulé. Cette approche nécessite des adaptations spécifiques, notamment une double itération par pas et des corrections d'échelle, afin de garantir la convergence dans l'intervalle $[-1, 1]$.

4.2.3 Comparaison Conceptuelle

En résumé, la différence fondamentale entre ces deux familles de fonctions peut être interprétée ainsi :

— Pour sin et cos, l'angle est connu et l'algorithme calcule les coordonnées correspondantes.

— Pour arctan et arcsin, la coordonnée est connue et l'algorithme reconstruit l'angle associé.

Cette dualité illustre la grande flexibilité de l'algorithme CORDIC, capable de traiter des fonctions directes et inverses au moyen d'un même noyau computationnel, simplement en modifiant le critère de décision du sens de rotation.

4.3 Précision Théorique et Convergence

La précision de l'algorithme CORDIC repose sur deux mécanismes complémentaires : la décroissance exponentielle des angles élémentaires σ_i et les propriétés de l'arithmétique en virgule flottante.

À l'itération i, la rotation appliquée est de l'ordre de $\arctan(2^{-i})$, qui se comporte asymptotiquement comme 2^{-i} pour les grandes valeurs de i. Chaque itération apporte donc une correction deux fois plus fine que la précédente. En pratique, après n itérations, l'erreur angulaire résiduelle est majorée par une quantité de l'ordre de :

$$\mathcal{O}(2^{-n})$$

ce qui signifie que le nombre de bits de précision augmente linéairement avec le nombre d'itérations.

Dans un contexte de calcul en simple précision IEEE 754, la mantisse comporte 24 bits significatifs (bit implicite inclus).

La précision finale est donc limitée non par l'algorithme lui-même, mais par le format de représentation des nombres flottants. L'algorithme CORDIC est dit *numériquement stable* dans ce cadre : les erreurs d'arrondi ne s'amplifient pas au fil des itérations, car chaque rotation est de plus en plus petite et contrôlée.

Enfin, le facteur de gain constant K est appliqué une seule fois lors de l'initialisation. Cette stratégie évite l'accumulation d'erreurs multiplicatives et contribue à garantir que les valeurs calculées de sin, cos ou des fonctions inverses atteignent une précision proche de l'ulp (*unit in the last place*) du format flottant utilisé.

5 Optimisations et Adaptations pour IMA

L'implémentation CORDIC pour la machine IMA intègre plusieurs optimisations spécifiques :

5.1 Représentation des Données

Les angles et les coordonnées sont stockés en format virgule flottante simple précision, correspondant aux capacités natives de l'IMA.

5.2 Réduction d'Angle

Avant d'entrer dans la boucle CORDIC principale, l'angle d'entrée est réduit à l'intervalle $[-\pi/2, \pi/2]$ en utilisant les propriétés de symétrie des fonctions trigonométriques. Cette réduction est essentielle pour assurer la convergence de l'algorithme et améliorer la précision des résultats.

5.3 Absence d'Instruction FMA

Contrairement aux méthodes polynomiales, l'algorithme CORDIC ne se prête pas à une fusion d'opérations de type FMA (Fused Multiply-Add). Les optimisations possibles se limitent essentiellement à des améliorations locales, comme la réduction des accès mémoire ou la suppression d'opérations de pile, sans modification significative de la structure de calcul.

6 Spécifications des algorithmes :

6.1 Fonction rapprochePi : Réduction d'Angle (première approche)

```
int rapprochePi(float angle) {
    int i = 0;

    while ((angle < -pi/2) || (angle > pi/2)) {
        if (angle < 0) {
            angle = angle + pi;
        }
        else {
            angle = angle - pi;
        }
        i = i + 1;
    }
    return i;
}
```

6.1.1 Contexte Mathématique

Les fonctions trigonométriques sont périodiques :

$$\sin(\theta + \pi) = -\sin(\theta) \tag{4}$$
$$\cos(\theta + \pi) = -\cos(\theta) \tag{5}$$

Cette propriété permet de ramener tout angle dans l'intervalle $[-\pi/2, \pi/2]$ tout en ajustant correctement le signe du résultat.

6.1.2 Fonctionnement

1. La fonction reçoit un angle en radians
2. Un compteur i est initialisé à zéro
3. Une boucle teste si l'angle est en dehors de $[-\pi/2, \pi/2]$
4. Si l'angle est trop petit, on ajoute π
5. Si l'angle est trop grand, on soustrait π
6. Le compteur i est incrémenté à chaque ajustement
7. La fonction retourne i, utilisé pour corriger le signe final

6.1.3 Exemple

Si l'angle d'entrée vaut $3\pi/4$:

— Premier passage : $3\pi/4 > \pi/2 \rightarrow$ soustraction de $\pi \rightarrow$ angle réduit $= -\pi/4$, compteur $i = 1$

— L'angle est maintenant dans $[-\pi/2, \pi/2]$

— Le sinus final sera multiplié par $(-1)^i = -1$ pour obtenir la valeur correcte

6.2 Gestion précise des erreurs d'arrondi

Le calcul numérique des fonctions trigonométriques repose sur des opérations en virgule flottante, lesquelles introduisent inévitablement des erreurs d'arrondi. En effet, les nombres réels continus ne peuvent pas être représentés exactement sur un nombre fini de bits. Des constantes fondamentales comme π ou 2π ne possèdent pas de représentation exacte en base 2, ce qui implique que toute valeur stockée en mémoire n'est qu'une approximation.

Dans la première approche naïve, ces constantes sont utilisées sous la forme d'un unique nombre flottant lors des opérations de réduction d'angle. Cette simplification masque cependant les erreurs de représentation, qui sont directement injectées dans les calculs lors des multiplications et des soustractions. Lorsque ces opérations impliquent des valeurs proches, notamment lors de la soustraction d'un multiple de π ou de 2π, les erreurs d'arrondi deviennent dominantes et entraînent une perte significative de précision sur le résultat final.

Ces limitations rendent nécessaire l'utilisation de représentations plus fines des constantes mathématiques. En particulier, la décomposition de π en une partie haute et une partie basse permet de corriger explicitement l'erreur introduite par l'approximation flottante principale. La partie basse, bien que de faible amplitude, joue un rôle crucial en réinjectant les bits de précision perdus lors de la réduction grossière, ce qui améliore sensiblement la stabilité numérique du calcul.

Ces erreurs sont généralement faibles lorsqu'elles sont prises isolément, mais elles peuvent devenir significatives lorsque plusieurs opérations arithmétiques sont enchaînées, notamment dans le cadre de la réduction d'angle.

6.2.1 Annulation numérique lors des réductions

Lors de la réduction d'un angle x par soustraction d'un multiple de 2π, on effectue une opération du type :

$$x \leftarrow x - k \cdot 2\pi$$

où k est un entier représentant le nombre de périodes complètes. Lorsque x est grand, les deux termes x et $k \cdot 2\pi$ sont proches en valeur absolue. Leur soustraction provoque alors une *annulation numérique* : les bits de poids fort, qui sont similaires dans les deux opérandes, s'éliminent, et le résultat ne conserve que les bits de poids faible.

Cette perte de bits significatifs entraîne une diminution importante de la précision relative du résultat. Autrement dit, même si x et $k \cdot 2\pi$ sont chacun représentés avec une bonne précision absolue, la valeur réduite peut être fortement bruitée par les erreurs d'arrondi.

6.2.2 Accumulation des erreurs

Le problème est aggravé lorsque la réduction est effectuée en une seule opération avec une constante approchée. En effet, l'erreur contenue dans la représentation flottante de 2π est multipliée par k, puis propagée lors de la soustraction. Plus la valeur de k est grande, plus cette erreur est amplifiée.

Ainsi, une réduction naïve de la forme :

$$x - k \cdot \widetilde{2\pi}$$

où $\widetilde{2\pi}$ est une approximation flottante de 2π, peut conduire à un résultat réduit significativement différent de la valeur mathématiquement correcte.

6.2.3 Décomposition haute et basse des constantes

Pour limiter ces erreurs, les constantes 2π et π sont décomposées en deux parties :

$$2\pi = 2\pi_{\text{HI}} + 2\pi_{\text{LO}}, \quad \pi = \pi_{\text{HI}} + \pi_{\text{LO}}$$

La partie `HI` contient les bits de poids fort et est choisie de manière à être exactement représentable en virgule flottante. La partie `LO`, beaucoup plus petite en valeur absolue, contient le résidu permettant de corriger l'erreur introduite par `HI`.

Lors des soustractions, les opérations sont alors effectuées en deux temps :

$$x \leftarrow x - k \cdot 2\pi_{\text{HI}}$$

$$x \leftarrow x - k \cdot 2\pi_{\text{LO}}$$

Cette stratégie permet de préserver davantage de bits significatifs dans le résultat final, car la soustraction de la partie `HI` effectue la réduction grossière, tandis que la partie `LO` ajuste finement le résultat sans provoquer de nouvelle annulation catastrophique.

6.2.4 Impact sur la précision finale

La gestion rigoureuse des erreurs d'arrondi est essentielle, car le calcul du sinus est particulièrement sensible à la précision de l'argument. Une erreur faible sur l'angle peut entraîner une erreur relative beaucoup plus importante sur la valeur du sinus, notamment lorsque l'angle est proche de points critiques comme 0, $\pi/2$ ou π.

En contrôlant explicitement les erreurs d'arrondi lors de la réduction d'angle, on garantit que l'angle transmis à l'algorithme de calcul final est aussi précis que possible, ce qui améliore directement la fiabilité et la stabilité numérique du résultat.

6.3 Calcul du sinus

Une fois l'angle correctement réduit et normalisé, le calcul du sinus peut être effectué de manière fiable. Cette séparation entre la phase de réduction d'angle et la phase de calcul effectif permet d'améliorer à la fois la précision numérique et la solidité de l'implémentation.

6.3.1 Traitement des cas particuliers

Avant d'exécuter l'algorithme principal, certains cas limites sont traités afin d'éviter des calculs inutiles et de limiter la propagation des erreurs d'arrondi :

```
if (f < 1.0E-10 && f > -1.0E-10) { return 0.0; }
if (f < 0.1 && f > -0.1) { return f - f*f*f/6; }
```

Pour des angles très proches de zéro, le sinus est soit assimilé à zéro, soit approximé par un développement de Taylor à l'ordre 3. Ces approximations sont particulièrement efficaces dans un voisinage réduit et permettent d'éviter l'exécution de l'algorithme itératif.

6.3.2 Réduction modulaire de l'angle

L'angle est ensuite ramené dans une plage bornée par une réduction modulo 2π, réalisée avec des constantes décomposées en parties haute et basse afin de limiter les erreurs d'arrondi :

```
k = (int)(f / TWO_PI_HI);
f = f - ((float)(k) * TWO_PI_HI);
f = f - ((float)(k) * TWO_PI_LO);
```

Cette méthode permet de corriger explicitement l'erreur introduite par la représentation flottante de 2π, et garantit une réduction plus précise qu'une soustraction naïve.

6.3.3 Normalisation dans $[-\pi, \pi]$

Après la réduction modulo 2π, l'angle est normalisé afin d'être contenu dans l'intervalle $[-\pi, \pi]$, ce qui facilite l'exploitation des symétries trigonométriques :

```
if (f > 3.1415927) {
    f = f - TWO_PI_HI;
    f = f - TWO_PI_LO;
} else if (f < -3.1415927) {
    f = f + TWO_PI_HI;
    f = f + TWO_PI_LO;
}
```

Cette étape assure que l'angle reste proche de zéro, condition favorable à la stabilité numérique des calculs suivants.

6.3.4 Réduction dans l'intervalle $[-\pi/2, \pi/2]$

Pour garantir une convergence optimale de l'algorithme de calcul du sinus, l'angle est finalement ramené dans l'intervalle $[-\pi/2, \pi/2]$ en utilisant les symétries de la fonction sinus :

```
if (f > 1.5707963) {
    f = PI_HI - f;
    f = f + PI_LO;
} else if (f < -1.5707963) {
    f = -PI_HI - f;
    f = f - PI_LO;
}
```

L'utilisation des constantes `PI_HI` et `PI_LO` permet de conserver une précision maximale lors de ces transformations.

6.3.5 Calcul final du sinus

Une fois l'angle réduit dans l'intervalle cible, le calcul du sinus est confié à un algorithme itératif spécialisé, appelé via :

```
return this.sinCordic(f);
```

Cet algorithme applique une suite de rotations élémentaires successives, basées sur des constantes pré-calculées, afin d'approcher progressivement la valeur du sinus. Le nombre d'itérations est fixé, ce qui garantit un temps d'exécution déterministe et une précision adaptée à une représentation en simple précision.

6.4 Calcul du cosinus

Le calcul du cosinus repose sur la même structure générale que celui du sinus : une réduction précise de l'angle suivie d'un calcul itératif sur un intervalle restreint. Les étapes de réduction modulo 2π et de normalisation sont strictement identiques et ne sont donc pas redétaillées ici.

La différence principale réside dans la manière dont les symétries trigonométriques sont exploitées ainsi que dans la gestion explicite du signe du résultat final.

6.4.1 Traitement des petits angles

Comme pour le sinus, des optimisations sont appliquées pour les angles proches de zéro, afin d'éviter des calculs itératifs inutiles :

```
if (f < 1.0E-10 && f > -1.0E-10) { return 1.0; }
if (f < 0.01 && f > -0.01) { return 1.0 - f*f/2; }
```

Ces approximations reposent sur le développement de Taylor du cosinus autour de l'origine et fournissent une excellente précision locale.

6.4.2 Réduction finale et gestion du signe

Après la réduction et la normalisation de l'angle dans l'intervalle $[-\pi, \pi]$, l'angle est ramené dans $[-\pi/2, \pi/2]$. Contrairement au sinus, cette opération peut modifier le signe du cosinus, ce qui nécessite un traitement spécifique.

Les identités utilisées sont :

$$\cos(x) = -\cos(\pi - x), \quad \cos(x) = -\cos(-\pi - x)$$

Elles sont implémentées de la manière suivante :

```
if (f > 1.5707963) {
    f = PI_HI - f;
    f = f + PI_LO;
    invertSign = 1;
} else if (f < -1.5707963) {
    f = -PI_HI - f;
    f = f - PI_LO;
    invertSign = 1;
}
```

Un indicateur booléen (`invertSign`) est utilisé pour mémoriser le changement de signe induit par cette transformation, sans perturber la phase de calcul numérique proprement dite.

6.4.3 Calcul itératif du cosinus

Une fois l'angle correctement réduit, le calcul du cosinus est effectué par un algorithme itératif de rotations successives, similaire à celui utilisé pour le sinus. La différence essentielle réside dans la valeur retournée à la fin du processus : la composante horizontale du vecteur est cette fois-ci conservée.

L'appel final est donc conditionné par l'indicateur de signe :

```
if (invertSign == 1) {
    return -this.cosCordic(f);
} else {
    return this.cosCordic(f);
}
```

L'algorithme `cosCordic` effectue un nombre fixe d'itérations et manipule un vecteur initial normalisé. À l'issue des rotations, la composante x du vecteur correspond à une approximation du cosinus de l'angle réduit, garantissant une précision suffisante en simple précision.

Cette approche permet de mutualiser la logique de réduction d'angle avec celle du sinus, tout en traitant explicitement les particularités algébriques propres au cosinus.

6.5 Calcul de l'arctangente

Le calcul de l'arctangente repose sur un algorithme itératif similaire à celui utilisé pour le sinus et le cosinus, mais dans un mode différent. Contrairement aux fonctions trigonométriques directes, l'arctangente est calculée ici à partir de la valeur de son argument, en déterminant progressivement l'angle correspondant.

L'implémentation repose sur un algorithme de type CORDIC en mode *vectoring*, dont l'objectif est de faire converger la composante verticale d'un vecteur vers zéro tout en accumulant l'angle correspondant.

6.5.1 Gestion du signe

La fonction $\arctan(x)$ étant impaire, le signe de l'argument peut être traité séparément afin de simplifier le calcul. L'algorithme travaille uniquement avec des valeurs positives, le signe étant réappliqué à la fin :

```
LOAD R1, R7
SUB #0.0, R7
BLT atan_neg_sign

atan_neg_sign:
    OPP R1, R1
    LOAD #1, R9
```

Un indicateur (`sign`) permet de mémoriser cette transformation sans impacter la phase de calcul principale.

6.5.2 Traitement des grandes valeurs

Afin d'améliorer la précision numérique pour les grandes valeurs de $|x|$, une transformation est appliquée lorsque $|x| > 1$:

$$\arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right)$$

Cette identité permet de ramener l'argument dans un intervalle favorable à la convergence de l'algorithme :

```
LOAD R1, R7
SUB #1.0, R7
BGT atan_do_inverse

atan_do_inverse:
    LOAD #1.0, R2
    DIV R1, R2
    LOAD R2, R1
    LOAD #1, R10
```

Un second indicateur (`inverse`) est utilisé pour ajuster le résultat final après convergence.

6.5.3 Initialisation du calcul itératif

L'algorithme est initialisé avec un vecteur (x, y) dont la composante verticale correspond à l'argument traité. L'angle accumulé est initialisé à zéro :

```
LOAD #1.0, R2      ; x
LOAD R1, R3        ; y
LOAD #0.0, R4      ; z
```

```
LOAD #0, R5         ; i
LOAD #1.0, R6       ; p
```

Ces valeurs constituent le point de départ des rotations successives.

6.5.4 Boucle de convergence

À chaque itération, le signe de la composante verticale du vecteur est testé afin de déterminer le sens de la rotation. L'angle correspondant est alors ajouté ou soustrait à l'angle accumulé :

```
LOAD R3, R7
SUB #0.0, R7
BLT atan_neg
BRA atan_pos
```

Les rotations successives permettent de réduire progressivement la composante verticale du vecteur, tandis que l'angle accumulé converge vers $\arctan(x)$.

6.5.5 Correction finale

Si l'argument a été inversé au début du calcul, l'angle obtenu est corrigé à l'aide de l'identité trigonométrique correspondante :

```
CMP #0, R10
BEQ atan_sign

LOAD #1.57079632679, R7
SUB R4, R7
LOAD R7, R4
```

Enfin, le signe initial est réappliqué si nécessaire :

```
CMP #0, R9
BEQ atan_return
OPP R4, R4
```

La valeur finale contenue dans le registre de l'angle est alors retournée comme résultat de la fonction.

Cette implémentation permet de calculer l'arctangente avec une précision suffisante en simple précision, tout en conservant une structure cohérente avec les fonctions trigonométriques précédemment définies.

6.6 Calcul de l'arcsinus

Le calcul de l'arcsinus repose également sur un algorithme de type CORDIC, mais en mode *vectoring inverse* : l'objectif est cette fois-ci de trouver l'angle dont le sinus vaut la valeur donnée. Cette approche diffère légèrement de celle utilisée pour le sinus et le cosinus, qui calculent directement le résultat à partir d'un angle.

6.6.1 Initialisation

L'entrée de la fonction est la valeur val pour laquelle on souhaite calculer l'arcsinus. Le vecteur de départ est initialisé avec une composante horizontale normalisée à 1, tandis que la composante verticale est initialisée à zéro. L'angle accumulé est également initialisé à zéro :

```
LOAD #1.0, R2          ; xn = 1.0
LOAD #0.0, R3          ; yn = 0.0
LOAD #0.0, R4          ; zn = 0.0
LOAD #0, R6            ; i = 0
LOAD #1.0, R7          ; exp2 = 1.0
```

Cette configuration permet de transformer progressivement le vecteur pour que sa composante verticale converge vers la valeur d'entrée.

6.6.2 Gestion du signe et direction de rotation

Contrairement aux fonctions précédentes, l'arcsinus doit faire converger le vecteur vers une valeur cible spécifique (la valeur d'entrée). À chaque itération, le signe de la rotation est déterminé en fonction de la différence entre la composante verticale actuelle et la valeur cible :

```
CMP R5, R3
BLE rot_start
OPP R8, R8
```

Cette logique garantit que le vecteur se rapproche progressivement de la position correspondant à l'angle dont le sinus vaut val.

6.6.3 Rotations successives

Les rotations sont appliquées deux fois par itération pour accélérer la convergence, chaque rotation ajustant simultanément les composantes x et y du vecteur, ainsi que l'angle accumulé z :

```
; xn = xn - sign * yn * exp2
; yn = yn + sign * xn_old * exp2
; zn = zn + sign * sigma[i]
```

Cette double rotation permet d'améliorer la précision pour un nombre limité d'itérations.

6.6.4 Correction d'échelle

Après chaque double rotation, une correction de la valeur cible est appliquée pour compenser la réduction progressive de l'échelle :

```
wn = wn + wn * exp2 * exp2
exp2 = exp2 / 2
```

Cette étape est spécifique à l'arcsinus et n'apparaît pas dans le calcul direct de sinus ou de cosinus.

6.6.5 Résultat final

Après l'ensemble des itérations, l'angle accumulé dans z correspond à l'arcsinus de la valeur d'entrée. Cette valeur est retournée par la fonction :

```
LOAD R4, R0            ; zn contient le résultat
```

6.6.6 Différences principales par rapport à sin et cos

— L'arcsinus calcule un angle à partir d'une valeur donnée, tandis que sinus et cosinus évaluent directement la fonction trigonométrique pour un angle donné.

— La direction des rotations dépend de la comparaison avec la valeur cible (composante verticale) plutôt que du signe de l'angle.

— Une correction d'échelle de la valeur cible est appliquée à chaque itération pour maintenir la convergence, ce qui n'est pas nécessaire pour sin ou cos.

— Le vecteur de départ est fixé à $(1, 0)$ pour normaliser la magnitude, alors que pour sin/cos, le vecteur initial correspond à l'angle lui-même.

Cette méthode permet de calculer l'arcsinus avec une précision suffisante en simple précision, tout en conservant une logique cohérente avec les autres fonctions trigonométriques implémentées.

6.7 Calcul de l'ULP (Unit in the Last Place)

La précision des fonctions trigonométriques implémentées dépend directement de la représentation flottante des nombres. L'*ULP* (*Unit in the Last Place*) est une mesure fondamentale qui correspond à la distance entre un nombre flottant donné et le nombre flottant le plus proche représentable en mémoire. Cette valeur est cruciale pour évaluer et contrôler les erreurs d'arrondi dans les calculs numériques.

La fonction `ulp(float f)` calcule l'ULP d'un nombre flottant f selon sa magnitude :

```
float ulp(float f) {
    float borneInf = 8388608;
    float borneSup = 16777216;
    float ulp = 1;
    if (f < 0) {
        f = -f;
    }
    if (f == 0) {
        return (float) (1.4E-45);
    }
    else if (f >= borneSup) {
        while (f >= borneSup) {
            borneInf = borneInf * 2;
            borneSup = borneSup * 2;
            ulp = ulp * 2;
        }
        return ulp;
    }
    else if (f < borneInf) {
        while (f < borneInf) {
            borneInf = borneInf / 2;
            ulp = ulp / 2;
        }
        return ulp;
    }
    else {
        return ulp;
    }
}
```

6.7.1 Explication de l'algorithme

— La fonction commence par travailler sur la valeur absolue de f, car l'ULP ne dépend pas du signe.

— Pour $f = 0$, elle renvoie la plus petite valeur positive représentable (1.4×10^{-45} en simple précision IEEE 754).

— Pour des nombres très grands ($f \geq$ borneSup), l'ULP est multiplié par 2 à chaque étape, ce qui reflète l'augmentation de l'espacement entre nombres flottants à mesure que l'exposant croît.

— Pour des nombres très petits ($f <$ borneInf), l'ULP est divisé par 2 à chaque étape, correspondant à la diminution de l'espacement pour les petites valeurs.

— Pour les nombres situés entre `borneInf` et `borneSup`, l'ULP vaut simplement 1 unité, correspondant à la résolution standard en simple précision.

6.7.2 Lien avec les fonctions trigonométriques

L'ULP est utilisée pour évaluer la précision des résultats des fonctions `sin`, `cos`, `atan` et `asin`. Elle permet notamment de :

— Déterminer la sensibilité aux erreurs d'arrondi lors des réductions d'angle et des rotations CORDIC.

— Vérifier que les approximations (par exemple le développement de Taylor pour les petits angles) restent dans la précision représentable par la machine.

— Comparer le résultat calculé avec la valeur exacte attendue en termes d'unités de l'ULP, ce qui fournit une métrique standardisée de l'erreur numérique.

Ainsi, la fonction `ulp` est un outil essentiel pour la **validation et le contrôle de précision** de toute bibliothèque mathématique en virgule flottante.

7 Gestion des Cas Particuliers et Erreurs

7.1 Vérification des Domaines d'Entrée

Pour la fonction `asin`, une vérification préalable s'assure que l'argument fourni se trouve bien dans l'intervalle $[-1, 1]$. Toute valeur en dehors de catte intervalle ne déclenchera pas une erreur, au lieu de ça on peut faire println de message d'erreur pour tester.

7.2 Réduction des Grands Angles

Pour les fonctions `sin` et `cos`, les angles d'entrée très grands sont automatiquement réduits à l'intervalle $[-\pi, \pi]$, grâce à la périodicité naturelle des fonctions trigonométriques.

7.3 Contrôle de la Précision Numérique

La précision des calculs est surveillée par plusieurs mécanismes :

— **Nombre d'itérations CORDIC** : 24 itérations pour atteindre une précision supérieure à celle de la mantisse des flottants simples

— **Tests exhaustifs** : vérification sur l'ensemble du domaine de définition

— **Fonction ulp** : mesure de la précision en unités de dernier rang (Unit in the Last Place)

8 Performance et Utilisation des Ressources

8.1 Analyse des Temps d'Exécution

Les performances observées dépendent directement de la structure itérative de l'algorithme CORDIC :

Fonction	Cycles	Commentaire
`sin/cos`	∼500	Mode rotation, 24 itérations
`atan`	∼600	Mode vectoring, tests supplémentaires
`asin`	∼700	Calcul de racine + atan
`ulp`	∼100	Manipulation directe du flottant

TABLE 1 – Temps d'exécution estimés (en cycles IMA)

8.2 Structure d'une Itération CORDIC

Chaque itération comprend :

— Un test de signe sur l'angle résiduel

— Deux additions/soustractions sur les composantes du vecteur

— Une mise à jour de l'angle résiduel

— Un décalage arithmétique correspondant à la division par 2^{-i}

Sur la machine IMA, ce corps de boucle représente environ 15 à 18 instructions, toutes exécutées à partir de registres, sans accès mémoire.

9 Compatibilité et Évolutivité

L'extension TRIGO a été conçue avec la compatibilité et l'évolutivité à l'esprit :

— Entièrement compatible avec le modèle objet de Deca

— Architecture modulaire permettant l'ajout de nouvelles fonctions

— Système de détection extensible à d'autres bibliothèques

— Le générateur CORDIC peut être étendu pour d'autres fonctions transcendantes (exponentielle, logarithme)

10 Tests de Validation

10.1 Tests pour des Valeurs Usuelles

Cette sous-section présente une série de tests visant à valider le bon fonctionnement des fonctions trigonométriques implémentées sur des valeurs usuelles, pour lesquelles les résultats théoriques sont bien connus. Ces tests constituent une première étape essentielle de validation, permettant de vérifier à la fois la correction fonctionnelle des algorithmes et leur précision numérique.

Chaque test consiste à comparer le résultat fourni par l'implémentation à une valeur de référence, puis à mesurer l'erreur absolue associée. Afin de tenir compte des erreurs d'arrondi inhérentes à l'arithmétique en virgule flottante, la comparaison est effectuée à l'aide d'une tolérance exprimée en unités de dernier rang (ULP).

Les fonctions sin et cos sont évaluées sur des angles remarquables tels que 0, $\frac{\pi}{6}$ et $\frac{\pi}{3}$. Ces cas permettent de vérifier le comportement au voisinage de zéro, ainsi que la bonne prise en compte des symétries et des réductions d'angle utilisées dans l'implémentation.

La fonction arctan est testée sur une large plage de valeurs, allant de cas simples comme $\arctan(1) = \frac{\pi}{4}$ jusqu'à des valeurs très grandes, pour lesquelles le résultat doit converger vers $\frac{\pi}{2}$. Ces tests valident notamment la gestion des grandes valeurs par le calcul de l'inverse et la convergence du mode vectorisation du CORDIC.

Enfin, la fonction arcsin est évaluée sur une valeur usuelle, $\arcsin\left(\frac{1}{2}\right) = \frac{\pi}{6}$, afin de vérifier la reconstruction correcte de l'angle à partir d'une valeur trigonométrique donnée.

L'ensemble des tests passe avec succès : les écarts mesurés restent systématiquement inférieurs aux tolérances définies, exprimées en multiples d'ULP. Ces résultats confirment la cohérence des implémentations et la maîtrise des erreurs numériques pour des cas représentatifs.

10.2 Test de Cohérence Fonctionnelle

Un test de cohérence a également été mis en place afin de vérifier la compatibilité entre les fonctions sin et arcsin. L'identité mathématique suivante est utilisée :

$$\sin(\arcsin(x)) = x \quad \text{pour } x \in [-1, 1]$$

Ce test consiste à parcourir l'intervalle $[-1, 1]$ avec un pas constant, à calculer successivement $\arcsin(x)$ puis $\sin(\arcsin(x))$, et à comparer le résultat obtenu à la valeur initiale x.

10.2.1 Code du Test

```
#include "Math.decah"
{
    Math m = new Math();
    float x;
    float y;
    float eps = 1.0E-5;

    x = -1.0;
    while (x <= 1.0) {
        y = m.sin(m.asin(x));
        println("x=" ,x ,
                " sin(asin(x))=" , y ,
                " erreur=" , (y - x));
        x = x + 0.1;
    }
}
```

10.2.2 Analyse des Résultats

Les erreurs observées sont en moyenne de l'ordre de 10^{-7}, soit de l'ordre 2^{-24} établie théoriquement et qui peut être observé pour par printlnx et printx. Ce niveau d'erreur est cohérent avec les limites de la représentation en virgule flottante simple précision (IEEE 754), pour laquelle l'unité de dernier rang autour de 1 vaut environ :

$$\text{ulp}(1) \approx 1.19 \times 10^{-7}$$

L'erreur mesurée correspond donc à un écart de l'ordre de une à quelques ULP, ce qui est attendu compte tenu :

— des erreurs d'arrondi cumulées ;

— de l'utilisation successive de deux algorithmes CORDIC ;

— des approximations numériques internes.

Ce test de cohérence confirme ainsi que les fonctions sin et arcsin sont compatibles entre elles et que les erreurs numériques restent maîtrisées.

10.3 Tests sur des Grandes Valeurs

Afin de vérifier la robustesse de l'implémentation de sin pour des angles très grands, nous avons utilisé un test basé sur la périodicité du sinus :

$$\sin(x) = \sin(x + 2\pi)$$

10.3.1 Code du Test

```
#include "Math.decah"
{
    Math m = new Math();
    float x;
    float s1;
    float s2;
    float twoPi;

    twoPi = 2 * m.getPi();
    x = 1.0E3;

    while (x <= 1.0E7) {
        s1 = m.sin(x);
        s2 = m.sin(x + twoPi);

        println("x=", x,
                " sin(x)=", s1,
                " sin(x+2pi)=", s2,
                " diff=", (s1 - s2));

        x = x * 10;
    }
}
```

10.3.2 Analyse des Résultats

Les résultats montrent que :

— Pour des valeurs de x allant de 10^3 jusqu'à environ 10^6, la différence $\sin(x) - \sin(x + 2\pi)$ est de l'ordre de 10^{-6}, soit quelques ULP autour de 1, ce qui reste très raisonnable en simple précision.

— Pour $x \approx 10^6$, l'écart augmente et peut atteindre l'ordre de 10^{-2}. Cela est dû à la **réduction d'argument modulo 2π**, qui devient moins précise lorsque x est très grand. En effet, la différence 2π est très petite par rapport à x, et le flottant simple précision ne peut pas la représenter exactement, ce qui engendre une perte de précision.

— Pour $x \gtrsim 10^7$, , la précision semble se rétablir et les différences reviennent à quelques ULP. Ce phénomène est lié à l'**ULP croissant des nombres flottants pour des grandes valeurs** : lorsque x devient énorme, l'ajout de 2π n'affecte plus la représentation en mémoire, et la réduction d'argument se comporte à nouveau correctement. On peut interpréter cela comme un "rebond" de la précision dû au pas discret de l'approximation flottante.

Ces observations illustrent une limitation classique des fonctions trigonométriques en virgule flottante : la précision est optimale pour des angles modérés, mais la réduction d'argument devient le facteur limitant pour des angles très grands. Néanmoins, même pour des nombres énormes, le sinus reste stable et le comportement périodique est globalement respecté.

10.4 Limitation : Accumulation d'Erreur avec des Multiples de π

Un test a été réalisé pour étudier le comportement du sinus et du cosinus lorsque l'on ajoute un **multiple de π** plus précis. L'idée est de vérifier l'identité périodique :

$$\sin(x + k\pi) = (-1)^k \sin(x), \quad \cos(x + k\pi) = (-1)^k \cos(x)$$

10.4.1 Code du Test

```
#include "Math.decah"
```

```
{
    Math m = new Math();
    float x;
    float s1;
    float s2;
    float c1;
    float c2;
    float pi_hi;
    float pi_lo;
    int k;
    float sign;
    float errSin;
    float errCos;
    float ulpSin;
    float ulpCos;

    pi_hi = 3.1415927;
    pi_lo = -8.742278E-8;

    x = 0.7;
    k = 1;

    while (k <= 10000000) {

        s1 = m.sin(x);
        s2 = m.sin(x + (float)(k) * pi_hi + (float)(k) * pi_lo);

        c1 = m.cos(x);
        c2 = m.cos(x + (float)(k) * pi_hi + (float)(k) * pi_lo);

        if ((k % 2) == 0) {
            sign = 1.0;
        } else {
            sign = -1.0;
        }

        errSin = s2 - sign * s1;
        errCos = c2 - sign * c1;

        ulpSin = m.ulp(sign * s1);
        ulpCos = m.ulp(sign * c1);

        println("k=", k,
                " sin(x+k*pi)=", s2,
                " err/ulp=", errSin / ulpSin);

        println("k=", k,
                " cos(x+k*pi)=", c2,
                " err/ulp=", errCos / ulpCos);

        k = k * 10;
    }
}
```

10.4.2 Analyse des Résultats

Ce test met en évidence un **problème classique de précision numérique** :

— Pour des valeurs modestes de k (de l'ordre de 1 à 100), les erreurs restent faibles, de l'ordre de quelques dizaines d'ULP seulement. La représentation double-partagée $\pi_{\text{hi}} + \pi_{\text{lo}}$ permet alors de conserver une bonne précision.

— Lorsque k devient très grand (au-delà de 1000 et jusqu'à 10^7), l'**imprécision de π** se cumule avec chaque multiplication par k. Chaque terme $k \cdot (\pi_{\text{hi}} + \pi_{\text{lo}})$ introduit une petite erreur, qui s'additionne et devient significative.

— Ce phénomène se traduit par une **augmentation de l'écart relatif** entre $\sin(x + k\pi)$ et $(-1)^k \sin(x)$, ou entre $\cos(x + k\pi)$ et $(-1)^k \cos(x)$.

— En pratique, cette limitation est due à la **représentation finie de π en simple précision** : au-delà d'un certain k, la réduction d'argument n'est plus exacte et l'erreur se propage.

Conclusion : Bien que l'utilisation de π_{hi} et π_{lo} permette d'obtenir une grande précision pour des multiples faibles de π, **la précision se dégrade progressivement lorsque k augmente**, illustrant une limite intrinsèque de la représentation flottante.

11 Conclusion

L'extension TRIGO représente une implémentation solide et efficace des fonctions trigonométriques pour le compilateur Deca et la machine IMA. En s'appuyant sur l'algorithme CORDIC, elle offre un excellent compromis entre précision mathématique, performance d'exécution et simplicité d'utilisation. L'architecture modulaire et les choix d'implémentation garantissent une intégration harmonieuse dans l'écosystème Deca tout en préservant la possibilité d'extensions futures.

12 Bibliographie

Références

[1] J. E. Volder, *The CORDIC Trigonometric Computing Technique*, IRE Transactions on Electronic Computers, vol. EC-8, no. 3, pp. 330–334, 1959.

[2] J. S. Walther, *A Unified Algorithm for Elementary Functions*, Proceedings of the Spring Joint Computer Conference, pp. 379–385, 1971.

[3] R. Andraka, *A Survey of CORDIC Algorithms for FPGA Based Computers*, Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, 1998.

[4] IEEE Standards Committee, *IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008)*, IEEE Computer Society, 2008.

[5] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, 2nd edition, 2002.

[6] J.-M. Muller et al., *Handbook of Floating-Point Arithmetic*, Birkhäuser, 2010.

[7] Ulrich Drepper, *How to Write Shared Libraries*, GNU Project — sections on mathematical libraries and argument reduction.

[8] W. J. Cody, W. Waite, *Software Manual for the Elementary Functions*, Prentice Hall, 1980.