Réduction d'argument basée sur les triplets pythagoriciens pour l'évaluation de fonctions trigonométriques

Hugues De Lassus Saint-Geniès, David Defour, Guillaume Revy

Compas'2015 : Parallélisme/ Architecture / Système/ Répartition
Lille, France, du 30 juin au 3 juillet 2015

Réduction d'argument basée sur les triplets pythagoriciens pour l'évaluation de fonctions trigonométriques

Hugues de Lassus Saint-Geniès, David Defour et Guillaume Revy

Université de Perpignan Via Domitia, DALI, F-66860, Perpignan, France
Université Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France
CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France

hugues.de-lassus@univ-perp.fr, david.defour@univ-perp.fr, guillaume.revy@univ-perp.fr

Résumé
L'évaluation logicielle de fonctions élémentaires se décompose généralement en trois grandes étapes : une réduction d'argument, une évaluation polynomiale et une reconstruction. Afin de garantir la précision du résultat final, ces évaluations s'accompagnent d'une étude rigoureuse des erreurs de méthode et d'arrondi. Une des grandes problématiques consiste à minimiser le nombre de sources d'erreur et/ou leur impact sur le résultat final. Le travail présenté dans cet article s'inscrit dans cet objectif en éliminant une des sources d'erreur dans le schéma d'évaluation de fonctions trigonométriques. Pour cela, nous montrons comment construire des données tabulées sans erreur utilisées lors de la réduction d'argument pour l'évaluation du sinus et du cosinus. La méthode proposée repose sur les générateurs de triplets pythagoriciens. Elle permet de diminuer la taille des tables tout en rendant les données tabulées plus précises.

Mots-clés : réduction d'argument, triplet pythagoricien, fonctions trigonométriques.

1. Introduction

Les formats et l'arithmétique des nombres flottants sont définis par la norme IEEE 754-2008 [10]. Pour les cinq opérations arithmétiques de base ($+$, $-$, $\times$, $/$, et $\sqrt{}$), ce standard requiert *l'arrondi correct* du résultat selon l'un des quatre modes d'arrondi (au plus près, vers $-\infty$, vers $+\infty$, vers 0). Cependant, pour les fonctions élémentaires (e.g. sin, log, exp), l'arrondi correct est seulement recommandé. Cela est dû au *Dilemme du Fabricant de Tables* [13, Ch. 12] et à la difficulté de développer des schémas d'évaluation efficaces pour ces fonctions.
Il existe différentes façons d'évaluer une fonction élémentaire comme sinus ou cosinus [5, 6, 7, 13, 18]. Le choix d'une méthode est guidé par la précision cible, la possibilité d'utiliser du matériel dédié, et le budget ressource associé. Dans cet article, nous nous intéressons aux méthodes d'évaluation logicielle de fonctions trigonométriques à base de valeurs tabulées et d'approximations polynomiales. L'évaluation de ces fonctions suit généralement deux phases, comme dans le projet CR-Libm. [1] Une première phase *rapide* utilise des opérations d'une précision similaire à celle des opérandes pour garantir quelques bits supplémentaires par rapport à la précision visée. Si l'arrondi correct ne peut pas être déterminé après cette première phase, une phase *lente* basée sur une précision étendue est utilisée.

1. Accessible à l'adresse `http://lipforge.ens-lyon.fr/www/crlibm/`.

Dans la suite de cet article nous considérons l'évaluation de la fonction $\sin(x)$, avec x un nombre flottant défini dans [10]. L'évaluation logicielle de cette fonction se décompose en quatre étapes :

1. Une *première* réduction d'argument basée sur l'identité :

$$\sin(x) = f_k\,(x - k\cdot\pi/2)\,, \quad \text{avec} \quad k\in\mathbb{Z} \quad \text{et} \quad f_k = \pm\sin \quad \text{ou} \quad f_k = \pm\cos$$

   selon la valeur de $k \mod 4$, qui permet de réduire l'intervalle à $(-\pi/4, \pi/4]$, puis, en utilisant la parité des fonctions trigonométriques, à $x^* \in [0, \pi/4]$. Notons que, $\pi/2$ étant irrationnel, l'argument réduit x^* l'est également. La précision de l'argument réduit x^* conditionnant la précision du résultat final, il est alors indispensable de réaliser cette étape en utilisant un algorithme adapté [2, 3, 4, 14]. L'argument réduit pourra donc éventuellement être représenté par plusieurs nombres flottants dont la somme fournit une bonne approximation de x^*. On trouve ici une *première source d'erreur* du processus d'évaluation.

2. Cet intervalle est encore trop grand pour approcher $f_k(x^*)$ par une évaluation polynomiale précise et de degré raisonnable. Une *deuxième* réduction d'argument, basée cette fois sur des données tabulées, est nécessaire. Cette solution a été proposée par Tang [17]. Elle consiste à diviser l'argument réduit x^* en deux parties x_h^* et x_l^*, définies par :

$$x_h^* = \lfloor x^* \times 2^p \rfloor \cdot 2^{-p} \quad \text{et} \quad x_l^* = x^* - x_h^*. \tag{1}$$

   La partie haute x_h^* est constituée des p bits de poids forts de x^*, dont on se sert pour adresser une table de $\lceil \pi/4 \times 2^p \rceil$ entrées contenant des valeurs précalculées $sin_h \approx \sin(x_h^*)$ et $cos_h \approx \cos(x_h^*)$ et arrondies dans le format de destination (simple, double, double-double). Selon qu'il faut évaluer $\pm\sin(x^*)$ ou $\pm\cos(x^*)$, on utilise l'une des deux formules suivantes :

$$\begin{aligned} \sin(x^*) &= \sin(x_h^* + x_l^*) = sin_h \cdot \cos(x_l^*) + cos_h \cdot \sin(x_l^*) \\ \cos(x^*) &= \cos(x_h^* + x_l^*) = cos_h \cdot \cos(x_l^*) - sin_h \cdot \sin(x_l^*). \end{aligned} \tag{2}$$

   Cette étape introduit une *deuxième source d'erreur* sur les termes sin_h et cos_h.

3. Il reste a réaliser les approximations polynomiales de $\sin(x_l^*)$ et $\cos(x_l^*)$ pour $x_l^* \in [0, 2^{-p}]$. Ceci introduit une *troisième source d'erreur*.

4. Enfin, le résultat final est obtenu par une *reconstruction* qui fusionne les termes obtenus dans les étapes 2 et 3 selon une des Équations (2). Inévitablement, cette étape rajoute une source d'erreur par l'emploi des opérations élémentaires $+$ et $\times$.

Il existe déjà des solutions satisfaisantes pour la première réduction d'argument, la génération et l'évaluation de polynômes d'approximation précis et efficaces, et la reconstruction [12].
La deuxième réduction d'argument a fait l'objet de propositions où l'objectif était d'équilibrer la première source d'erreur présente dans l'argument réduit et la deuxième source d'erreur présente dans les données tabulées. La méthode de Gal réduit les erreurs d'arrondis des valeurs tabulées, en autorisant un *terme correctif* $corr$ dans les x_h^* [8, 9]. Pour chaque valeur de x_h^*, la variation $corr$ est choisie de sorte que $\cos(x_h^* + corr)$ et $\sin(x_h^* + corr)$ soient très proches de nombres représentables en machine. Les données tabulées sont ainsi plus précises, ce qui simplifie les calculs intermédiaires. En revanche, il est nécessaire de stocker les termes correctifs. La recherche des valeurs $corr$ a été améliorée par Stehlé et Zimmermann [16].
Dans cet article, nous montrons comment aller plus loin et éliminer complètement la deuxième source d'erreur en tabulant des nombres entiers représentables en flottant IEEE, qui donnent

accès à des valeurs rationnelles proches de $\cos(x_h^*)$ et $\sin(x_h^*)$. Cela simplifie l'étape de reconstruction ainsi que la construction des preuves, et réduit l'espace mémoire nécessaire pour les tables. Pour arriver à ces améliorations, nous nous appuyons sur les triplets pythagoriciens, car ils permettent d'obtenir des valeurs rationnelles de sinus et cosinus.
Le reste de cet article est organisé comme suit : La Section 2 détaille notre approche et comment nous parvenons à représenter certains termes sans erreur. Puis, la Section 3 illustre l'intérêt de cette approche à travers un ensemble de résultats expérimentaux, et montre que nous pouvons calculer des tables jusqu'à 10 bits d'indice avec des coûts limités en temps et en mémoire. Enfin, la Section 4 montre que notre méthode permet des gains en mémoire et en latence par rapport aux méthodes existantes.

2. Tabulation de termes exacts

Dans cette section, nous présentons une méthode proche de celle de Gal, basée sur un terme correctif de x_h^*. Notre objectif est toutefois de pouvoir sélectionner des points pour lesquels il est possible de tabuler des valeurs *exactes*. Nous lions leur recherche aux *triplets pythagoriciens*. La suite de la section présente le principe de notre approche, ainsi que les triplets pythagoriciens et la méthode proposée pour sélectionner les triplets pertinents pour la construction des tables.

2.1. Principe de la méthode

Notre approche consiste à construire des tables dont certaines valeurs sont stockées exactement. Plus particulièrement, rappelons qu'une table contient $\lceil \pi/4 \times 2^p \rceil$ entrées, et pour chaque entrée, les valeurs $\sin_h$ et $\cos_h$ doivent pouvoir être stockées sans erreur. Pour ce faire :

1. Les valeurs de $\sin_h$ et $\cos_h$ doivent être des nombres rationnels, c'est-à-dire :
$$\sin_h = s_n/s_d \quad \text{et} \quad \cos_h = c_n/c_d \quad \text{avec} \quad s_n, s_d, c_n, c_d \in \mathbb{N} \quad \text{et} \quad s_d, c_d \neq 0.$$
2. Pour faciliter la reconstruction, les dénominateurs doivent être identiques : $s_d = c_d$.
3. Pour éviter une division par s_d durant la reconstruction, le dénominateur s_d doit être identique à une valeur k pour chaque entrée, de telle sorte qu'il puisse être directement intégré dans les coefficients des polynômes. Cette valeur k peut alors être vue comme le *plus petit commun multiple* (PPCM) des dénominateurs de toutes les entrées.
4. Pour réduire la taille des tables, s_n et c_n doivent être représentables en machine.

Après avoir trouvé des nombres qui vérifient ces quatre propriétés, la reconstruction devient :
$$\sin(x^*) = s_n \cdot C(x_l^* - \mathrm{corr}) + c_n \cdot S(x_l^* - \mathrm{corr}),$$
avec :

— $C(x)$ et $S(x)$ deux polynômes d'approximation définis pour $x \in [0, 2^{-p}]$ par :
$$C(x) = \cos(x)/k \quad \text{et} \quad S(x) = \sin(x)/k,$$

— et les valeurs tabulées s_n, c_n et corr définies par :
$$x_h^* = \arcsin(s_n/k) - \mathrm{corr} = \arccos(c_n/k) - \mathrm{corr}.$$

Les valeurs $\sin_h$ et $\cos_h$ sont alors deux valeurs rationnelles. Mais seuls les deux numérateurs c_n et s_n seront stockés dans la table, les divisions ayant été intégrées à l'évaluation polynomiale.

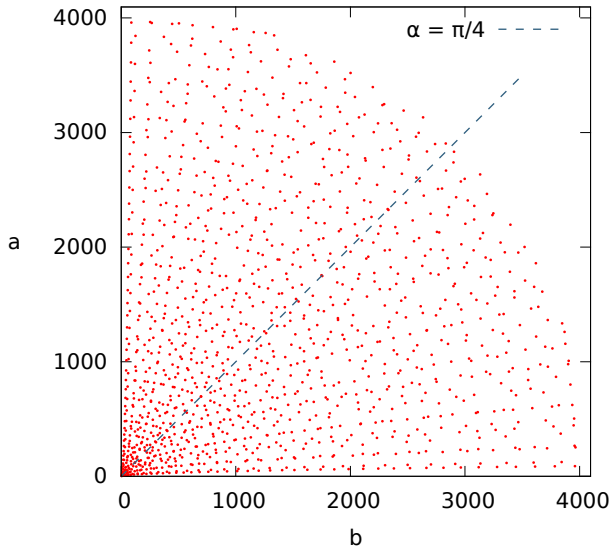

FIGURE 1 – Triplets pythagoriciens primitifs avec $c < 2^{12}$.

2.2. Triplets pythagoriciens

Un *triplet pythagoricien* est un triplet d'entiers (a, b, c), où a, b et c vérifient :

$$a^2 + b^2 = c^2, \quad \text{avec} \quad a, b, c \in \mathbb{N}^*.$$

D'après le théorème de Pythagore, un triplet pythagoricien renvoie aux longueurs des côtés d'un triangle rectangle. Les valeurs de $\sin_h$ et $\cos_h$ peuvent alors être définies comme les quotients de ces longueurs. On en déduit qu'à tout triplet pythagoricien (a, b, c), on peut associer un angle $\theta \in [0, \pi/2]$ tel que $\sin(\theta) = a/c$ et $\sin(\theta) = b/c$.
Un triplet pythagoricien pour lequel les fractions a/c et b/c sont irréductibles est appelé triplet pythagoricien *primitif*. Un triplet primitif et ses multiples renvoient à des triangles semblables, et représentent donc le même angle θ. Par conséquent, nous ne nous intéresserons ici qu'aux triplets pythagoriciens primitifs. Un exemple de triplet primitif est le triplet $(3, 4, 5)$ qui renvoie à l'angle $\theta = \arcsin(3/5) \approx 0.6435$ rad, c'est-à-dire, environ 58°.

2.3. Construction et sélection de l'ensemble des triplets pythagoriciens

Construction. Il existe une infinité de triplets pythagoriciens primitifs, qui couvrent un large intervalle d'angles. Ceci est illustré sur la Figure 1, qui montre tous les triplets pythagoriciens primitifs dont l'hypoténuse est strictement inférieure à 2^{12}. L'ensemble des triplets primitifs possède une structure d'arbre ternaire [1, 15]. Ainsi, l'arbre de Barning-Hall [1] se construit à partir d'un triplet (a, b, c) dont on déduit trois fils en le multipliant par les matrices suivantes :

$$\begin{pmatrix} 1 & -2 & 2 \\ 2 & -1 & 2 \\ 2 & -2 & 3 \end{pmatrix}, \quad \begin{pmatrix} -1 & 2 & 2 \\ 2 & -1 & 2 \\ -2 & 2 & 3 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 3 \end{pmatrix}.$$

Tous les triplets primitifs peuvent donc être générés à partir du triplet $(3, 4, 5)$, en prenant en compte la symétrie entre les triplets (a, b, c) et (b, a, c). Par exemple, après la première itération, on obtient les triplets $(5, 12, 13)$, $(15, 8, 17)$, et $(21, 20, 29)$, et leur symétrique $(12, 5, 13)$, $(8, 15, 17)$, et $(20, 21, 29)$.

Sélection. Parmi les triplets pythagoriciens primitifs, il faut en sélectionner un par entrée de la table, de telle sorte que chaque triplet (a, b, c) minimise le terme correctif $corr$ défini par :

$$corr = \theta - x_h^*, \quad \text{avec} \quad \theta = \arcsin(a/c) \quad \text{et} \quad x_h^* = i \cdot 2^{-p},$$

où i représente l'indice de l'entrée dans la table ($i \geq 0$). Comme on l'a vu en Section 2.1, au lieu de stocker a, b, c et $corr$, notre approche consiste à stocker A et B de la forme

$$A = (a/c) \cdot k \qquad \text{et} \qquad B = (b/c) \cdot k, \quad \text{avec} \quad k \in \mathbb{N}^*$$

et k unique pour toute la table, et tels que A et B soient exactement représentables.

3. Implantation et résultats numériques

Dans cette section, nous présentons les deux méthodes que nous avons utilisées pour générer des tables de valeurs ayant les propriétés décrites en Section 2. Du fait de l'explosion combinatoire lors de la génération des triplets, l'approche *exhaustive* permet d'atteindre en un temps raisonnable des tables indexées par au plus 7 bits. Pour des tailles supérieures, il est nécessaire d'utiliser des méthodes *heuristiques*.

3.1. Recherche exhaustive

3.1.1. Algorithme

Comme nous l'avons vu en Section 2.1, l'objectif est de trouver un petit PPCM k parmi les hypoténuses des triplets générés. Soit p le nombre de bits utilisés pour adresser la table et n le nombre de bits utilisés pour représenter les hypoténuses des triplets stockés. L'algorithme de recherche se divise en trois étapes, où n est initialisé à 4 :

1. On génère tous les triplets (a, b, c), avec $c < 2^n$,
2. On recherche un petit PPCM k, commun à toutes les entrées de la table,
3. Si k est trouvé, on construit la table de valeurs $(A, B, corr)$. Sinon, on recommence l'algorithme avec $n = n + 1$.

Nous avons implanté cet algorithme en C++ et nous avons inclus diverses optimisations pour accélérer les calculs et réduire la consommation mémoire. Par exemple, nous pouvons remarquer que le triplet dégénéré $(0, 1, 1)$ permet de représenter l'angle $\theta = 0°$ avec un terme correctif nul. Nous pouvons le choisir pour l'entrée d'indice 0 dans la table, ce qui nous permet de ne pas avoir à considérer cette entrée et ses triplets associés lors de la phase de génération.

3.1.2. Résultats numériques

La Table 1 présente les résultats obtenus sur une machine Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60 GHz (32 cœurs) et 125 Go de RAM, et sous environnement GNU/Linux. Ces résultats donnent les valeurs n et k_{min} trouvées, les temps d'exécution, ainsi que le nombre de triplets et d'hypoténuses différents stockés.
La Table 1 montre que l'on peut construire des tables indexées par $p = 6$ bits en 6 secondes. Pour $p = 7$ bits d'entrée, on passe alors à 17 minutes. Notons que dans cette dernière configuration, il est nécessaire de manipuler en mémoire plus de cinq millions de triplets et deux millions d'hypoténuses. La solution exhaustive n'est donc pas envisageable pour des valeurs de $p \geq 8$.

3.2. Recherche heuristique

Afin de considérer, en un temps et une consommation mémoire raisonnables, des valeurs p plus grandes ($p \geq 8$), nous avons cherché à caractériser les triplets et les PPCM déjà trouvés

TABLE 1 – Résultats obtenus avec l'approche *exhaustive* pour $p \in \{3, \cdots, 7\}$.

p	n	k_{min}	temps (s)	triplets	hypoténuses
3	10	725	$\ll 1$	181	125
4	14	10 625	0,01	2689	1712
5	17	130 645	0,14	21 588	12 463
6	21	1 676 285	6	338 660	179 632
7	25	32 846 125	1000	5 365 290	2 635 323

TABLE 2 – Résultats obtenus avec l'heuristique proposée pour $p \in \{6, \cdots, 10\}$.

p	k_{min}	n	temps (s)	triplets	hypoténuses
6	1 676 285	21	0.12	1566	43
7	32 846 125	25	0.88	3308	48
8	243 061 325	28	4.63	5217	49
9	12 882 250 225	34	269	10 174	55
10	370 668 520 625	39	8563	14 888	56

pour des p plus petits ($p \leq 7$). On remarque que les PPCM obtenus sont toujours le produit de petits facteurs premiers, et plus particulièrement de petits nombres premiers *pythagoriciens*, c'est-à-dire les nombres premiers de la forme $4n + 1$. Pour $p \leq 7$, ces facteurs appartiennent à l'ensemble $\mathcal{P}$ des premiers pythagoriciens inférieurs ou égaux à 61 :

$$\mathcal{P} = \{5, 13, 17, 29, 37, 41, 53, 61\}.$$

Nous avons alors développé une heuristique qui ne stocke que les triplets pythagoriciens primitifs dont l'hypoténuse est de la forme

$$c = \prod_i p_i^{r_i}, \text{ avec } p_i \in \mathcal{P}, \text{ et } \begin{cases} r_i \in \{0, 1\} & \text{si } p_i \neq 5 \\ r_i \in \mathbb{N}^* & \text{sinon} \end{cases}. \tag{3}$$

Les résultats obtenus par cette heuristique sont consignés dans la Table 2. La dernière colonne correspond au nombre d'hypoténuses sélectionnées. On observe que non seulement cette heuristique a une consommation mémoire nettement inférieure à notre algorithme exhaustif, mais elle retrouve aussi les résultats obtenus par cet algorithme au moins jusqu'à $p = 7$, en des temps largement inférieurs. En effet, pour $p = 7$ par exemple, elle permet de ne stocker que 48 hypoténuses différentes et de générer la table en moins d'une seconde, contre plus de deux millions d'hypoténuses et 17 minutes pour la recherche exhaustive.
Le facteur limitant de notre algorithme n'est donc plus la recherche du PPCM, qui consiste à vérifier seulement quelques dizaines d'hypoténuses, mais la génération des triplets. En effet, la décision de stocker ou non un triplet (a, b, c) est coûteuse : il faut vérifier si c est de la forme (3).

4. Comparaisons avec les méthodes existantes

Nous avons présenté une réduction d'argument basée sur des points exacts ainsi qu'une méthode efficace pour générer ces points. Nous comparons cette solution à la solution originale de Tang [17] et à l'optimisation de Gal [8]. On considère un schéma d'évaluation de la fonction sinus en deux étapes qui cible l'arrondi correct en double précision. La phase rapide et la phase précise ont pour objectifs respectifs des précisions relatives d'au moins 2^{-66} et 2^{-150}. On choisit $p = 10$ pour notre table, ce qui correspond à $\lceil \pi/4 \times 2^{10} \rceil = 805$ entrées.
Pour faciliter la comparaison, on ne considère que le nombre d'accès mémoire (MA) requis par la seconde réduction d'argument et le nombre d'opérations flottantes (FLOP) effectuées

TABLE 3 – Coûts des additions et des multiplications en précision étendue.

Addition	Nb. de FLOP
$E_2 = E_2 + E_2$	12
$E_3 = E_3 + E_3$	27

Multiplication	Nb. de FLOP
$E_2 = E_1 \times E_2$	20
$E_3 = E_1 \times E_3$	47

Multiplication	Nb. de FLOP
$E_2 = E_1 \times E_1$	17
$E_2 = E_2 \times E_2$	26
$E_3 = E_3 \times E_3$	107

pendant l'étape de reconstruction. Aussi, on considère que des algorithmes d'expansion sont utilisés quand une grande précision est requise, comme par exemple dans la bibliothèque CR-Libm. La Table 3, extraite de [11], sera utilisée comme référence pour calculer le coût de ces algorithmes quand aucun FMA n'est disponible. La notation E_n désigne une expansion de taille n. Avec ce formalisme, E_1 représente un nombre flottant habituel.

4.1. Méthode de Tang

Pour atteindre une précision de 66 bits, la solution de Tang doit accéder à des valeurs de sin_h et cos_h stockées en expansions de taille 2. Ces valeurs sont ensuite multipliées par les deux résultats des évaluations polynomiales, considérés comme des expansions de taille 2 également. La phase rapide a donc le coût total suivant : 4 accès mémoire double précision, ainsi que 2 multiplications $E_2 \times E_2$ et 1 addition $E_2 + E_2$, ce qui représente 64 opérations flottantes.
Dans le cas où la phase précise est lancée, cette dernière a besoin d'accéder à 2 valeurs tabulées supplémentaires afin de représenter sin_h et cos_h en expansions de taille 3. La reconstruction consiste alors en 2 multiplications $E_3 \times E_3$ et 1 addition $E_3 + E_3$. Cela représente un total de 6 accès mémoire, 241 opérations flottantes, et une table occupant 38 640 octets.

4.2. Méthode de Gal

La technique utilisée par Gal permet d'atteindre 63 bits de précision, et l'amélioration de Stehlé et Zimmermann atteint les 74 bits. Dans ce cas, en considérant l'approche de Stehlé et Zimmermann, un seul nombre flottant double précision est nécessaire pour les termes sin_h, cos_h et $corr$. La reconstruction consiste à multiplier les deux premiers termes avec les deux résultats des évaluations polynomiales, stockés en expansions de taille 2. Ainsi, la phase rapide requiert 2+1 accès mémoire double précision, 1 addition pour le terme correctif, 2 multiplications $E_1 \times E_2$ et 1 addition $E_2 + E_2$. Le coût total est donc ici de 3 accès mémoire et 53 opérations flottantes.
Les 150 bits de précision requis par la phase précise nécessitent 2 nombres flottants supplémentaires pour chacune des valeurs tabulées. Le terme correctif est alors intégré au résultat final en utilisant une addition $E_3 + E_3$. Le reste des opérations s'effectue lui aussi en expansions de taille 3. Le coût total pour la phase précise est donc de 6 accès mémoires double précision supplémentaires, 2 multiplications $E_3 \times E_3$ et 2 additions $E_3 + E_3$, ce qui représente 9 accès mémoire et 268 opérations flottantes pour une table de 57 960 octets.

4.3. Méthode des points exacts

Comme on peut le remarquer dans la Table 2, la solution proposée dans cet article requiert au plus 39 bits pour stocker les nombres A et B, c'est-à-dire qu'un seul nombre flottant double précision par terme est nécessaire. Le coût de la phase rapide est donc le même que celui de Gal à la Section 4.2. Cependant, pour la phase précise, les valeurs tabulées A et B sont exactes. Donc seul le terme correctif $corr$ est stocké en expansion de taille 3 et requiert 2 accès mémoire supplémentaires pour atteindre les 150 bits de précision. Le terme correctif est intégré au résultat final par une addition $E_3 + E_3$. Les multiplications correspondent à $E_3 = E_1 \times E_3$ car les résultats des évaluations polynomiales sont stockés en expansion de taille 3. L'addition finale est réalisée en expansion de taille 3. Cela représente un total de 5 accès mémoire et 148 opérations flottantes, pour une taille de table de 32 200 octets.

TABLE 4 – Comparaison du nombre d'accès mémoire (MA), d'opérations flottantes (FLOP), et de la taille des tables pour trois méthodes à base de valeurs tabulées et pour $p = 10$.

Méthode	Phase rapide	Phase précise	Taille de la table (octets)
Tang	4 MA + 64 FLOP	6 MA + 241 FLOP	38640
Gal	**3 MA + 53 FLOP**	9 MA + 268 FLOP	57960
Proposée	**3 MA + 53 FLOP**	**5 MA + 148 FLOP**	**32200**

4.4. Comparaison des résultats

Ces estimations de coûts sont résumées dans la Table 4, qui contient le nombre d'opérations flottantes et d'accès mémoire pour la phase rapide et la phase précise ainsi que la taille de chaque table calculée. On remarque tout d'abord que la réduction d'argument proposée requiert moins de mémoire par entrée dans la table que les autres solutions. Il faut en effet 48 octets par entrée pour la méthode de Tang et 72 octets par entrée pour celle de Gal contre seulement 40 octets par entrée pour la table de points exacts. Cela représente un gain en mémoire d'environ 17% et 44% par rapport à Tang et Gal, respectivement.

Concernant le nombre d'opérations flottantes, notre solution a une performance similaire à celle de Gal pour la phase rapide. Pour la phase précise, on observe un gain d'environ 39% et 45% par rapport à Tang et Gal, respectivement.

Enfin, on remarque que notre solution réduit le nombre d'accès mémoire par rapport à toutes les méthodes étudiées. La phase rapide effectue 3 accès mémoire, tandis que l'approche de Tang en fait 4, ce qui représente une amélioration de 25%. Pour la phase précise, on passe de 6 accès pour Tang et 9 pour Gal à 5 accès pour notre méthode, c'est-à-dire, une amélioration d'environ 17% et 44%, respectivement.

5. Conclusions et perspectives

Dans cet article, nous présentons une nouvelle approche pour implanter la réduction d'argument pour l'évaluation de fonctions trigonométriques basée sur des valeurs tabulées. Notre méthode s'appuie sur les triplets pythagoriciens, ce qui permet de simplifier et d'accélérer l'évaluation de ces fonctions. Nous nous sommes concentrés sur le sinus et le cosinus, mais le concept reste valide pour d'autres fonctions. Comparée à d'autres solutions, la table de *points exacts* élimine l'erreur d'arrondi sur certaines valeurs tabulées, et la concentre dans l'argument réduit sur lequel sont faites les approximations polynomiales. Nous réduisons ainsi jusqu'à 45% la taille des tables, le nombre d'accès mémoire, et le nombre d'opérations flottantes impliquées dans le processus de reconstruction.

Nos recherches futures suivent trois axes : Premièrement, il serait intéressant d'intégrer notre table de points exacts à une implantation complète des fonctions sinus et cosinus pour observer son impact réel. Cela pourrait être réalisé au sein du projet CR-Libm. Deuxièmement, comme on a pu le voir en Section 3.2, les hypoténuses intéressantes sont des nombres composés de petits facteurs premiers pythagoriciens. En approfondissant cette idée, on pourrait caractériser ces hypoténuses et les calculer directement au lieu de générer un énorme ensemble de triplets puis de sélectionner les meilleurs. Une première piste serait d'utiliser l'algorithme de Bresenham. Cette façon de faire pourrait accélérer grandement le processus de génération des tables. Troisièmement, nous nous sommes efforcés de chercher le plus petit commun multiple pour que les valeurs tabulées soient stockées sur le plus petit nombre de bits. Cette propriété est essentielle dans les implantations matérielles, où chaque bit est important. Pour des implantations logicielles, il serait plus intéressant de regarder quelles valeurs tabulées peuvent être stockées dans un format donné (i.e. un nombre flottant double précision) et en même temps minimiser

l'erreur sur les termes correctifs. Cela pourrait mener à des termes correctifs représentables en expansions de taille strictement inférieures à n, et donc économiser des accès mémoire ainsi que les opérations flottantes associées.

Remerciements
Ce travail a été réalisé dans le cadre du projet ANR MetaLibm (ANR-13-INSE-0007).

Bibliographie

1. Barning (F. J. M.). – On Pythagorean and quasi-Pythagorean triangles and a generation process with the help of unimodular matrices. *(Dutch) Math. Centrum Amsterdam Afd. Zuivere Wisk. ZW-001*, 1963.
2. Brisebarre (N.), Defour (D.), Kornerup (P.), Muller (J.-M.) et Revol (N.). – A new range-reduction algorithm. *IEEE Transactions on Computers*, vol. 54, 2005, pp. 331–339.
3. Cody (W. J.). – Implementation and testing of function software. – In *Problems and Methodologies in Mathematical Software Production*, *Lecture Notes in Computer Science*, volume 142, pp. 24–47, 1982.
4. Cody (W. J.) et Waite (W.). – *Software manual for the elementary functions*. – New Jersey, Prentice Hall, 1980.
5. DasSarma (D.) et Matula (D. W.). – Faithful bipartite ROM reciprocal tables. – In *Proc. 12th IEEE Symposium on Computer Arithmetic*, pp. 17–28. IEEE Computer Society, 1995.
6. de Dinechin (F.) et Tisserand (A.). – Multipartite table methods. *IEEE Transactions on Computers*, vol. 54, n3, 2005, pp. 319–330.
7. Defour (D.), de Dinechin (F.) et Muller (J.-M.). – A new scheme for table-based evaluation of functions. – In *36th Asilomar Conference on Signals, Systems, and Computers*, pp. 1608–1613, 2002.
8. Gal (S.). – Computing Elementary Functions : a New Approach for Achieving High Accuracy and Good Performance. – In *Symposium on Accurate Scientific Computations*, pp. 1–16, 1986.
9. Gal (S.) et Bachelis (B.). – An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, vol. 17, 1991, pp. 26–45.
10. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008, pp. 1–70.
11. Lauter (C. Q.). – Basic building blocks for a triple-double intermediate format, 2005.
12. Muller (J.-M.). – *Elementary functions - algorithms and implementation (2. ed.)*. – Birkhäuser, 2006, I–XXII, 1–265p.
13. Muller (J.-M.), Brisebarre (N.), de Dinechin (F.), Jeannerod (C.-P.), Lefèvre (V.), Melquiond (G.), Revol (N.), Stehlé (D.) et Torres (S.). – *Handbook of Floating-Point Arithmetic*. – Birkhäuser Boston, 2010.
14. Payne (M.) et Hanek (R.). – Radian Reduction for Trigonometric Functions. *SIGNUM Newsletter*, vol. 18, 1983, pp. 19–24.
15. Price (H. L.). – The Pythagorean Tree : A New Species, 2008.
16. Stehlé (D.) et Zimmermann (P.). – Gal's Accurate Tables Method Revisited. – In *17th IEEE Symposium on Computer Arithmetic (ARITH'17)*, pp. 257–264, 2005.
17. Tang (P. T. P.). – Table-Lookup Algorithms for Elementary Functions and Their Error Analysis. – In *10th IEEE Symposium on Computer Arithmetic (ARITH'10)*, pp. 232–236, 1991.
18. Ziv (A.). – Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Trans. Math. Software*, vol. 17, 1991, pp. 410–423.