Elementary functions: towards automatically generated, efficient, and vectorizable implementations

Hugues de Lassus Saint-Geniès

▶ To cite this version:

Hugues de Lassus Saint-Geniès. Elementary functions: towards automatically generated, efficient, and vectorizable implementations. Other [cs.OH]. Université de Perpignan, 2018. English. NNT: 2018PERP0010 . tel-01841424

HAL Id: tel-01841424

https://theses.hal.science/tel-01841424v1

Submitted on 17 Jul 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Collège Doctoral du Languedoc-Roussillon

THÈSE

Pour obtenir le grade de Docteur

Délivré par l'**Université de Perpignan Via Domitia**

Préparée au sein de l'école doctorale **305 – Énergie et Environnement**
Et de l'unité de recherche **DALI – LIRMM – CNRS UMR 5506**

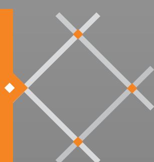
Spécialité: **Informatique**

Présentée par **Hugues de Lassus Saint-Geniès**
hugues.de-lassus@univ-perp.fr

Elementary functions: towards automatically generated, efficient, and vectorizable implementations

Version soumise aux rapporteurs. Jury composé de :

M. Florent de Dinechin	Pr.	INSA Lyon	Rapporteur
Mme Fabienne Jézéquel	MC, HDR	UParis 2	Rapporteur
M. Marc Daumas	Pr.	UPVD	Examinateur
M. Lionel Lacassagne	Pr.	UParis 6	Examinateur
M. Daniel Menard	Pr.	INSA Rennes	Examinateur
M. Éric Petit	Ph.D.	Intel	Examinateur
M. David Defour	MC, HDR	UPVD	Directeur
M. Guillaume Revy	MC	UPVD	Codirecteur



À la mémoire de ma grand-mère Françoise Lapergue
et de Jos Perrot, marin-pêcheur bigouden.

REMERCIEMENTS

Je commencerai bien sûr par remercier mon directeur de thèse David Defour et mon codirecteur Guillaume Revy pour leurs conseils avisés, leur ténacité concernant ma thèse ainsi que leur soutien dans les moments difficiles. Je réalise combien j'ai pu mettre leur patience à rude épreuve ! Grâce à eux, j'ai énormément appris tant scientifiquement que personnellement. Merci aussi à David pour le vélo de course *vintage*. Il m'aura servi presque quotidiennement de moyen de transport urbain ultra-rapide dans la jungle de voitures perpignanaises, mais aussi de fidèle destrier pour gravir mes premiers cols catalans. Je crois qu'il est maintenant temps de lui trouver un nouveau doctorant.

Comment ne pas aussi remercier les rapporteurs de cette thèse, Fabienne Jézéquel et Florent de Dinechin, qui ont porté la lourde charge de relire, corriger et valider ce manuscrit dans le cours temps qui leur était imparti. Leurs retours m'ont permis d'améliorer la qualité de ce manuscrit. Un grand merci également aux examinateurs de cette thèse Marc Daumas, Lionel Lacassagne, Daniel Menard, et Éric Petit, d'avoir accepté de faire partie de mon jury et de s'être intéressés à mes travaux au point de venir de loin au « centre du monde ». En particulier, merci aux membres de mon comité de suivi de thèse, Marc et Florent, de s'être investis pleinement dans ce rôle en offrant de leur temps et un regard externe sur mes travaux, ainsi qu'en étant à l'écoute et en me donnant des conseils utiles.

Je tiens aussi à remercier les membres de l'équipe DALI, actuels ou passés, que j'ai pu côtoyer ces dernières années : Alexis Werey, Amine Najahi, Bernard Goossens, Christophe Negre, David Parello, Jean-Marc Robert, Matthieu Martel, Nasrine Damouche, Philippe Langlois, pour leur soutien ou leur aide à divers moments de ma thèse ; Guillaume Cano pour ses discussions intéressantes ; Chemseddine Chorah pour les soirées (voire les nuits) passées au labo, ses blagues et sa bonne humeur, et j'en passe ; Rafife Nheili pour, entre autres, son accueil à mon arrivée dans l'équipe DALI, pour sa bonne humeur et ses encouragements, pour les dizaines de pauses-café nécessaires à l'aboutissement de cette thèse ; Cathy Porada pour ses encouragements et sa compassion pendant la rédaction ; et enfin Sylvia Munoz pour son aide administrative immense et indispensable, qui a rendu ma vie de doctorant bien plus facile au quotidien.

Je remercie également l'école doctorale de l'UPVD, et plus particulièrement Jocelyne Pla qui a suivi avec bienveillance mon dossier pendant plus de trois ans.

Je souhaite aussi remercier les membres et partenaires du projet ANR MetaLibm pour leur accueil et leur sympathie, et notamment Nicolas Brunie pour sa disponibilité, son enthousiasme et son accueil à Grenoble quand j'ai montré l'envie de travailler sur Metalibm-

lugdunum. C'était un plaisir et une chance de travailler (mais aussi de faire un tour de bicyclette !) avec lui.

Côté « envers du décor », je remercie, sans les nommer, les nombreux enseignants qui m'ont captivé pendant leurs cours, mes maîtres de stages ou collègues à Vérimag et chez Argosim qui m'ont conduit au doctorat, ainsi qu'Inria Learning-Lab pour son MOOC Python mis à profit très rapidement et l'Université Paris-Diderot pour son MOOC OCaml très enrichissant, tous deux suivis grâce à la plateforme FUN-MOOC. Cette thèse ne serait pas non plus la même sans les nombreux logiciels gratuits voire libres que j'ai utilisés. Merci, donc, aux contributeurs des nombreux projets GNU ayant servi cette thèse (GCC, GDB, glibc. . . la liste est trop longue !), mais aussi ceux de LaTeX, Ti*k*Z, gnuplot, Zsh, Git, et Vim, entre autres.

En thèse, malgré ce qu'on peut croire, il existe une vie en dehors du laboratoire ! Cette thèse n'aurait sûrement jamais vu le jour sans le soutien infaillible de proches et amis : les doctorants et anciens doctorants de l'UPVD Rawa El Fallah, Hadjer et Anis Benallal, Félix Authier, Jannine Avila, pour n'en citer que quelques uns, pour leur support au quotidien, le partage de nos difficultés mais aussi de nos succès, ainsi que les soirées films, les jeux, les après-midis au lac de la Raho, et toutes les autres activités qui permettent de changer un peu d'air ; les joueurs de badminton de l'université, notamment Philippe, Brice, Jean-Marc, et Christian ; mes voisins et « voisins par adoption », Louise, Oussama, Margaux, Marion, Jean, Sarah, Laure et Quentin, pour leur disponibilité, les repas, les jeux, et autres excellents moments passés entre amis ; ceux de plus longue date maintenant : les jumeaux Raphaël et Albane, qui sont toujours là quand il faut, Cédric, Thibaut, et Nicolas pour les retrouvailles à Grenoble, Aix-en-Provence, ou ailleurs ; Kévin, Denis, Clément, Mathieu, Jérémy, ainsi que Florian, magicien et « *CEMI*-coloc » pas qu'à moitié, pour les retrouvailles à Montpellier ou à Grenoble ; Charles, pour ses tentatives de m'inscrire à toutes les compétitions auxquelles il participe ; Habib, pour sa disponibilité et son soutien ; Gwénaëlle, pour m'avoir parfaitement « vendu » Grenoble comme objectif de 5/2. La liste pourrait s'allonger encore longtemps, et j'en oublierais très probablement ; aussi je remercierai sans les citer, mes amis du Mexique, de France, et d'ailleurs, afin qu'ils sachent combien leur amitié a pu m'être précieuse tout au long de cette thèse.

Enfin, je souhaite remercier ma famille, ou plutôt *mes* familles. D'abord mes parents Cécile et Jacques, qui n'y comprennent jamais rien (ou font semblant), s'inquiètent parfois, mais m'encouragent toujours ; mes grandes sœurs Raphaële, Edwige, et Marguerite, pour les très bons souvenirs rapportés de week-ends à Londres, Cahors, ou Paris ; Cyril, Simon, et Éric, pour leur second degré, leur enthousiasme, et leur intérêt au sujet de ma thèse, respectivement ; mes neveux et nièces Charlotte, François, Saskia et Jeanne, parfois bruyants mais tellement adorables ; ma marraine Geneviève et mon oncle Jean, pour, entre mille autres choses, leur accueil pour des vacances reposantes. Au passage, merci à Ouided, François, Arnaud et Inès ; mon

parrain Hubert pour ses remèdes bordelais calmant les douleurs post-opératoires mais aux effets secondaires assez troubles ; Isabelle, pour sa porte toujours grande ouverte, la cuisine pour ceux qui ont une *triple dalle*, et j'en passe ; Étienne, aussi pour sa porte grande ouverte, l'initiation au ski de randonnée, et j'en passe également ; Bastien et Mathis, deux frères en or, compagnons des dix premières secondes d'une Étape du Tour, mais heureusement de nombreux autres moments de qualité.

Guénola, tu l'attendais depuis si longtemps : cette thèse s'achève enfin ! Depuis le début tu m'as fait confiance. Malgré la distance qui nous séparait la plupart du temps, tu m'as soutenu, rassuré, poussé à me dépasser. Tu as aussi minutieusement relu tous mes articles écrits dans un anglais obscur à la recherche de la moindre coquille, tu m'as fait prendre l'air aux moments où j'en avais besoin, et tu as encaissé mes sautes d'humeur dans les nombreuses périodes de stress. Pour tout cela et bien plus encore, merci du fond du cœur.

RÉSUMÉ SUBSTANTIEL EN FRANÇAIS

Extrait du guide à l'usage du candidat au Doctorat en vue de la soutenance de thèse (version du 19 février 2018) de l'école doctorale n°305 E² Énergie Environnement de l'Université de Perpignan Via Domitia :

> "La langue des thèses et mémoires dans les établissements publics et privés d'enseignement est le français, sauf exceptions justifiées par les nécessités de l'enseignement des langues et cultures régionales ou étrangères ou lorsque les enseignants sont des professeurs associés ou invités étrangers. La maîtrise de la langue française fait partie des objectifs fondamentaux de l'enseignement. [...] Lorsque cette langue n'est pas le français, la rédaction est complétée par un résumé substantiel en langue française.
>
> ED 305 : une synthèse d'une vingtaine de pages en français, intégrée au manuscrit, mettant en évidence les apports principaux du travail de thèse."

INTRODUCTION

De plus en plus de découvertes scientifiques sont faites grâce à la puissance de calcul qu'offrent des super-ordinateurs comme la grille de calcul du CERN [2, 23].[1] Par exemple, on peut considérer que la découverte du boson de Higgs en 2012 doit sa rapidité, au moins en partie, au traitement informatisé des 13 petaoctets (13×10^{15} o) de données brutes que produisent les expériences ATLAS et CMS chaque année [19]. Ces données nécessitent en effet des calculs complexes de reconstruction des phénomènes physiques avant de pouvoir être exploitées. Cependant, le temps passé par la grille à évaluer des fonctions élémentaires telles que l'exponentielle ou le logarithme peut représenter jusqu'à un quart du temps de calcul total [73, 138].

En effet, beaucoup de processeurs généralistes ne disposent pas d'instructions matérielles permettant d'approximer efficacement l'ensemble des fonctions élémentaires recommandées par la norme IEEE 754 pour l'arithmétique flottante [76]. Il est alors nécessaire d'utiliser des bibliothèques de fonctions mathématiques (couramment appelées "libm") qui diffèrent souvent en termes de précision ou de performance.

Le premier défi que pose cette thèse est la conception d'implémentations performantes de fonctions élémentaires en arithmétique flottante, et en particulier leur optimisation pour les architectures matérielles proposant des instructions vectorielles SIMD. Mais trouver le bon équilibre entre performances et précision des calculs est un

[1] Conseil Européen pour la Recherche Nucléaire. https://home.cern/about

travail de longue haleine. C'est pourquoi un second défi est *l'automatisation* des procédés d'écriture de bibliothèques mathématiques (aussi appelées libms). Cela permettrait aux utilisateurs de créer des implémentations de fonctions élémentaires adaptées à leurs besoins et optimisées pour leurs architectures matérielles.

En ce sens a été lancé le projet ANR MetaLibm[1] en 2014. Ce projet a pour but de développer des outils de génération de code pour les fonctions mathématiques ainsi que pour les filtes numériques [22]. Il implique les acteurs suivants : Pequan/LIP6 (Sorbonne Université), Inria/SOCRATE (INSA Lyon), le CERN, et DALI/LIRMM (Université de Perpignan Via Domitia). Cette thèse s'inscrit dans le cadre de ce projet. Nos travaux portent sur la génération de codes hautes performances sous contraintes architecturales pour l'implémentation de fonctions mathématiques.

0.1 ARITHMÉTIQUE FLOTTANTE

Le Chapitre 1 de cette thèse rappelle l'état de l'art sur lequel se basent nos travaux :

- l'arithmétique flottante et plus particulièrement sa standardisation avec la norme IEEE 754-2008 ;
- l'évaluation des fonctions élémentaires, de façon générale mais avec un accent sur les implémentations logicielles ;
- et les outils actuels de génération pour automatiser en partie ou en totalité l'implémentation de fonctions élémentaires.

0.1.1 *Arithmétique flottante IEEE 754*

L'arithmétique des ordinateurs est fondamentalement limitée par la quantité de mémoire disponible. Ainsi, seul des ensembles finis de nombres peuvent être représentés par nos machines. Pour approximer les nombres à virgule, de nombreuses représentations existent, mais une des plus communément utilisées est celle définie par la norme IEEE 754-2008 [76]. Ce standard définit plusieurs formats, binaires ou décimaux, ainsi que l'arithmétique dite « flottante » pour les formats dédiés au calcul. Cette thèse se concentre sur les formats arithmétiques binaires comme `binary32` ou `binary64`, décrits dans la Table 1.1 (p.10).

L'arithmétique de ces formats est différente de l'arithmétique sur les nombres réels, car chaque nombre, opérande ou résultat, a une précision finie. Ainsi, des erreurs d'arrondi peuvent avoir lieu dès lors qu'un opérande ou un résultat n'est pas *représentable* dans son format de destination. Pour les formats utilisés dans cette thèse, le standard IEEE 754-2008 requiert que les opérations arithmétiques dites « de base » que sont l'addition, la soustraction, multiplication, la division,

[1] See `http://metalibm.org`.

mais aussi la racine carrée et l'opération fusionnée « multiplication puis addition » (FMA), soient *correctement arrondies*, c'est à dire que le résultat final doit être identique à l'arrondi du résultat infiniment précis.

Cependant, pour d'autres opérations, et notamment des *fonctions élémentaires* comme le sinus, l'exponentielle ou le logarithme, cette propriété est seulement *recommandée*. En effet, contrairement aux opérations de base, ces fonctions sont généralement plus difficiles à arrondir correctement, car beaucoup produisent des résultats *transcendants*. Un nombre transcendant n'a pas de suite périodique de chiffres dans sa partie fractionnaire, donc il ne peut être ni un nombre flottant de précision finie, ni exactement au milieu de deux nombres flottants de précision finie. Cependant, il peut être extrêmement proche d'une de ces valeurs. Cela pose un dilemme quant à la précision supplémentaire nécessaire pour arrondir correctement ce type de nombres, appelé *dilemme du fabricant de tables* (DFT), qui peut être beaucoup plus grande que la précision de destination.

0.1.2 *Évaluation de fonctions élémentaires*

L'évaluation des fonctions élémentaires repose en général sur des compromis entre performance et précision, à cause du DFT. Certaines méthodes visent l'arrondi correct, et doivent donc prévoir des algorithmes plus précis (et donc plus coûteux) pour les cas difficiles à arrondir. Mais il est possible de séparer l'évaluation en deux étapes pour gagner en performance : une première étape dite *rapide* fournit un résultat correctement arrondi *la plupart du temps*, mais si l'arrondi peut ne pas être correct, alors une deuxième étape dite *précise* est exécutée et renvoie le résultat attendu. À l'inverse, d'autres méthodes privilégient la performance à la précision en garantissant des bornes d'erreur plus souples, par exemple bornées par 1 ulp (on parle alors d'*arrondi fidèle*).

Dans cette thèse on s'intéresse à l'évaluation logicielle de fonctions élémentaires se décomposant en trois grandes étapes : une *réduction d'argument*, une *évaluation polynomiale* et une *reconstruction*. La réduction d'argument consiste à utiliser des propriétés mathématiques comme la périodicité, la parité, ou des identités remarquables afin de réduire le domaine d'évaluation à un domaine restreint. Par exemple, le cosinus étant 2π-périodique, on peut restreindre son évaluation à l'intervalle $[0, 2\pi]$. D'autres propriétés permettent de réduire encore cet intervalle à $[0, \pi/4]$. Sur un intervalle raisonnablement petit, il est devient intéressant d'approximer la fonction par un polynôme car le matériel est en général efficace pour ce genre de calculs. Cependant, si le degré nécessaire est trop grand pour atteindre la précision voulue, il est souvent possible de le diminuer à l'aide de *tables de correspondances*. Il s'agit de valeurs précalculées et stockées en mémoire qui permettent de réduire encore un peu plus l'intervalle d'évaluation.

Pour un polynôme quelconque, il y a en général de nombreuses manières de l'évaluer. On appelle ces méthodes des *schémas d'évalua-*

tion polynomiale. Un des schémas les plus utilisés est celui dit de Horner, mais nous montrons dans le Chapitre 3 que d'autres schémas peuvent être plus efficaces.

Enfin, la reconstruction consiste à combiner les résultats des réductions d'argument mathématique et à base de table avec ceux des approximations polynomiales de façon à reformer le résultat final.

Chacune de ces étapes induit des erreurs d'approximation ou d'évaluation qu'il convient de quantifier pour garantir des bornes d'erreur. L'implémentation de ces étapes ainsi que le calcul de bornes d'erreur pour différentes précisions d'entrées/sorties sont des tâches plutôt laborieuses et sujettes à de nombreuses erreurs humaines [92]. C'est pourquoi on cherche à automatiser toujours plus les différentes phases de l'implémentation de fonctions, via des générateurs de code.

0.1.3 *Génération de code pour les fonctions élémentaires*

Sollya, un environnement pour le développement de codes numériques

Sollya est un langage de script ainsi qu'une bibliothèque C pouvant effectuer certains calculs symboliques sur de nombreuses fonctions élémentaires, mais aussi calculer différentes approximations polynomiales comme celles de Taylor, de Remez ou encore FP-minimax [26, 27]. Parmi ses fonctionnalités, Sollya fournit les routines suivantes :

- `guessdegree` essaye de deviner le degré minimal requis pour approximer une fonction sur un intervalle donné avec une certaine marge d'erreur ;
- `taylor`, `remez` et `fpminimax` peuvent générer des polynômes d'approximation pour les fonctions élémentaires [16, 26, 27] ;
- `implementpoly` peut générer du code C qui implémente une évaluation polynomiale optimisée en double, double-double ou triple-double, suivant le schéma de Horner [50].

CGPE, générateur de schémas d'évaluation polynomiale

Bien que Sollya fournisse `implementpoly` pour évaluer un polynôme, seul le schéma de Horner est considéré. Or, on peut souhaiter atteindre des débits d'opérations plus élevés en utilisant d'autres schémas d'évaluation. C'est ce que propose CGPE, un logiciel en ligne de commande et une bibliothèque C++ pouvant générer du code efficace pour l'évaluation polynomiale sur différentes architectures [144]. CGPE peut notamment générer des schémas d'évaluation exposant le plus de parallélisme d'instruction (ILP). CGPE est aussi capable de générer des schémas efficaces satisfaisant une certaine borne d'erreur. Dans ce cas, CGPE peut formellement vérifier cette propriété en générant une preuve Gappa, qui est présenté ci-dessous.

Gappa, pour formaliser et certifier le calcul d'erreur

L'analyse d'erreur est une tâche pénible et source d'erreurs, mais peut être automatisée grâce à des outils comme Gappa [37, 51, 52]. Gappa peut automatiser et vérifier une analyse d'erreur formalisée. Il gère notamment les formats binary32, binary64 et binary128 [115].

Le projet ANR MetaLibm

MetaLibm est le nom de multiples projets de génération automatique de code qui ont fusionné pour donner naissance au projet ANR MetaLibm, au sein duquel cette thèse s'est déroulée.

Un de ces projets est Metalibm-lutetia, conçu pour être utilisé par des développeurs avec ou sans expertise en calcul numérique. En effet, il permet de générer du code pour des expressions mathématiques générales en prenant en compte des spécifications fonctionnelles comme la précision ou l'intervalle d'évaluation. Kupriianova et Lauter ont développé un algorithme de découpage en sous-domaines qui permet d'optimiser l'approximation de ces expressions mathématiques par plusieurs polynômes [93].

Le second projet est Metalibm-lugdunum, plutôt conçu pour les développeurs de bibliothèques mathématiques, mais pouvant aussi être utilisé comme un générateur d'implémentations efficaces pour des développeurs sans expertise numérique. Cet environnement repose sur un méta-langage permettant la description de plusieurs implémentations abstraites et factorisées, partageant des algorithmes communs. Ainsi, ces algorithmes communs peuvent être réutilisés à l'envi afin de gagner du temps de développement. Ce sont finalement les passes d'optimisation et de génération de code qui transcrivent le méta-code en implémentations efficaces pour différentes architectures et spécifications [20, 22]. Par exemple, le méta-code suivant décrit la multiplication $x \cdot x$ dans le méta-language, où x est la variable d'entrée de la fonction à implémenter.

```
y = Multiplication(x, x, tag="y", precision =
    self.precision)
```

Si la cible est du code portable C11 en précision binary32, Metalibm génère l'expression `y = x * x;`, tandis que si la cible est x86_avx2 en précision binary32 avec huit mots par vecteurs, ce sera le code

```
carg = GET_VEC_FIELD_ADDR(vec_x);
tmp = _mm256_load_ps(carg);
y = _mm256_mul_ps(tmp, tmp);
```

où `vec_x` est l'équivalent vectoriel de x.

Le troisième projet est Metalibm-tricassium. Il concerne les filtres numériques pour notamment le traitement du signal. Ceux-ci ont en effet de nombreux points communs avec les polynômes ou les fonctions rationnelles [109, 171].

Programmation génératrice pour le calcul

Pour le calcul numérique, il existe de plus en plus de langages dédiés ou de compilateurs source à source ciblant différentes optimisations [20, 33, 35, 36, 48, 49, 54, 58–60, 80, 81, 83, 84, 112, 116, 117, 126, 131, 134, 145, 161, 162, 165, 166]. Cette liste est non-exhaustive et pourrait probablement être mise à jour tous les mois. Parmi ces projets, certains visent à améliorer voire à certifier la précision des calculs en arithmétique flottante ou virgule fixe, comme Sardana/Salsa [33, 80, 81] ou Rosa/Daisy [35, 36, 84]. D'autres se focalisent sur de meilleurs compromis entre performances et précision en matériel [83, 134, 166], comme FloPoCo [48, 49, 54], ou en logiciel (PeachPy [58–60]), ou les deux [20].

0.2 TABLES EXACTES POUR LES FONCTIONS ÉLÉMENTAIRES

On a vu que les fonctions élémentaires sont omniprésentes dans de nombreuses applications de calcul haute performance. Cependant, leur évaluation est souvent une approximation. Pour atteindre de grandes précisions, on utilise souvent des propriétés mathématiques, des valeurs tabulées et des approximations polynomiales [43, Ch. 2]. Typiquement, chaque étape de l'évaluation combine des erreurs d'approximation (p. ex. en remplaçant une fonction élémentaire par un polynôme) et des erreurs d'évaluation en précision finie (p. ex. l'évaluation d'un tel polynôme). Les valeurs tabulées ne font pas exception, et contiennent généralement des erreurs d'arrondi dues à la transcendance de la plupart des fonctions élémentaires [111].

Dans le Chapitre 2, nous présentons une méthode permettant de supprimer ces erreurs d'arrondi, qui peut être intéressante quand au moins deux termes sont tabulés dans chaque entrée de la table. Notre technique transfère toutes les potentielles erreurs d'arrondi dans un seul terme correctif, ce qui permet d'économiser asymptotiquement deux fois plus de bits que les méthodes de l'état de l'art. Pour les fonctions trigonométriques et hyperboliques $\sin, \cos, \sinh, \cosh$, nous montrons que les triplets pythagoriciens permettent la construction de telles tables en un temps raisonnable. Lorsque l'on vise l'arrondi correct en double précision pour ces fonctions, nous montrons aussi que cette méthode pourrait permettre de réduire, pendant la reconstruction, jusqu'à 29% les accès mémoire et jusqu'à 42% le nombre d'opérations flottantes.

Ces résultats ont fait l'objet de deux communications dans des conférences [97, 98] et d'une publication dans un journal [96].

0.2.1 *Évaluation à base de tables*

Sans perte de généralité, on considère dans la suite uniquement l'évaluation de la fonction $\sin(x)$. Pour atteindre l'arrondi correct de cette fonction en un temps moyen raisonnable, on divise habituellement le schéma d'évaluation en deux phases. 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 [34]. Chaque phase peut se décomposer en quatre étapes :

1. Une *première* réduction d'argument basée sur des identités mathématiques permet de réduire l'argument x en un argument $x^* \in [0, \pi/4]$. Précisons que x^* est un nombre généralement irrationnel quand x est, lui, exact. La précision de x^* conditionnant la précision du résultat final, il est alors indispensable de réaliser cette étape en utilisant un algorithme adapté [18, 28, 29, 137]. L'argument réduit est alors éventuellement 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. L'intervalle de x^* est encore trop grand pour approximer précisément $f_k(x^*)$ avec un polynôme de petit degré. Une deuxième réduction d'argument est alors effectuée, basée cette fois sur des données tabulées. Cette solution a été détaillée par Tang [164]. Elle consiste à diviser l'argument réduit x^* en deux parties x_h^* et x_l^*, définies par :

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

On se sert alors des p bits de poids fort de x_h pour adresser une table de $\lfloor \pi/4 \times 2^p \rceil$ entrées contenant des valeurs précalculées $\mathtt{sin}_h \approx \sin(x_h^*)$ et $\mathtt{cos}_h \approx \cos(x_h^*)$ et arrondies dans un format de destination, par exemple double-double. Le format double-double consiste à approximer un nombre z sur deux mots binary64 qui ne se chevauchent pas. On utilise alors une des deux formules :

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

Cette étape introduit une deuxième source d'erreur, les approximations $\mathtt{sin}_h$ et $\mathtt{cos}_h$.

3. Des approximations polynomiales de $\sin(x_l^*)$ et $\cos(x_l^*)$ sont alors évaluées. Ceci introduit une troisième source d'erreur.

4. Enfin, le résultat final est obtenu par une *reconstruction* qui utilise les termes obtenus dans les étapes 2 et 3 selon une des Équations (2). Cette étape est encore une source d'erreur via les opérations en précision finie $+$ 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 [121].

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^* [68, 69]. Pour chaque valeur de x_h^*, la variation corr est choisie de sorte que $\cos(x_h^* + \mathrm{corr})$ et $\sin(x_h^* + \mathrm{corr})$ 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 [159].

Dans le Chapitre 2, nous montrons comment aller plus loin et éliminer complètement la deuxième source d'erreur en tabulant des nombres entiers représentables sur un nombre flottant IEEE 754, qui donnent accès à des valeurs rationnelles proches de $\cos(x_h^*)$ et $\sin(x_h^*)$ ou de $\cosh(x_h^*)$ et $\sinh(x_h^*)$. Pour cela, nous nous appuyons sur les triplets pythagoriciens.

0.2.2 *Tabulation de termes exacts*

Notre objectif est de sélectionner des points pour lesquels il est possible de tabuler des valeurs *exactes*. Pour les fonctions trigonométriques et hyperboliques sinus et cosinus, nous utilisons les *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.

Principe de la méthode

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

1. Les valeurs de $\sin_h$ et $\cos_h$ doivent être des nombres rationnels, c'est-à-dire :

$$\sin_h = s_n/s_d \quad \text{et} \quad \cos_h = c_n/c_d$$
$$\text{avec} \quad s_n, s_d, c_n, c_d \in \mathbb{N} \quad \text{et} \quad s_d, c_d \neq 0.$$

2. Pour faciliter la reconstruction, les dénominateurs doivent être identiques : $s_d = c_d$. Cela permet de factoriser la division par les dénominateurs.

3. Pour éviter l'évaluation de cette division durant la reconstruction, le dénominateur s_d doit être égal à une même valeur k pour chaque entrée. Il est ainsi possible de l'intégrer 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 :

$$\sin(x^*) = s_n \cdot C(x_l^* - corr_i) + c_n \cdot S(x_l^* - corr_i),$$

avec :

- s_n, c_n et $corr$ les valeurs tabulées définies par :

$$x_h^* = \arcsin(s_n/k) - corr_i = \arccos(c_n/k) - corr_i.$$

- $C(x)$ et $S(x)$ deux polynômes d'approximation définis pour $x \in [-2^{-p-1} + \max_i(corr_i), 2^{-p-1} - \min_i(corr_i)]$ par :

$$C(x) \approx \cos(x)/k \quad \text{et} \quad S(x) \approx \sin(x)/k.$$

Les valeurs sin_h et cos_h sont alors rationnelles. Mais seuls les deux numérateurs c_n et s_n seront stockés dans la table, les divisions étant précalculées dans les coefficients des polynômes.

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) peut être associé un angle $\theta \in [0, \pi/2]$ tel que $\sin(\theta) = a/c$ et $\cos(\theta) = b/c$.

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°.

Construction et sélection de l'ensemble des triplets pythagoriciens

Il existe une infinité de triplets pythagoriciens primitifs, qui couvrent un large intervalle d'angles. Ceci est illustré sur la Figure 2.5 (p.44), 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 [7, 139]. Ainsi, l'arbre de

Barning-Hall [7] se construit à partir d'un triplet (a, b, c) dont on déduit trois nœuds fils en multipliant les matrices

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

par ce triplet considéré comme un vecteur colonne.

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 $(20, 21, 29)$.

Parmi les triplets pythagoriciens primitifs, nous en sélectionnons un par entrée de la table, de telle sorte que chaque triplet (a, b, c) minimise la magnitude du terme $corr$ 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 ($i \geqslant 0$). En effet, chaque écart $x_{h,k}^* - \theta_k$ correspond au terme correctif $corr_k$ présent dans les évaluations polynomiales. En les réduisant, on réduit la taille de l'intervalle sur lequel on évalue les polynômes. Comme on l'a vu, 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.

0.2.3 *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 précédemment. 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*.

Recherche exhaustive

Algorithme

L'objectif est de trouver l'ensemble composé d'une hypoténuse par entrée qui minimise le PPCM k. On note 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. Génération de tous les triplets pythagoriciens primitifs (a, b, c), avec $c < 2^n$,
2. Recherche de l'ensemble composé d'un triplet par entrée qui minimise le PPCM k,
3. Si k est trouvé *parmi* les hypoténuses générées, construction de la table de valeurs (A, B, corr). Sinon, reprise de l'algorithme avec $n \leftarrow n + 1$.

Nous avons implémenté 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.

Résultats numériques

La Table 2.1 (p.54) présente les résultats obtenus sur un serveur Linux CentOS (Intel Xeon CPU E5-2650 v2 @ 2.60 GHz, 125 Go de RAM). 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 montre que l'on peut construire des tables indexées par $p = 6$ bits en 7 secondes. Pour $p = 7$ bits d'entrée, on passe alors à 31 secondes. Notons que dans cette dernière configuration, il est nécessaire de stocker en mémoire plus de 1.3 millions de triplets et 0.3 million d'hypoténuses. On observe que notre solution exhaustive n'est alors pas envisageable pour des valeurs de $p \geqslant 8$.

Recherche heuristique

Afin de considérer, en un temps et une consommation mémoire raisonnables, des valeurs p plus grandes ($p \geqslant 8$), nous avons cherché à caractériser les triplets et les PPCM déjà trouvés pour des p plus petits ($p \leqslant 7$). La Table 2.5 (p.57) montre leur décomposition en facteurs premiers. En analysant cette table, 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 $4n + 1$ [64]. Pour $p \leqslant 7$, ces facteurs appartiennent à l'ensemble $\mathcal{P}$ des premiers pythagoriciens inférieurs ou égaux à 73 :

$$\mathcal{P} = \{5, 13, 17, 29, 37, 41, 53, 61, 73\}.$$

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 2.7 (p.58). La dernière colonne correspond au nombre d'hypoténuses comprises entre 2^{n-1} et 2^n, et sélectionnées par l'heuristique. On observe que non seulement cette heuristique a une consommation mémoire nettement inférieure à notre algorithme exhaustif, avec optimisation, 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 69 hypoténuses différentes et de générer la table en moins d'une seconde, contre plus de 0.3 million d'hypoténuses et 31 secondes pour notre algorithme exhaustif.

Le facteur limitant de cette heuristique algorithme n'est plus la recherche du PPCM, qui consiste à vérifier seulement quelques dizaines d'hypoténuses, mais la sélection des triplets. Nous avons donc développé une seconde heuristique utilisant une technique décrite par Fässler dans [64]. Cette technique permet de construire des triplets pythagoriciens partageant une même hypoténuse. Les résultats de cette heuristique sont présentés dans la Table 2.9 (p.63). Ils montrent qu'on peut construire des tables dans la limite du stockage possible sur des nombres binary64 en un temps encore plus raisonnable.

0.2.4 *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 aux solutions de Tang et de Gal. 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 pendant l'étape de reconstruction. Aussi, on considère que des algorithmes de calcul en expansions sont utilisés quand une grande précision est requise, comme c'est le cas par exemple dans la bibliothèque CR-Libm. La Table 2.10 (p.65), extraite de [34, § 2.3, 101, § 3.2, 102], sera utilisée comme référence pour calculer le coût de ces algorithmes quand aucune instruction FMA (Fused-Multiply and Add) n'est disponible dans le jeu d'instructions. La notation E_n désigne une expansion de taille n, c'est-à-dire, un nombre flottant représenté en machine comme la somme non évaluée de n nombres flottants IEEE 754. Avec ce formalisme, E_1 représente un nombre flottant habituel. Le coût de ces opérations élémentaires en précision étendue est égal au nombre d'instructions flottantes nécessaires pour les réaliser.

Comparaison des résultats

Nos estimations sont résumées dans la Table 2.11 (p.67), qui contient le nombre d'opérations flottantes et d'accès mémoire pour la phase rapide et la phase précise ainsi que l'occupation mémoire de chaque table en octets/ligne. 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 56 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 29% par rapport à Tang et Gal, respectivement.

Concernant le nombre d'opérations flottantes, notre solution un gain jusqu'à 42% peut être noté. Enfin, on remarque que notre solution peut réduire le nombre d'accès mémoire. La phase rapide requiert 3 accès mémoire, tandis que l'approche de Tang en nécessite 4, ce qui représente une amélioration de 25%. Pour la phase précise, on passe de 6 accès pour Tang et 7 pour Gal à 5 accès pour notre méthode, c'est-à-dire, une amélioration jusqu'à 29%.

0.2.5 *Conclusions et perspectives*

Dans ce premier chapitre, 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. Notre solution est la seule parvenant à *éliminer* l'erreur d'arrondi sur les approximations tabulées, pour la concentrer dans les approximations polynomiales. Nous réduisons ainsi jusqu'à 29% la taille des tables, le nombre d'accès mémoire, et le nombre d'opérations flottantes impliquées dans le processus de reconstruction.

Nous percevons deux axes de perspectives : Premièrement, il serait intéressant d'intégrer notre table de points exacts à une implémentation complète des fonctions sinus et cosinus pour observer son impact réel. Deuxièmement, une heuristique du même type que celle basée sur la technique de Fässler mais pour les fonctions hyperboliques serait souhaitable. Une piste serait d'utiliser les résultats décrits dans [147].

0.3 VECTORISATION DE SCHÉMAS POLYNOMIAUX

Les fonctions élémentaires sont souvent calculées à l'aide d'approximations polynomiales, dont l'efficacité dépend directement de celle du schéma d'évaluation sous-jacent. Le troisième chapitre de cette thèse montre que le schéma classiquement utilisé (Horner) est rarement le plus performant. En effet, d'autres schémas exploitent mieux les parallélismes des architectures modernes, en réduisant les dépendances de données. Ces résultats ont pour objectif d'être intégrés à un

générateur de code performant pour l'évaluation polynomiale dans le cadre de l'approximation de fonctions. Ils ont fait l'objet d'une publication à la conférence ComPAS 2016 [99], tout en représentant une étape importante pour le Chapitre 4, qui a conduit à la soumission d'un article à la conférence ASAP 2018 [95] (en cours de relecture).

0.3.1 *Contexte et Motivations*

L'implémentation de fonctions élémentaires (sin, log, ...) en logiciel passe souvent par une approximation polynomiale de degré raisonnable, sur un intervalle restreint [123, § 11.4]. L'évaluation d'un tel polynôme, c'est-à-dire l'ordre dans lequel les opérations sont effectuées, est fixé statiquement par un *schéma d'évaluation* dans le code, pour plusieurs architectures matérielles. Le schéma le plus utilisé est celui de Horner (illustré Figure 3.1 (p.75)), qui est essentiellement séquentiel. Sa latence est donc relativement élevée [124]. Cependant, sélectionner manuellement un schéma plus rapide pour une architecture donnée peut s'avérer long et complexe.

Notre objectif est de générer automatiquement des schémas qui profitent au mieux des architectures modernes, notamment du parallélisme d'instructions (ILP) et du parallélisme de données. Ce chapitre présente une étude de la performance de différents schémas sur une architecture de type Intel Haswell[1]. Nous montrons deux résultats utiles : d'une part, des schémas exposant beaucoup d'ILP peuvent avoir à la fois une latence et un débit meilleurs que des schémas plus séquentiels, un résultat qui contredit les préconisations usuelles [123, § 11.5]. D'autre part, quand la production d'un résultat dénormalisé est très coûteuse, nous notons que l'intervalle d'évaluation prévu pour un polynôme donné est un facteur déterminant pour le choix d'un bon schéma.

0.3.2 *Rappels sur l'évaluation polynomiale*

Soit P un polynôme de degré n à coefficients flottants non nuls, comme définis dans la norme IEEE 754-2008 [76]. On veut évaluer P en un point x dans le même format. Dans cette thèse, on ne considère que les schémas d'évaluation sans adaptation de coefficients. (Plus de détails sur l'adaptation de coefficients peuvent être trouvés dans [62, 135].)

Le schéma le plus courant est celui de Horner. Il permet d'évaluer $P = \sum_{i=0}^{n} a_i X^i$ en n additions et n multiplications. Par exemple, pour $n = 7$, on aura :

$$\begin{aligned} P(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (a_3 + x \cdot (a_4 \\ + x \cdot (a_5 + x \cdot (a_6 + x \cdot a_7)))))). \end{aligned} \quad (4)$$

[1] https://software.intel.com/haswell

Son intérêt est triple : il minimise le nombre de multiplications, n'a pas besoin de stocker de résultat intermédiaire [90, p. 486] et il est stable numériquement [14, ch.9]. Quand le critère de choix est la précision, Horner est donc un bon schéma. Mais, on l'a vu, sa latence est élevée. Or, il existe des schémas plus rapides. Par exemple, le schéma de Horner d'ordre 2, noté Horner-2, a le parenthésage suivant au degré 7 :

$$\begin{aligned} P(x) = &(a_0 + x^2 \cdot (a_2 + x^2 \cdot (a_4 + x^2 \cdot a_6))) \\ &+ x \cdot (a_1 + x^2 \cdot (a_3 + x^2 \cdot (a_5 + x^2 \cdot a_7))). \end{aligned} \tag{5}$$

Ce schéma expose de l'ILP, comme illustré à la Figure 3.2 (p.76) : les mônomes de degrés pair et impair peuvent être évalués en parallèle. Sa latence est donc environ divisée par deux [90, p. 488].

Le schéma d'Estrin, illustré en Figure 3.3 (p.76), expose encore plus d'ILP en découpant l'arbre d'évaluation selon les puissances de x de la forme 2^i [61]. Par exemple, pour $n = 7$, on aura :

$$\begin{aligned} P(x) = &((a_0 + a_1 \cdot x) + x^2 \cdot (a_2 + a_3 \cdot x)) \\ &+ x^4((a_4 + a_5 \cdot x) + x^2(a_6 + a_7 \cdot x)). \end{aligned} \tag{6}$$

Dans le cas d'un polynôme de degré $n = 2^k - 1$, l'arbre d'évaluation d'Estrin est bien équilibré, mais il l'est un peu moins dans le cas général. Sa latence est en $O(\log n)$.

Avec une architecture simplifiée infiniment parallèle ayant respectivement des latences de 3 et 5 cycles pour l'addition et la multiplication flottantes[1], pour $n = 7$ et en utilisant (4), (5) et (6), on peut prévoir des latences de 56, 37 et 24 cycles, pour, respectivement, Horner, Horner-2 et Estrin. Avec un FMA en 5 cycles, on peut s'attendre, respectivement, à des latences de 35, 25 et 15 cycles. Il est donc important de choisir un bon schéma d'évaluation. Cependant, le nombre de schémas possibles au degré n croît rapidement avec n du fait de la combinatoire, ce qui rend impossible une recherche exhaustive dès que $n \geqslant 6$ [144, ch.6].

D'autres schémas ainsi que la question de l'optimalité asymptotique sur parallélisme fini ou infini ont été étudiés dans [113, 124, 125]. Dans le cadre de l'évaluation de fonctions élémentaires, nous avons comparé les performances de Horner à deux schémas classiques : Horner-2 et Estrin ; puis, à tous ceux de plus faible latence pour le degré 12 sur notre architecture simplifiée.

0.3.3 *Comparaison des performances de schémas d'évaluation polynomiale*

Dans cette section, nous analysons les résultats de mesures de performance obtenus pour les schémas de Horner, Horner-2 et Estrin, compilés avec GCC 5.2.0 (option `-O2` activée), sur un processeur Intel Haswell bénéficiant des extensions AVX2 et FMA. Celles-ci opèrent sur 16 registres de 256 bits, en scalaire (sur un mot de poids faible) ou en vectoriel, et disposent d'un additionneur et de deux multi-

[1] Latences minimales documentées pour l'architecture Haswell.

plieurs/FMA [66, § 10.14]. GCC 5.2.0 est capable de générer des instructions vectorielles via une passe d'autovectorisation, que nous utilisons pour produire du code vectorisé.

Nous mesurons le débit inverse, c'est-à-dire le nombre moyen de cycles entre deux résultats. Ainsi, plus il est faible, plus le débit correspondant est grand. Pour un degré fixé, il dépend *a priori* du schéma et de l'architecture, que nous faisons varier en autorisant ou non l'utilisation du FMA. Sur ce type d'architecture, tant qu'aucun calcul ne dénormalise, nous observons que les schémas exposant le plus d'ILP ont un meilleur débit que les schémas séquentiels.

Afin que l'expérience soit représentative de l'évaluation de fonctions, nous faisons varier le degré du polynôme entre 3 et 32, et nous utilisons les coefficients de la série de Taylor de $\log(1+x)$ en 0, calculés avec Sollya[1]. Nous réalisons plusieurs évaluations sur des vecteurs de 2^{14} nombres flottants uniformément répartis dans l'intervalle $[2^{-12}; 2^{-5}]$.

Les Figures 3.4 (p.78) et 3.5 (p.79) présentent les débits inverses mesurés en simple précision pour Horner, Horner-2 et Estrin, avec ou sans vectorisation, avec ou sans FMA, respectivement. Sans surprise, le débit diminue avec le degré pour tous les schémas. En revanche, on observe une perte de performance drastique à partir du degré 16, uniquement pour le schéma d'Estrin. Dans la section suivante, nous expliquons cette perte de performance par le fait que le résultat intermédiaire x^{16}, calculé par Estrin, dénormalise dans 13% des cas. Sans FMA, Estrin a le meilleur débit jusqu'au degré 15 : pour ce degré, il est plus de deux fois meilleur que Horner et 50% meilleur que Horner-2, pour les deux versions (scalaire et vectorisée). Au degré 16, il est de 27 à 67% inférieur à Horner et de 49 à 79% inférieur à Horner-2, pour les versions scalaire et vectorisée, respectivement. Avec FMA, les trois schémas se valent jusqu'au degré 15, avec un désavantage pour Horner non vectorisé. À partir du degré 16, seul Horner-2 est meilleur que Horner, d'environ 40%. Globalement, jusqu'au degré 32, aucun des trois schémas ne souffre du nombre limité de registres, le séquenceur de GCC parvenant à conserver les résultats intermédiaires dans ces derniers. Cela est encourageant pour l'utilisation de schémas très parallèles dans le cadre de l'évaluation de fonctions.

En refaisant les mêmes expériences en double précision, on n'observe pas la perte de performance obtenue pour Estrin en simple précision. Cela conforte l'hypothèse que ceci est causé par un résultat intermédiaire dénormalisé. La diminution du débit avec le degré est alors quasi-linéaire, et le schéma d'Estrin a un débit toujours supérieur à Horner dès le degré 10 dans tous les cas. Nous concluons qu'une latence faible n'entraîne pas toujours une chute du débit, ce qui peut rendre des schémas comme Horner-2 ou Estrin plus intéressants si l'on veut privilégier la performance à la précision. Ce résultat est assez contre-intuitif car sur parallélisme fini, on pouvait s'attendre

[1] Voir `http://sollya.gforge.inria.fr/` et [26].

à une saturation des unités arithmétiques pour les schémas très parallèles.

0.3.4 *Influence de l'intervalle d'évaluation*

Dans la suite, on note $RN(()x)$ l'arrondi dans une direction quelconque d'un réel x en simple précision. Les polynômes utilisés pour l'évaluation de fonctions élémentaires sont valides sur un petit intervalle. Les autres entrées sont soit traitées séparément, soit ramenées à l'intervalle d'évaluation. Cet intervalle est un voisinage du point d'approximation qui est souvent 0, c'est le cas pour $\ln(1+x)$, $\exp(x)$, ou $\sin(x)$. Les entrées de petites magnitudes élevées à certaines puissances peuvent alors dénormaliser rapidement. Par exemple, pour $x \in [2^{-12}; 2^{-5}]$, x^{16} dénormalise en simple précision dans environ $(2^{-126/16} - 2^{-12})/2^{-5} \approx 13\%$ des cas.

Les Figures 3.8 (p.81) et 3.9 (p.82) montrent les latences mesurées pour deux schémas d'élévation à la puissance 16 de nombres flottants simple précision dans le même intervalle. L'un correspond à une exponentiation naïve, l'autre à une exponentiation binaire. Leurs latences minimales sont de l'ordre de quelques cycles, mais quand x devient suffisamment petit pour que x^{16} dénormalise, on observe un palier pour les deux schémas. Les latences observées passent ainsi de quelques cycles à plus d'une centaine. D'après Agner Fog, cette pénalité provient de traitements micro-codés, pour toutes les opérations dont les opérandes sont normaux et dont le résultat est dénormalisé, ainsi que pour une multiplication dont un opérande est normal et l'autre dénormalisé [66, § 10.9]. Les paliers suivants observés pour la méthode naïve commencent quand les mônomes de degré inférieur dénormalisent. On remarque aussi que la version binaire ne retrouve une latence de quelques cycles que bien après que $RN(()x^{16}) = 0$, ce qui vient en partie contredire le manuel de Fog [66, § 10.9]. Cela suggère que le micro-code n'est exécuté que pour un intervalle précis de résultat. Par ailleurs, lorsque $RN(()x^{15}) = 0$, i.e. $|x| \leqslant 2^{-10}$, on peut voir un palier descendant pour la version naïve. Cela montre que la multiplication par zéro ne souffre pas de l'exécution de micro-code.

Ainsi, sur certaines architectures, le fait qu'un résultat intermédiaire dénormalise diminue grandement le débit des calculs, et notamment des évaluations polynomiales. Il est alors pertinent de s'assurer que, pour un schéma d'évaluation donné, l'intervalle d'entrée prévu n'engendre pas de dénormalisés. Ce résultat pourrait permettre d'améliorer la sélection automatique de schémas performants sur l'ensemble de son intervalle d'utilisation.

0.3.5 *Vers des schémas plus performants*

L'outil CGPE[1] permet de calculer tous les schémas d'évaluation polynomiale de plus faible latence sur parallélisme infini, pour un degré

[1] Voir http://cgpe.gforge.inria.fr/ et [120, 144].

et une architecture donnés. Nous l'avons utilisé pour générer ceux de degré 12, ce qui représente 11 944 schémas de latence 26 avec l'architecture considérée. Les débits de ces schémas compilés sans FMA et avec autovectorisation, ont ensuite été mesurés pour des entrées dont certaines puissances dénormalisent. Les Figure 3.10 (p.83), 3.11 (p.84), et 3.12 (p.84) représentent ces 11 944 mesures par des points. Les traits verticaux distinguent les schémas selon le nombre de multiplications (de 17 à 25) qu'ils impliquent.

Sur la Figure 3.10 (p.83), on remarque que *tous* les schémas de plus faible latence ont un *meilleur* débit que les schémas de Horner et Horner-2, et seuls une dizaine de schémas sont aussi bons qu'Estrin. Ce sont ceux qui impliquent le moins de multiplications. Assez logiquement, on observe aussi des paliers indiquant des débits plus faibles à mesure que le nombre de multiplications augmente. La Figure 3.11 (p.84) vient confirmer que le schéma d'Estrin peut être beaucoup plus performant que beaucoup de schémas de latence minimale, car il peut être en mesure de produire moins de résultats intermédiaires dénormalisés. En revanche, la Figure 3.12 (p.84) montre qu'Estrin est environ 12 fois moins bon que Horner et Horner-2 quand la dénormalisation intervient à partir de x^8, alors qu'une cinquantaine de schémas générés restent plus performants que Horner et Horner-2, notamment parmi ceux qui impliquent peu de multiplications. Cela va dans le sens de nos conclusions précédentes, c'est-à-dire, qu'il faut s'assurer que le schéma que l'on choisit, pour une architecture qui lève des exceptions pour des résultats dénormalisés, ne produise pas lui-même de résultats dénormalisés sur son intervalle d'évaluation.

Une étude plus approfondie des meilleurs schémas générés par CGPE pourrait nous permettre de distinguer des motifs, réutilisables dans le cadre de l'approximation de fonctions par des polynômes univariés. Dans le cas contraire, une mesure automatisée des performances de ces schémas sur une architecture cible pourrait nous fournir des critères de choix. *A posteriori*, une borne sur la précision du résultat final pourrait être calculée afin de vérifier si un schéma sélectionné est conforme aux exigences demandées.

0.3.6 *Conclusions*

Dans le cadre de l'évaluation polynomiale pour l'approximation de fonctions, nous avons observé que certaines dénormalisations pouvaient fortement impacter les performances. C'est donc un facteur essentiel pour choisir un schéma performant sur tout son intervalle d'évaluation. Nous avons aussi vu que des schémas exposant beaucoup d'ILP comme le schéma d'Estrin ou ceux générés par CGPE ont souvent un meilleur débit que les schémas séquentiels comme Horner.

Ces résultats pourraient être la base de recherches plus approfondies. Il pourrait être intéressant d'intégrer des critères de choix dans CGPE pour éliminer les schémas qui, pour un intervalle d'évaluation et un ensemble de coefficients donnés, pourraient engendrer des

pertes de performances. Une piste serait d'utiliser Gappa à cette fin. Enfin, l'influence qu'a le choix d'un bon schéma sur les performances de l'évaluation de fonctions élémentaires mériterait d'être mesurée.

0.4 MÉTA-IMPLÉMENTATION DE LOGARITHMES

Afin de générer du code efficace, les environnements de développement devraient être en mesure d'exploiter toute source d'amélioration des performances. Par exemple, les extensions SIMD telles que SSE, AVX ou AVX-512 supportent des instructions vectorielles, permettant ainsi d'augmenter le débit d'applications exposant du parallélisme de données. En effet, ces instructions peuvent traiter actuellement jusqu'à 16 mots binary32 ou 8 mots binary64 avec un débit et une latence annoncés identiques aux instructions scalaires équivalentes. Le Chapitre 4 de cette thèse a pour objectif de montrer que la conception d'une implémentation de fonction élémentaire peut s'abstraire de plusieurs notions indépendantes des formats d'entrée/sortie ou de l'architecture visée en utilisant des générateurs de code tels que Metalibm-lugdunum. Dans le cadre de l'implémentation de fonctions élémentaires vectorielles, il n'est ainsi plus nécessaire d'écrire le code vectorisé à la main. En ajoutant du support pour la génération d'instructions vectorielles, c'est l'environnement Metalibm-lugdunum qui se charge entièrement de la vectorisation. Le développeur n'a alors plus qu'à concevoir des « méta-algorithmes » d'évaluation, *vectorisables* pour plus de performances.

Nous présentons dans ce chapitre une collaboration avec Nicolas Brunie[1], créateur et principal développeur de l'environnement Metalibm-lugdunum[2]. Concrètement, nous présentons :

- une réduction d'argument classique mais réalisée sans branchements afin d'utiliser au mieux les ressources vectorielles de notre cible, AVX2 ;
- un « méta-algorithme » du logarithme naturel, avec arrondi fidèle, pour tout format de précision $p \geqslant 2$;
- une méta-implémentation de cet algorithme dans l'environnement Metalibm-lugdunum, avec le support vectoriel nécessaire pour la cible AVX2 ;
- et des mesures de performances des implémentations générées pour les formats binary32 et binary64 sur une architecture AVX2.

0.4.1 *Vectorisation de fonctions élémentaires*

Depuis le milieu des années 1990, avec notamment le développement de l'imagerie en trois dimensions pour de nombreux domaines (médical, scientifique, architecture, divertissement. . .), les jeux d'instruc-

[1] Kalray, https://kalray.eu.
[2] https://github.com/kalray/metalibm

tions SIMD se sont démocratisés [130, 136]. En effet, ces derniers proposent théoriquement de démultiplier le débit d'applications exposant un parallélisme de données massif en réalisant des instructions matérielles *vectorielles*, c'est-à-dire des instructions effectuant la même opération mais sur des données multiples.

Afin d'exploiter ce parallélisme pour l'évaluation des fonctions élémentaires, il convient d'utiliser des algorithmes qui se vectorisent efficacement. De tels algorithmes ne comportent habituellement pas de branchements ni de tables de correspondances qui dépendent des arguments. Dans ce chapitre, nous développons un algorithme vectorisable efficacement sur le jeu d'instruction SIMD AVX2, sans branchements mais avec une table de correspondances, pour l'évaluation du logarithme naturel en arithmétique flottante de précision arbitraire $p \geqslant 2$. Cet algorithme est implémenté dans l'environnement Metalibm-lugdunum [20, 22] et permet de générer du code C scalaire ou vectoriel pour l'évaluation du logarithme naturel dans les formats binary32 et binary64, avec arrondi fidèle garanti. Cette précision est en effet suffisante pour certaines applications de physique théorique [138]. Il est possible, à faible coût, de générer des implémentations offrant un meilleur débit au prix d'une précision dégradée et non garantie, mesurable a posteriori, ce qui peut parfois être suffisant [73]. Comme notre algorithme ne dépend pas de la précision, il est envisageable de générer des implémentations pour d'autres formats comme binary16 ou binary128.

0.4.2 *Conception d'un méta-logarithme vectorisable*

On considère la fonction $x \mapsto \log(x)$ avec x un nombre flottant de précision $p \geqslant 2$ tel que $x > 0$ et $x \notin \{\mathrm{NaN}, \pm\infty\}$. On pose $x = m \cdot 2^e$ avec $m \in [1, 2 - 2^{1-p}]$.

La réduction d'argument que nous utilisons est classique pour les implémentations scalaires. Soit r_i une approximation de $1/m$. Alors, en utilisant l'identité $\log(x) = e\log(2) + \log(m)$, le logarithme peut se calculer de la façon suivante :

$$\log(x) = (e + \tau) \cdot \log(2) - \log(2^\tau \cdot r_i) + \log(1 + u),$$

avec $u = r_i \cdot m - 1$, et $\tau = [m > \sqrt{2}]$. On introduit ici τ afin d'éviter une *cancellation* catastrophique qui peut arriver lorsque $e = -1$ et $m \approx 2$, comme dans [143]. Cette réduction d'argument permet d'éviter des branchements conditionnels, ainsi que de profiter de l'approximation inverse rapide (`rcp`) souvent disponible sur les architectures vectorielles récentes.

Quand x est un nombre dénormalisé, une technique couramment utilisée est une remise à l'échelle de x avant tout autre calcul [34]. On écrit donc $x' = x \cdot 2^\lambda$ avec $\lambda \in [0 \,..\, p - 1]$ le nombre de zéros en tête

dans la représentation binaire de x. Alors on a $x' = m' \cdot 2^{e'} \geqslant 2^{e_{\min}}$ avec $m' \in [1, 2 - 2^{1-p}]$, et, finalement :

$$\log(x) = (e' + \tau - \lambda) \cdot \log(2) - \log(2^{\tau} \cdot r_i) + \log(1 + u). \quad (7)$$

La quantité r_i devient donc une approximation de $1/m'$, tronquée sur i bits, et $u = r_i \cdot m' - 1$. La valeur $-\log(2^{\tau} \cdot r_i)$ est lue dans une table indexée par les i premiers bits fractionnaires de r_i.

Nous montrons que u peut-être représenté exactement sous la forme d'un nombre flottant de précision p si le paramètre i est choisi de telle sorte que :

$$i \leqslant e_{\max} - 2 \quad \text{and} \quad \kappa \cdot (2^{i} + 2^{-1}) < 2^{-1}, \quad (8)$$

où κ est une borne d'erreur relative sur l'approximation de $1/m'$ par l'instruction `rcp`. C'est pourquoi nous privilégions la génération d'une instruction `fma` pour calculer u exactement, disponible sur la plupart des architectures récentes. Sans cette possibilité, une solution est de générer une séquence de calcul en arithmétique multi-mots comme par exemple la précision double-double [53, 101].

Nous montrons finalement comment garantir l'arrondi fidèle du résultat r_h en distinguant trois cas qui nous fournissent trois bornes d'erreur sur $|\log(x) - r|$ où $r = r_h + r_l$ est le résultat intermédiaire sur deux mots de précision p. Ces bornes nous ont ensuite permis de certifier que notre méta-implémentation renvoie l'arrondi fidèle du logarithme de x.

0.4.3 *Méta-implémentation dans Metalibm-lugdunum*

Dans cette section, nous montrons comment calculer x' en utilisant uniquement des instructions en arithmétique entière, permettant ainsi d'éviter les surcouts liés au traitement des nombres dénormalisés en arithmétique flottante. Le Listing 4.1 (p.96) présente ainsi un algorithme efficace et vectorisable qui calcule λ avec uniquement des opérations entières. Le calcul de $x' = x \cdot 2^{\lambda}$ revient ensuite simplement à décaler x de λ bits vers la gauche, qui est une opération rapide et vectorisable.

Un tel algorithme a été implémenté sous la forme d'un "méta-bloc" dans Metalibm-lugdunum. Metalibm-lugdunum est un environnement de développement permettant de générer rapidement du code logiciel ou matériel efficace, principalement pour les fonctions élémentaires et le calcul numérique. Il repose sur un langage spécifique capable d'abstraire des concepts comme la cible matérielle ou la précision, entre autres. Sur ce principe, nous avons ajouté du support pour le générateur de code ciblant les architectures Intel AVX2, afin que Metalibm puisse générer du code vectorisé utilisant les instructions `vrcpps` ou `vgatherdps`, par exemple.

En ce qui concerne l'évaluation polynomiale, Metalibm-lugdunum propose un générateur de schéma de Horner ainsi qu'un autre pour

le schéma d'Estrin. Afin de générer d'autres schémas potentiellement plus efficaces automatiquement, nous avons développé et intégré une interface entre CGPE et Metalibm-lugdunum nommée PythonCGPE. Comme CGPE est écrit en C++ et Metalibm en Python, notre choix s'est porté sur Cython pour l'implémentation de cette interface.

0.4.4 *Mesures de performances du code généré*

Nous avons mesuré le débit inverse de plusieurs routines vectorisées, soit écrites manuellement, soit générées automatiquement, ainsi que plusieurs routines de la GNU Libmvec présente dans glibc 2.26. Ces résultats sont présentés dans la Table 4.3 (p.101). Également, nous avons mesuré l'impact du traitement des dénormalisés et de la taille de notre table sur le débit de 28 implémentations générées automatiquement. Ces mesures sont reportées dans la Table 4.4 (p.103).

Bien que les routines de la Libmvec se montrent plus performantes que nos implémentations, celles-ci ne sont précises qu'à 4 ulps quand les nôtres renvoient toujours un arrondi fidèle, à au plus 1 ulp du résultat exact. De plus, les routines de la Libmvec sont écrites une par une, directement en assembleur. L'avantage de notre méthode est de pouvoir générer en quelques secondes de nombreuses implémentations en C afin d'observer quels sont les meilleurs compromis, par exemple ici pour trouver quels paramètres donnent le débit le plus élevé.

0.4.5 *Conclusions*

Dans ce chapitre, nous avons voulu montrer à travers l'exemple du logarithme les avantages offerts par la programmation générative. Nous avons effectivement montré comment générer de multiples implémentations vectorisées avec une garantie sur la précision du résultat final, et ce indépendamment des formats d'entrée/sortie. Des perspectives envisagées sont un support étendu du backend Intel vectoriel, notamment pour supporter les instructions AVX-512, le support de formats comme binary16 ou binary128, ou encore la méta-implémentation entièrement factorisée de logarithme en base β paramétrable.

CONCLUSION GÉNÉRALE

Cette thèse avait pour objectif le développement de techniques et d'algorithmes pour la génération de code performant appliquée à l'évaluation des fonctions élémentaires. Dans le Chapitre 2, nous avons présenté une méthode originale pour générer des tables de correspondances sans erreur utiles pour les fonctions trigonométriques et hyperboliques. Jusque là, aucune méthode, à notre connaissance, n'était parvenue à supprimer l'erreur sur les valeurs tabulées pour ces fonctions.

Dans le Chapitre 3, nous avons analysé l'impact sur le débit apporté par la vectorisation de schémas d'évaluation polynomiale. Premièrement, nous avons confirmé des résultats précédents montrant que certaines opérations en arithmétique flottante pouvaient sévèrement réduire le débit d'applications dès lors que des nombres dénormalisés sont impliqués. L'évaluation polynomiale fait partie de ces applications. Deuxièmement, nous avons observé que de nombreux schémas d'évaluation polynomiale exposant beaucoup de parallélisme d'instruction pouvaient avoir un plus grand débit que des schémas classiques (tels Horner) sur des architectures Intel Haswell.

Finalement, dans le Chapitre 4, nous avons défendu le concept de « méta-libms », c'est-à-dire des environnements de développement permettant de *décrire* une fonction mathématique (à divers niveaux d'abstraction) plutôt que de fournir des implémentations très spécifiques, et qui sont ensuite capable de emphgénérer du code source optimisé pour certains critères, comme les formats d'entrée-sortie, la précision, l'architecture cible, ou bien la vectorisation. Nous avons montré, à travers notre méta-implémentation du logarithme, nos contributions à l'un de ces environnments : Metalibm-lugdunum. Ces contributions incluaient notamment un plus grand support des générateurs de code pour les architectures Intel x86 SIMD, ce qui nous a permis de générer des implémentations du logarithme entièrement vectorisées en C portable ou pour les architectures AVX2. L'avantage de cette méthode est qu'elle permet de factoriser du code commun à de nombreuses implémentations, ce qui autorise des gains en temps de développement potentiellement très importants.

Perspectives

Dans le Chapitre 2, nous avons présenté une heuristique basée sur des algorithmes de Fässler, bien plus performante que notre méthode exhaustive, mais seulement pour les fonctions trigonométriques. Une heuristique similaire pour les fonctions hyperboliques serait intéressante et pourrait probablement se baser sur les travaux de Rothbart et Paulsell [147].

Par ailleurs, intégrer nos tables et mesurer leurs performances dans les implémentations en précision intermédiaires (4096 bits) de Johansson [85] pourrait être intéressant pour estimer leur impact dans des programmes réels.

Dans le Chapitre 3, nos mesures de performances de schémas d'évaluation polynomiale nous font envisager plusieurs perspectives à court et à long terme. D'abord, il serait intéressant d'étudier l'impact sur les performances du choix d'un schéma pour l'évaluation d'une fonction élémentaire. Des tests préliminaires semblent montrer un faible impact avec des polynômes de petit degré. Cependant, au sein d'un environnement comme Metalibm-lutetia [22], se basant sur l'évaluation de nombreux polynômes, les conclusions pourraient être différentes.

Ensuite, il pourrait être intéressant de développer CGPE pour que celui-ci puisse éliminer des schémas si, sur un intervalle cible, ceux-ci peuvent produire des résultats dénormalisés. Gappa pourrait être utilisé pour implémenter une telle heuristique. CGPE pourrait alors garantir qu'un schéma donné ne subira pas de pertes de performances dues à des résultats intermédiaires dénormalisés, par exemple.

Une autre perspective serait de fusionner les générateurs de schémas d'évaluation polynomiale de CGPE dans les différents projets Metalibm. L'interface PythonCGPE présentée au Chapitre 4 permet déjà accéder à des générateurs de CGPE dans Metalibm-lugdunum, mais mériterait peut-être d'être développée.

Finalement, dans le Chapitre 1, section 1.3, mais aussi dans le Chapitre 4, nous avons présenté et défendu les concepts de « méta-libm » et plus généralement de « programmation génératrice » qui semblent prometteurs pour produire des bibliothèques mathématiques *sur mesure* pour les applications haute performance. De plus, cela pourrait réduire les temps de développement et de maintenance en favorisant la factorisation de codes numériques.

De potentielles perspectives pourraient donc être : la fusion de méta-implémentations similaires comme les logarithmes dans différentes bases, en suivant [143], dans Metalibm-lugdunum ; le support des formats binary16 et binary128 et des fonctions en précision mixte. Metalibm-lugdunum propose déjà des opérateurs FMA en précision mixte utiles pour, par exemple, l'apprentissage automatique [21]. En ce sens, il pourrait être intéressant de dresser une liste des opérateurs et formats utiles mais manquants dans les différents projets Metalibm afin de les implémenter ; l'inventaire des méta-blocs qu'il serait intéressant de factoriser. Par exemple, dans Metalibm-lugdunum, il existe une méta-réduction d'argument de Payne et Hanek [137] qui pourrait probablement être abstraite un peu plus afin de pouvoir la réutiliser dans d'autres méta-fonctions ; et, finalement, un sujet de recherche à moyen terme pourrait être la détection automatique et en boîte noire de comportements asymptotiques dans Metalibm-lutetia. En effet, cet environnement fournit déjà un support pour détecter automatiquement et en boîte noire certaines propriétés de périodicité et de symétrie [22]. Cependant, en premier lieu, une approche en boîte blanche semble plus accessible, car pouvant raisonner sur des propriétés algébriques connues. Une telle analyse pourrait permettre d'améliorer l'algorithme de découpage en sous-domaines utilisé par Metalibm-lutetia.

CONTENTS

Dédicace i

Remerciements iii

RÉSUMÉ SUBSTANTIEL EN FRANÇAIS vii
Introduction . . . vii
0.1 Arithmétique Flottante . . . viii
0.1.1 Arithmétique flottante IEEE 754 . . . viii
0.1.2 Évaluation de fonctions élémentaires . . . ix
0.1.3 Génération de code pour les fonctions élémentaires . . . x
0.2 Tables Exactes pour les Fonctions Élémentaires . . . xii
0.2.1 Évaluation à base de tables . . . xii
0.2.2 Tabulation de termes exacts . . . xiv
0.2.3 Implantation et résultats numériques . . . xvi
0.2.4 Comparaisons avec les méthodes existantes . . . xviii
0.2.5 Conclusions et perspectives . . . xix
0.3 Vectorisation de Schémas Polynomiaux . . . xix
0.3.1 Contexte et Motivations . . . xx
0.3.2 Rappels sur l'évaluation polynomiale . . . xx
0.3.3 Comparaison des performances de schémas d'évaluation polynomiale . . . xxi
0.3.4 Influence de l'intervalle d'évaluation . . . xxiii
0.3.5 Vers des schémas plus performants . . . xxiii
0.3.6 Conclusions . . . xxiv
0.4 Méta-Implémentation de Logarithmes . . . xxv
0.4.1 Vectorisation de fonctions élémentaires . . . xxv
0.4.2 Conception d'un méta-logarithme vectorisable . . . xxvi
0.4.3 Méta-implémentation dans Metalibm-lugdunum . . . xxvii
0.4.4 Mesures de performances du code généré . . . xxviii
0.4.5 Conclusions . . . xxviii
Conclusion Générale . . . xxviii

Contents xxxi

List of Figures xxxv

List of Tables xxxvii

Introduction 1

1 COMPUTATION OF ELEMENTARY FUNCTIONS 7
1.1 Floating-Point Arithmetic . . . 8
1.1.1 IEEE 754-2008 binary floating-point formats . . . 10
1.1.2 IEEE 754-2008 required operations and rounding attributes 12
1.1.3 IEEE 754-2008 recommended operations . . . 14

1.2 Evaluation of Elementary Functions 14
1.2.1 The Table Maker's Dilemma 16
1.2.2 Algorithms for the evaluation of elementary functions . 18
1.2.3 SIMD vectorization of elementary functions 23
1.3 Code Generation for Elementary Functions 26
1.3.1 Tools geared towards code generation 26
1.3.2 The MetaLibm ANR project 28
1.3.3 Generative programming for numerical algorithms . . . 29

2 EXACT LOOKUP TABLES FOR ELEMENTARY FUNCTIONS 31
2.1 Introduction 32
2.1.1 Overview of this chapter 32
2.2 Table-Based Evaluation Schemes 33
2.2.1 Usual framework 33
2.2.2 Focus on table-lookup range reductions 35
2.3 Design of an Exact Lookup Table 40
2.4 Application to Sine and Cosine Functions 42
2.4.1 Pythagorean triples and their properties 42
2.4.2 Construction of primitive Pythagorean triples 43
2.4.3 Selection of primitive Pythagorean triples 47
2.5 Implementations and Numeric Results 51
2.5.1 Exhaustive search 52
2.5.2 Characterizing exhaustive results: a first heuristic 57
2.5.3 A much faster heuristic using Fässler's algorithms . . . 59
2.6 Comparisons with Other Methods 64
2.6.1 Costs for the trigonometric functions 65
2.6.2 Costs for the hyperbolic functions 68
2.7 Example on the Trigonometric Sine 69
2.7.1 Tang's table lookup method 71
2.7.2 Exact table lookup method 71
2.7.3 Comparison between both methods 71
2.8 Conclusions and Perspectives 72

3 VECTORIZING POLYNOMIAL SCHEMES 73
3.1 Introduction 73
3.2 Reminders About Polynomial Evaluation 74
3.3 Benchmarking Polynomial Evaluation Schemes 76
3.3.1 Benchmarking classic schemes 78
3.3.2 Influence of the evaluation interval 81
3.3.3 Towards more efficient evaluation schemes 83
3.4 Conclusion and Perspectives 85

4 THE POWER OF META-IMPLEMENTATIONS 87
4.1 Introduction 88
4.2 Algorithm for $\log(x)$ With Faithful Rounding 89
4.2.1 Range reduction 89
4.2.2 How to determine r_i, u, τ, and $-\log(2^\tau \cdot r_i)$? 90
4.2.3 How to ensure faithful rounding? 92
4.3 Guidelines to Build Accurate Enough Programs 93

4.3.1 Program to handle the general flow 94
4.3.2 How to handle the special input 1 in the general flow? . 95
4.4 Implementation Details 95
4.4.1 Branchless program 95
4.4.2 Certified evaluation program 97
4.5 Towards Automated Implementations 98
4.5.1 The Metalibm-lugdunum framework 98
4.5.2 The case of polynomial evaluation schemes 99
4.6 Numerical Results 100
4.6.1 Experimental protocol 100
4.6.2 Performance of the $\log(x)$ function 102
4.6.3 Impact of the table size and subnormal handling 102
4.7 Conclusion and Perspectives 103

General Conclusion and Perspectives 105

Bibliography 109

Abstracts 128

LIST OF FIGURES

Figure 1 Refraction of light at the interface between air and PMMA. 2
Figure 1.1 IEEE 754 Binary Interchange Floating-Point Format. 11
Figure 1.2 Effect of IEEE 754 rounding-direction attributes for positive results. 13
Figure 1.3 Overview of a typical evaluation scheme for an elementary function $x \mapsto f(x)$. 22
Figure 2.1 Graphical representation of the first range reduction for the trigonometric functions. 33
Figure 2.2 Visual representation of the second range reduction with a regularly-spaced trigonometric table. 36
Figure 2.3 Visual representation of the second range reduction with an example of Gal's trigonometric table. 38
Figure 2.4 Visual representation of the Pythagorean triple $(3,4,5)$. 43
Figure 2.5 Primitive Pythagorean triples with a hypotenuse $c < 2^{12}$. 44
Figure 2.6 Visual representation of PPTs and their symmetric counterparts falling into a trigonometric exact lookup table. 46
Figure 2.7 Number of PPTs per entry of a trigonometric table for $p = 7$ with a hypotenuse less than 2^{18}. 48
Figure 2.8 Number of PPTs per entry of a trigonometric table for $p = 7$ with a hypotenuse less than 2^{14}. 49
Figure 2.9 Visual representation of the trigonometric exact lookup table from table 2.3 in the Euclidean plane. 56
Figure 2.10 Visual representation of the hyperbolic exact lookup table from table 2.4 in the Euclidean plane. 56
Figure 2.11 Bit-length n of k with respect to the table index size p for the trigonometric functions. 60
Figure 2.12 Bit-length n of k with respect to the table index size p for the hyperbolic functions. 60
Figure 3.1 Horner's scheme for a degree-3 polynomial. . 75
Figure 3.2 Second-order Horner's scheme for a degree-3 polynomial. 76
Figure 3.3 Estrin's scheme for a degree-3 polynomial. . . 76
Figure 3.4 Means of reciprocal throughputs of three usual schemes in binary32 precision, without FMA. . 78

Figure 3.5 Means of reciprocal throughputs of three usual schemes in binary32 precision, with FMA. . . . 79
Figure 3.6 Means of reciprocal throughputs of three usual schemes in binary64 precision, without FMA. . 80
Figure 3.7 Means of reciprocal throughputs of three usual schemes in binary64 precision, with FMA. . . . 80
Figure 3.8 Latencies of the computation of x^{16} in binary32 by naive exponentiations. 81
Figure 3.9 Latencies of the computation of x^{16} in binary32 by binary exponentiations. 82
Figure 3.10 Mean reciprocal throughputs of lowest-latency schemes of degree 12, sorted by increasing number of multiplications. x^n is normal for $n \in [1; 12]$. 83
Figure 3.11 Mean reciprocal throughputs of lowest-latency schemes of degree 12, sorted by increasing number of multiplications. x^n is subnormal when $n = 12$. 84
Figure 3.12 Mean reciprocal throughputs of lowest-latency schemes of degree 12, sorted by increasing number of multiplications. x^n is subnormal when $n = 8$. 84

LIST OF TABLES

Table 1.1 Basic Binary Floating-Point Formats Parameters. 10
Table 1.2 Selection of IEEE 754-2008 Recommended Correctly Rounded Functions. 15
Table 1.3 Register Size and Availability of Intel's SIMD Extensions. 24
Table 2.1 Exhaustive Search Results for sin and cos. . . . 54
Table 2.2 Exhaustive Search Results for sinh and cosh. . 54
Table 2.3 A Trigonometric Exact Lookup Table Computed for $p = 4$. 55
Table 2.4 A Hyperbolic Exact Lookup Table Computed for $p = 5$. 55
Table 2.5 Prime Factorization of Found Common Multiples for sin and cos. 57
Table 2.6 Prime Factorization of Found Common Multiples for sinh and cosh. 57
Table 2.7 Heuristic Search Results for sin and cos. 58
Table 2.8 Heuristic Search Results for sinh and cosh. . . 58
Table 2.9 "Fässler's" Heuristic Search Results for sin and cos. 63
Table 2.10 Costs of Addition and Multiplication of Expansions. 65
Table 2.11 Computation and Memory Costs for Three Table-Based Range Reductions for Trigonometric and Hyperbolic Functions. 67
Table 2.12 Tang's Table for $p = 4$ With 16-Bit Precision. . 70
Table 3.1 Theoretical latencies for common schemes on a simplified Haswell architecture. 77
Table 4.1 Suitable Values for i (Satisfying Property 1) for Different Combinations of (e_{max}, κ). 91
Table 4.2 Implementation Parameters and Table and Polynomial Coefficient Sizes (# Bytes) for Various Values of i When $p = 24$. 98
Table 4.3 Measured Performances of our Implementations Compared to the GNU Libmvec. 101
Table 4.4 Measured Performances (CPE) on AVX2 of our Generated Implementations, in Binary32 Arithmetic and for Vectors of Size 4 (v4) or 8 (v8), With (sub) or Without (no-sub) Subnormal Handling. 103

INTRODUCTION

> "I have never been so wrong, in all my life."
>
> Thorin Oakenshield,
> in *The Hobbit: An Unexpected Journey*
> by Peter Jackson.

In 2012, a new elementary particle of mass 125 GeV was discovered independently by two CERN experiments: CMS (Compact Muon Solenoid), and ATLAS (A Toroidal LHC ApparatuS) [2, 23].[1] A bursting fever quickly spread in the general media that CERN was about to announce that this new particle could well be the famous Higgs boson. The Higgs boson is an elementary particle that was expected by the Standard Model of physics since the 1960's but that had never been experimentally observed with enough confidence. Up to now, in 2018, many experiments around the world have gone along with the hypothesis that indeed, this new particle is the one that physicists had been looking for for the past 50 years [1, 24, 25, 63, 87, 88, 157]. This discovery was both comforting the Standard Model and disturbing our understanding of *mass*. It was such a breakthrough that the following year, Peter Higgs and François Englert, two of the six theoretical physicists who predicted the existence of the particle, received the 2013 Nobel Prize in Physics for the theoretical discovery they had made in 1964.

But what few media related was that the experimental discovery would probably have taken much more time without the computing power provided by CERN's Worldwide LHC Computing Grid. The CERN experiments produced 13 petabytes (13×10^{15} B) of raw data in 2010 [19]. To give a perhaps more striking comparison, this represents a pile of Blu-Ray discs about twice as high as the Eiffel Tower. In addition, these data require complex processing to precisely reconstruct the high energy particle collisions, using state-of-the-art mathematical models. Such complex data processing is automated by deferring all the computations to specialized software on the Grid. However, as pointed out by CMS scientists, the time spent by the software evaluating elementary functions such as the exponential or the logarithm can represent a good share of the overall execution time [73, 138].

In physics, another, perhaps simpler example where the sine function is used is the refraction of light at the interface of two media m_1 and m_2 in the Snell-Descartes law:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2$$

[1] CERN (Conseil Européen pour la Recherche Nucléaire [*European Organization for Nuclear Research*]) studies the most basic constituent of matter: the fundamental particles. See https://home.cern/about.

where n_i and θ_i are, respectively, the index of refraction and the angle of incidence of the medium m_i. This well-known physical phenomenon is illustrated in Figure 1 at the interface between air and Poly(methyl methacrylate) (PMMA). Even with a rather high measuring error, we can see that $\theta_{air} \approx 60°$ and that $\theta_{PMMA} \approx 35°$. Using the free-software "basic calculator" GNU bc, anyone[1] can devise an approximation of the refractive index of PMMA in the visible spectrum, assuming it does not depend on the wavelength and that the refractive index of air is 1:

```
echo 'pi=4*a(1); n_pmma = s(60*pi/180)/s(35*pi/180); \
    n_pmma' | bc --mathlib
```

This shell command line outputs a refractive index of 1.5099... with 16 more digits. This value lays inside the bounds 1.485 and 1.510 found in the literature for the visible spectrum [8]. Simply said, the above command line asks bc to approximate π and output the ratio between $\sin(60°)$ and $\sin(35°)$. But under these apparently simple operations, bc had to approximate two sines (plus an arctangent to approximate π) and a few multiplications and divisions. As the result

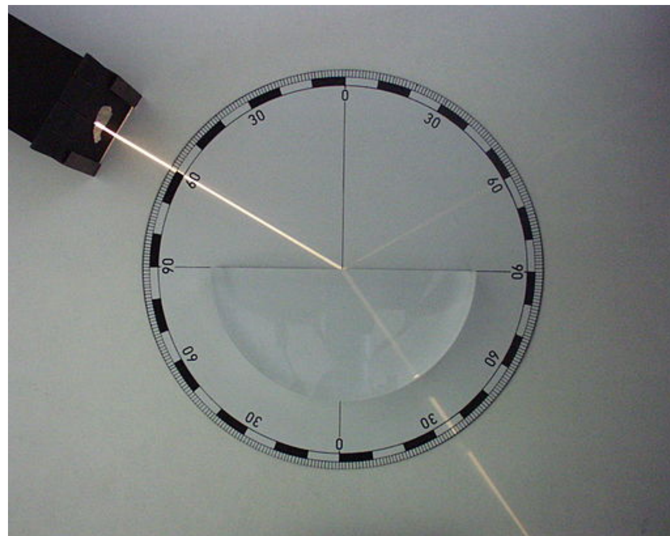


Figure 1 – Refraction of light at the interface between air and PMMA. Credit: Zátonyi Sándor, https://commons.wikimedia.org/wiki/File:Fénytörés.jpg, CC-BY-SA 3.0.

is displayed almost instantaneously, the naive user I was before the beginning of my Ph.D. would have thought that this computation was easy, and with only one final rounding. I had never been so wrong!

CHALLENGES OF THIS THESIS

The underlying challenges of this thesis are the design of efficient software implementations of elementary functions in floating-point arithmetic, with a particular focus on their optimization for current

[1] Well, perhaps only GNU users, to be honest.

vector micro-architectures providing SIMD instructions. Indeed, after the basic arithmetic operations that are addition, subtraction, multiplication, division, and square root, *elementary functions* such as the sine and cosine functions, the exponential or the logarithm functions, are the second most basic resource in scientific computing. Their computation or approximation has a long history. It dates back to at least Babylonian mathematics, for which many tables of reciprocals and square roots have been found. The famous Plimpton 322 clay tablet (circa 1800 BC), which lists 15 Pythagorean triples, might even have served to obtain sine and cosine approximations, but this assertion is still debated [89]. From the end of XVIII$^{\text{th}}$ century to the beginning of XX$^{\text{th}}$ century, approximation theory was thoroughly developed, progressively leading to optimal polynomial and rational approximations to continuous functions [158]. As polynomial and rational approximations are only made of elementary arithmetic operations, they represent a very efficient way to "compute" transcendental functions. However, even using these mathematical objects, efficiently and accurately approximating a function is already a pretty hard task to do, as we will see in Chapter 2 for trigonometric and hyperbolic functions, or in Chapter 4 for the logarithm function.

In the current context, most modern general-purpose processors are not designed to accurately approximate even basic elementary mathematical functions. On some general-purpose processors though, there exist hardware instructions (that is, *code* that the *silicon* can understand) that can approximate the sine or the cosine of a value. But we see at least three issues with these hardware instructions:

1. They are not versatile, i.e. they can only be used on fixed-length inputs [78, § 8.3.7];
2. They are not very accurate on their whole domain [78, § 8.3.10];
3. They are usually split into many micro-operations or μ-*ops*, i.e. basic hardware instructions such as memory loads or arithmetic instructions [65, 66]. This makes them very slow compared to more basic instructions.

Instead of using these problematic hardware instructions, an accurate computing software tool like `bc` or `sollya`[1] will call software routines that may have more latency but may also be more accurate, if not correctly rounded. In Chapter 1, we will define such terms as *correctly rounded*, but for the time being, we only try to give the reader some intuitions about the evaluation of elementary functions. Although rather expensive, these software routines terminate in a very short time that is not perceptible to the human eye. We are talking in milliseconds here, but this is already much more than what a processor can do for basic tasks. Adding two numbers, for example, only takes a few nanoseconds, hence being about 1 000 000 faster.

Therefore, finding the right trade-off between performance and accuracy of elementary mathematical functions can be of paramount

[1] Sollya is a software tool used in this thesis and described in Chapter 1.

importance when millions of such computations are involved, e.g. in the CERN CMS experiment software. Hence, a current challenge is to develop techniques and tools to automate the process of writing mathematical libraries (called *libm*'s) optimized for a given micro-architecture, instead of providing general purpose libraries. This way the user can generate his/her own libm for his/her specific problem, that is, for example, optimized for custom input ranges or hardware.

In this direction, a particular effort started in 2014 as the ANR[1] MetaLibm project,[2] whose goal is to develop such generation tools for mathematical functions and digital filters [22]. This project has involved PEQUAN (LIP6, Sorbonne Université / CNRS), Socrate (Citi, INSA Lyon / Inria), CERN, and DALI (LIRMM, Université de Perpignan Via Domitia / CNRS) teams. Members of MetaLibm tackled the above goal from different angles, described in more detail in Chapter 1. A first angle consists in open-ended code generators. Particularly, we consider a "function" (in every sense) as a black box to be implemented in floating-point arithmetic, and we produce source code valid for a specific input range [92, 94]. A second angle addresses the design of function implementations that are (more or less) compliant with the standard libm, and optimized for different software and hardware [47, 95, 100, 110, 138]. A third angle is to focus on the particular case of digital filter implementations in fixed-point arithmetic [172–174]. This thesis takes place in the context of this project. Our work targets the generation of high performance codes under architectural constraints, for the implementation of mathematical functions. The contributions are detailed below.

CONTRIBUTIONS AND OUTLINE OF THIS THESIS

This thesis is organized in six chapters, including this introduction, a state-of-the-art, and a conclusion. The end of this section is devoted to the brief presentation of each scientific chapter.

Chapter 1 – Computation of Elementary Functions

This chapter gives a brief overview of computer arithmetic. Particularly, it defines what we call *floating-point arithmetic*, and related objects and functions used in this thesis. Then, it presents state-of-the-art designs for the evaluation of elementary functions, and code generation frameworks that can help automate performance tuning, with a particular focus on the techniques and tools used in this thesis.

Chapter 2 – Exact Lookup Tables for Elementary Functions

Various approaches have been developed to build trigonometric and hyperbolic functions. Among these, let us cite the table-lookup based

[1] *L'Agence ationale de la recherche*, the French National Research Agency.
[2] See `http://metalibm.org`.

approaches. These tables generally hold tabulated values of the functions to be implemented. However these values are *approximations* embracing rounding errors. Hence, their use makes it more complicated to guarantee a certain accuracy on the output.

This chapter presents a new approach to design "exact lookup tables", that is, where error has been removed, together with efficient heuristics to build them. These are based on mathematical objects called *Pythagorean triples*. First, some mathematical background on Pythagorean triples allows us to formally prove a lower bound on the main computed values. Second, we use relations existing between these functions and Pythagorean triples to give exact representations of tabulated values instead of intrinsically rounded values that have been used up to now. In comparison to Tang's and Gal's methods, two well-known ones, and when targeting correct rounding in double precision, we estimate theoretical gains of up to 29% and 42% in memory accesses and floating-point operations, respectively, during the reconstruction step.

This work has led to two publications in conferences, at Compas 2015 [98] and ASAP 2015 [97], and one publication in an international journal, IEEE Transactions on Computers [96].

Chapter 3 – Vectorizing Polynomial Schemes

Polynomial approximations take an important part in the evaluation of elementary functions [123]. They can often be combined with table-lookup methods. The efficiency of their evaluation on a given micro-architecture can greatly impact the performance of the whole function implementation.

This chapter presents an analysis of the performances of classic polynomial evaluation schemes, namely Horner's and Estrin's, as well as of automatically-generated polynomial evaluation schemes, in the context of auto-vectorization. Polynomial evaluation schemes correspond to implementations of polynomial approximations. Auto-vectorization is the ability for a compiler to automatically transform scalar instructions into *vectorized* instructions that can treat multiple data simultaneously. We observe that on Intel Haswell architectures, the production of subnormal numbers as a result of arithmetic operations can increase the latency of such operations by a factor up to 50.

This work has led to one publication in a conference, at Compas 2016 [99].

Chapter 4 – The Power of Meta-Implementations

Vectorization can increase the throughput of compute-intensive applications on supporting architectures [138]. The diversity of micro-architectures (Skylake, Haswell, Sandy Bridge, or Xeon Phi, ... for x86 micro-architectures, for example) complicates the development

of efficient implementations that exploit at best the specificities of the underlying hardware. To tackle this issue, we favor the use of code generation tools to automate the implementation process optimized for specific hardware.

This chapter concerns meta-implementations of elementary functions, vectorization, and code generation. We present the design and implementation process of a *vectorizable* natural logarithm within the Metalibm-lugdunum code generator framework. Particularly, it presents our contributions to the Metalibm-lugdunum framework, which consist in the extension of the Intel x86 backends, the introduction of reusable *meta-blocks*, the integration of PythonCGPE that enables the automatic generation of efficient polynomial schemes, and the vectorized meta-logarithm itself. Careful optimizations to avoid penalties due to subnormal numbers or to help the code generator produce vectorized code are presented for our meta-implementation. Comparisons with the free GNU Libmvec available in glibc 2.22 are reported and show that meta-implementations can be competitive while bringing decisive advantages such as easier code maintenance and reduced development times.

This work was done in collaboration with Kalray, and it has led to a submission to ASAP 2018 [95], which is currently under review.

1

COMPUTATION OF ELEMENTARY FUNCTIONS

> "You take the red pill: you stay in Wonderland, and I show you how deep the rabbit-hole goes."
>
> Morpheus, in *The Matrix*, by the Wachowski brothers.

In this chapter, our goal is to give the reader the keys to understanding the rest of this thesis. Hence, let us recall a few basic bricks about how *computers* do what they are meant to do: *computing*. Since the late 1950's, transistors have formed the core components of electronic circuits. Transistors are *active* electronic components, which means they need an external source of energy to work. They can be used as electronic switches, in one of two states commonly called "on" and "off". These two available states make Boolean logic relatively easy to implement through logic gates. In turn, Boolean logic allows to build *universal Turing machines*, such as our modern computers [128]. A universal Turing machine is a special kind of theoretical computer that can compute *anything* that is *computable*, provided an infinite amount of memory. Thus, if we forget the "infinite memory" requirement of a truly universal Turing machine, modern computers, because they implement Boolean logic, are fundamentally able to compute anything that is computable. Computability theory is a fascinating but complex field of research that is not the subject of this thesis. Hence, we will simply admit that the elementary functions we deal with in this thesis are all *computable* for all their (computable) inputs. The interested reader can find more details in [91].

Back to our two-state transistors, Boolean logic makes *binary arithmetic* easy to implement, which is why nowadays, the vast majority of computers use binary digits (*bits*, whose value can be either 0 or 1) to convey information and perform basic computations such as addition or multiplication [31]. However, this has not always been the case. Indeed, many civilizations have embraced the radix-10 number system probably because they have been used to counting on their two pentadactyl ("five-finger") hands.[1] Therefore, most calculating tools and mechanical computers, such as abaci or the Pascaline, used radix 10 or radix 5. In the early development of two-state electronic

[1] The author humbly counts himself as part of such civilizations, with his two pentadactyl hands, although he also does his best to be in the set of people counting on their 10 hands, having 101 fingers each.

calculators, many early computers would still internally use decimal digits coded on four bits (a representation format generally called Binary Coded Decimal or BCD), mostly for commercial applications [75, p. 20, 31]. Most handheld calculators, e.g. the Texas Instruments TI-89 and TI-92 calculators, Casio calculators, and Hewlett-Packard calculators, are still using decimal arithmetic internally [31, 119]. Financial software often do the same, usually through programming language facilities. Such an arithmetic is still useful when exact representation of radix 10 decimal numbers is required, because binary arithmetic generally cannot represent these numbers exactly, as we will see in Section 1.1.

The memory of a computer is generally addressed by *words* made of a finite number of bits. Thus, the set of representable real numbers on one word is essentially finite. Therefore, only finite subsets of real numbers can fit in a machine word, and allow approximations of real-number arithmetic. Among the many representation systems used for these approximations on modern computers, the one required by the IEEE 754 standard for floating-point arithmetic[1] has been the most pervasive since its adoption in 1985 [123]. Hence we begin this chapter with a short introduction to floating-point arithmetic, and especially its IEEE 754-2008 standardization, because it is the arithmetic we use all along this thesis. Then we present general ideas for the design and implementations of efficient and accurate elementary functions, as well as pitfalls of approximating transcendental functions. Finally, we focus on state-of-the-art efforts that deal with assisted or automatic generation of high-performance code for the evaluation of elementary functions.

1.1 FLOATING-POINT ARITHMETIC

We are accustomed to using *fixed-point* arithmetic on a daily basis: when calculating a bill, or measuring volumes or weights, the radix point usually keeps a *fixed* position. Consider the following fixed-point computation in radix 10:

$$42 + 0.17 = 42.17.$$

Here, the decimal point is fixed before and after the addition: it always indicates where the *integer* part of a number ends and where its *fractional* part begins. However, when dealing with very large or very small numbers, it is much more common to use the *scientific notation*, where the radix point can *float*. Consider Coulomb's constant. It is the proportionality factor in Coulomb's law, which describes the force of interaction between two electrically charged particles. The constant is noted k_e and it can be made exact [57]:

$$k_e = 8\,987\,551\,787.368\,176\,4\ \mathrm{N\,m^{-2}\,C^{-2}}.$$

[1] IEEE stands for the Institute of Electrical and Electronics Engineers. More info can be found at `https://ieee.org/about/` and in [76, 77] for the 754 standards.

Notice how the scientific notation

$$8.987\,551\,787\,368\,176\,4 \times 10^9 \ \mathrm{N\,m^{-2}\,C^{-2}}$$

presents the same information, but is easier to read and *understand*. Indeed, since the position of the decimal point is embedded in the exponent 9, we immediately know the *order of magnitude* of the constant compared to the fixed point notation, which requires counting the number of decimal digits. Moreover, in many fields, it is often more useful to know the order of magnitude of a numerical result than all its digits. Finally, multiplication using the scientific notation is easier: only a multiplication with small significands is needed, while the exponents are simply added together. For instance, two elementary charges[1] that are one meter apart are subject to a force of interaction, the magnitude F of which is given by Coulomb's law:

$$\begin{aligned} F &= k_e \times e^2 \\ &\approx (8.99 \times 10^9) \times (2.57 \times 10^{-38}) \\ &= (8.99 \times 2.57) \times 10^{9-38} \\ &= 23.1043 \times 10^{-29} \ \mathrm{N}. \end{aligned}$$

Next, a *renormalization* step and a *rounding* step can be performed so that F be rewritten $F \approx 2.31 \times 10^{-28}$. In this example, the renormalization step shifted the decimal point and adjusted the *exponent* value (-29 was incremented to -28) so that the *significand* 2.31043 stay between 1 and 10. Then the rounding step kept only three significant figures, rounding the significand to the nearest 3-digit number 2.31.

Floating-point arithmetic is in many ways very similar to scientific notation. It consists in representing a real number x as the product of a nonnegative significand (or *mantissa*) $m \geqslant 0$ and an integral power (the *exponent*) e of a *radix* β. To be able to represent negative numbers, a *sign* bit $s \in \{0, 1\}$ is added to the representation so that finally:

$$x = (-1)^s \cdot m \cdot \beta^e. \qquad (1.1)$$

In this thesis we mainly consider $\beta = 2$, also known as *binary* floating-point arithmetic, in floating-point environments conforming to the IEEE 754-2008 standard [76]. However, whenever we believe it can be useful, we try to give intuitive examples in decimal arithmetic ($\beta = 10$). In the following, we present only the IEEE 754-2008 features that are useful in this thesis:

- floating-point formats and representation,
- required operations and rounding modes,
- and recommended correctly rounded functions.

[1] The elementary charge, usually noted e, is the electric charge carried by a proton. Its value is about 1.602×10^{-19} C.

Table 1.1 – Basic Binary Floating-Point Formats Parameters.

Parameter	binary32	binary64	binary128
k, storage width in bits	32	64	128
p, precision in bits	24	53	113
$emax$	127	1023	16383

1.1.1 *IEEE 754-2008 binary floating-point formats*

Because the storage of a floating-point number is only possible on a finite amount of memory, the data s, m, e must be stored on a finite number of bits. For the sign s, one bit is enough, but for the fields m and e, the IEEE 754-2008 standard specifies different lengths or ranges, which partially define floating-point *formats*. For binary floating-point arithmetic, i.e. when $\beta = 2$, a floating-point format is almost fully characterized by three integers [123, § 2.1]:

- its *precision* $p \geqslant 2$, which is the maximum number of significant digits of m;
- the minimum exponent $e_{\min}$;
- and the maximum exponent $e_{\max}$.

Since the 2008 revision requires that $e_{\min} = 1 - e_{\max}$ for all formats, the above list can be reduced to p and one of the exponents extrema. Furthermore, floating-point numbers that conform to such a format are required to be *normalized* so that their representation be unique. In this case, for a precision-p floating-point number $x = (-1)^s \cdot m \cdot 2^e$:

- either $m = 1.m_1 \cdots m_{p-1}$ and $e_{\min} \leqslant e \leqslant e_{\max}$, and x is called a *normal* number;
- or $m = 0.m_1 \cdots m_{p-1}$ and $e = e_{\min}$, and x is called a *subnormal* number.

For instance, in a radix-10 floating-point format with precision 3 and $e_{\min} = -2$, the number $-123 = -1.23 \cdot 10^2$ is normal, while the number $0.0042 = 4.2 \cdot 10^{-3} = 0.42 \cdot 10^{-2}$ is subnormal. Note that subnormal numbers have at most $p - 1$ significant digits. Subnormal numbers allow for a gradual flush towards zero, preserving some properties but sometimes at much more expensive computation costs as we will see in Chapter 3.

The two formats that we use in this thesis are the *binary32* (sometimes called *single precision*) and the binary64 (sometimes called *double precision*) floating-point formats. The parameters specifying these formats as well as the binary128 format are summarized in Table 1.1.

As we write this thesis, the latest version of the C programming language (ISO C11) provides only binary32 and binary64 formats as

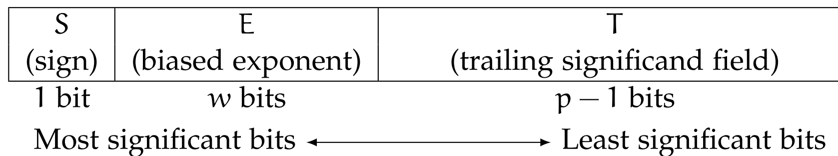


Figure 1.1 – IEEE 754 Binary Interchange Floating-Point Format.

basic types with the `float` and `double` keywords, respectively [82]. Although a *double-extended* format can be available on some systems and compilers via the `long double` type specifier, ISO C11 only requires it to be *at least* as precise as the binary64 format. However, some compilers such as GCC (at least since version 4.8) provide the binary128 format on some target architectures as a language extension, via the `__float128` or `_Float128` types. On x86_64 architectures, which have no hardware support for binary128, arithmetic operations on binary128 words are usually carried out in software.[1]

Finally, to fully characterize a floating-point format, the IEEE 754-2008 standard also specifies *binary interchange formats*, i.e. the representation of binary floating-point numbers in machine words:

- a sign bit $S \in \{0, 1\}$;
- a w-bit biased exponent $E \in [0 \mathinner{..} 2^w - 1]$, where $[a \mathinner{..} b]$ denotes the set of integers no less than a and no greater than b;
- and a $(p-1)$-bit trailing significand field $T = m_1 m_2 \cdots m_{p-1}$.

This is summarized in Figure 1.1. Then a binary interchange format (S, E, T) is interpreted as a floating-point number if and only if E is in the range $[0 \mathinner{..} 2^w - 2]$. Then the represented number is

- the normal number $(-1)^S \cdot (1 + T \cdot 2^{-p}) \cdot 2^{E - e_{\max}}$ if $E \geqslant 1$ and $T \neq 0$;
- or the subnormal number $(-1)^S \cdot T \cdot 2^{e_{\min}+1-p}$ if $E = 0$. Thus the signed zero ± 0 is represented when $T = 0$.

In the case $E = 2^w - 1$, the interchange format represent special values:

- the signed infinity $\pm\infty$ if $T = 0$;
- or "not a number" (NaN) if $T \neq 0$. NaNs[2] are used to signal invalid operations without interrupting the program flow, and therefore propagate through arithmetic computations[3].

In this thesis, we indifferently denote by $\mathbb{F}$ the different sets of values that can be represented by binary interchange formats.

[1] `https://gcc.gnu.org/onlinedocs/gcc/Floating-Types.html`
[2] Not to be confused with Indian naans. These are delicious.
[3] Except for some special cases of specific elementary functions, with which this thesis does not deal, fortunately.

Before the wide adoption of the IEEE 754 standard, there were various floating-point arithmetics that were often incompatible with one another [123, § 1.1]. For instance, the UNIVAC 1107 used radix-2 with a precision $p = 27$ and an exponent of width $w = 8$ biased by 128 [168], whereas the IBM System/360 provided an environment with $\beta = 16$, $p = 24$ or 56 and $w = 7$ with a bias of 64 for its single and double precision formats, respectively [75]. Furthermore, these architectural differences were the cause of major portability and results reproducibility issues, as well as difficulties in writing numerical algorithms [86]. The need for standardization was met with an involvement of the main manufacturers and academics in the redaction of the first IEEE 754 standard for binary floating-point arithmetic proposal, which was accepted in 1985 [77]. This version of the standard was quickly adopted by hardware manufacturers so that, in 2018, most if not all floating-point units are IEEE-754 compliant. The 2008 version presented in this section is a major revision of the 1985 one. It standardized the Fused Multiply and Add (FMA) operation, as well as quadruple precision (binary128), and included decimal floating-point arithmetic, among other changes. A minor revision is being prepared for release in 2018.[1]

1.1.2 *IEEE 754-2008 required operations and rounding attributes*

Basic arithmetic operations such as addition, subtraction, multiplication, division, square root, and FMA are required to be *correctly rounded* by the IEEE 754-2008 standard. An operation is said to be correctly rounded if it always return the rounding of its infinitely precise result according to a rounding-direction attribute. There are four required rounding-direction attributes for the correct rounding property in IEEE 754-2008 binary floating-point arithmetic, illustrated in Figure 1.2:

ROUNDTIESTOEVEN (RN) Return the floating-point number closest to the infinitely precise result. If the mathematical result is exactly halfway between two floating-point numbers, return the floating-point number that has an even least significant digit.

ROUNDTOWARDPOSITIVE (RU) Return the floating-point number closest to and no less than the infinitely precise result.

ROUNDTOWARDNEGATIVE (RD) Return the floating-point number closest to and no greater than the infinitely precise result.

ROUNDTOWARDZERO (RZ) Return the floating-point number closest to and no greater in magnitude than the infinitely precise result.

Hence, whenever an atomic operation such as a floating-point addition is performed, a rounding occurs. For a computed result $\hat{y}$ corre-

[1] See http://754r.ucbtest.org/.

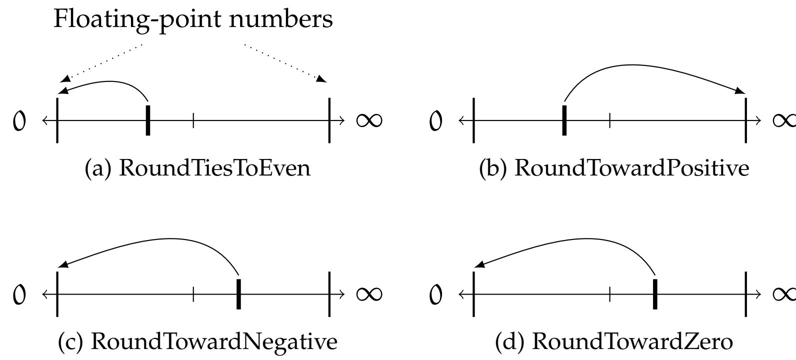


Figure 1.2 – Effect of IEEE 754 rounding-direction attributes for positive results.

sponding to the mathematical, infinitely precise result y, the rounding error can be quantified by the *absolute error*

$$\delta = |y - \hat{y}|\,.$$

Often, we may use the ulp function to quantify the absolute error of the operation. ulp stands for "unit in the last place". Several definitions of the ulp function coexist, but we choose Definition 1, from [123, p. 33]:

Definition 1 (Goldberg's definition extended to real numbers). *If* $|x| \in [2^e, 2^{e+1})$, *then* $\text{ulp}(x) = 2^{\max(e, e_{min}) - p + 1}$.

For example, in binary32,

$$42 \in [2^5, 2^6) \implies \text{ulp}(42) = 2^{-18}.$$

This means that the smallest binary32 number greater than 42 is $42 + \text{ulp}(42) = 42 + 2^{-18}$, and that their midpoint (which is not a binary32 number) is $42 + \frac{1}{2}\,\text{ulp}(42)$. Note that $\text{ulp}(0) = 2^{e_{\min} - p + 1} = 2^{-149}$ for the binary32 format. Using Figure 1.2, one can get the intuition that the error δ of a correctly-rounded arithmetic operation returning x is such that

- $|\delta| \leqslant \frac{1}{2}\,\text{ulp}(x)$ for the RN attribute;
- or $|\delta| \leqslant \text{ulp}(x)$ for the RU, RD or RZ attributes.

Sometimes, e.g. for a sequence of operations, we may measure the absolute error, or the *relative error* of the computed result [70]. For the latter, we use Definition 2:

Definition 2 (Relative error). *If* $x \in \mathbb{R}^*$ *is approximated by a finite-precision floating-point number* $\hat{x}$, *then the relative error of* $\hat{x}$ *with respect to* x *is*

$$\left|\frac{x - \hat{x}}{x}\right|.$$

This metric informs us about the approximate number of significant digits in the computed result of an operation: if the relative

error is close to β^{-n}, then there are roughly n radix-β significant digits in the result. For example, $x = 0.1$ is not representable in binary32. Indeed,

$$\mathrm{RN}(0.1) = 0.100\,000\,001\,490\,116\,119\,384\,765\,625.$$

Hence, the relative error due to rounding 0.1 in binary32 is

$$\left|\frac{0.1 - \mathrm{RN}(0.1)}{0.1}\right| \approx 1.4999\cdots \times 10^{-8}.$$

This means that RN(0.1) has roughly eight (slightly less, actually) significant decimal digits. This metric is often used in the context of elementary functions, in conjunction with the absolute error. In the 2008 standard, elementary functions are part of *recommended operations*, which are the topic of the next subsection.

1.1.3 *IEEE 754-2008 recommended operations*

Besides required operations presented in Section 1.1.2, the IEEE 754-2008 standard specifies *recommended* correctly rounded operations, written as named functions. These operations are neither required, nor, if provided, necessarily correctly-rounded. A selection of these functions is presented in Table 1.2. As can be seen, these operations correspond to various elementary functions, some of which being algebraic, like the Euclidean length $\sqrt{x^2 + y^2}$ or the reciprocal square root $1/\sqrt{x}$, while most of them are *transcendental*, like trigonometric and hyperbolic functions $\sin, \cos, \sinh, \cosh$, or logarithm functions $\log, \log_2, \log_{10}$.

The major issue arising with transcendental functions is their correct rounding, as it is hard to tell in advance how much accuracy is required to ensure this property [107]. Hence, as we will see in Section 1.2, many libraries have been trading accuracy for performance by providing *faithfully-rounded* elementary functions. Faithful rounding is the property for an arithmetic operation whose infinitely precise result is x to return *one of* $\mathrm{RU}(x)$ or $\mathrm{RD}(x)$. Note that it is *not* a rounding attribute. Therefore the absolute error δ of a faithfully rounded result x is such that $|\delta| \leqslant \mathrm{ulp}(x)$.

The difficulty to correctly round elementary functions is related to the Table Maker's Dilemma, which we briefly explain in the next section. Then we present classic designs of elementary functions evaluation, as well as the recent efforts to design efficient *vectorized* functions by exploiting the available low-level parallelism of modern processors architectures.

1.2 EVALUATION OF ELEMENTARY FUNCTIONS

As we have seen in the Introduction, elementary functions are pervasive in many scientific or financial fields, and their evaluation of-

Table 1.2 – Selection of IEEE 754-2008 Recommended Correctly Rounded Functions.

Operation	Function	Domain
exp	e^x	$[-\infty, +\infty]$
log log2 log10	$\log_e(x)$ $\log_2(x)$ $\log_{10}(x)$	$[0, +\infty]$
logp1	$\log_e(1+x)$	$[-1, +\infty]$
pow(x, y)	x^y	$[-\infty, +\infty] \times [-\infty, +\infty]$
sin cos tan sinPi cosPi	$\sin(x)$ $\cos(x)$ $\tan(x)$ $\sin(\pi \cdot x)$ $\cos(\pi \cdot x)$	$(-\infty, +\infty)$
asin acos	$\arcsin(x)$ $\arccos(x)$	$[-1, +1]$
atan	$\arctan(x)$	$[-\infty, +\infty]$
atanPi	$\arctan(x)/\pi$	
atan2(y, x)	$\arctan(y/x)$	$[-\infty, +\infty] \times [-\infty, +\infty]$
atan2Pi(y, x)	$\arctan(y/x)/\pi$	
sinh cosh tanh	$\sinh(x)$ $\cosh(x)$ $\tanh(x)$	$[-\infty, +\infty]$
asinh	$\operatorname{arsinh}(x)$	$[-\infty, +\infty]$
acosh	$\operatorname{arcosh}(x)$	$[+1, +\infty]$
atanh	$\operatorname{artanh}(x)$	$[-1, +1]$

ten require tradeoffs between accuracy and performance. However, since most are transcendental functions, they are subject to the Table Maker's Dilemma (TMD).

1.2.1 *The Table Maker's Dilemma*

Most functions of Table 1.2 are transcendental, i.e. for most finite precision floating-point inputs, they return a transcendental number. Transcendental numbers do not have a repetitive sequence of digits in their fractional part, so they can be neither a finite precision floating-point number nor a midpoint between two finite precision floating-point numbers, but they may be extremely close to such values. As can be seen in Figure 1.2, rounding values can change at these boundaries, which is why they are sometimes called *rounding breakpoints* [123, p. 406]. Therefore, the accuracy required to correctly round an approximation of a transcendental number can be much higher than the target precision.

For instance, let us take the cosine function $\cos(x)$, with, say, the RU rounding direction attribute. This example was given in [107]. The first decimal digits of $\cos(1.149)$ are

$$0.409\,400\,000\,428\,5\cdots. \tag{1.2}$$

Therefore, in precision $p = 4$ decimal digits, the correct rounding towards positive (RU) of $\cos(1.149)$ is 0.4095. But because the exact value is a transcendental number, it cannot be computed in a finite number of steps. Hence, let us assume that an intermediate result $\hat{y}$ was first computed on twice as many digits as p with an absolute error δ such that $|\delta| \leqslant 10^{-6}$. In this thesis, we abstract absolute error notation by writing δ_i to denote an absolute error less than or equal to β^i in magnitude, e.g. $|\delta_{-6}| \leqslant 10^{-6}$ in this example. Therefore let $\delta = \delta_{-6}$. Then we have

$$\cos(1.149) = \hat{y} + \delta_{-6}.$$

If, e.g. $\hat{y} = 0.409\,399\,91$, then $\delta_{-6} = -8.042\,858\,708\,1\cdots \times 10^{-8}$. We see that $\text{RU}(\hat{y}) = 0.4094$, which is not the correctly rounded result. Moreover, without knowing the exact value of δ_{-6}, we know that

$$\cos(1.149) \in \left[\hat{y} - 10^{-6}, \hat{y} + 10^{-6}\right] = [0.409\,398\,91, 0.409\,400\,91].$$

From this equation, we deduce that the correct rounding (RU) of $\cos(1.149)$ is either 0.4094 or 0.4095, but as is, we cannot decide. The absolute error of the intermediate result $\hat{y}$ is too high. From the digits in (1.2), we see that computing an approximation within an error bound δ_{-10} such that $|\delta_{-10}| \leqslant 10^{-10}$ would be enough to correctly round $\cos(1.149)$. But for other inputs, the maximum error bound may be less than δ_{-10}! Hence a natural question comes to mind:

> "What is the maximum absolute error bound δ_{-m} on $\hat{y}$ such that for all possible x, $\mathrm{RU}(\hat{y}) = \mathrm{RU}(\cos(x))$?"

This is what the TMD is about: *solving* the TMD roughly means finding the minimal integer m answering such a question for a given format and rounding attribute, for all possible inputs of a function.

For the binary32 format, exhaustive search can solve the TMD in reasonable time. But as the TMD is exponentially hard to solve with the size of the inputs, exhaustive search is much more compute intensive for the binary64 format. Fortunately, clever algorithms have been designed to solve the TMD for many elementary functions in binary64 [71, 105–107]. It is now known that approximating most functions on roughly three times the format precision will allow for correct rounding. Yet, this extra-precision is seldom needed and has a substantial overhead. That is why correct rounding is often achieved using a strategy in several steps known as *Ziv's onion-peeling*, which can be explained by the *stochastic* or *random model*.

In the random model, it is assumed that an elementary function behaves like a uniform pseudo-random bit generator, i.e. that every bit of a result has a probability $1/2$ to be either a zero or a one. This is not true, but experiments show that this simple model is in fact quite accurate [123]. A direct consequence of this model is that any bit string of length n has a probability 2^{-n} to appear at any position in the result. In particular, if $\{d^n\}$ represents a bit string made of n bits set to d, then the bit strings $\{0\}^n$, $\{1\}^n$, $1\{0\}^{n-1}$ and $0\{1\}^{n-1}$ have probability 2^{-n} to happen after the bits that are to be stored. According to the random model, this means that correct rounding may be achieved for any rounding direction with probability $1-2^{-n}$, provided a temporary result is computed with n digits more than the targeted precision. For example, in binary64, if an intermediate result is computed on 66 bits, which means 13 additional bits compared to the target precision $p = 53$, the random model predicts that about 1 result in $2^{13} = 8192$ will be rounded with no guarantee. This corresponds to a guaranteed rounding rate near 99.999%.

Ziv followed this model to implement his correctly rounded mathematical library [181]. The principle is intuitive. First, compute an approximation with n extra bits of precision, which allows correct rounding most of the time. If correct rounding is not possible, compute a new approximation on more bits. As the algorithm was not proven to always terminate in a reasonable amount of operations, Ziv limited the extra precision of his last step, so that the probability of rounding incorrectly was astronomically small [46].

The CR-Libm effort combined Ziv's strategy with the solutions to the TMD to provide a binary64 correctly-rounded libm [34]. They pushed for the concept of only two steps in Ziv's algorithm: a first "quick" step that allows correct rounding most of the time, and, following the TMD solutions, an "accurate" step that guarantees correct rounding for hard-to-round cases. Such design is now very common for correctly-rounded transcendental functions. The decision of running the accurate phase relies on a simple test involving the compu-

tation of a tight error bound [41, 167]. In the next section, we will see different methods of approximations that allow such "quick" and "accurate" steps in elementary function evaluations.

1.2.2 *Algorithms for the evaluation of elementary functions*

As mentioned earlier, elementary functions recommended by the IEEE 754 standard are mostly transcendental. This entails that they generally need an unbounded number of arithmetic operations to be evaluated. Therefore, different techniques have been developed to *approximate* such functions with a finite number of arithmetic operations. In this section, we present different algorithms that serve the purpose of approximating elementary functions.

Approximation methods

One of the simplest and earliest methods to approximate a function is to store precomputed values in *mathematical tables*, and to look up the value corresponding to an input when needed. This technique was used centuries ago by the Babylonians, in Greece, and in Ancient India, to get fast approximations of the basic trigonometric functions. The precomputation step was obviously the most difficult one. It may rely on mathematical identities, e.g. $\cos(2\theta) = 1 - 2\sin(\theta)^2$, applied to already known values such as $\sin(\pi/4) = \sqrt{2}/2$ [3], or on rational approximations, e.g. using Pythagorean triples, as we will see in Chapter 2. During the Renaissance, many handbooks of mathematical functions provided mathematical tables that were used to approximate a function by hand, possibly using interpolations. They generally provided values for equally spaced inputs, as a linear approximation would provide approximately the same accuracy whichever the input. For instance, Bernegger and Meyer's mathematical tables provided sine, tangent, and secant values for all arc minutes in $[0°, 90°]$ [11].

For digital computers, table-lookup algorithms have been used in many software and hardware implementations [44, 55, 148, 163, 164, 175]. Used in conjunction with *mathematical range reductions* (see Section 1.2.2), they make a trade-off between computation costs and memory. The larger the table, the smaller the approximation error. But for performance reasons and because L1 data (L1d) caches are small, tables typically do not hold more than a few thousand rows [42]. Although the mathematical range reduction can narrow the approximation interval so that a medium degree polynomial may be used, it is sometimes desirable to further reduce the interval of approximation so as to have very few arithmetic operations to perform. Table-based range reduction schemes allow for such additional reduction. They simply rely on a time/memory trade-off by storing precomputed values of the function(s) needed to reconstruct the final result.

In 1991, Tang suggested to use *regularly* spaced tables in hardware implementations of elementary functions that are already reduced to medium size intervals [164]. In the three examples given in his article, he uses equally spaced values for 2^x on $[-1, 1]$ and $\log(x)$ on $[1, 2]$, while a different yet regular tabulation is used for $\sin(x)$ on $[0, \pi/4]$. Indeed, for the trigonometric function, Tang's tabulated values are indexed by two variables j and k such that the input "breakpoints" are of the form $2^{-j}(1 + k/8)$, for $j \in [1\,..\,4]$ and $k \in [0\,..\,7]$. By choosing the breakpoint that is closest to the input, the interval of approximation is reduced to $[-1/32, 1/32]$. This small interval allows for polynomial approximations of small degree given the targeted precision. At the same time, the unequal breakpoint spacing may account for finer-grained values when close to zero, where the sine reaches its maximum rate of change.

On one hand, such a tabulation allows for optimizations in the computation of the reduced arguments and in addressing the table, since the breakpoints are easily computable. On the other hand, it is not really able to optimize the accuracy of all tabulated values. Indeed, since most elementary functions suffer from the TMD, images of rational values may fall very close to a midpoint of their storage format. Therefore the error on such tabulated values, provided they are correctly rounded to the nearest in their storage format, may be as much as 0.5 ulp [70]. Other table-based range reductions have been suggested in order to increase the accuracy of tabulated values, such as Gal's accurate tables.

In 1985, Gal suggested a method of tabulation for elementary functions that allows a virtually increased accuracy for tabulated values [68]. By using the random model described in Section 1.2.1, Gal shows that for any function f conforming to this model and for any $p > 0$, there is approximately one x in any set of 2^n values such that $f(x)$ has a string of n identical bits after its first p-th bits. Gal's probabilistic hypothesis was comforted by picking pseudo-random samples around regularly-spaced values so that a table of *almost* regularly-spaced values could be built with virtually extended accuracy. Although the precomputation cost is exponential in the number of virtual bits gained for each tabulated value, Gal and Bachelis later showed that it was reasonable to build tables that reached 9-extra bits of accuracy [69]. This means that each tabulated value can be stored with p-bit precision while having a virtual precision of $p + 9$ bits, which makes the maximum relative error of each value shrink down to 2^{-10} ulp.

In 2005, an improvement to the computation of Gal's accurate tables was suggested by Stehlé and Zimmermann [159]. By cleverly correlating Gal's table entries and the search for hard-to-round cases of an elementary function, they reduce the relative error upper bound to 2^{-20} ulp for each tabulated value. For the trigonometric functions, which require two simultaneously-bad cases for each entry, their algorithm achieves a heuristic complexity of about $2^{n/2}$ instead of the 2^n complexity of Gal's search based on the random model. This allowed

them to build accurate tables for the functions 2^x and $\sin(x)$ on the interval $[1, 2]$.

Other methods of tabulations have been developed, especially for hardware implementations. For example, we can cite the "(M, p, k)-friendly points" method for the trigonometric functions [17, 175], or bipartite tables [148], and the generalized multipartite tables [55] for any elementary function. More detail about the "(M, p, k)-friendly points" are given in Chapter 2. Table-based methods can provide a first rough approximation, but they quickly require exponential memory if one wants more accurate values for any input. Hence, combining small tables with efficient numerical approximations has been a design choice for many elementary functions for a long time.

Power series are among the first general numerical methods to approximate most elementary functions [123, ch.11]. The most common approximation functions are *polynomials*, but sometimes *rational approximations* (i.e. a quotient of two polynomials) are preferred. Perhaps the most famous rational approximations are Padé approximants, as they often give better approximations than Taylor series [6].

However, polynomial approximations are often preferred in libms as they do not involve any division, which is often an expensive operation. There are various kinds of polynomial approximations, among which:

TAYLOR SERIES They interpolate the function to approximate at a single point of multiplicity $n + 1$, and have the advantage of conserving parity properties. However, their numerical accuracy deteriorates the closer to the boundaries of the approximation interval [123].

CHEBYSHEV APPROXIMATIONS They interpolate the function to approximate at Chebyshev points, i.e. the $n + 1$ *translated roots* of the $(n + 1)$th Chebyshev polynomial T_{n+1}. They have nice properties of accuracy, comparable to *minimax* approximations [121, §3.1, 108].

MINIMAX POLYNOMIALS They are the best polynomial approximations to a function in terms of *maximum absolute error*, since they *minimize* it. Remez gave an iterative algorithm to efficiently compute the coefficients of such polynomials [16, 27, 142].

These approximations are commonly used for elementary functions implementations. Taylor series are generally chosen for their simplicity and parity-saving property, which then allows to halve the number of polynomial coefficients. Minimax polynomials are often chosen for their optimality in terms of accuracy, but Chebyshev are sometimes preferred because Remez' algorithm is slower than a simple Chebyshev interpolation for large degrees. As said above, there exist many other kinds of polynomial approximations. We may also cite Bernstein polynomials [158], pervasively used for Bézier curves in computer graphics, although they were not used in this thesis.

When it comes to *evaluating* a polynomial, Horner's rule is "one of the first things a novice programmer is usually taught" [90, p. 486]. Actually, it is a rather sequential way of evaluating a polynomial $p(x)$. Suppose that p is of degree n and that its coefficients are all nonzero. Then Horner's rule gives the following *polynomial evaluation scheme*:

$$p(x) = ((p_n x + p_{n-1})x + \cdots)x + p_0.$$

Because of its elegant simplicity, but also thanks to its numerical accuracy and many other properties [15] Horner's rule is pervasively used for elementary function evaluations.

On hardware architectures that benefit from Instruction-Level Parallelism (ILP) through the use of *pipelining*, Horner's scheme can suffer from performance issues because each operation requires the intermediate result that is produced immediately before, hence causing *pipeline stalls* [74]. Therefore, various polynomial evaluation schemes have been proposed for univariate [61, 62, 90, 113, 124, 133, 135], bivariate [120, 144], or multivariate polynomials [13, 150], to optimize their performance.

As we will see in more detail in Chapter 3, it can sometimes be preferable to *automatically* generate efficient polynomial evaluation schemes, although the exponential search-space makes it still a complex and open research problem [90, 144].

Finally, let us also cite the rather famous CORDIC algorithm [114, 132, 170]. COordinate Rotation DIgital Computer was designed and extended to compute trigonometric and hyperbolic functions using simple arithmetic operations (namely, addition, subtraction and bit shift) and table lookups. It was developed in 1959 for hardware that did not have any multiplier, and is still often used on pocket calculators and reconfigurable hardware (FPGAs). On highly-constrained hardware, with little memory dedicated to numerical code, CORDIC gives a good ratio between provided functions and code size, thanks to its versatility [123, 127, 164].

All the above methods alone have quite limited achievable accuracy for the large input ranges offered by IEEE 754 floating-point numbers. This is why they are often combined with each other or with *range reductions*, which we briefly present in the next section.

Range Reductions

Polynomial or rational approximations are usually designed to be accurate on small intervals. This way, the degree of the involved polynomials can generally stay low enough for the evaluation to be fast. Additionally, table lookups should not hold more than a few thousands values for performance reasons. Therefore, whenever a function is to be efficiently approximated by a table-lookup or a numerical approximation (or a combination of both), range reductions are often performed. Range reductions allow to shrink the interval on which these approximations take place, so that they can stay efficient. They generally involve mathematical properties of the function

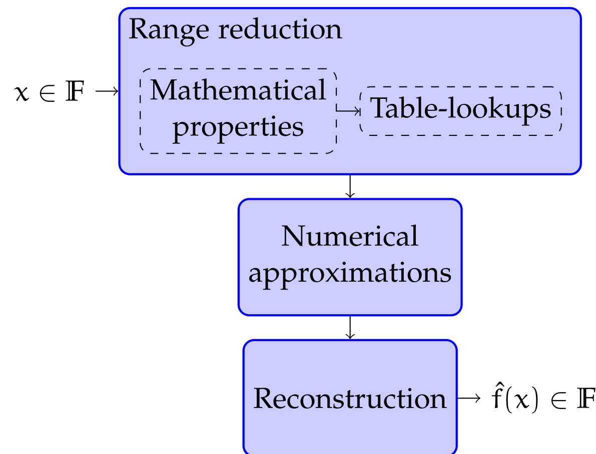


Figure 1.3 – Overview of a typical evaluation scheme for an elementary function $x \mapsto f(x)$. Note that it results in $\hat{f}(x)$, and not in $f(x)$, as it is an approximation in finite-precision arithmetic.

that is approximated, so that one or more (simpler) functions have to be approximated on small intervals. The final result is generally computed by a *reconstruction step*, which roughly "reverts" the range reduction, by combining all intermediate results from the range reduction and from the numerical approximations. Then, a typical evaluation scheme combining these four steps can be sketched as in Figure 1.3. The mathematical properties used for range reduction generally include:

PARITY If a function defined on $\mathbb{R}$ is even or odd, then it is sufficient to evaluate it on $\mathbb{R}_{\geqslant 0}$ and to change the sign of the result when needed.

PERIODICITY If a function f defined on $\mathbb{R}$ is such that

$$\exists C \in \mathbb{R}, \forall k \in \mathbb{Z}, \forall x \in \mathbb{R}, f(x + k \cdot C) = f(x),$$

then it is sufficient to evaluate it on $[-C/2, C/2]$ after performing an *additive reduction* by subtracting $k \cdot C$ from x. This reduction step must be carefully done especially if C cannot be stored exactly in memory [18, 28, 29, 137].

SYMMETRY If two functions f and g defined on $\mathbb{R}$ are such that

$$\exists C \in \mathbb{R}, \forall x \in \mathbb{R}, f(x + C) = g(x),$$

then $g(x - C)$ can be evaluated instead of $f(x)$ if it is advantageous. This can allow for further range reductions, such as for the trigonometric functions.

FUNCTION-SPECIFIC IDENTITIES These are often used for range reductions and may overlap above properties. For example:

$$\begin{aligned}
\log(ab) &= \log a + \log b;\\
\sin(a+b) &= \sin a \cos b + \sin b \cos a;\\
\cos\left(x + (2n+1)\frac{\pi}{2}\right) &= (-1)^n \sin x;\\
\cosh(a+b) &= \cosh a \cosh b + \sinh a \sinh b;\\
\sinh(x) &= \frac{e^x - e^{-x}}{2};\\
\sinh(n \log 2) &= 2^{n-1} - 2^{-n-1};\\
\cosh^2 x - \sinh^2 x &= 1.
\end{aligned}$$

The logarithm function, for example, can be mathematically transformed into $\log(x) = \log(m \cdot 2^e) = \log(m) + e\log(2)$, so as to only have to approximate the log function on $[1, 2)$ for normal numbers [111].

In this section, we have seen that in the context of high performance libms, the whole evaluation scheme of an elementary function is often split into several steps: range reductions, table-lookups, numerical approximations and the reconstruction of the final result. As compute-intensive applications using elementary functions (such as the CERN CMS experiment software) often present Data-Level Parallelism (DLP), they try to exploit as much parallelism offered by computers as possible. At the hardware level, exploiting DLP can be achieved with Vector, SIMD and GPU architectures [66, 74]. Hence, a current challenge is to develop both efficient and accurate *vectorized* libms, which is the topic of the next section.

1.2.3 *SIMD vectorization of elementary functions*

Exhibiting arithmetic parallelism is easy in many applications such as image processing, finite element analysis or 3D graphics. As Moore's law accurately predicted that the number of transistors per area unit would double each 18 months [118], hardware designers really began to take advantage of these extra transistors by supporting arithmetic parallelism in the 1990s. Such support was partly enabled by the democratization of *Single Instruction Multiple Data* (SIMD) architectures [130, 136].

SIMD architectures provide arithmetic and logic operators that are atomically applied to multiple data. Common SIMD architectures such as Intel's or AMD's are implemented as vector extensions to the Instruction Set Architecture (ISA), and are also known as SIMD Within A Register (SWAR) units. These units operate on vector registers whose length currently ranges from 64 to 512 bits, as can be seen in Table 1.3. For instance, Intel's Advanced Vector Instructions (AVX) and AVX2 operate on 256-bit registers that can be split into e.g.

Table 1.3 – Register Size and Availability of Intel's SIMD Extensions.

Extension name	Register size (bits)	ISA
MMX	64	Pentium P55C
SSE	128	Pentium III
SSE2		Pentium 4
AVX	256	Sandy Bridge
AVX2		Haswell
AVX512	512	Knights Landing

four double precision (binary64) or eight single precision (binary32) floating-point numbers.

These extensions can increase the throughput of arithmetic-intensive programs, but they usually do not implement vectorized elementary functions, i.e. elementary functions that would return a SIMD vector of outputs from a vector of inputs. However, manufacturers or independent developers have distributed mathematical libraries that provides such functions. They naturally rely on SIMD instructions provided by the hardware, or on the auto-vectorizing capabilities of modern compilers. Most of them are implemented in the C or C++ languages. The commercial Intel SVML library[1] provides intrinsic support for vectorized mathematical functions, but it is not freely available. Intel's Math Kernel Libraries provides vectorized mathematical functions optimized for large arrays of inputs. In this library, each function is implemented in three different flavors: high accuracy (HA), low accuracy (LA), and enhanced performance (EP). The maximum *measured* absolute error of the various routines ranges from 0.5 ulp for basic functions in their HA flavor to several thousand ulps – i.e. without any significant digit – for some EP routines.[2]

Other, non-commercial implementations have been proposed. In 2015, Intel contributed the GNU Libmvec to glibc 2.22 [151]. GNU Libmvec claims reasonable testing to pass 4-ulp maximum relative error, on each function domain and only in rounding-to-nearest mode. The latest version (from glibc 2.26) is only optimized for performance, although HA and LA flavors have been mentioned for future developments. It is written in x86 assembly though, which reduces its portability to other architectures and makes it more difficult to maintain and extend. For now, the functions may or may not handle special cases, and scalar callbacks to the original GNU libm functions may be used, which can lower the average performance. AMD also contributed the proprietary but free ACML_MV (no longer available) and the AMD LibM.[3] AMD ACML_MV was announced returning re-

[1] https://software.intel.com/en-us/node/524289

[2] https://software.intel.com/sites/products/documentation/doclib/mkl/vm/functions/_accuracyall.html

[3] http://developer.amd.com/amd-cpu-libraries/amd-math-library-libm

sults with a maximum error of at most 1 ulp, but with unpredictable behavior for subnormal inputs.

The CERN's VDT is another open source effort aiming at performance [138]. It is written in C++ and relies on modern compilers autovectorizing abilities to generate vectorized routines for large arrays. In terms of accuracy, the logarithm function is announced with at most two bits differing from GNU libm results in single and double precision, eight bits being the overall maximum difference for the arccos function in double precision.

Influenced by the Cephes library,[1] like the VDT library, Pommier's SSE[2] and Garberoglio's AVX[3] versions of "Mathfun" headers offer SIMD routines for the sin, cos, exp, and log functions in single precision. Like EP implementations, Mathfun does not claim high accuracy. However the authors write that the trigonometric functions are measured to be faithful on the range $[-8192, 8192]$. Their performance benchmarks show an average reciprocal throughput close to 30 cycles per element (CPE) for the SSE version, while the AVX version is said to be close to a 1.7x speedup compared to SSE routines.

Two recent projects are Yeppp! [60], and SLEEF [153]. Yeppp! targets EP implementations by relying on the PeachPy assembly kernel generator [58, 59], while the SIMD Library for Evaluating Elementary Functions (SLEEF) provides HA and LA routines [153].

In 2016, Lauter proposed an open-source auto-vectorizable mathematical library written in high level scalar C [100]. Its main feature is probably to be system and hardware independent, since it does not rely on any other libraries and only makes use of standard C constructs. This makes this library available on any architecture for which an auto-vectorizing backend is implemented in the compiling toolchain. The library was developed with both binary32 and binary64 formats in mind, but until now efforts have been concentrated on the binary64 routines only. It currently provides nine elementary functions such as trigonometric functions, exponential and logarithm, and claims ongoing work for functions harder to vectorize such as the power function $(x, y) \mapsto x^y$ or the bivariate arctangent $(x, y) \mapsto \arctan(y/x)$. Each implementation targets enhanced performance at the expense of accuracy: a maximum error of 8 ulps was measured for the whole set of functions. The implementation, which is self-sufficient as it requires no other library, relies on compiler autovectorizers like the CERN's VDT. For this reason, conditional branches and table-lookups are entirely avoided.

Note that none of these mathematical libraries provide a guaranteed accuracy, since, to our knowledge, documented accuracies are obtained by non-exhaustive testing. Furthermore, writing high performance libms is a tedious, error-prone task that often leads to monolithic, general-purpose libms. As for any monolithic library, one can

[1] http://www.netlib.org/cephes/
[2] http://gruntthepeon.free.fr/ssemath/
[3] http://software-lisc.fbk.eu/avx_mathfun/

think that general-purpose libms are to tailored software what COTS[1] circuits are to full-custom ASICs[2] in hardware design: they usually meet the requirements, but something else could fulfill specific needs much better. As the knowledge about floating-point arithmetics and elementary functions grows, the current challenge is now to automatize the generation of code for libms, or provide tools that help developers in this time-consuming task. This is the topic of the next section.

1.3 CODE GENERATION FOR ELEMENTARY FUNCTIONS

The design and implementation of a whole libm usually involve high development costs [92]. Furthermore, real-world applications using mathematical functions are so diverse that they often need specific levels of accuracy over specific input intervals with respect to their targeted performance (e.g. throughput or latency) [138]. For instance, different research fields such as earthquake engineering or acoustics may use the same trigonometric functions but with very different input intervals. The first may use angles in the interval $[-\pi, \pi]$ as in [140, p. 290], while the second might use much wider intervals such as $\left[0, 10^4\right]$ as in [32]. Such diverse needs push for *tailored* implementations of each function [45]. As this possibly means an exponential number of different implementations, recent efforts have been put on automatic code generation [20, 22, 40, 48, 92, 101, 103, 143, 144] to let the machines handle the exponential development costs.

The results of these efforts have been to reduce development costs down to a few man-months for a whole custom libm. Many software tools that can generate or help at generating code for elementary function evaluation have been developed over the recent years. Now, these efforts have been aiming at bringing development costs further down to a few man-hours. We present here such tools that were used for this thesis.

1.3.1 *Tools geared towards code generation*

Sollya: an environment for the development of numerical codes

Sollya[3] is an extensible scripting language and a C library that can perform symbolic computations on many elementary functions, as well as compute different polynomial approximations. Among its features, Sollya provides the following algorithms:

- `guessdegree` tries to guess the minimal polynomial degree that is required to approximate a function on some interval within an error bound;

[1] Commercial off-the-shelf.

[2] "Full-custom application-specific integrated circuits" is a self-defined expression. They potentially maximize the hardware performance.

[3] http://sollya.gforge.inria.fr/

- `taylor`, `remez` and `fpminimax` try to generate Taylor, Remez, or FP-minimax polynomial approximations, for elementary or special functions [16, 26, 27];
- `implementpoly` can generate C code that implements a polynomial object in optimized double, double-binary64 or triple-binary64 precision following Horner's scheme [50].

A code generator for polynomial evaluation: CGPE

Although Sollya provides `implementpoly` to generate optimized-precision C codes for polynomial evaluation, only the Horner's scheme is considered. As we will see in Chapter 3, one might want to achieve higher computing throughput or lower latency by using different evaluation schemes. This is the purpose of CGPE.

CGPE[1] is an interactive software and a C++ library that can generate efficient code for polynomial evaluations on multiple targets [144]. It is able to generate polynomial schemes for bivariate polynomials and return the ones that exhibit the most Instruction Level Parallelism (ILP), i.e. schemes in which many arithmetic operations are independent, hence that could be run in parallel. Given bivariate polynomial coefficients, input intervals and an error bound, CGPE can also try to generate an efficient polynomial evaluation scheme that satisfies the error bound for the polynomial on its input intervals. In this case, CGPE can formally *verify* such error bound using Gappa, which is briefly presented in the next section.

Formalization and code certification

Finite precision implementations usually document their approximation errors or provide guarantees on the relative error on the outputs compared to the mathematical results. Such guarantees can be obtained by a tight analysis of evaluation and approximation errors. This analysis is tedious and error-prone, but can be automated with tools such as Gappa [37, 51, 52].

Gappa[2] combines interval arithmetic and logical mathematical rewritings to automate and verify intelligent forward error analyses. It can manipulate many kinds of real numbers, including binary32, binary64 or binary128 floating-point numbers [115]. For example, the following Gappa script is able to determine a tight interval in which lies the binary32 square of a binary32 approximation of a real number in $[-0.1, 0.1]$.

```
@rnd = float<ieee_32, ne>;
x = rnd(Mx);
{
 Mx in [-0.1,0.1] -> rnd(x*x) in ?
}
```

[1] http://cgpe.gforge.inria.fr/

[2] *Génération Automatique de Preuves de Propriétés Arithmétiques* [automatic proof generation of arithmetic properties]. http://gappa.gforge.inria.fr/

Gappa then returns

```
Results:
  float<24,-149,ne>(x * x) in [0, 10737419b-30 {0.01,
     2^(-6.64385)}]
```

which means that `x*x` is in the interval `[0, 10737419b-30]` in dyadic notation, that is, in `[0, 0x1.47ae16p-7]` in hexadecimal notation.

1.3.2 *The MetaLibm ANR project*

MetaLibm is the name of multiple automatic code generation projects[1] that eventually merged to form the MetaLibm ANR project,[2] in which this thesis takes place. All of these projects have been aiming at generating source code for the evaluation of elementary functions or for digital filters, but they differ in their respective approaches.

Metalibm-lutetia, a black-box generator

One of them is the Metalibm-lutetia project, which adopts a top-down approach. It was designed for wide adoption by developers with or without numerical expertise. Indeed, it is a framework that tries to generate code for general mathematical expressions, with I/O precision, range and target accuracy specifications.

First, it tries to automatically detect periodicities or symmetries that can allow for range reductions. Then, for the remaining interval on which the function must be approximated, it applies an algorithm of *domain splitting*: the interval is split into as many subintervals as needed so that the function can be approximated by different numerical approximations on each of these intervals. For polynomial approximations, a halting condition for splitting can be a maximum value for the polynomial degree. Kupriianova and Lauter developed a split-then-merge algorithm that uses such a condition to perform domain splitting and then merge as many subintervals as possible, in order to optimize the generated code [93].

Metalibm-lugdunum, a libm-oriented generator

The second project is Metalibm-lugdunum.[3] This framework enables to describe the implementation of a function using a meta-language in a Python syntax. More particularly the descriptions consist in abstracting evaluation schemes as a set of high-level building blocks. Then, the MetaLibm backend can optimize and translate the whole description for different target architectures (C11, x86-intrinsics C11, OpenCL, VHDL, ...) and numerical formats (floating-point, fixed-point, ...) [20, 22].

[1] http://www.metalibm.org/
[2] http://www.metalibm.org/ANRMetaLibm
[3] https://github.com/kalray/metalibm

As an example, the following listing describes the multiplication $x \cdot x$ in this meta-language, where x is the input variable of our program.

```
y = Multiplication(x, x, tag="y", precision =
    self.precision)
```

Then, if the generic C11 architecture is targeted, Metalibm will simply generate the statement `y = x * x`, while if x86 architectures with AVX2 are targeted, it will produce the following piece of C code, where `vec_x` is the vector equivalent of x.

```
carg = GET_VEC_FIELD_ADDR(vec_x);
tmp = _mm256_load_ps(carg);
y = _mm256_mul_ps(tmp, tmp);
```

Metalibm-lugdunum is more oriented towards libm developers as it follows a bottom-up approach. However, it can optimize generated implementations from abstract evaluation schemes for various goals, such as (among others) target accuracy, input interval, vector size, or compliance to the IEEE 754 standard. Therefore, it is also designed to be used as a toolbox, e.g. by theoretical physicists needing a tailor-made, efficient OpenCL implementation of the logarithm or the exponential functions.

As abstract evaluation schemes can be target-agnostic, Metalibm pushes for factoring shared algorithms across architectures, different I/O precisions, or target accuracies. Such design aims at shrinking the development time for mathematical libraries. Facilities such as automatic testing for performance and accuracy, and formal proof generation are included, which represents an additional time-saving opportunity for meta-libm developers and final users.

Metalibm-tricassium, a filter-oriented generator

Digital signal processing (DSP) numerical algorithms have much in common with polynomial and rational functions. Therefore, the Meta-Libm project also aims at generating efficient and accurate source codes for the implementation of digital filters on various architectures [109, 171].

1.3.3 *Generative programming for numerical algorithms*

As said by Rompf et al., there has been an ever-growing appeal to "go meta" [146]. For numerical programs, the number of domain specific languages (DSL) and source-to-source compilers is skyrocketing. They now target various optimizations goals for improved, automatically generated or transformed, numerical programs [20, 33, 35, 36, 48, 49, 54, 58–60, 80, 81, 83, 84, 112, 116, 117, 126, 131, 134, 145, 161, 162, 165, 166]. This list is far from being exhaustive and could probably be updated every month. We will just present some of the goals that such "meta-optimizers" target. Without going into much

detail, some of them aim at improving and certifying the accuracy of floating-point or fixed-point programs such as Sardana/Salsa [33, 80, 81] or Rosa/Daisy [35, 36, 84]. Other explore better tradeoffs between efficiency and accuracy for hardware [83, 134, 166], such as FloPoCo [48, 49, 54], software (PeachPy [58–60]), or both [20].

2

EXACT LOOKUP TABLES FOR ELEMENTARY FUNCTIONS

> "It is a bit uncertain who first realized that [Primitive Pythagorean Triples] had a beautiful family tree, but a 1934 paper in Swedish could be the earliest (Berggren 1934 [10])."
>
> Bernhart and Price [12].

As we have seen in Chapter 1, elementary mathematical functions are pervasively used in many applications such as electronic calculators, computer simulations, or critical embedded systems. However, their evaluation is usually an approximation. To reach high accuracies, they usually make use of mathematical properties, precomputed tabulated values and accurate polynomial approximations [43, Ch.2]. Each step of the evaluation process generally combines error of approximation (e.g. replacing an elementary function by a polynomial) and error of evaluation on finite-precision floating-point arithmetic (e.g. the evaluation of such a polynomial). Whenever they are used, tabulated values usually embed rounding error inherent to the transcendence of most elementary functions [111]. Several interesting techniques have been developed to reduce this rounding error as much as possible, yet, to our knowledge, none has ever managed to remove it entirely.

In this chapter, we present a general method that enables to build *error-free* tabulated values that is worthwhile whenever at least two terms are tabulated in each table row. Our technique transfers all potential rounding errors on tabulated values to one small corrective term, which allows to asymptotically save twice as many bits as state-of-the-art techniques. For the trigonometric and hyperbolic functions sin, cos, sinh, and cosh, we show that Pythagorean triples can lead to such tables in reasonable time and memory usage. When targeting correct rounding in double precision for the same functions, we also demonstrate that this method might save memory accesses and floating-point operations by up to 29% and 42%, respectively, during the reconstruction step.

2.1 INTRODUCTION

The algorithms developed for the evaluation of elementary functions, such as the logarithm, the exponential, or the trigonometric and hyperbolic functions, can be classified into at least two categories. The first category concerns algorithms based on small and custom operators combined with tabulated values that target small accuracies, typically less than 30 bits [44, 55, 56]. The second category concerns algorithms that usually target single or double precision (24 or 53 bits of precision) with an implementation on general processors that relies on the available hardware units [34, 181].

Implementations of those functions that target correct rounding are usually divided into two or more phases [121]. A *quick phase* is first performed: it is based on a fast approximation that provides a few more bits than the targeted format, which makes correct rounding possible most of the time at a reasonable cost. When correct rounding is not possible, a slower *accurate phase* is executed. The quick phase uses operations with a precision slightly greater than the targeted precision, while the accurate phase is based on a larger precision. For instance, in the correctly-rounded library CR-Libm, the quick phase for the sine and cosine functions in double precision targets 66 bits while the accurate phase corresponds to 200 bits [34, § 9]. Then, in order to guarantee that an implementation actually computes correctly-rounded results, a proof of correctness must be built. This proof is based on the mandatory number of bits required to ensure correct rounding, which is linked with the search for the worst cases for the TMD [104].

The design of such correctly-rounded implementations is a real challenge, as it requires to control and limit every source of numerical error [27]. Indeed these implementations involve various steps, such as range reduction, polynomial evaluation, or reconstruction, during all of which errors may occur. Since those errors accumulate and propagate up to the final result, any solution or algorithm that reduces them can have an impact on the simplicity of the proof of correctness and the performance of the implementation.

2.1.1 *Overview of this chapter*

This chapter presents works on the trigonometric functions sin and cos that were published and presented in [97], their extension to the hyperbolic functions sinh and cosh that were published in [96], as well as a faster heuristic to build trigonometric exact lookup tables that has, to our knowledge, never been published before. Our main contributions are twofold:

1. a general method that eliminates rounding errors from tabulated values and transfers them into a single corrective term to be added to the reduced argument, which, in most cases, already contains rounding errors;

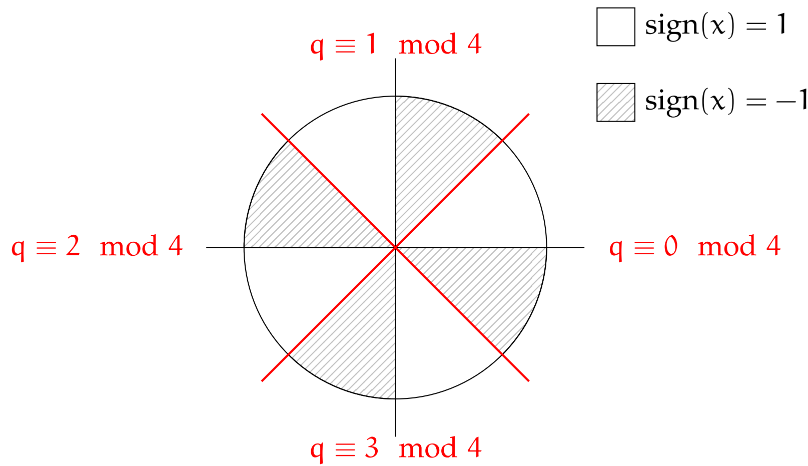


Figure 2.1 – Graphical representation of the first range reduction for the trigonometric functions.

2. and a formally justified algorithm that shows how to apply this method to trigonometric and hyperbolic functions.

In Section 2.2, we recall classic evaluation schemes for the trigonometric and hyperbolic functions. Section 2.3 details the properties of the proposed tables and demonstrates how it is possible to remove two sources of error involved in this step. Section 2.4 presents our theoretical approach to build exact lookup tables for trigonometric and hyperbolic functions using Pythagorean triples. Then, Section 2.5 presents some experimental results that show that we can precompute exact lookup tables up to 12 indexing bits reasonably fast with the help of suitable heuristics. A quantitative comparison with classic approaches is made in Section 2.6. They show that our method can lower the table sizes and the number of floating-point operations performed during the reconstruction step. Finally, an example with a toy table is also presented in Section 2.7. We conclude and discuss perspectives in Section 2.8.

2.2 TABLE-BASED EVALUATION SCHEMES

2.2.1 *Usual framework*

In order to better understand the challenges that we face when designing evaluation schemes for elementary functions, let us detail each step of a classic function evaluation process with a focus on methods based on table lookups through the example of the trigonometric and hyperbolic sine and cosine. For this purpose, let y be a machine-representable floating-point number, taken as the input to the considered functions. The design of these evaluation schemes is usually very similar to the four-step process that we have seen in Chapter 1 (p.22):

1. A *first range reduction*, based on mathematical properties, narrows the domain of the function to a smaller one. For the

trigonometric functions, properties of periodicity and symmetry lead to evaluating $f_q \in \{\pm\sin, \pm\cos\}$ at input

$$|x| = |y - q \cdot \pi/2| \in [0, \pi/4],$$

where $q \in \mathbb{Z}$ and the choice of f_q depends on the function to evaluate, $q \bmod 4$, and $\mathrm{sign}(x)$ [121]. Figure 2.1 sums up how $q \bmod 4$ and $\mathrm{sign}(x)$ may change the function to evaluate.

For the hyperbolic functions, let

$$|x| = |y - q \cdot \ln(2)| \in [0, \ln(2)/2]\,,$$

with $q \in \mathbb{Z}$. Addition formulas can then be used together with the analytic expressions for the hyperbolic functions involving the exponential [111], which gives, for the sine:

$$\begin{aligned}\sinh(y) = &\left(2^{q-1} - 2^{-q-1}\right) \cdot \cosh|x| \\ &+ (-1)^{\mathrm{sign}\,x} \cdot \left(2^{q-1} + 2^{-q-1}\right) \cdot \sinh|x|\,.\end{aligned}$$

2. A *second range reduction*, based on tabulated values, further reduces the range on which polynomial evaluations are to be performed. The argument x is split into two parts, x_h and x_ℓ, such that:

$$x = x_h + x_\ell \quad \text{with} \quad |x_\ell| \leqslant 2^{-p-1}. \tag{2.1}$$

The term x_h is the p-bit value of the form

$$x_h = i \cdot 2^{-p} \quad \text{with} \quad i = \lfloor x \cdot 2^p \rceil\,, \tag{2.2}$$

where $\lfloor a \rceil$ denotes the rounding of a floating-point number a to the nearest integral value.[1] The integer i is used to address a table of $n = \lfloor \kappa \cdot 2^{p-1} \rceil + 1$ rows, with $\kappa \in \{\pi/2, \ln(2)\}$. This table holds precomputed values of either trigonometric or hyperbolic sines and cosines of x_h, which we indifferently name S_h and C_h.

3. Meanwhile, *polynomial approximations* of the trigonometric or hyperbolic sine and cosine on the interval $[-2^{-p-1}, 2^{-p-1}]$, denoted by P_S and P_C, are evaluated at input x_ℓ.

4. Finally, a *reconstruction step* allows to compute the final result using the precomputed values retrieved at step 2 and the computed values from step 3. For the trigonometric sine, if we assume that $q \equiv 0 \bmod 4$ and $\mathrm{sign}(x) = 1$ so that $f_q = \sin$, one has to perform the following reconstruction:

$$\sin(y) = S_h \cdot P_C(x_\ell) + C_h \cdot P_S(x_\ell),$$

[1] Tie cases are generally rounded to the nearest even integral value, e.g. $\lfloor 2.5 \rceil = 2$.

while the reconstruction for the hyperbolic sine is:

$$\begin{aligned}\sinh(y) = &\left(2^{q-1} - 2^{-q-1}\right) \cdot \left(C_h \cdot P_C\left(x_\ell\right) + S_h \cdot P_S\left(x_\ell\right)\right) \\ &+ (-1)^{\operatorname{sign} x} \\ &\cdot \left(2^{q-1} + 2^{-q-1}\right) \cdot \left(S_h \cdot P_C\left(x_\ell\right) + C_h \cdot P_S\left(x_\ell\right)\right).\end{aligned}$$

Satisfactory solutions already exist to address the first range reduction [18, 38, 137], the generation and evaluation of accurate and efficient polynomial evaluation schemes [16, 50, 120], and the reconstruction step. The interested reader can find more details in [121].

All these steps generally add errors of approximation and/or evaluation. Here, we address the second range reduction based on tabulated values for the trigonometric and hyperbolic sine and cosine. At this step, each tabulated value embeds a rounding error. Our objective is to remove the error on these values, and consequently transfer the error due to range reduction into the reduced argument used in the polynomial evaluations. Precisely the proposed method relies on finding precomputed values with remarkable properties that simplify and accelerate the evaluation of these functions. These properties are threefold:

1. Each pair of values holds the exact images of a reference argument under the functions, that is, without any rounding error. For this purpose, we require these values to be rational numbers;

2. The numerator and denominator of each rational value are exactly representable in a representation format available in hardware, e.g., binary64;

3. And finally, these rational values all share the same denominator. This enables to incorporate the division by this denominator directly into the polynomial coefficients.

These three properties lead to tabulated values that are exact *and* representable on single machine words. For the trigonometric and hyperbolic functions, Pythagorean triples give rational numbers that satisfy such properties.

2.2.2 *Focus on table-lookup range reductions*

The second range reduction uses fast but inaccurate approximations. In practice it is often implemented using table lookups.[1] This section presents three solutions that address this step with the sine function to illustrate them. However, these methods are general and also apply to the trigonometric cosine as well as to the hyperbolic functions.

[1] For instance, see the GNU `libm` in `glibc 2.25`.

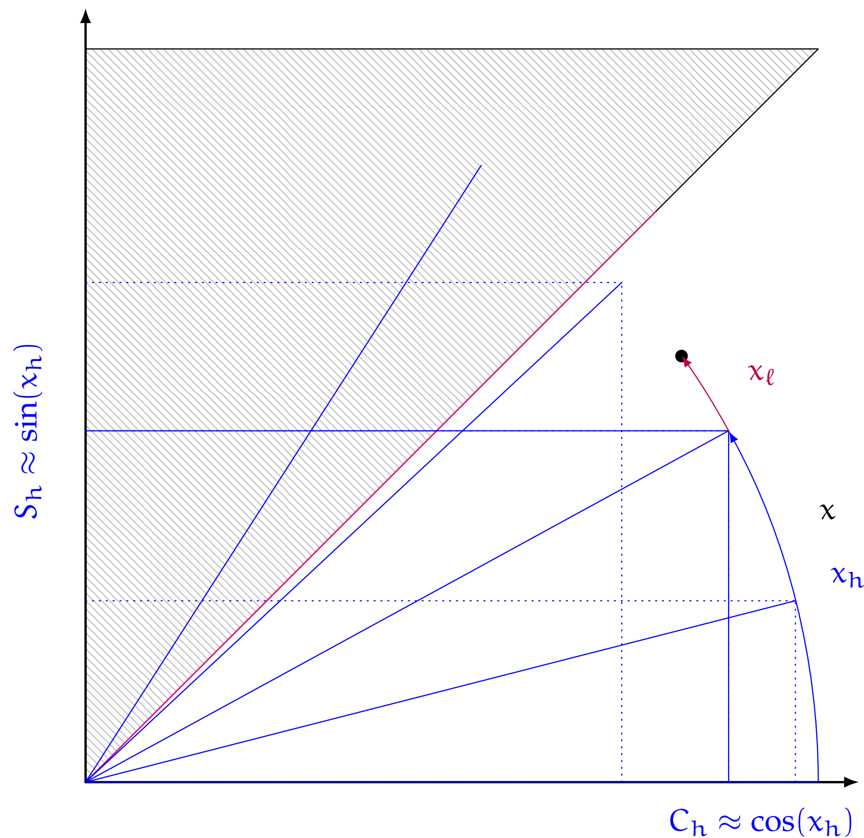


Figure 2.2 – Visual representation of the second range reduction with a regularly-spaced trigonometric table.

Tang's tables

Tang proposed a general method to implement elementary functions that relies on hardware-tabulated values [164]. Given the reduced argument x as in Equation (2.1), Tang's method uses the upper part x_h to retrieve two tabulated values S_h and C_h that are good approximations of $\sin(x_h)$ and $\cos(x_h)$, respectively, rounded to the destination format. Figure 2.2 gives a visual representation of the second range reduction with an example of Tang's tables for the trigonometric functions. If an implementation targets correct rounding, then those approximations are generally stored as *floating-point expansions*. A floating-point expansion of size n usually consists in representing a given number as the unevaluated sum of n non-overlapping floating-point numbers so that the rounding error be reduced compared to a single floating-point number [152]. If we denote by $\mathrm{RN}_i(x)$ the rounded value of a number x to the nearest floating-point number of precision i, let us note ε_{-i} its relative rounding error such that $|\varepsilon_{-i}| \leqslant 2^{-i}$, then, for the trigonometric functions, we have:

$$\begin{aligned} S_h &= \mathrm{RN}_{53j}(\sin(x_h)) = \sin(x_h) \cdot (1 + \varepsilon_{-53j}) \\ \text{and} \quad C_h &= \mathrm{RN}_{53j}(\cos(x_h)) = \cos(x_h) \cdot (1 + \varepsilon_{-53j}), \end{aligned}$$

where j is the size of the binary64 expansions used to represent S_h and C_h.

In parallel to the extraction of the values S_h and C_h, the evaluation of two polynomial approximations $P_S(x)$ and $P_C(x)$ can be performed. They respectively approximate the sine and cosine functions over the interval covered by x_ℓ, namely $[-2^{-p-1}, 2^{-p-1}]$. Finally, the result of $\sin(x)$ is reconstructed as follows:

$$\sin(x) \approx S_h \cdot P_C(x_\ell) + C_h \cdot P_S(x_\ell).$$

Tang's method is well suited for hardware implementations on pipelined and out-of-order architectures. It takes advantage of the capability on these architectures to access tabulated values in memory and to perform floating-point computations concurrently. Once the argument x is split into the two parts x_h and x_ℓ, memory units can provide the two tabulated values S_h and C_h, while floating-point units evaluate the polynomials P_S and P_C. As the degree of the polynomials decreases when the table size increases, the objective is to find parameters so that the polynomial evaluations take as long as memory accesses, on average [42].

Tang's tables store images of regularly spaced inputs, rounded to the destination format. This rounding error is problematic when seeking a correctly rounded implementation in double precision, since worst cases for trigonometric and hyperbolic functions require more than 118 bits of accuracy [105, 106]. A solution consists in storing values on expansions of size 3 and using costly extended-precision operators [34].

Gal's accurate tables

In Tang's method, S_h and C_h are approximations of $\sin(x_h)$ and $\cos(x_h)$, respectively. They are rounded according to the format used in the table and the targeted accuracy for the final result. To increase the accuracy of these tabulated values, Gal suggested a method to transfer some of the errors due to rounding over the reduced argument [68]. This consists in introducing small, exact corrective terms on the values x_h, hereafter denoted by $corr$. Figure 2.3 illustrates how such a table can be used for the trigonometric functions. For each input entry x_h of the table, one $corr$ term is carefully chosen to ensure that both $\sin(x_h + corr)$ and $\cos(x_h + corr)$ are "very close" to floating-point machine numbers. In [69], Gal and Bachelis were able to find $corr$ values such that

$$\begin{aligned} S_h &= \mathrm{RN}_{53j}(\sin(x_h + corr)) \\ &= \sin\left(x_h + corr\right) \cdot \left(1 + \varepsilon_{-10-53j}\right) \\ \text{and} \quad C_h &= \mathrm{RN}_{53j}(\cos(x_h + corr)) \\ &= \cos\left(x_h + corr\right) \cdot \left(1 + \varepsilon_{-10-53j}\right) \end{aligned}$$

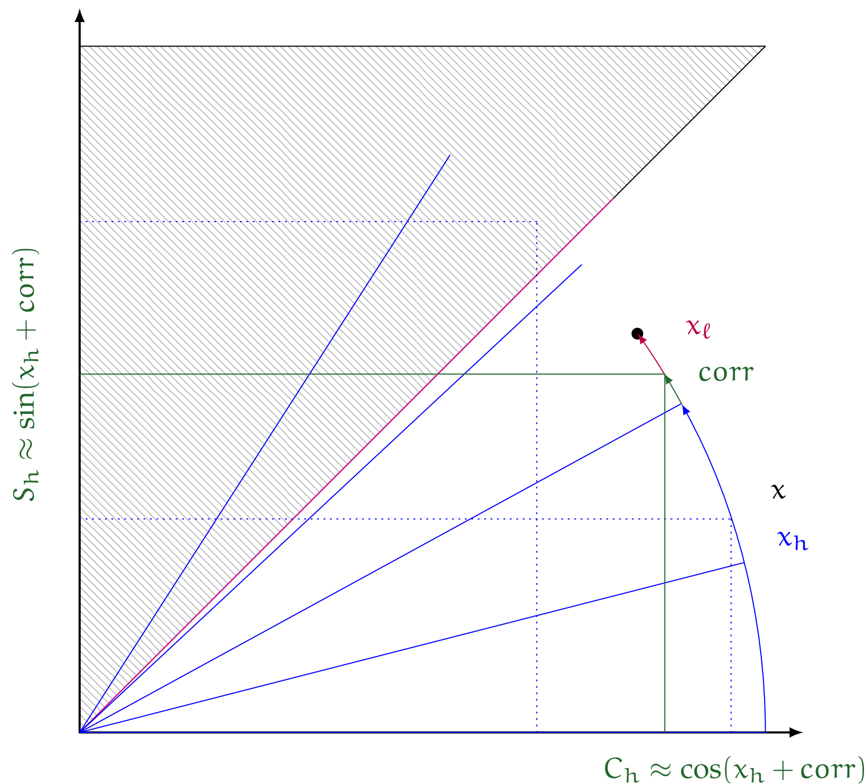


Figure 2.3 – Visual representation of the second range reduction with an example of Gal's trigonometric table.

for each row of an 8-bit-indexed table. This corresponds to 10 extra bits of accuracy for both tabulated values of sine and cosine compared to Tang's tabulated values, thanks to a small perturbation corr on the values x_h. For the trigonometric and hyperbolic functions, such corr values are produced by a pseudo-random sampling with an expected sample size of 2^{18} [159]. Indeed, using a statistically pseudo-random model, Gal expects these corr values to have about 18 bits of significand each. This method entails storing the corrective terms corr along with the values S_h and C_h, which will usually make Gal's accurate tables larger than Tang's tables when targeting correct rounding. The value $\sin(x)$ is reconstructed as follows:

$$\sin(x) = S_h \cdot P_C(x_\ell - \mathrm{corr}) + C_h \cdot P_S(x_\ell - \mathrm{corr}),$$

where $P_C(x)$ and $P_S(x)$ are polynomials approximating $\cos(x)$ and $\sin(x)$, respectively, on the interval covered by $x_\ell - \mathrm{corr}$.

Gal's solution requires a rather expensive search in order to find one corrective term for each entry x_h, within a search space that grows exponentially with the number of extra bits for S_h and C_h. Stehlé and Zimmermann proposed an improvement based on lattice reduction and worst cases to accelerate the precomputation [159]. They were able to increase the accuracy of Gal's tables by 11 extra bits, which translates to 21 extra bits compared to Tang's tables.

Gal's technique virtually increases the accuracy of tabulated values by a few extra bits, slightly reducing the error bounds in the reconstruction step. However, there are still errors in the reduced argument as well as in tabulated approximations.

Brisebarre et al.'s (M, p, k)*-friendly points*

In 2012, Brisebarre, Ercegovac, and Muller proposed a new method for trigonometric sine and cosine evaluation with a few table lookups and additions in hardware [17]. It was improved and implemented by Wang and the same authors in 2014 [121, 175]. Their approach consists in tabulating four values a, b, z, and $\hat{x}$, defined as:

$$z = 1/\sqrt{a^2 + b^2} \quad \text{and} \quad \hat{x} \approx \arctan(b/a),$$

where a and b are *small* particular integers. The reconstruction then corresponds to:

$$\sin(x) = \big(b \cdot \cos(x - \hat{x}) + a \cdot \sin(x - \hat{x})\big) \cdot z.$$

The values $(a, b, z, \hat{x})$ are chosen amongst specific points with integer coordinates (a, b) called (M, p, k)-friendly points. These points are recoded using the *canonical recoding*, which minimizes the number of nonzero digits in table entries [141]. More precisely, a and b must be positive integers less than M, so that the number $z = 1/\sqrt{a^2 + b^2}$ has less than k nonzero bits in the first p bits of its *canonical recoding*. Therefore, with this method, a and b can be stored exactly, but not z nor $1/z$ in general.

However these requirements make hardware multiplications by a, b, and z much more acceptable, since they are equivalent to just a few additions. Compared to other hardware evaluation schemes, the authors claim that their solution can reduce the area on FPGAs for 24-bit accuracy by 50%. Overall, this is a suitable method for competitive low and medium-precision hardware approximations. But although it could be an interesting question to investigate, this technique does not seem specifically suited for software implementations nor for high precisions.

Furthermore, as far as our knowledge goes, there have not been any similar application of this approach to the evaluation of the hyperbolic sine and cosine. For these functions, we may imagine a similar method using small integers $a > b$, so that $z = 1/\sqrt{a^2 - b^2}$ makes (a, b) an (M, p, k)-friendly point. By defining $\hat{x} \approx \operatorname{atanh}(b/a)$, this method might also be suitable for the evaluation of hyperbolic functions. However, as the original authors have not made any statement about such an application, further investigations remain to be done to prove its feasibility.

2.3 DESIGN OF AN EXACT LOOKUP TABLE

After the first range reduction, the reduced number x is assumed to be an irrational number. Therefore, it has to be rounded to some finite precision, which means that after the second range reduction, only x_ℓ contains a rounding error. Gal's method adds an exact corrective term corr to x_ℓ that allows to increase the accuracy of transcendental tabulated values. Instead, we suggest not to worry about *inexact* corrective terms, as long as they make tabulated values *exactly representable*. This way, the error is solely concentrated in the reduced argument $x_\ell - \mathrm{corr}$ used during the polynomial evaluations.

Now, let us characterize these exactly representable values and then explain how the evaluation scheme can benefit of this property. For this purpose, let f and g be the functions to approximate, $\kappa \in \{\pi/2, \ln(2)\}$ the additive constant for the first range reduction, and y an input floating-point number. As seen in the introduction, the reduced argument x obtained after the first range reduction is such that

$$x = |y - q \cdot \kappa| \quad \text{with} \quad q \in \mathbb{Z} \quad \text{and} \quad 0 \leqslant x \leqslant \kappa/2.$$

As y is rational, q is an integer, and κ is irrational, then unless $q = 0$, x must be irrational, and it has to be rounded to a floating-point number $\hat{x}$, with a precision j greater than the targeted format such that: $\hat{x} = x \cdot (1 + \varepsilon_{-j})$. We should mention that $\hat{x}$ is generally represented as a floating-point expansion of size 2 or more to reach an accuracy of at least j bits. As seen in Equation (2.1), the second range reduction splits $\hat{x}$ into two parts, x_h and x_ℓ, such that

$$\hat{x} = x_h + x_\ell \quad \text{and} \quad |x_\ell| \leqslant 2^{-p-1}.$$

As shown in Equation (2.2), the value x_h is then used to compute an address i in the table T made of n rows. Let us assume that this table T holds exact precomputed values of $k \cdot f(x_h + \mathrm{corr}_i)$ and $k \cdot g(x_h + \mathrm{corr}_i)$, where k is an integer scale factor that makes both tabulated values integers, and corr_i is an irrational corrective term, precomputed for each table entry i, such that $|\mathrm{corr}_i| \leqslant 2^{-p-1}$. To build such values, we assume the following properties:

1. The functions f and g are right invertible on the domain $[0, (n - 1/2) \cdot 2^{-p}]$. This allows corrective terms to be determined.

2. Corrective terms are such that $|\mathrm{corr}_i| \leqslant 2^{-p-1}$.

3. For each table entry i, the values $f(x_h + \mathrm{corr}_i)$ and $g(x_h + \mathrm{corr}_i)$ are *rational* numbers with the same denominator, that is:

$$f(x_h + \mathrm{corr}_i) = \frac{\eta_i}{k_i} \quad \text{and} \quad g(x_h + \mathrm{corr}_i) = \frac{\gamma_i}{k_i}$$

with $\eta_i, \gamma_i \in \mathbb{Z}$ and $k_i \in \mathbb{Z}^*$.

4. Let $k = \mathrm{lcm}(k_0, \ldots, k_{n-1})$, i.e. the least common multiple (LCM) of the denominators k_i. Let

$$F_i = k \cdot f(x_h + \mathrm{corr}_i) \quad \text{and} \quad G_i = k \cdot g(x_h + \mathrm{corr}_i).$$

It is clear that the values $F_i = k \cdot \eta_i / k_i$ and $G_i = k \cdot \gamma_i / k_i$ are integers, but we will also assume that they are representable as one floating-point machine number each, e.g. they fit on 53 bits if using the binary64 format.

With numbers satisfying those properties, the reconstruction step corresponds to

$$f(x) = G_i \cdot P_{f/k}(x_\ell - \mathrm{corr}_i) + F_i \cdot P_{g/k}(x_\ell - \mathrm{corr}_i)$$

for some i in $[0 \,..\, n-1]$, where

- $P_{f/k}(x)$ and $P_{g/k}(x)$ are polynomial approximations of the functions $f(x)/k$ and $g(x)/k$, respectively, on the interval

$$\left[-2^{-p-1} - \max_i(\mathrm{corr}_i), 2^{-p-1} - \min_i(\mathrm{corr}_i)\right],$$

- and F_i, G_i, and $\mathrm{RN}_j(\mathrm{corr}_i)$ are gathered from T. The integers F_i and G_i can be stored without error as one floating-point number each. The third tabulated value $\mathrm{RN}_j(\mathrm{corr}_i)$ is a rounding of the corrective term corr_i such that:

$$x_h + \mathrm{corr}_i \in f^{-1}\left(\frac{F_i}{k}\right) \cup g^{-1}\left(\frac{G_i}{k}\right).$$

It can be seen that the new reduced argument $x_\ell - \mathrm{corr}_i$ covers a wider range than just x_ℓ. In the worst case, this interval can have its length doubled, making the polynomial approximations potentially more expensive. Actually, those polynomials approximate functions that usually do not variate much on such reduced intervals, so that their degree need rarely be increased to achieve the same precision as for an interval that may be up to twice as narrow. This was tested with some of our tables with Sollya's `guessdegree` function [26]. In unfortunate cases for which the degree should be increased, rarely more than one additional monomial will be required, which barely adds two floating-point operations (one addition and one multiplication). Furthermore, such additional operations can often benefit from instruction parallelism in the evaluation scheme.

Also note that instead of considering polynomial approximations of $f(x)$ and $g(x)$ directly, we propose to incorporate the division by k into the polynomial coefficients. Therefore, we consider approximations $P_{f/k}(x)$ and $P_{g/k}(x)$ of $f(x)/k$ and $g(x)/k$, respectively, which avoid the prohibitive cost of the division and the associated rounding error.

2.4 APPLICATION TO SINE AND COSINE FUNCTIONS

The proposed lookup table for the second range reduction brings several benefits over existing solutions. However, building such tables of error-free values is not trivial, since the integers F_i, G_i, and k are not always straightforward, especially for transcendental functions. For the trigonometric and hyperbolic sine and cosine, we rely on some useful results on *Pythagorean triples* [156]. These objects are briefly described in the first part of this section. Then, we present a method to efficiently generate Pythagorean triples using the Barning-Hall tree. Finally, we detail the selection criteria for the generated Pythagorean triples to build exact lookup tables for the trigonometric and hyperbolic functions.

2.4.1 *Pythagorean triples and their properties*

Pythagorean triples are a set of mathematical objects from Euclidean geometry which have been known and studied since ancient Babylonia [30, ch. 6]. There exist several definitions of Pythagorean triples that usually differ on authorized values. We choose the following one:

Definition 3. *A triple of positive integers* (a, b, c) *is a* Pythagorean triple *if and only if* $a^2 + b^2 = c^2$.

A Pythagorean triple (a, b, c) for which a, b, and c are coprime is called a *primitive Pythagorean triple* (PPT). In the following, we will refer to the set of PPTs as $\mathbb{PPT}$. Recall that we are looking for *several* rational numbers which hold exact values for the trigonometric and hyperbolic functions. Yet a PPT and its multiples share the same rational values a/b, b/c, ... For example, the well known PPT $(3, 4, 5)$ and all its multiples can be associated to the ratio $a/b = 3/4$. Therefore we can restrict our search to primitive Pythagorean triples only, and apply a scale factor afterwards if needed.

According to the fundamental trigonometric and hyperbolic identities, we have

$$\forall x \in \mathbb{R}, \begin{cases} \cos(x)^2 + \sin(x)^2 = 1 \\ \cosh(x)^2 - \sinh(x)^2 = 1. \end{cases} \tag{2.3}$$

It follows from Definition 3 and Equation (2.3) that all Pythagorean triples can be mapped to rational values of trigonometric or hyperbolic sine and cosine. Indeed, let (a, b, c) be a Pythagorean triple. Without loss of generality, we can assume that $b \neq 0$. Then we have:

$$\begin{aligned} a^2 + b^2 = c^2 &\iff \left(\frac{a}{c}\right)^2 + \left(\frac{b}{c}\right)^2 = 1 \\ &\iff \left(\frac{c}{b}\right)^2 - \left(\frac{a}{b}\right)^2 = 1. \end{aligned}$$

Hence, for each Pythagorean triple (a, b, c), assuming $b \neq 0$,

$$\exists \theta \in [0, \pi/2[, \quad \cos(\theta) = \frac{b}{c} \quad \text{and} \quad \sin(\theta) = \frac{a}{c}$$
$$\exists \varphi \in \mathbb{R}_{\geqslant 0}, \quad \cosh(\varphi) = \frac{c}{b} \quad \text{and} \quad \sinh(\varphi) = \frac{a}{b}.$$

The consequences of these properties are twofold:

1. A PPT and its multiples share the same angles θ and φ.

2. Any Pythagorean triple can be mapped to the sides of a right triangle (possibly degenerated into a segment), whose hypotenuse is the third item c of the triple. This is illustrated in Figure 2.4.

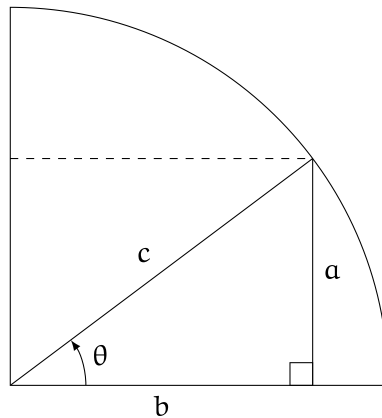


Figure 2.4 – Visual representation of the Pythagorean triple $(3, 4, 5)$.

Hence, in the following, the word "hypotenuse" is used to refer to the third item of a Pythagorean triple, while the word "legs" is used to refer to its first and second items. Also, we often treat Pythagorean triples as row or column vectors, which will prove to be a useful way to manipulate these objects.

2.4.2 *Construction of primitive Pythagorean triples*

The set of primitive Pythagorean triples $\mathbb{PPT}$ is countably infinite [7, 72, 139]. Figure 2.5 represents the subset of all PPTs with a hypotenuse $c < 2^{12}$. It shows that PPTs rapidly cover a wide range of angles over $[0, \pi/2]$ as c increases. But, as our exact lookup tables need one triple per row, one may ask if there will *always* be at least one PPT that matches each row, no matter how *narrow* the entry interval of a row can be.

Actually, a result due to Shiu states that the set of trigonometric angles covered by Pythagorean triples is dense in $[0, \pi/4]$. This is formalized by Theorem 1.

Theorem 1 (Shiu 1983 [154]). $\forall \theta \in [0, \pi/4], \forall \delta > 0$, *there exists a primitive Pythagorean triple with a corresponding trigonometric angle* θ', *such that*

$$|\theta - \theta'| < \delta.$$

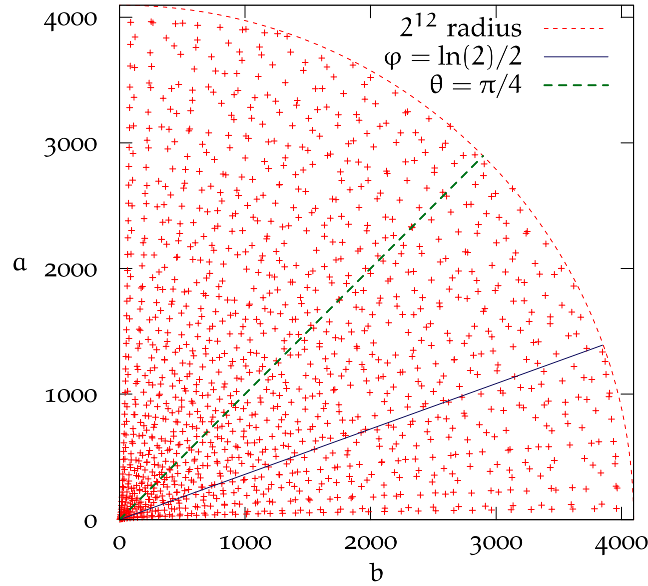


Figure 2.5 – Primitive Pythagorean triples with a hypotenuse $c < 2^{12}$.

Proof. (From [154]. We quote his proof because an analogous one will be used for Corollary 1.) Let $(a, b, c) \in \mathbb{PPT}$. It is well-known that, assuming that a is even, (a, b, c) can be rewritten $(2mn, m^2 - n^2, m^2 + n^2)$ where m, n are coprime integers satisfying $m > n > 0$. It follows that

$$\tan(\theta') = \frac{m^2 - n^2}{2mn} = \frac{1}{2}\left(\frac{m}{n} - \frac{n}{m}\right).$$

Let us write $t = \tan(\theta')$ and $r = m/n$ so that we have $r^2 - 2tr - 1 = 0$ and hence

$$r = t + \sqrt{t^2 + 1} = \tan(\theta') + \sec(\theta').$$

Note that we do not have $r = t - \sqrt{t^2 + 1}$ because $r > 0$.

It is now easy to prove the theorem. Let $0 \leqslant \theta \leqslant \pi/4$ and $u = \tan(\theta) + \sec(\theta)$. One can choose a sequence of rational numbers $r_0, r_1, r_2, \ldots$ converging to u, where

$$r_k = \frac{m_k}{n_k}, \quad k = 0, 1, 2, \ldots$$

such that m_k and n_k are positive and coprime [4]. Let $x = 2m_k n_k, y = m_k^2 - n_k^2, z = m_k^2 + n_k^2$ with corresponding angle θ_k. The angles $\theta_0, \theta_1, \theta_2, \ldots$ tend to θ, therefore the θ_k's can approximate θ arbitrarily closely, as required. □

By symmetry of the Pythagorean triples (a, b, c) and (b, a, c), the density of the aforementioned set can be extended to the interval $[0, \pi/2]$. A corollary is the density of the set of hyperbolic angles

covered by Pythagorean triples in $\mathbb{R}_+$. This is stated by Corollary 1 below.

Corollary 1. $\forall \varphi \in \mathbb{R}_+, \forall \delta > 0$, *there exists a primitive Pythagorean triple with an associated hyperbolic angle* φ', *such that*

$$\left|\varphi - \varphi'\right| < \delta.$$

Proof. By analogy, we replace $\tan(\varphi')$ by $\sinh(\varphi')$, and $\sec(\varphi')$ by $\cosh(\varphi')$ in Shiu's proof of Theorem 1. We have

$$\sinh(\varphi') = \frac{m^2 - n^2}{2mn} = \frac{1}{2}\left(\frac{m}{n} - \frac{n}{m}\right).$$

Let us write $s = \sinh(\varphi')$ and $r = m/n$ so that we have $r^2 - 2sr - 1 = 0$ and hence

$$r = s + \sqrt{s^2 + 1} = \sinh(\varphi') + \cosh(\varphi').$$

Let $\varphi \geqslant 0$ and $u = \sinh(\varphi) + \cosh(\varphi)$. One can choose a sequence of rational numbers $r_0, r_1, r_2, \ldots$ converging to u, where

$$r_k = \frac{m_k}{n_k}, \quad k = 0, 1, 2, \ldots$$

such that m_k and n_k are positive and coprime. Let $x = 2m_k n_k, y = m_k^2 - n_k^2, z = m_k^2 + n_k^2$ with corresponding hyperbolic angle φ_k. It follows that the hyperbolic angles $\varphi_0, \varphi_1, \varphi_2, \ldots$ tend to φ and that the φ_k's can approximate φ arbitrarily closely, as required. □

Although we are now certain that there will always be an infinite number of PPTs for each row of our tables, these theorems do not give any bounds on the size of the PPTs. In practice, we will see that the PPT sizes allow us to build tables in double precision indexed by usual values of indexing bits (up to 12). Also, we still have to find an easy means to generate PPTs efficiently. We choose to use the Barning-Hall tree [7, 72], which exhibits a ternary structure that links any PPT to three different PPTs. It allows to recursively generate PPTs quite efficiently: From any PPT represented as a column vector, the Barning-Hall tree allows to compute three new PPTs by multiplying the former with the three matrices

$$\begin{pmatrix} 1 & -2 & 2 \\ 2 & -1 & 2 \\ 2 & -2 & 3 \end{pmatrix} \quad \begin{pmatrix} -1 & 2 & 2 \\ -2 & 1 & 2 \\ -2 & 2 & 3 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 3 \end{pmatrix}. \tag{2.4}$$

It has been proven that all PPTs can be generated from the root $(3, 4, 5)$ with increasing hypotenuse lengths [139].

For every generated PPT (a, b, c), we also consider its *symmetric* PPT (b, a, c), because it may be interesting for three reasons:

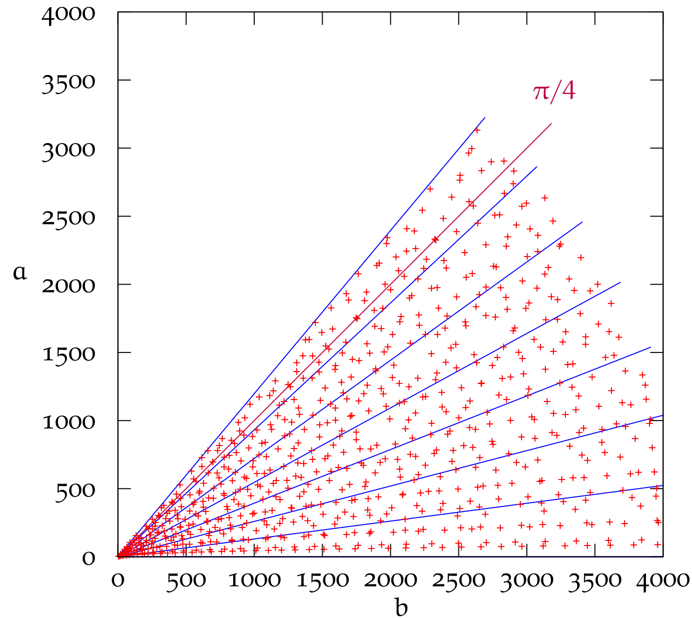


Figure 2.6 – Visual representation of PPTs and their symmetric counterparts falling into a trigonometric exact lookup table.

1. For the trigonometric functions, Figure 2.6 illustrates the following advantages:

   (a) If $\arcsin(b/c) > \pi/4$, one has $\arcsin(a/c) < \pi/4$, which falls into the range of the exact lookup table.

   (b) Whenever $\arcsin(b/c) \approx \pi/4$, one also has $\arcsin(a/c) \approx \pi/4$, and both triples may either fall into two different rows of the exact lookup table, which can help reduce the value k (since they share a common hypotenuse), or fall into the same subinterval, which can help find a better corrective term.

2. For the hyperbolic functions, since the reduced argument x lies in $[0, \ln(2)/2]$, interesting PPTs must satisfy $a/b \in \sinh([0,(n+1/2)\cdot 2^{-p}]) \supset [0, \sqrt{2}/4]$. Therefore, if $a \geqslant 4b/\sqrt{2}$, only the symmetric PPT (b, a, c) falls into the range of the lookup table.

Hence the first step of PPT generation using the Barning-Hall tree is the following: multiplying the matrices in Equation (2.4) by the root $(3, 4, 5)$ taken as a column vector, one gets the three new PPTs $(5, 12, 13)$, $(15, 8, 17)$, and $(21, 20, 29)$, and their symmetric counterparts $(12, 5, 13)$, $(8, 15, 17)$, and $(20, 21, 29)$. In the following, note that we always consider the “degenerated” PPT $(0, 1, 1)$, because it gives us an exact corrective term for the first table entry, without making the LCM k grow.

2.4.3 *Selection of primitive Pythagorean triples*

For each row i of the table indexed by $\lfloor x_h \cdot 2^p \rceil$, we want to select exactly one PPT (a, b, c) with a corresponding value $\theta = \arcsin(a/c)$ or $\varphi = \operatorname{arsinh}(a/b)$ such that:

$$
\begin{aligned}
|x_h - \theta| &< 2^{-p-1} \quad \text{for trigonometric functions, or} \\
|x_h - \varphi| &< 2^{-p-1} \quad \text{for hyperbolic functions.}
\end{aligned}
$$

The existence of such values θ and φ is a consequence of Theorem 1 and its Corollary 1.

In the following, we define our *corrective terms*, which we denote by corr, as:

$$
\begin{aligned}
\mathrm{corr} &= \theta - x_h \quad \text{for trigonometric functions, or} \\
\mathrm{corr} &= \varphi - x_h \quad \text{for hyperbolic functions.}
\end{aligned}
$$

Once one PPT has been selected for each row i, a naive solution would consist in storing exactly each a_i, b_i, c_i in the lookup table T, plus an approximation of corr on as many bits as necessary. Instead, as presented in Section 2.3, we suggest to store two integers C_h and S_h of the form

$$
C_h^{(i)} = \frac{b_i}{c_i} \cdot k \quad \text{and} \quad S_h^{(i)} = \frac{a_i}{c_i} \cdot k \tag{2.5}
$$

for the trigonometric functions, or

$$
C_h^{(i)} = \frac{c_i}{b_i} \cdot k \quad \text{and} \quad S_h^{(i)} = \frac{a_i}{b_i} \cdot k \tag{2.6}
$$

for the hyperbolic functions, where $k \in \mathbb{N}^*$ is the same for all table entries. In order to reduce the memory footprint of the table T, we look for *small* table entries C_h and S_h. This entails looking for a value k that is rather small, which can be simplified as a search for a small value in a set of least common multiples.

We showed in [97] that a straightforward approach, consisting in finding the minimum of the set of all possible LCMs, was quickly out of reach with current technology, as soon as tables had more than a few rows. For example, for the trigonometric functions, Figure 2.7 shows the number of PPTs per entry for $p = 7$, i.e. when there are 102 entries, such that the generated hypotenuses were less than 2^{18}. 18 corresponds to the minimum number of bits for the hypotenuses so that there be at least one PPT per entry. In this figure, the mean number of PPTs per entry is about 413, and the standard deviation is close to 33. Because we are looking for a *small* least common multiple k of one PPT hypotenuse per table entry, this means that a *naive* exhaustive search would have to compute the LCM for every set made of 101 elements, each of which having 413 different possible values in average. Think of it as finding the right combination of a padlock that has 101 rotating discs, each of which having approximately

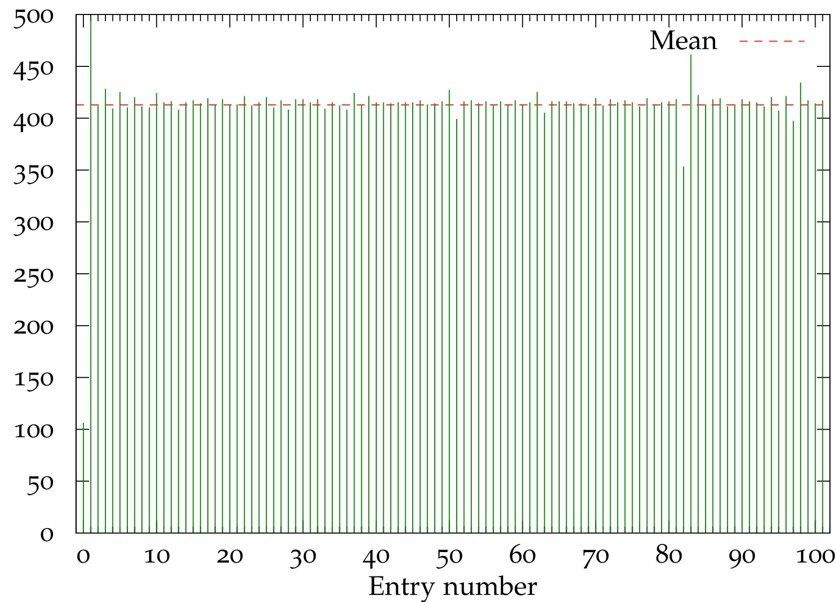


Figure 2.7 – Number of PPTs per entry of a trigonometric table for $p = 7$ with a hypotenuse less than 2^{18}. Each table entry contains at least one PPT.

413 symbols: this corresponds to $413^{101} \approx 10^{173}$ combinations! This is obviously way too many — there are only about 10^{80} atoms in the observable universe [39, pp. 37-43] — to permit an exhaustive search. Indeed, even if we could test a billion combinations per second, it would take approximately 10^{156} years to find the right one, which is about 10^{146} as much as the age of our universe.

It can be observed in Figure 2.7 that the number of PPTs for the table entry 0, that is, near the angle 0, is equal to 106 whereas the mean is around 413. Actually, there are advantages of not even considering this entry, by selecting the *degenerated* PPT $(0, 1, 1)$. This PPT corresponds to the angle $\theta = 0°$ or $\varphi = 0$ exactly, with a corrective term $\mathrm{corr} = 0$. We see three advantages of selecting this PPT:

1. The search space is reduced: we need not worry about the first entry;

2. Its hypotenuse is equal to 1, which will not impact the search of LCM, as $\mathrm{lcm}(k, 1) = 1$.

3. Its corresponding corrective term is exact and null: $\mathrm{corr} = 0$.

In Figure 2.8, we have reported the number of PPTs per entry for $p = 7$, such that the generated hypotenuses were less than 2^{14}. Here, 14 corresponds to the minimum number of bits for the hypotenuses so that there be at least one PPT per entry, with the first entry excluded from the search space. In this figure, the mean number of PPTs per entry is about 26 and the standard deviation is close to 5. Therefore, if the first entry is excluded, the search space is reduced to about $26^{100} \approx 10^{141}$ combinations to test. Again, this is still a way too large

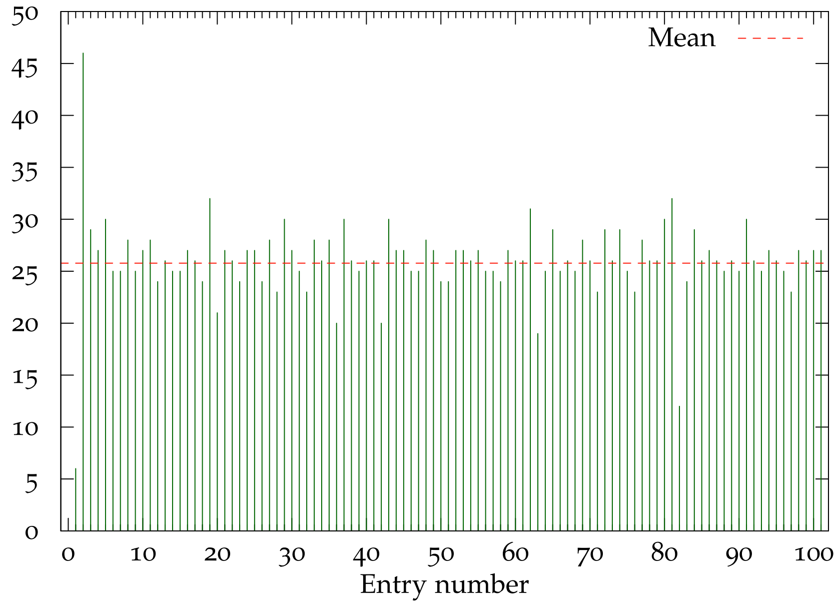


Figure 2.8 – Number of PPTs per entry of a trigonometric table for $p = 7$ with a hypotenuse less than 2^{14}. Only the first entry does not contain any PPT.

number of combinations to compute the smallest LCM in a reasonable time. To further reduce the space search, the LCM will not be computed but rather be looked for amongst generated hypotenuses.

To reduce the computational time, the solution we propose in [97] consists in looking for an LCM directly *amongst* generated values: hypotenuses (for the trigonometric functions), or bigger legs (for the hyperbolic functions) [96]. We call this solution "exhaustive search" in the sequel, as it *always* gives the smallest LCM. This claim may be counter-intuitive since one may ask why the smallest LCM would necessarily appear as the denominator side of some PPT. We prove this claim in [96]. It is done in three steps, which give Theorems 3 and 4. First, we start by proving that for any table, there is a set of PPTs that fills it, for which the denominator sides (either hypotenuses or bigger legs) have the least LCM possible:

Lemma 1. *If* $k \in \mathbb{N}^*$ *is the least element in the set of possible LCMs for a table of* n *rows, then there is a set of* primitive *Pythagorean triples* $\{T_i = (a_i, b_i, c_i)\}_{i \in \{0,\ldots,n-1\}}$, *with* T_i *belonging to row* i, *such that:*

$$\begin{aligned} \operatorname{lcm}(c_0, \ldots, c_{n-1}) = k & \quad \textit{for the trigonometric functions, and} \\ \operatorname{lcm}(b_0, \ldots, b_{n-1}) = k & \quad \textit{for the hyperbolic functions.} \end{aligned}$$

Proof. By construction. For readability, and without loss of generality, we will only consider the LCM for the trigonometric functions. The set of possible LCMs for a table T of n rows is non-empty as a consequence of Theorem 1. Thus, by the Well-Ordering Principle, we know that k exists [5, p. 13].

Since k exists, let $\left\{T'_0, \ldots, T'_{n-1}\right\}$ be a set of Pythagorean triples that fill the table T, such that $\mathrm{lcm}(c'_0, \ldots, c'_{n-1}) = k$. Note that the T'_i's need not necessarily be *primitive*. If there exists $i \in \{0, \ldots, n-1\}$ such that T'_i is non primitive, then there exists an integer $\alpha > 1$ and $T_i \in \mathbb{PPT}$ such that $T'_i = \alpha \cdot T_i$. Thus, we have

$$k = \mathrm{lcm}(c'_0, \ldots, \alpha \cdot c_i, \ldots, c'_{n-1}).$$

Hence α divides k. Since k is the *least* element of the set of possible LCMs, $k/\alpha < k$ cannot be a valid LCM. Therefore the prime factors of α must be shared with other c'_j. Hence, it is possible to write

$$k = \mathrm{lcm}(c'_0, \ldots, c_i, \ldots, c'_{n-1}).$$

As mentioned in Section 2.4.1, T_i and T'_i share the same angles θ and φ, and consequently they belong to the same subdivision of the table T. By repeating this process c'_i by c'_i as long as necessary, one can construct a set of *primitive* Pythagorean triples $\{T_i\}_{i \in \{0, \ldots, n-1\}}$ such that $k = \mathrm{lcm}(c_0, \ldots, c_{n-1})$, which concludes the proof. □

Second, let us recall two important results concerning primitive Pythagorean triples.

Theorem 2. *Let n be a positive integer. First n is the hypotenuse of a primitive Pythagorean triple if and only if all of its prime factors are of the form $4k+1$. Second, when $n > 2$, n is a leg of a primitive Pythagorean triple if and only if $n \not\equiv 2 \mod 4$.*

Proof. For the first result, see [155, ch. XI.3, 169]. For the second result, see [9, p. 116, 155, ch. II.3]. □

Now, for the trigonometric functions, we formulate Theorem 3, which states that the smallest LCM k is the hypothenuse of a PPT.

Theorem 3. *If $k \in \mathbb{N}^*$ is the least element of the set of possible LCMs for a table of n rows for trigonometric functions, then k is the hypotenuse of a* primitive *Pythagorean triple.*

Proof. Lemma 1 taught us that each T_i could be made primitive. Thus, by Theorem 2, we have:

$$\forall i \in \{0, \ldots, n-1\},\ \exists \left\{\beta_{p,i}\right\}_{p \in \mathbb{P}_\pi} \in \mathbb{N}^{\mathbb{P}_\pi},\ c_i = \prod_{p \in \mathbb{P}_\pi} p^{\beta_{p,i}}$$

where $\mathbb{P}_\pi$ is the set of Pythagorean primes, i.e. the set of prime numbers congruent to 1 modulo 4.[1] By construction, we have

$$k = \mathrm{lcm}(c_0, \ldots, c_{n-1}) = \prod_{p \in \mathbb{P}_\pi} p^{\max_i \beta_{p,i}}$$

[1] OEIS Foundation Inc. (2018), The On-Line Encyclopedia of Integer Sequences, http://oeis.org/A002144.

which means that k is a product of Pythagorean primes. By Theorem 2, k is the hypotenuse of at least one primitive Pythagorean triple, which concludes the proof. □

Finally, for the case of hyperbolic functions, Theorem 4 shows that the smallest LCM k is a leg of a PPT.

Theorem 4. *If $k \in \mathbb{N}^*$ is the least element of the set of possible LCMs for a table of n rows for hyperbolic functions, then k is a leg of a* primitive *Pythagorean triple.*

Proof. By Theorem 2, we only need to prove that $k \not\equiv 2 \mod 4$.

Using Lemma 1, we know that each $T_i = (a_i, b_i, c_i)$ can be made primitive. Thus, using the aforementioned equivalence, we have

$$\forall i \in \{0, \dots, n-1\},\ b_i \not\equiv 2 \mod 4. \tag{2.7}$$

Hence, either every b_i is odd, or there exists $i \in \{0, \dots, n-1\}$ such that b_i is even. In the first case, the LCM k of all b_i's is obviously odd too. In the latter case, b_i is even implies that it is a multiple of 4 as a consequence of Equation (2.7), which entails that k is also a multiple of 4. Therefore, by construction,

$$k = \operatorname{lcm}(b_0, \dots, b_{n-1}) \not\equiv 2 \mod 4,$$

which concludes the proof. □

Theorems 3 and 4 justify why we call our search amongst generated hypotenuses or legs "exhaustive", because by searching *exhaustively* amongst increasing PPT hypotenuses or legs, the smallest LCM *will* be found eventually. However, we will see in the next section that such an exhaustive search is exponentially expensive with p. Therefore we will also detail the design and implementation of efficient heuristics to build exact lookup tables.

2.5 IMPLEMENTATIONS AND NUMERIC RESULTS

In this section, we present how we implemented the proposed method to generate exact lookup tables as described in Section 2.4. We have designed three solutions to look for a small common multiple k. The first solution, presented in Section 2.5.1, is based on an *exhaustive* search and allows us to build tables indexed by up to 7 bits in a reasonable amount of time. The second solution uses a *heuristic* approach, which reduces memory consumption during the execution and the search time. Presented in Section 2.5.2, it allows us to build tables indexed by 10 bits much faster than the exhaustive search. Finally, the third solution, originally proposed as a perspective in [96], uses a much more efficient heuristic for the trigonometric functions. It allows us to build tables up to the limit of binary64-representable entries, that is, indexed by 14 bits, in a few seconds. This third solution might also be adaptable to the hyperbolic functions, as explained in Section 2.5.3.

2.5.1 *Exhaustive search*

This section first describes the design of our exhaustive algorithm, then discusses the different choices that we must make when several PPTs are available inside one table row for the found LCM. Finally, numeric results are presented for various precisions p along with two toy exact lookup tables for the trigonometric and hyperbolic functions.

Algorithm

As seen in Section 2.4.3, we restrain our search to the set of generated hypotenuses c or to the set of generated legs b, for the trigonometric and hyperbolic functions, respectively. In other words, we require that the LCM k be the hypotenuse or a leg of one of the generated PPTs. Moreover, for the hyperbolic functions, one has $\sinh(\ln(2)/2) < 1$, so that the search can safely be limited to the set of bigger legs only.

To perform such an exhaustive search, we designed a C++ program that takes as input the number p of bits used to index the table T and tries to generate an exact lookup table. To achieve this, the algorithm looks for the smallest integer k, amongst the generated values, that is a multiple of at least one PPT hypotenuse or leg per entry. By Theorems 3 and 4, this search is guaranteed to find the smallest LCM. The algorithm is the following: Start with a maximum denominator size (in bits) of $n = 4$ and then follow these four steps:

1. Generate all PPTs (a, b, c) such that $c \leqslant 2^n$ for the trigonometric functions, or $c \leqslant 2^n / \cos((N - 1/2) \cdot 2^{-p})$, where N is the number of table entries, for the hyperbolic functions. The latter inequality guarantees that every interesting leg for the search step will be generated. It is during this step that the Barning-Hall matrices from Equation (2.4) are used.

2. Store only the PPTs that belong to a table entry, i.e. the potentially interesting ones.

3. Search for the LCM k among the PPT hypotenuses c or legs b that lie in $[2^{n-1}, 2^n]$, for the trigonometric or the hyperbolic functions, respectively.

4. If such a k is found, build values $(S_h, C_h, RN_j(corr_i))$ for every row using Equation (2.5) or (2.6) and return an exact lookup table. Otherwise, set $n \leftarrow n + 1$ and go back to step 1.

Possible options for selecting only one triple per row

Sometimes, whenever k is found, several PPTs are possible for a same table row. To build the final table, however, only one PPT per row is to be used. Then the selection of only one of the available PPTs depends on which goal we want to achieve. Said another way, the choice of one PPT over another may simply rely on an optimization function

to minimize or maximize. Here, we make the choice of selecting the PPT for which $\arcsin(a/c)$ or $\operatorname{arsinh}(a/b)$ is the closest to x_h, as this minimizes the ratio between corr and x_ℓ. This virtually increases the precision of the corrected reduced argument $x_\ell - \mathrm{corr}$.

But we see at least three alternatives to this solution:

1. We could try to minimize the value $\max_i(\mathrm{corr}_i) - \min_j(\mathrm{corr}_j)$. This could be interesting to store smaller corrective terms $\delta_i = \mathrm{corr}_i - \min_j(\mathrm{corr}_j)$, and incorporate $\min_j(\mathrm{corr}_j)$ into the polynomial approximations. But minimizing such a value might be much more expensive as its time complexity is $\prod_{i=0}^{n-1} m_i$, where m_i is the number of possible triples for the entry i.

2. In a way similar to Gal's strategy explained in Section 2.2.2, we could choose the PPT for which the corrective term has as many identical bits as possible after its least significant stored bits. If there are enough, the precision of the corrective term is also virtually extended, which could allow for smaller tables.

3. Finally, we could relax the constraints on the k_i's to allow greater values (i.e. non optimal) of the LCM k, so that the corrective terms may have even lower magnitudes or be even closer to machine numbers.

The last option can be done in different fashions, but we will only describe one to explain the idea to the reader. As we have seen in Section 2.4.3, the greater p, the lower the magnitudes of the corrective terms. Therefore it becomes possible to build tables for lower values of p with exponentially lower magnitudes of corrective terms. For instance, imagine that we want to build a hyperbolic table indexed by 7 bits. Instead of building the one for which k is optimal, let us first build a table indexed by 10 bits. Now let us call i_7 and i_{10} the row indices of both tables, respectively. Then, we only need to keep the rows i_{10} from the table indexed by 10 bits such that:

$$i_{10} \cdot 2^{-10} = i_7 \cdot 2^{-7},$$

which means the set

$$\left\{i_{10} = i_7 \cdot 2^3 \mid i_7 \in \left[0 \,..\, \left\lfloor 2^6 \ln 2 \right\rfloor\right]\right\} = \{i_{10} = 8 i_7 \mid i_7 \in [0 \,..\, 44]\}.$$

Since the corrective terms in the table indexed by 10 bits have a magnitude bounded by 2^{-11}, then so will the corrective terms of our improved table indexed by 7 bits, whereas they were only bounded by 2^{-8} for a classic table. Therefore, just by doing this, it is possible to reduce the bound on the corrective terms by a factor 8 while keeping exactly stored values of sinh and cosh.

The reader may object that for some configurations of tables, say indexed by p bits and p' bits such that $p < p'$, the last row cannot be built because

$$\max i_{p'} < 2^{p'-p} \max i_p.$$

Table 2.1 – Exhaustive Search Results for sin and cos.

p	k	n	Time (s)	PPTs	Hypotenuses
3	425	9	$\ll 1$	87	33
4	5525	13	$\ll 1$	1405	428
5	160 225	18	0.2	42 329	11 765
6	1 698 385	21	7	335 345	87 633
7	6 569 225	23	31	1 347 954	335 645
8	$> 2^{27}$	> 27	> 6700	$> 21\,407\,993$	$> 4\,976\,110$

Table 2.2 – Exhaustive Search Results for sinh and cosh.

p	k	n	time (s)	PPTs	Bigger legs
3	144	8	$\ll 1$	23	12
4	840	10	$\ll 1$	86	43
5	10 080	14	$\ll 1$	1202	610
6	171 360	18	9	18 674	9312
7	1 081 080	21	328	147 748	72 476
8	$> 2^{24}$	> 24	$> 60\,000$	$> 1\,188\,585$	$> 574\,800$

This case happens e.g. for $p = 10$ and $p' = 13$ for the hyperbolic functions: $\max i_{13} = \lfloor 2^{12} \ln 2 \rceil = 2839$ and $\max i_{10} = \lfloor 2^{9} \ln 2 \rceil = 355$, which gives no solution for $\max i_{13} = 2^3 \cdot 355 = 2840$. In such a case, selecting the last row of the table indexed by p' bits would suffice to build the last row of the table indexed by p bits. This, however, may theoretically increase the upper bound on the magnitudes of the improved corrective terms up to 2^{-p-1} because of the last row. Instead, it is possible to build a custom table indexed by p' bits with $\max i_{p'} = 2^{p'-p} \max i_p$. In our example, this would correspond to a table with 2841 rows.

This technique could increase the precision of each corrective term at the cost of greater PPTs. As long as the exact stored values are still machine-representable, then this method remains interesting for software implementations. However, for hardware implementations, this might be less interesting since it involves more storage bits than a regular table.

Numeric results

Tables 2.1 and 2.2 show the results obtained for the trigonometric and hyperbolic functions, respectively. Timings were measured on a server with an Intel Xeon E5-2650 v2 @ 2.6 GHz processor (16 physical cores) with 125 GB of RAM running on GNU/Linux. For the number p of bits that is targeted, the tables describe the value k that was found, followed by the number n of bits used to represent k (that is, $n = \lceil \log_2(k) \rceil$), the time (in seconds) taken by our program to run, and the numbers of PPTs and denominators considered during the LCM search.

Table 2.3 – A Trigonometric Exact Lookup Table Computed for $p = 4$.

Index	S_h	C_h	corr
0	0	5525	+0x0.0000000000000p+0
1	235	5520	-0x1.46e9e7603049fp-6
2	612	5491	-0x1.cad996fe25a24p-7
3	1036	5427	+0x1.27ac440de0a8cp-10
4	1360	5355	-0x1.522b2a9e8491dp-10
5	1547	5304	-0x1.d6513b89c7237p-6
6	2044	5133	+0x1.038b12ae4eba1p-8
7	2340	5005	-0x1.53f734851f48bp-13
8	2600	4875	-0x1.49140da6fe454p-7
9	2880	4715	-0x1.d02973d03a1f6p-7
10	3315	4420	+0x1.2f1f464d3dc25p-6
11	3500	4275	-0x1.7caa112f287aep-10
12	3720	4085	-0x1.735972faced77p-7
13	3952	3861	-0x1.fa6ed9240ab1ap-7

Table 2.4 – A Hyperbolic Exact Lookup Table Computed for $p = 5$.

Index	S_h	C_h	corr
0	0	10 080	+0x0.0000000000000p+0
1	284	10 084	-0x1.93963974f0cb6p-9
2	651	10 101	+0x1.0b316b3c740d1p-9
3	1064	10 136	+0x1.7c74108520aebp-7
4	1190	10 150	-0x1.d8f891d50d1a1p-8
5	1560	10 200	-0x1.13297ef8b55bbp-9
6	1848	10 248	-0x1.535fdc36d3139p-8
7	2222	10 322	-0x1.fe04ef1053a97p-15
8	2560	10 400	+0x1.5891c9eaef76ap-10
9	2940	10 500	+0x1.a58844d36e49ep-8
10	3237	10 587	+0x1.b77a5031ebc86p-9
11	3456	10 656	-0x1.dcf49bb32dc17p-8

As can be seen, it was possible to find k and to build tables indexed by up to $p = 7$ bits in a reasonable amount of time. However, it is clear that the number of dynamic memory allocations, which are mainly used to store the triples and the denominators, grows exponentially with p. Consequently, it was not possible to find k for $p \geqslant 8$ with our hardware configuration.

Table 2.3 describes an exact lookup table for the trigonometric functions when $p = 4$, where $k = 5525$ and the absolute value of the corrective terms is at most `0x1.d6513b89c7237p-6`, that is, ≈ 0.0287 for input index $i = 5$. Table 2.4 presents an exact lookup table for the hyperbolic functions when $p = 5$, where $k = 10\,080$ and the absolute value of the corrective terms is at most `0x1.7c74108520aebp-7`, that is, ≈ 0.0116 for input index $i = 3$. Figures 2.9 and 2.10 give a visual representation of those tables in the Euclidean plane, depicting the

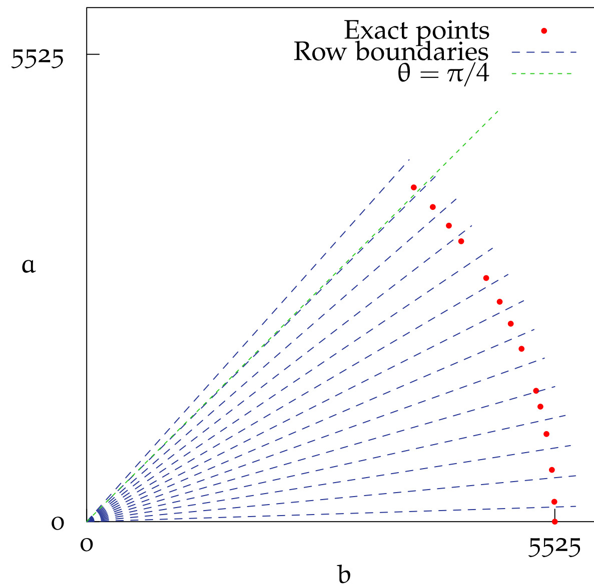


Figure 2.9 – Visual representation of the trigonometric exact lookup table from table 2.3 in the Euclidean plane.

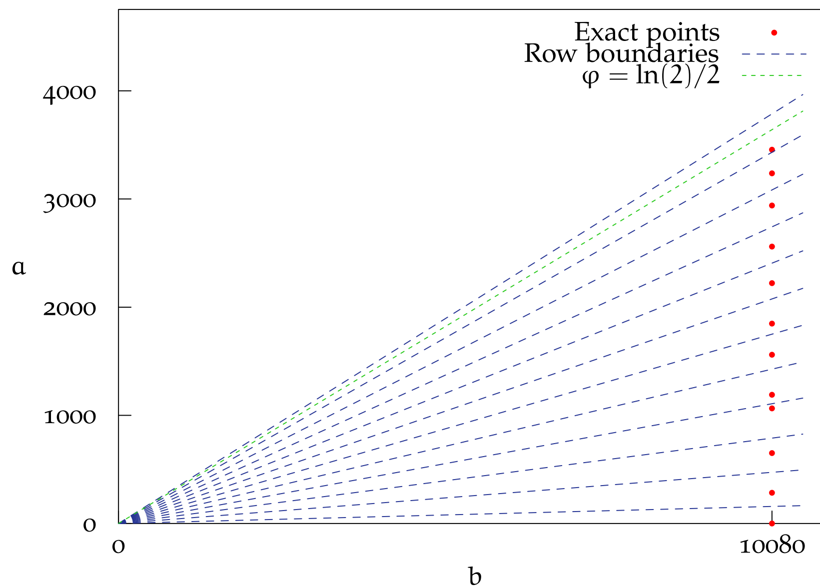


Figure 2.10 – Visual representation of the hyperbolic exact lookup table from table 2.4 in the Euclidean plane.

stored triples by dots and the limits of each entry interval by dashed lines.

2.5.2 *Characterizing exhaustive results: a first heuristic*

To build tables indexed by a larger number of bits, a more efficient solution must be found. In order to drastically reduce the search space, we have developed two heuristics, one for the trigonometric functions and one for the hyperbolic functions. These heuristics try to characterize the denominator sides (hypotenuses or bigger legs) to either keep or reject PPTs during the generation step.

Table 2.5 – Prime Factorization of Found Common Multiples for sin and cos.

k	Prime factorization
425	$5^2 \cdot 17$
5525	$5^2 \cdot 13 \cdot 17$
160 225	$5^2 \cdot 13 \cdot 17 \cdot 29$
1 698 385	$5 \cdot 13 \cdot 17 \cdot 29 \cdot 53$
6 569 225	$5^2 \cdot 13 \cdot 17 \cdot 29 \cdot 41$

Table 2.6 – Prime Factorization of Found Common Multiples for sinh and cosh.

k	Prime factorization
144	$2^4 \cdot 3^2$
840	$2^3 \cdot 3 \cdot 5 \cdot 7$
10 080	$2^5 \cdot 3^2 \cdot 5 \cdot 7$
180 180	$2^2 \cdot 3^2 \cdot 5 \cdot 7 \cdot 11 \cdot 13$
1 081 080	$2^3 \cdot 3^3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$

For this purpose, let us observe the decomposition in prime factors of each k found using the exhaustive search. Such decompositions are given in Tables 2.5 and 2.6 for both the trigonometric and the hyperbolic tables. These factorizations show that every k in the table is a composite number divisible by relatively small primes. Furthermore, for the trigonometric functions, all those small primes are of the form $4n+1$, as per Theorem 2.

Therefore, each heuristic that we propose follows a simple rule: for the trigonometric functions, only store PPTs with a hypotenuse of the form:

$$c = \prod_{p \in \mathbb{P}_\pi} p^{r_p} \quad \text{with} \quad \begin{cases} r_p \in \mathbb{N}^* & \text{if} \quad p = 5 \\ r_p \in \{0, 1\} & \text{if} \quad 13 \leqslant p \leqslant 73 \\ r_p = 0 & \text{if} \quad p > 73 \end{cases} \tag{2.8}$$

Table 2.7 – Heuristic Search Results for sin and cos.

p	k	n	time (s)	PPTs	Hypotenuses
3	425	9	$\ll 1$	42	8
4	5525	13	$\ll 1$	211	17
5	160 225	18	$\ll 1$	996	40
6	1 698 385	21	0.1	2172	66
7	6 569 225	23	0.4	3453	69
8	314 201 225	29	9.5	10 468	100
9	12 882 250 225	34	294	20 312	109
10	279 827 610 985	39	9393	33 057	110

Table 2.8 – Heuristic Search Results for sinh and cosh.

p	k	n	time (s)	PPTs	Bigger legs
3	144	8	$\ll 1$	23	12
4	840	10	$\ll 1$	65	24
5	10 080	14	$\ll 1$	247	79
6	180 180	18	$\ll 1$	917	193
7	1 081 080	21	0.3	1743	248
8	17 907 120	25	3.2	3909	388
9	147 026 880	28	23	5802	400
10	2 793 510 720	32	350	9012	502

where $\mathbb{P}_\pi$ is the set of Pythagorean primes:

$$\mathbb{P}_\pi = \{5, 13, 17, 29, 37, 41, 53, 61, 73, \dots\}.$$

For the hyperbolic functions, the heuristic only stores PPTs with a bigger leg of the form:

$$b = \prod_{p \in \mathbb{P}} p^{r_p} \quad \text{with} \quad \begin{cases} r_p \in \mathbb{N}^* & \text{if} \quad p < 5 \\ r_p \in \{0, 1\} & \text{if} \quad 5 \leqslant p \leqslant 23 \\ r_p = 0 & \text{if} \quad p > 23 \end{cases} \tag{2.9}$$

where $\mathbb{P}$ is the set of all primes:

$$\mathbb{P} = \{2, 3, 5, 7, 11, 13, 17, 19, 23, \dots\}.$$

Results, timings, and numbers of considered triples and potential LCMs (hypotenuses or legs) for this heuristic are given in Tables 2.7 and 2.8 for the trigonometric and the hyperbolic functions, respectively. As can be seen, this algorithm considers a number of potential LCMs several orders of magnitude less than the exhaustive search solution. This reduces drastically the memory usage and execution times. For instance, for the trigonometric functions, when $p = 7$, only 3453 triples are stored, compared to the 1 347 954 triples for the exhaustive algorithm. In this first case, the execution time was reduced from 31 seconds to 0.4 seconds. And for the hyperbolic functions,

when $p = 7$, only 1743 triples are stored, while there were 147 748 when using the exhaustive algorithm. In this second case, the execution time was reduced from 328 seconds to 0.3 seconds.

With this heuristic, the bottleneck is no longer the memory but the selection of PPTs during their generation. Indeed, this selection is carried out on the elements of an exponentially-growing set. And checking if a given integer satisfies either Equation (2.8) or Equation (2.9) requires checking if it is multiple of certain prime numbers, which is a rather expensive test involving integer division.

2.5.3 *A much faster heuristic using Fässler's algorithms*

In [96], we concluded that it could be interesting to further characterize potential small LCMs in order to speed up the table precomputation process even more. On one hand, since precomputing large tables ($p \geqslant 10$) was still exponentially expensive, we were unable to generate them for $p > 10$ in a reasonable time. On the other hand, such large tables are rarely useful because of the exponential memory footprint they involve compared to the small computing benefits they bring in return. However, in this section, we will detail another heuristic that allows us to go a little bit further, for two reasons. First, we find this theoretical problem mathematically interesting in itself. Second, and more importantly, as we have seen in Section 2.5.1, solving it could help us improve the tables for lower values of p. Furthermore, the theoretical background of this heuristic gives us lower bounds for the optimal k.

How to generate Pythagorean triples sharing a common hypotenuse?

Before we present our second heuristic, let us observe the variations of the previously found LCMs bit-lengths n with respect to the table index sizes p. They are represented in Figure 2.11 and Figure 2.12, for the trigonometric and hyperbolic functions, respectively. Using the gnuplot 4.6 `stats` command[1], we performed ordinary least squares (OLS) linear regressions on both datasets, showing very good linear correlation coefficients between n and p. We can see that if the extrapolation is meaningful, which seems a plausible hypothesis, then it would be impossible to store tabulated values as exact binary64 floating-point numbers starting from $p \approx 13$ or 14 for the trigonometric functions and from $p \approx 15$ or 16 for the hyperbolic functions.

Therefore, we choose to set our ultimate objective to generating trigonometric and hyperbolic tables in a reasonable amount of time, and indexed by at most those values ($p = 14$ or $p = 16$).

In the following, we focus only on the trigonometric functions. We propose to directly compute Pythagorean triples (PTs) sharing a common hypotenuse instead of generating huge sets of PPTs and then selecting the relevant ones. To do this, we will use some of Fässler's inspiring mathematical results taken from [64, pp. 505-511]. Fässler

[1] See `http://gnuplot.info/` and [177, p. 173].

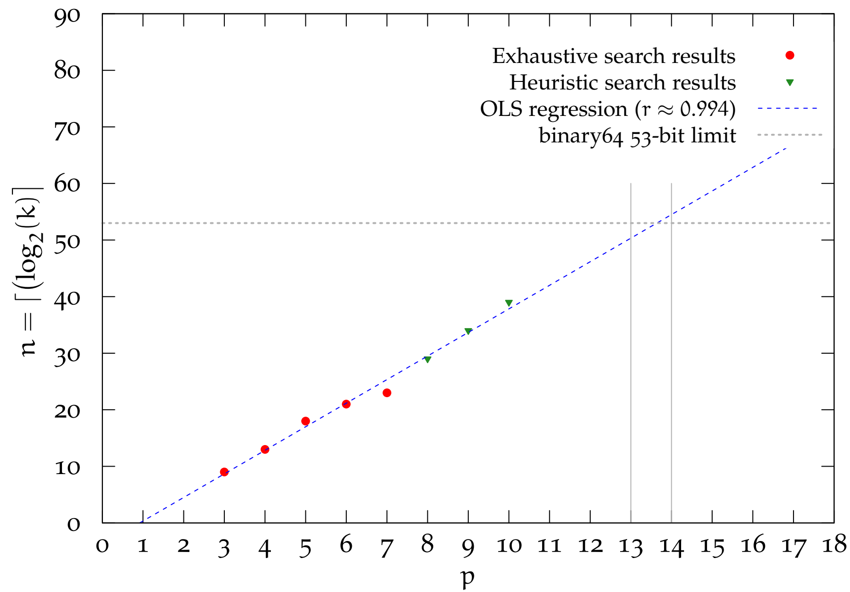


Figure 2.11 – Bit-length n of k with respect to the table index size p for the trigonometric functions.

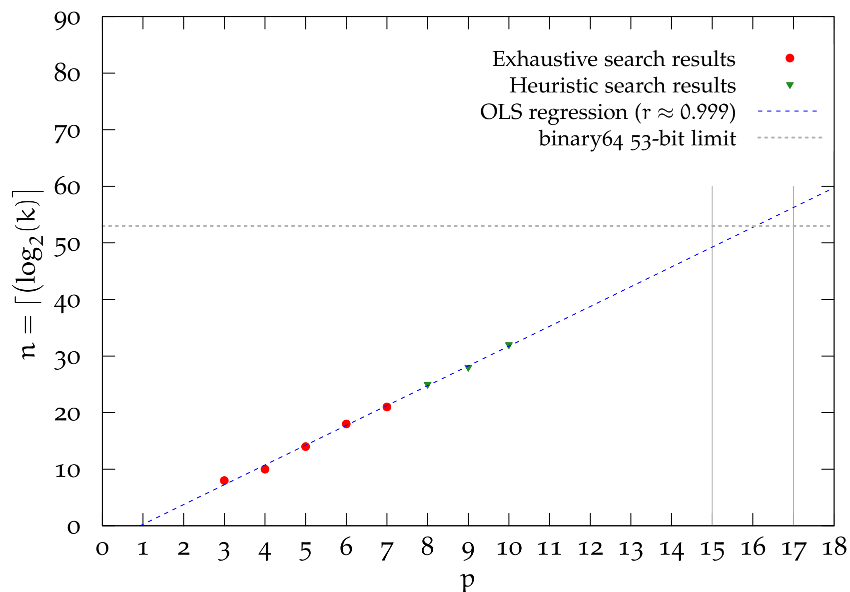


Figure 2.12 – Bit-length n of k with respect to the table index size p for the hyperbolic functions.

makes use of several results from number theory: First, he brings back an antique way of representing *any* PPT (a, b, c), which dates back to at least the time of Euclid:

$$\begin{aligned} a = 2xy, \quad & b = x^2 - y^2, \quad c = x^2 + y^2 \\ \text{with} \quad x > y > 0, \quad & \gcd(x, y) = 1, \quad x + y \equiv 1 \mod 2. \end{aligned} \tag{2.10}$$

Second, as already stated in Theorem 2, the hypotenuse c is necessarily of the form

$$c = p_1^{\beta_1} \cdot p_2^{\beta_2} \cdots p_n^{\beta_n} \tag{2.11}$$

with pairwise different primes $p_i \equiv 1 \mod 4$. Fässler assumes that each $\beta_i \geqslant 1$, and we follow his assumption. Then, he reminds us about a famous theorem:

Theorem 5 (Fermat's theorem on sums of two squares). *A prime number p is of the form $4m + 1$ if and only if it can be written as a unique sum of two squares $p = x^2 + y^2$, up to a permutation of the squares:*

$$\exists (x, y) \in \mathbb{N}^2 \mid p = x^2 + y^2 \iff p \equiv 1 \mod 4.$$

With these results in mind, Fässler uses the following identity, known as either the Brahmagupta-Fibonacci identity or the Diophantus identity [160, §III.19]

$$(r^2 + s^2) \cdot (u^2 + v^2) = (ru \pm sv)^2 + (rv \mp su)^2 \tag{2.12}$$

to show that since each prime factor p_i of c can be represented as a sum of two squares, then it is recursively possible to write c as 2^{n-1} sums of two squares where n is the number of prime factors of c. Furthermore, the representations given by Equation (2.12) comply with all the requirements of Equation (2.10), so that all the 2^{n-1} sums of two squares can be used to generate 2^{n-1} PPTs sharing the same hypotenuse c using Equation (2.10). For instance, there are only two PPTs that share the common hypotenuse $65 = 5 \cdot 13$: First, we write $5 = 1^2 + 2^2$ and $13 = 2^2 + 3^2$. Using Equation (2.12), we have:

$$\begin{aligned} 65 = \left(1^2 + 2^2\right) \cdot \left(2^2 + 3^2\right) &= (2 \pm 6)^2 + (3 \mp 4)^2 \\ &= 8^2 + 1^2 = 4^2 + 7^2. \end{aligned}$$

Then using Equation (2.10), we can compute the *only* two PPTs (a, b, c) that share the hypotenuse $c = 65$, namely $(16, 63, 65)$ and $(56, 33, 65)$. Actually, Fässler acknowledges that these results were progressively developed by Fermat, Euler, Lagrange and Legendre. A corollary of these results is what Fässler names the "Formula of Legendre":

Corollary 2 (Formula of Legendre). *If c has the form in Equation (2.11), then there are exactly M Pythagorean triples, where*

$$M = \frac{(2\beta_1 + 1) \cdot (2\beta_2 + 1) \cdots (2\beta_n + 1) - 1}{2}.$$

Since we know the number of Pythagorean triples that are needed for a given table, we are now able to find a more precise lower bound of k thanks to Corollary 2.

Improved lower bound for the optimal least common multiple

Corollary 2 gives us the exact number of Pythagorean triples (PTs) sharing the same hypotenuse c. Hence the lower bound on k can be improved by requiring that the number M of PTs sharing the same hypotenuse k be at least $\lfloor 2^{p-1}\pi/2 \rceil$. For instance, for $p = 5$, our trigonometric table has 26 entries, so we need to find at least 25 PTs sharing the same hypotenuse c. There is one less triple needed because we exclude the first entry from the search, as seen in Section 2.4.3.

Let us assume for a moment that $k = p_1 \cdot p_2 \cdots p_n$, i.e. that each $\beta_i = 1$ for $1 \leqslant i \leqslant n$. From Corollary 2, we have $(3^n - 1)/2$ triples sharing the same hypotenuse c. In order to have at least 25 PTs with the least k, we need the first n Pythagorean primes such that:

$$\frac{3^n - 1}{2} \geqslant 25 \iff n \geqslant \frac{\log(51)}{\log(3)} \iff n \geqslant 4.$$

So if each $\beta_i = 1$ and $n = 4$, there are exactly $(3^4 - 1)/2 = 40$ PTs. Thus the minimum value of such a hypotenuse is the product of the first four Pythagorean primes: $c \geqslant 5 \cdot 13 \cdot 17 \cdot 29 = 32\,045$. Actually, our first assumption that each $\beta_i = 1$ may be too simple. Indeed, for $n \geqslant 2$, if we remove the greatest Pythagorean prime factor of c, and add x to $\beta_1 = 1$ so that now $\beta_1 = 1 + x$, we have:

$$M = \frac{(2x+3) \cdot 3^{n-2} - 1}{2}.$$

Hence, solving $M \geqslant \frac{3^n - 1}{2}$ for x, one finds $x \geqslant 3$. This means that for any $n \in \mathbb{N}^*$, if $\beta_1 \geqslant 4$, then there are at least as many PTs sharing the hypotenuse $c_1 = 5^{\beta_1} \prod_{2 \leqslant i \leqslant n-1} p_i$ as there are PTs sharing the hypotenuse $c_2 = \prod_{1 \leqslant i \leqslant n} p_i$. Solving $c_1 < c_2$ for p_n, we find $p_n > 5^{\beta_1}$ with $\beta_1 \geqslant 4$, i.e. $p_n \geqslant 625$. The least p_n satisfying this inequality is $p_{55} = 641$. Therefore,

$$c_1 = 5^4 \cdot \prod_{2 \leqslant i \leqslant 54} p_i < c_2 = \prod_{1 \leqslant i \leqslant 55} p_i$$

is the least example showing than c_2 may not be a lower bound for k. Fortunately, we have no practical need for such a consideration because $M = \frac{3^{54} - 1}{2}$ is greater than 10^{25}, which is way greater than our practical use: M generally does not exceed 10^5 for our tables. Hence we will compute a lower bound on k with each β_i set to 1.

Table 2.9 – "Fässler's" Heuristic Search Results for sin and cos.

p	k	n	time (s)	PPTs	Hypotenuses
3	425	9	$\ll 1$	43	8
4	5525	13	$\ll 1$	225	16
5	160 225	18	$\ll 1$	1071	34
6	1 698 385	21	0.1	2317	49
7	6 569 225	23	0.2	3757	50
8	314 201 225	29	0.8	10 861	44
9	12 882 250 225	34	2	19 243	27
10	279 827 610 985	39	6	26 947	11
11	3 929 086 318 625	42	6	29 434	3
12	286 823 301 259 625	49	15	32 581	1
13	not found	> 53	1	32 576	0

Description of the algorithm

Now that we have improved our knowledge about PTs sharing a common hypotenuse and the lower bound for k, let us explain how we can use Equation (2.12) in a much faster heuristic to generate trigonometric tables. First, we must determine the decompositions of the first Pythagorean primes up to 73 as sums of two squares. This can be done by a precomputed simple exhaustive search: Assuming $c > b > a > 0$, we find

$$\sqrt{c/2} < b \leqslant \sqrt{c-1} \quad \text{and} \quad 1 \leqslant a \leqslant \sqrt{c-1} - 1.$$

For example, if $c = 73$, we can search for $(a, b) \in [1..8] \times [7..8]$, which is almost instantaneous: $73 = 3^2 + 8^2$. Second, if the trigonometric table needs N triples, we initialize the lower bound for k with

$$\beta_i = 0 \quad \forall i \geqslant \frac{\log(2N+1)}{\log(3)} \quad \text{and} \quad \beta_i = 1 \quad \text{otherwise.}$$

Then our heuristic follows these six steps:

1. Generate an arbitrarily small set $\mathbb{K}$ of possible LCMs k, using $1 \leqslant \beta_1 \leqslant 3$ for $p_1 = 5$ and $\beta_i \in \{0, 1\}$ otherwise, according to the lower bound found for k.

2. Set $c \leftarrow \min\{c \in \mathbb{K}\}$.

3. Generate all PTs sharing the same hypotenuse $c = \prod_i p_i^{\beta_i}$, by composing Equation (2.12);

4. If there is at least one PT in each table entry, generate the final table and terminate;

5. Else, remove c from $\mathbb{K}$.

6. If $\mathbb{K} = \emptyset$, return an error. Else go back to Step 2.

The results using this heuristic are summarized in Table 2.9. As opposed to the linear model prevision that we drew from Figure 2.11, you can see that our heuristic was not able to find an LCM k that could be stored exactly in a single binary64 floating-point number for $p = 13$. However, we can see that this method allows to build exact trigonometric tables up to $p = 12$ within reasonable amounts of time and memory usage, and that it finds the same LCMs k as both the exhaustive method and the first heuristic up to $p = 10$, which comforts the results found for $p \in \{11, 12\}$.

In this section, we did not address the generation of exact lookup tables for the hyperbolic functions. For this problem, an efficient algorithm that generates all the Pythagorean triples sharing a common bigger leg would be needed. A first lead might be to follow the results presented in [147] by Rothbart and Paulsell. An open question remains to be answered: does this heuristic actually give optimal results for binary64 exact lookup tables whenever we use unlimited values for the β_i's?

2.6 COMPARISONS WITH OTHER METHODS

We have presented a range reduction for the trigonometric and hyperbolic functions based on exact lookup tables and a method to efficiently build such tables. In order to compare this solution with the other solutions presented in Section 2.2.2, we consider a two-phase evaluation scheme for the trigonometric and hyperbolic functions that targets correct rounding for the rounding to nearest in double precision. The quick and the accurate phases target a relative error less than 2^{-66} and 2^{-150}, respectively [104, 123]. We choose $p = 10$ which corresponds to 805 rows in the trigonometric table and 356 rows in the hyperbolic table.

In order to ease comparisons, we consider only the number of memory accesses required by the second range reduction and the number of floating-point operations (FLOPs) involved in the reconstruction step, and table sizes. We will consider that expansion algorithms are used whenever high accuracy is required as it is the case in the correctly rounded library CR-Libm [34]. Let us recall that an expansion of size n consists in an unevaluated sum of n non-overlapping floating-point numbers that represents a given number with a larger precision than the one available in hardware [152]. We will take Table 2.10 extracted from [34, § 2.3, 101, § 3.2, 102] as the reference costs for those algorithms. The notation E_n stands for an expansion of size n in double precision, so that, with this notation, E_1 represents a regular binary64 word.

Table 2.10 – Costs of Addition and Multiplication of Expansions.

Operation	Floating-point operations	
	Without FMA	With FMA
$E_2 = E_1 + E_2$	11	
$E_2 = E_2 + E_2$	12	
$E_3 = E_1 + E_3$	16	
$E_3 = E_3 + E_3$	27	
$E_2 = E_1 \times E_2$	20	6
$E_2 = E_2 \times E_2$	26	11
$E_3 = E_1 \times E_3$	47	19
$E_3 = E_3 \times E_3$	107	43

2.6.1 *Costs for the trigonometric functions*

Tang's solution

In order to reach an accuracy of 66 bits, Tang's solution requires access to tabulated values S_h and C_h that are stored as expansions of size 2. These values need to be multiplied by the two results of the polynomial evaluations, which can be considered stored as expansion of size 2 as well. The total cost of the quick phase becomes: 4 double-precision memory accesses, 2 multiplications $E_2 \times E_2$, and 1 addition $E_2 + E_2$, that is, 34 FLOPs (64 without FMA).

In case the quick phase failed to return correct rounding, the accurate phase is launched. This phase requires access to 2 extra tabulated values to represent S_h and C_h as expansions of size 3. Those values need to be multiplied by the two results of the polynomial evaluations, which can be considered stored as expansion of size 3 as well. The total cost of the accurate phase becomes: 2 extra memory accesses, 2 multiplications $E_3 \times E_3$, and 1 addition $E_3 + E_3$. This corresponds to 6 memory accesses, 113 FLOPs (241 without FMA), and a 38 640 byte table.

Gal's solution

Using Gal's method, the corrective terms allow around 63 bits of accuracy, and Stehlé and Zimmermann's improvement allows to reach 74 bits. By considering Stehlé's approach, only one double-precision number is required for S_h, C_h and the corrective term, which fits on about 20 bits, to reach an accuracy of 66 bits. Again, S_h and C_h need to be multiplied by the two results of the polynomial evaluations, which can be considered stored as expansions of size 2. Thus the quick phase requires $2 + 1$ double-precision memory accesses, 1 addition $E_2 = E_1 + E_2$ for the corrective term, 2 multiplications $E_1 \times E_2$, and 1 addition $E_2 + E_2$. The cost of the quick phase with this table becomes 3 memory accesses and 35 FLOPs (63 without FMA).

To reach the 150-bit accuracy required by the accurate phase, it is necessary to get 2 extra floating-point numbers for S_h and C_h, which

are not exact. The corrective term is then incorporated in the final result using an addition $E_3 = E_1 + E_3$. The remaining operations need to be done using size-3 expansions. The total cost for the accurate phase becomes: 4 extra memory accesses, 2 multiplications $E_3 \times E_3$, 1 addition $E_1 + E_3$, and 1 addition $E_3 + E_3$, that is, 7 memory accesses, 129 FLOPs (257 without FMA), and a 45 080-byte table.

Exact lookup tables solution

With our solution, as shown in Table 2.7, at most 39 bits are required to store S_h and C_h, that is, only one floating-point number per entry. Our corrective terms are inexact, so that the cost of the quick phase is the same as Gal's approach in Section 2.6.1, except that the addition for the corrective term is an addition $E_2 = E_2 + E_2$, which accounts for 1 extra FLOP and 1 extra memory access. Hence, the quick phase for the exact lookup table solution is 36 FLOPs (64 without FMA), which is as much as Tang's solution when no FMA is available.

However, for the accurate phase, values S_h and C_h that were accessed during the quick phase are exact, and do not require any extra memory access. The corrective term is stored as an expansion of size 3 and it requires 1 extra memory access to reach the 150 bits of accuracy. The addition for the corrective term is performed using an addition with expansions of size 3. Multiplications correspond to $E_3 = E_1 \times E_3$ as the results of the polynomial evaluations can be considered as expansions of size 3. The final addition is done using an $E_3 = E_3 + E_3$ operation. That is, the total cost of this step becomes 5 memory accesses and 92 FLOPs (148 without FMA), for a 32 200 byte table.

Comparison results

Table 2.11 synthesizes the estimated costs for those three range reduction algorithms based on tabulated values. This table reports the number of memory accesses and FLOPs for the quick and accurate phases, together with the size in bytes of each row of the precomputed table. First, it can be seen that the proposed table-based range reduction requires less memory per table row than other solutions. Tang's method requires 48 bytes per row, Gal's method 56 bytes and our exact lookup table 40 bytes. This is an improvement in memory usage of $\approx$ 17% and $\approx$ 29% over Tang's and Gal's methods, respectively. Second, regarding the number of operations, our solution requires two more FLOPs than Tang's solution for the quick phase (0 without FMA), and one more FLOP than Gal's solution. This represents an overhead of $\approx$ 6% (0% without FMA) and $\approx$ 2%, respectively. This small downside comes with at least three advantages:

1. The accurate phase shows greater gains. Indeed, for this phase, there is an improvement in favor of our approach of $\approx$ 19% (39% without FMA) and $\approx$ 29% (42% without FMA) over Tang's and Gal's methods, respectively.

Table 2.11 – Computation and Memory Costs for Three Table-Based Range Reductions for Trigonometric and Hyperbolic Functions. The number of memory accesses and the number of floating-point operations (without FMA in parentheses) are reported.

Method	Table Loads		Flops for the reconstruction (without FMA)				Size (Bytes/row)
			Trigonometric		Hyperbolic		
	Quick	Accurate	Quick	Accurate	Quick	Accurate	
Tang	4	6	**34** (64)	113 (241)	102 (192)	339 (723)	48
Gal	**3**	7	35 **(63)**	129 (257)	**93 (179)**	355 (739)	56
Proposed	4	**5**	36 (64)	**92 (148)**	94 (180)	**270 (510)**	**40**

2. Our method reduces the number of memory accesses during the accurate phase. Indeed, the number of accesses is reduced from 7 to 5 compared to Gal's approach. This is an improvement of $\approx$ 29%, which translates to $\approx$ 17% compared to Tang's solution.

3. Error bounds might be easier to determine, as half the terms in the reconstruction are exact. Following this, the maximum acceptable error for polynomials might be loosened and lead to polynomial approximations of lesser degree, which would improve the overall performance.

2.6.2 *Costs for the hyperbolic functions*

The costs for memory accesses and table sizes for the hyperbolic functions are the same as the ones for the trigonometric functions. Indeed, although the exact lookup tables are smaller for the hyperbolic functions because $\ln(2)/2 \approx 0.347$ is less than $\pi/4 \approx 0.785$, they both contain as many words per row. We recall the reconstruction step for the hyperbolic functions:

$$\begin{aligned}\sinh(y) = {} & \left(2^{n-1} - 2^{-n-1}\right) \cdot \\ & \quad \left(C_h \cdot P_C\left(x_\ell\right) + S_h \cdot P_S\left(x_\ell\right)\right) \\ & \pm \left(2^{n-1} + 2^{-n-1}\right) \cdot \\ & \quad \left(S_h \cdot P_C\left(x_\ell\right) + C_h \cdot P_S\left(x_\ell\right)\right).\end{aligned} \tag{2.13}$$

If we do not count the computation of the terms $\left(2^{n-1} \pm 2^{-n-1}\right)$, there are 6 multiplications, and 3 additions. Indeed these terms can be neglected in the cost estimations as they are easy to generate using shifts and adds in integer arithmetic.

Tang's and Gal's solutions

On the first hand, let us consider Tang's solution: Overall, the table requires 17 088 bytes. For the quick phase, we must use E_2 expansions for 6 multiplications and 3 additions, as we have seen in Equation (2.13). These extended-precisions operations account for 102 FLOPs (192 without FMA). During the accurate phase, all computations are done using E_3 expansions, which require a total of 339 FLOPs (723 without FMA).

On the other hand, using Gal's method, the table takes 19 936 bytes and one addition $E_2 = E_1 + E_2$ for $x_\ell - \mathrm{corr}$ is added to the cost computations. The quick phase accounts now for 4 multiplications $E_1 \times E_2$ between tabulated values and polynomial results, 2 multiplications $E_2 \times E_2$ for the remaining terms, 1 addition $E_2 = E_1 + E_2$, and 3 additions $E_2 = E_2 + E_2$. Hence, the cost of the quick phase with this table is 93 FLOPs (179 without FMA). For the accurate phase, all operations need to be done using size-3 expansions, except for the addition of the corrective term $E_3 = E_1 + E_3$. The total cost for the

accurate phase becomes: 355 floating-point operations (739 without FMA).

Our exact lookup tables solution

Again, as for the trigonometric functions, the cost of the quick phase is the same as Gal's approach plus one extra memory access and one extra FLOP for the addition of the inexact corrective term, which is 94 FLOPs (180 without FMA). However, the table is smaller as it takes only 14 240 bytes.

For the accurate phase, as it was the case for the trigonometric functions, values S_h and C_h are exact and do not require any extra memory access. Only the corrective term is stored as an expansion of size 3 and requires 1 extra memory accesses in order to reach an accuracy of 150 bits. Hence, the accurate phase corresponds to 4 multiplications $E_1 \times E_3$, 2 multiplications $E_3 \times E_3$, and 4 additions $E_3 + E_3$. Thus the total cost of this step is 270 floating-point operations (510 without FMA).

Comparison results

As in Section 2.6.1, we compare the three methods using synthesized results from Table 2.11. Regarding memory usage, we have quantitatively the same benefits as for the trigonometric functions. Regarding the number of floating-point operations, our solution has the same small overhead of one FLOP for the quick phase compared to Gal's solution, but it requires 8% (6% without FMA) less FLOPs than Tang's solution. For the accurate phase, there is an improvement in favor of our approach of $\approx$ 20% (29% without FMA) and $\approx$ 24% (31% without FMA) over Tang and Gal, respectively. This is mainly due to the huge cost of multiplications $E_3 \times E_3$ that our method avoids in 4 out of 6 multiplications.

2.7 EXAMPLE ON THE TRIGONOMETRIC SINE

We propose to illustrate the use of the proposed error-free tabulated values compared to Tang's solution, which is probably the most pervasive, using Table 2.3 and Table 2.12. We consider the evaluation of the sine function for the input value $y = 10$ when targeting 8 bits of accuracy. When necessary, storage precision and intermediate computations will be done on 16 bits of precision in order to emulate 8-bit expansions of size 2, like what is commonly used for the quick phase. For clarity, bit fields will be represented in radix 2 (suffix "$_2$"). First, the reduced argument x is computed on 16 bits:

$$\begin{aligned} q &= \lfloor 10 \cdot 2/\pi \rceil = 6 \\ x &= 10 - q \cdot \pi/2 \\ &= 1.001001101000001_2 \cdot 2^{-1} \cdot (1 + \varepsilon_{-16}) \\ &\approx 0.575225830078125. \end{aligned}$$

Table 2.12 – Tang's Table for $p = 4$ With 16-Bit Precision.

Index	S_h	C_h
0	`0x0.0000p+0`	`0x1.0000p+0`
1	`0x1.ffaap-5`	`0x1.ff00p-1`
2	`0x1.feaap-4`	`0x1.fc02p-1`
3	`0x1.7dc2p-3`	`0x1.f706p-1`
4	`0x1.faaep-3`	`0x1.f016p-1`
5	`0x1.3ad2p-2`	`0x1.e734p-1`
6	`0x1.7710p-2`	`0x1.dc6cp-1`
7	`0x1.b1d8p-2`	`0x1.cfc6p-1`
8	`0x1.eaeep-2`	`0x1.c152p-1`
9	`0x1.110ep-1`	`0x1.b11ep-1`
10	`0x1.2b92p-1`	`0x1.9f36p-1`
11	`0x1.44ecp-1`	`0x1.8bb2p-1`
12	`0x1.5d00p-1`	`0x1.76a0p-1`
13	`0x1.73b8p-1`	`0x1.6018p-1`

As explained earlier, x is essentially inexact and has to be rounded (it is rounded to 16 bits in this case). By construction, x belongs to the interval $[-\pi/4, \pi/4]$, but in practice, properties of symmetry are used so that it finally lies in $[0, \pi/4]$. We have $q = 6$, which entails $q \equiv 2 \mod 4$. This means that an additional rotation by π radians was made during the first range reduction, so that the sign changed and we now have $\sin(10) = -\sin(x)$.

Now let us compute x_h and x_ℓ, such that $x = x_h + x_\ell$ with $x_h = i \cdot 2^{-4}, i \in [0, 13]$ and $|x_\ell| \leqslant 2^{-5}$. This leads to

$$\begin{aligned} x_h &= 9 \cdot 2^{-4} \quad \text{and} \\ x_\ell &= x - x_h \\ &= 1.10100001_2 \cdot 2^{-7} \cdot (1 + \varepsilon_{-16}). \end{aligned}$$

For row index $i = 9$, Table 2.12 gives us

$$\begin{aligned} S_h &= 1.000100010000111_2 \cdot 2^{-1} \cdot (1 + \varepsilon_{-16}) \\ C_h &= 1.101100010001111_2 \cdot 2^{-1} \cdot (1 + \varepsilon_{-16}), \end{aligned}$$

while Table 2.3 gives us

$$\begin{aligned} S_h &= 2880 \\ C_h &= 4715 \\ \text{corr} &= -1.110100000010101_2 \cdot 2^{-7} \cdot (1 + \varepsilon_{-16}) \end{aligned}$$

with $k = 5525$ as mentioned in Section 2.5.1. Now is when the different table-based methods diverge. They will be analyzed separately.

2.7.1 *Tang's table lookup method*

Using Tang's table, the evaluation of two polynomial approximations to $\sin(x)$ and $\cos(x)$ is performed with an output precision of 12 bits at input $x_\ell = 1.10100001_2 \cdot 2^{-7}$:

$$\begin{aligned} \sin(x_\ell) &\approx 1.10100001_2 \cdot 2^{-7} \cdot (1+\varepsilon_{-12}) \\ \cos(x_\ell) &\approx 1.0_2 \cdot (1+\varepsilon_{-12}). \end{aligned} \tag{2.14}$$

The final result is then computed on 8 bits using tabulated values S_h and C_h with the results from Equation (2.14):

$$\begin{aligned} \sin(10) &= -\sin(x) \\ &\approx -S_h \cdot 1.0_2 - C_h \cdot 1.10100001_2 \cdot 2^{-7} \\ &\approx -1.0001011_2 \cdot 2^{-1} \cdot (1+\varepsilon_{-8}) \\ &\approx -0.54296875. \end{aligned}$$

2.7.2 *Exact table lookup method*

With the proposed solution, the evaluation of polynomial approximations to $\sin(x)/k$ and $\cos(x)/k$ are performed with an output precision of 12 bits at input $x_\ell - \mathrm{corr} \approx 1.10111000100101_2 \cdot 2^{-6}$:

$$\begin{aligned} \sin(x_\ell - \mathrm{corr})/k &\approx 1.01000110101_2 \cdot 2^{-18} \\ \cos(x_\ell - \mathrm{corr})/k &\approx 1.011110111_2 \cdot 2^{-13}. \end{aligned} \tag{2.15}$$

Finally, the result is reconstructed as follows, using the error-free tabulated values S_h and C_h from Table 2.3 and the approximations from Equation (2.15):

$$\begin{aligned} \sin(10) &= -\sin(x) \\ &= -S_h \cdot \cos(x_\ell - \mathrm{corr})/k - C_h \cdot \sin(x_\ell - \mathrm{corr})/k \\ &\approx -1.0001011_2 \cdot 2^{-1} \\ &\approx -0.54296875. \end{aligned}$$

2.7.3 *Comparison between both methods*

One can see that for the exact lookup table method, the error is quickly concentrated into the reduced argument x_ℓ, then into $x_\ell - \mathrm{corr}$ and finally into the polynomial approximations. The main advantage of this method relies on the fact that S_h and C_h embed no rounding error at all, which allows for an easier, faster, and more elegant reconstruction step.

2.8 CONCLUSIONS AND PERSPECTIVES

In this chapter, we have presented a new approach to address table-based range reductions in the evaluation process of elementary functions. We have shown that this method allows for simpler and faster evaluation of these functions. For the trigonometric and hyperbolic functions, we used Pythagorean triples to build exact lookup tables that concentrate the error into the polynomial evaluations. Compared to other solutions, our exact lookup tables eliminate the rounding errors on certain tabulated values, and transfer these errors into the remaining reduced argument, which generally already contains rounding error. When targeting correct rounding in double precision, we have shown that our method allows to reduce the table sizes, the number of memory accesses and the number of floating-point operations involved in the reconstruction step by up to 29%, 29% and 42%, respectively, compared to Tang's and Gal's tables. This benefit could be even higher for greater targeted accuracies since all tabulated values except the corrective terms are exactly stored and do not depend on the targeted accuracy. Therefore we encourage integration and testing of our exact lookup tables into the efficient implementations of the trigonometric functions in the medium-precision range detailed in [85].

Finally, in this chapter we have only focused on the trigonometric and hyperbolic functions, as they both benefit from the use of Pythagorean triples. However, the concept remains valid and worthwhile for other functions, provided at least two terms need to be tabulated per row and that exact values can be found for such terms. Hence it may be interesting to investigate whether and how other elementary functions could benefit from these exact lookup tables.

3

VECTORIZING POLYNOMIAL SCHEMES

> "Multitasking: A clever method of simultaneously slowing down the multitude of computer programs that insist on running too fast."
>
> Software Terms,
> https://www.gnu.org/fun/jokes/software.terms.html

As we have seen, the evaluation of elementary functions often makes use of polynomial approximations. Their theoretical performance is closely linked to the order in which multiplications and additions are performed. Such an order is commonly referred to as an *evaluation scheme*. In this chapter, we intend to show that the widespread Horner scheme is in fact rarely the most efficient. Other evaluation schemes can better take advantage of modern computer architectures by reducing data dependencies. Vectorization is a key factor to improve throughput performance for large, independent computations [138]. Hence, focusing on single-instruction multiple-data vector architectures (SIMD), we benchmarked three classic schemes as well as thousands of automatically generated ones, which we vectorized using auto-vectorizing capabilities of the GNU Compiler Collection (GCC). From our various measurements, we confirmed the important impact on performance of the chosen scheme as well as the input interval for the evaluation. These conclusions are used in Chapter 4, which targets the efficiency of automatically-generated source code.

3.1 INTRODUCTION

Software implementations of elementary functions often use one or more polynomial evaluations of reasonable degree, on a small input interval [123, § 11.4]. The evaluation of those polynomials, i.e. the order in which the arithmetical operations are executed, is usually statically decided at the time the source code is written. This order is called a *polynomial evaluation scheme*. They may differ for custom implementations for specific architectures, but generic codes often use a single scheme for several architectures.[1]

Horner's scheme, or sometimes Horner's rule, is one of the most commonly used schemes [90, §4.6.4]. It is often chosen for its simplicity of implementation, its good accuracy properties, and because

[1] See for instance the GNU `libm` implementations for generic architectures.

it minimizes operation count, hence having good potential throughput [15]. Figure 3.1 gives a visual representation of this scheme, which shows that it is essentially sequential. Therefore its latency on architectures benefiting from unbounded parallelism is theoretically higher than the latency of more parallel schemes such as Estrin's [124]. Regardless of potential precision loss, one may want to use a more parallel scheme to reduce the latency or increase the throughput. However, manually selecting a fast scheme on a given hardware can be a very long and complex task as the polynomial degree increases.

Our objective is to automatically generate polynomial schemes that benefit most from features of modern architectures. More particularly, we are interested in leveraging Instruction Level Parallelism (ILP) and data parallelism through the Single Instruction Multiple Data (SIMD) execution model. In this chapter, the throughput performances of different polynomial schemes are studied on an Intel Haswell architecture[1]. We present two useful results:

1. High-ILP schemes can have both better latency *and* throughput than more sequential schemes like Horner, which goes against the usual recommendations [123, § 11.5].

2. We observe that the input interval for a given polynomial can be a key factor for the choice of a fast evaluation scheme, especially when this interval is around zero.

The chapter is organized as follows: Section 3.2 presents the classic polynomial schemes that are studied later on. Section 3.3 analyzes the measured throughputs of several schemes on an Intel Haswell CPU. Finally, Section 3.4 gives perspectives about efficient code generation for polynomial schemes with respect to vectorized implementations and ILP-leveraging environments.

3.2 REMINDERS ABOUT POLYNOMIAL EVALUATION

Let $P \in \mathbb{F}_n[X]$ be a degree-n polynomial with nonzero representable floating-point coefficients. Suppose that we want to evaluate this polynomial at a certain point x in the same floating-point format, using a Simplified, Not Pipelined and Infinitely Parallel Architecture (SNPIP) whose addition and multiplication latencies are respectively 3 and 5 cycles,[2], and that has no FMA. We do *not* focus on evaluation schemes with coefficient adaptation. The main reason of such a restriction is that coefficient adaptation can change polynomial coefficients and can lead to catastrophic cancellations near the roots of the polynomial. The interested reader can find out more about coefficient adaptation in [62, 135]. In our case, the number of additions remains constant and increasing instruction parallelism can only increase the number of multiplications [144].

[1] See `https://software.intel.com/haswell`.

[2] Latencies for Haswell architectures [79].

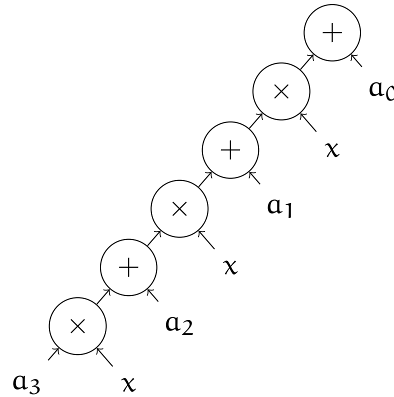


Figure 3.1 – Horner's scheme for a degree-3 polynomial.

One of the most common schemes is probably Horner's, which is often taught as a general and elegant method to evaluate a polynomial [90, §4.6.4]. It allows to evaluate $P = \sum_{i=0}^{n} a_i X^i$ in n additions and n multiplications. For instance, for $n = 7$, one has:

$$P(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (a_3 + x \cdot (a_4 + x \cdot (a_5 + x \cdot (a_6 + x \cdot a_7)))))). \tag{3.1}$$

Its interest is threefold:

1. It minimizes the number of multiplications that are needed;
2. There is no need to store intermediate results, which accounts for a lower pressure put on hardware registers and thus potential reductions of register spilling [90, p. 486];
3. In the context of elementary function evaluation, it is known to be numerically stable [14, ch.9].

Therefore, when the selection criterion is accuracy, Horner can be seen as a "good" scheme. But, as seen before, its latency on an architecture like SNPIP is higher than many schemes due to its intrinsic sequentiality. For example, the second-order Horner's scheme, which we denote Horner-2, is parenthesized this way for $n = 7$:

$$P(x) = \left(a_0 + x^2 \cdot \left(a_2 + x^2 \cdot \left(a_4 + x^2 \cdot a_6\right)\right)\right) + x \cdot \left(a_1 + x^2 \cdot \left(a_3 + x^2 \cdot \left(a_5 + x^2 \cdot a_7\right)\right)\right). \tag{3.2}$$

This scheme shows some ILP, as illustrated in Figure 3.2: Odd- and even-degree monomials can be evaluated in parallel, at the cost of an additional multiplication. Its latency is thus asymptotically divided by two compared to Horner's rule [90, p. 488].

Estrin's scheme, illustrated in Figure 3.3, shows even more ILP by splitting the evaluation tree according to powers of x of the form 2^i [61]. For example, still for $n = 7$, Estrin's scheme is:

$$P(x) = \left((a_0 + a_1 \cdot x) + x^2 \cdot (a_2 + a_3 \cdot x)\right) + x^4 \left((a_4 + a_5 \cdot x) + x^2 (a_6 + a_7 \cdot x)\right). \tag{3.3}$$

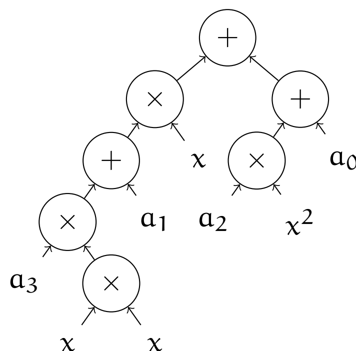


Figure 3.2 – Second-order Horner's scheme for a degree-3 polynomial.

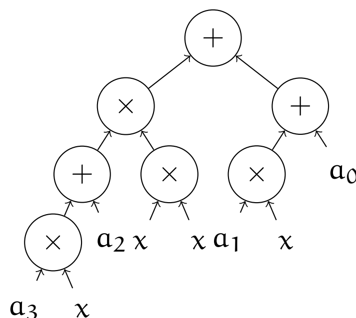


Figure 3.3 – Estrin's scheme for a degree-3 polynomial.

In the case of a degree $n = 2^k - 1$ polynomial, Estrin's evaluation tree is well-balanced, but a bit less in the general case. Its latency on unbounded parallelism is in $O(\log n)$.

Using the SNPIP architecture, for $n = 7$ and with Equations (3.1), (3.2) and (3.3), we can theoretically expect the latencies given in Table 3.1 for all the aforementioned evaluation schemes. We can see that the choice of an evaluation scheme has a direct impact on its efficiency. Therefore it can be important to choose the right evaluation scheme when efficiency is required. However, the combinatorics of the number of existing schemes for a degree-n polynomial is incredibly huge, which makes exhaustive search impossible as soon as $n \geqslant 6$. Fortunately, heuristics exist to allow non-exhaustive search to higher degrees up to $n \approx 12$ [144, ch.6].

Other schemes as well as the asymptotic optimality problem on bounded or unbounded parallelism have been studied in [113, 124, 125]. Here, in the context of elementary function evaluation, and especially their vectorization, we performed various performance measurements on scalar and autovectorized classic schemes as well as on many generated schemes, that are presented in Section 3.3.

3.3 BENCHMARKING POLYNOMIAL EVALUATION SCHEMES

In this section, benchmark results are analyzed for different evaluation schemes that were compiled with the GNU Compiler Collection

Table 3.1 – Theoretical latencies for common schemes on a simplified Haswell architecture.

Scheme	Latency (cycles)	
	without FMA	with FMA
Horner	56	35
Horner-2	37	25
Estrin	24	15

(GCC) 5.2, `-O2` optimization flag enabled, for an Intel Haswell CPU featuring the AVX2 and FMA instruction set extensions. These extensions can operate on 16 registers of 256 bits, either on a single word or on multiple words contained in the same register. The Haswell architecture is deeply pipelined, and has one floating-point adder and two multipliers/FMA, which allows for a throughput of four instructions per cycle [66, § 10]. Therefore it is able to leverage ILP on independent multiplications and independent additions/multiplications, although this is clearly bounded parallelism. GCC 5.2 is able to generate vector instructions *via* an autovectorization pass, which we used to produce vectorized binary code using scalar C source code.

The reciprocal throughput is a common metric for benchmarks. It corresponds to the mean number of cycles between two independent results: the lower, the better. In the following, we implicitly convert this metric by dealing with *throughput*: the higher, the better. Unless otherwise stated, this is the metric we use in all our benchmarks. For a given degree, it *a priori* depends upon the chosen scheme and the architecture, for which we alternate between enabling and disabling the generation of FMA instructions. On the targeted Haswell architecture, as long as no operation involves or produces subnormal numbers, we observe that the more ILP a scheme exhibits, the better its throughput.

In Section 3.3.1, we perform and analyze throughput benchmarks of the three aforementioned "classic" schemes, making the polynomial degree vary from 3 to 32, as well as enabling or disabling compiler autovectorization and FMA instruction generation. We run the same benchmarks for both the binary32 and the binary64 precisions, which proves to be useful when trying to explain huge performance drops in the binary32 precision. Then, in Section 3.3.2, we run latency micro-benchmarks to gain confidence in our hypothesis that the interval on which a polynomial is designed to be evaluated can have an important impact on performance. Finally, in Section 3.3.3, we perform several benchmarks on Horner, Horner-2 and Estrin's schemes against all schemes of theoretical lowest latency on our simplified architecture for a degree-12 polynomial, and for various evaluation intervals. They confirm that the choice of an efficient polynomial evaluation scheme can greatly vary with the evaluation interval. They also show that on architectures like Haswell, when no subnormal number

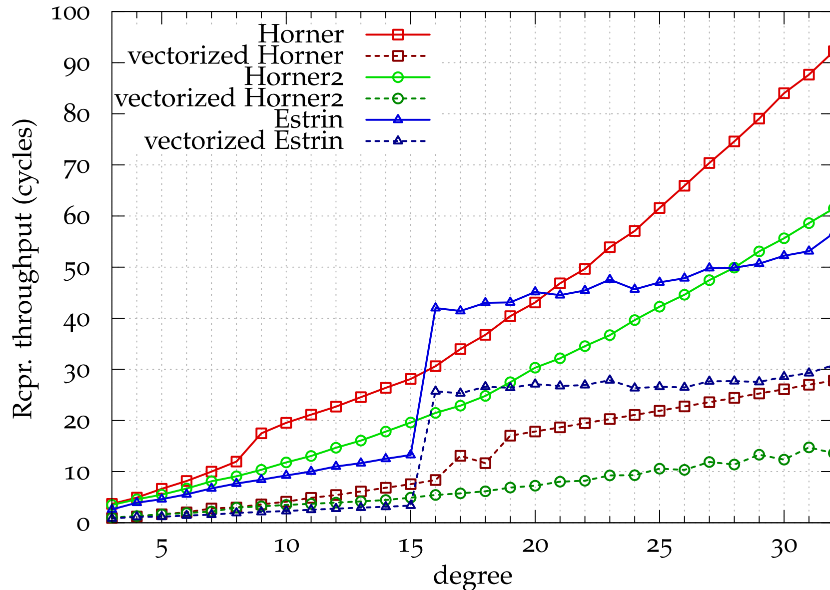


Figure 3.4 – Means of reciprocal throughputs of three usual schemes in binary32 precision, without FMA.

is produced during the evaluation, the less the number of multiplications involved, the greater the throughput.

3.3.1 *Benchmarking classic schemes*

In this section, in order to better represent the context of elementary functions evaluation, we show benchmarks for polynomials whose degree ranges between 3 and 32, and we use the Taylor coefficients for the function $x \mapsto \log(1 + x)$ around 0, computed with Sollya 6.0 [26]. This corresponds to the following Sollya command:

```
> P = taylor(log1p(x), degree, 0);
```

which generates a Taylor approximation of degree `degree` in the working precision. Then the polynomial coefficients are rounded to the target precision, e.g. binary32, with the following command:

```
> P_binary32 = roundcoefficients(P, [|single...|]);
```

Several evaluations are benchmarked on large vectors of 2^{14} floating-point numbers uniformly distributed in the interval $[2^{-12}; 2^{-5}]$.

Figures 3.4 and 3.5 present the reciprocal throughputs measured for binary32 precision Horner, Horner-2 and Estrin schemes, with or without vectorization enabled and with or without FMA enabled. As expected, the higher the polynomial degree, the lower the throughput. However, starting from degree 16, we can see a huge performance loss for Estrin's scheme only. In Section 3.3.2, we explain this by the fact that the intermediate result x^{16}, computed by Estrin's scheme only, produces subnormal numbers in approximately 13% of calls. Apart from this problem, we can notice that, without FMA, Estrin has the best throughput until the degree 15: For this degree, it is more than

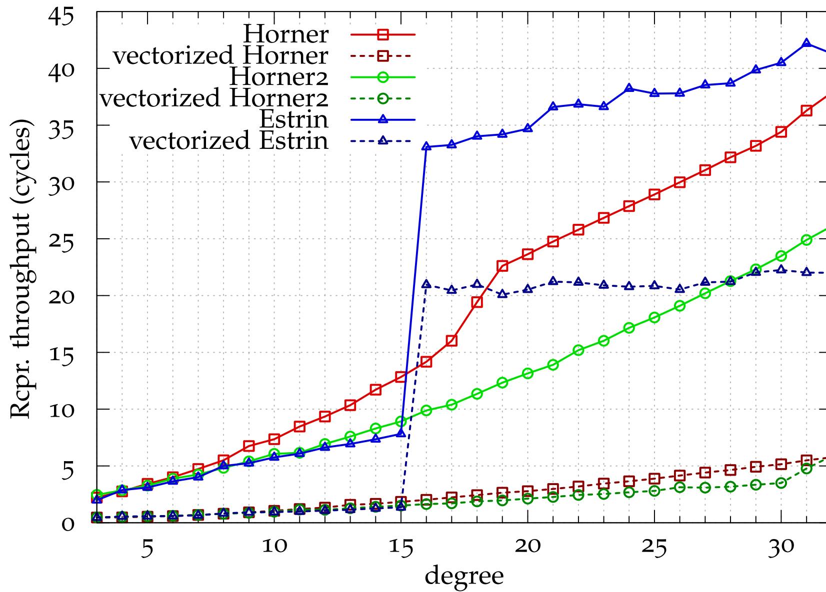


Figure 3.5 – Means of reciprocal throughputs of three usual schemes in binary32 precision, with FMA.

twice as good as Horner's scheme and 50% better than Horner-2's scheme, for both the scalar and vectorized versions. For the degree 16, when the subnormal production issue arises, Estrin's throughput is from 27 to 67% less than Horner's and from 49 to 79% less than Horner-2's, for the scalar and vectorized versions, respectively. With FMA enabled, the three schemes have comparable throughputs until degree 15, although the scalar version of Horner's scheme has the least throughput. Starting from degree 16, only Horner-2 is better than Horner, by 40%. Until degree 32, none of the three schemes seem to suffer from the limited number of registers. Indeed, GCC's scheduler manages to keep intermediate results in registers during each evaluation. This is an incentive for more use of highly parallel schemes in the context of elementary functions evaluation, for which the polynomial approximations are not likely to be of degrees higher than 32.

By replicating the same experiment but in binary64 precision, we do not observe the performance loss we had for Estrin's scheme in binary32 precision. This is clearly shown on Figures 3.6 and 3.7. This reinforces the hypothesis that the throughput loss was caused by the production of subnormal results. The throughput decrease is then almost linear with respect to the polynomial degree. Furthermore, Estrin's scheme has a greater throughput than Horner's starting from degree 10 in all cases: with or without FMA, with or without vectorization. We conclude that a lower latency does not always imply a throughput decrease, which can make schemes like second-order Horner or Estrin more interesting than Horner if performance is more important than precision. This conclusion may be a little counter-intuitive, since bounded parallelism can make us ex-

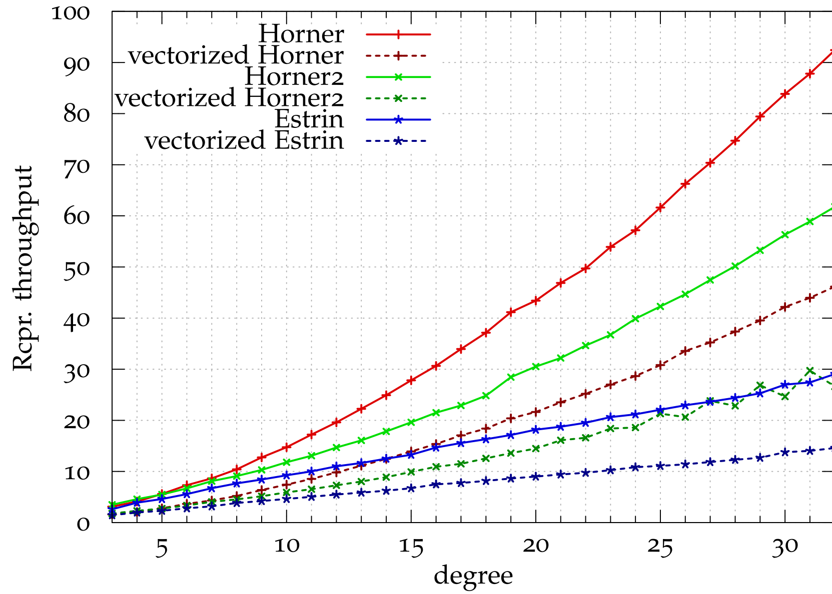


Figure 3.6 – Means of reciprocal throughputs of three usual schemes in binary64 precision, without FMA.

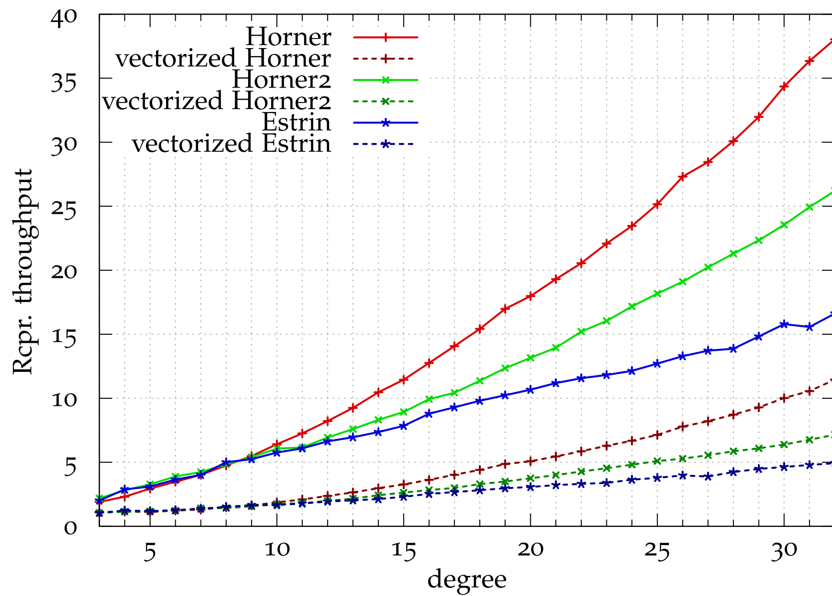


Figure 3.7 – Means of reciprocal throughputs of three usual schemes in binary64 precision, with FMA.

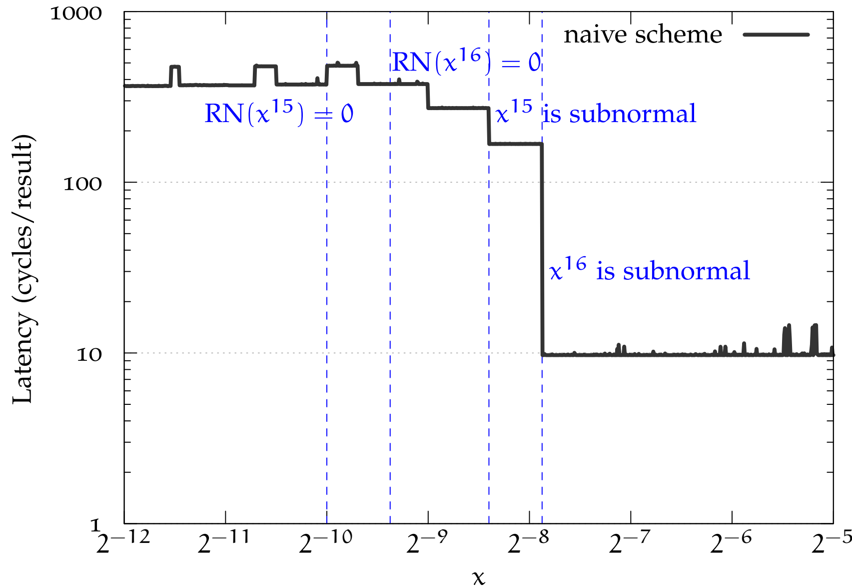


Figure 3.8 – Latencies of the computation of x^{16} in binary32 by naive exponentiations.

pect some kind of saturation point on arithmetic units for high ILP evaluation schemes.

3.3.2 *Influence of the evaluation interval*

In the following, we note $\mathrm{RN}(x)$ the rounding of a real number x to the nearest (ties to even) binary32 floating-point number. The polynomials used for the evaluation of elementary functions are usually precise enough on rather small intervals. Inputs that lie outside of these intervals are either treated separately, or reduced to one of them. This interval is generally a neighborhood of the point of approximation of the elementary function, which is often 0. It is the case for $\ln(1+x)$, $\exp(x)$, or $\sin(x)$. As a result, small magnitude inputs, if raised to certain powers, can rapidly produce subnormal numbers. For instance, for $x \in [2^{-12}; 2^{-5}]$, which was the interval used in the benchmarks in the previous section, x^{16} is a binary32 subnormal for about $(2^{-126/16} - 2^{-12})/2^{-5} \approx 13\%$ cases.

Figures 3.8 and 3.9 show the measured latency of two schemes of raising a binary32 floating-point number x in the aforementioned interval to the 16th power. The first one corresponds to a naive exponentiation composed of 15 multiplications, while the other one uses a binary or "divide-and-conquer" (D&C) approach. Their minimal latencies are a few cycles long, except when x is small enough so that x^{16} is a subnormal. Then a threshold can be observed: the measured latencies switch from a few cycles to more than 100. According to Agner Fog, these high cycle counts come from micro-coded special treatments for all operations whose operands are normalized floating-point numbers and whose result is a subnormal, as well as for multiplications with one normal and one subnormal operands [66,

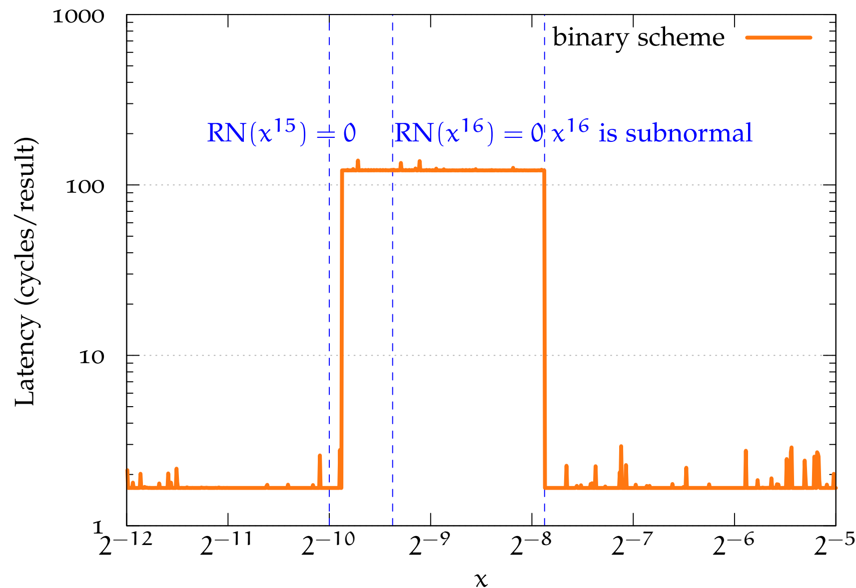


Figure 3.9 – Latencies of the computation of x^{16} in binary32 by binary exponentiations.

§ 10.9]. The naive evaluation scheme shows other thresholds, which correspond to monomials of lower degree that also produce subnormal numbers. The D&C approach gets back to having a latency of a few cycles *after* $\mathrm{RN}(x^{16}) = 0$. This may contradict Fog's manual [66, § 10.9], which states:

> "There is no penalty for overflow, underflow, infinity or not-a-number results."

This suggests that the micro-code gets executed only for a precise interval of outputs. Also, when $\mathrm{RN}(x^{15}) = 0$, i.e. $|x| \leqslant 2^{-10}$, a decreasing threshold can be observed for the naive scheme. This is interpreted by the fact that a multiplication by zero is not subject to this micro-code treatment penalty.

Therefore, on some architectures, the fact that an intermediate result produces a subnormal can make computation performances decrease a lot. This conclusion will justify some algorithmic choices that we make in Chapter 4, when designing a high-throughput vectorized logarithm. In the context of elementary function evaluation, this performance issue can also affect polynomial evaluations. They are often valid approximations on intervals of small magnitudes, therefore higher-degree monomials can quickly produce subnormals. When efficiency is required it is thus relevant, if not *necessary*, to ensure that, for a given evaluation scheme, the input interval will not produce subnormal numbers. This conclusion could allow for better automatic selection of polynomial evaluation schemes, which would then be efficient on their whole interval of possible inputs.

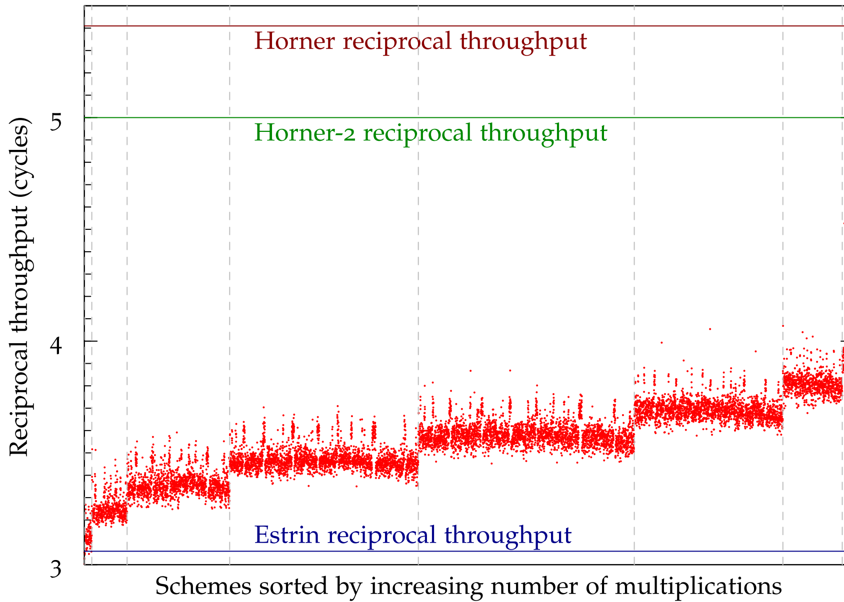


Figure 3.10 – Mean reciprocal throughputs of lowest-latency schemes of degree 12, sorted by increasing number of multiplications. x^n is normal for $n \in [1; 12]$.

3.3.3 *Towards more efficient evaluation schemes*

CGPE,[1] which stands for Code Generation for Polynomial Evaluation, is a software tool that can, among other features, compute low-latency polynomial evaluation schemes on unbounded parallelism for a given degree and architecture [120, 144]. Using our simplified SNPIP architecture defined in Section 3.2, we used CGPE to generate low-latency schemes of degree 12. This represents a total of 11 944 schemes, each having a latency of 26 cycles. The reciprocal throughputs of these schemes were measured for different inputs. Some of those inputs were chosen because they produce subnormal numbers when raised to a certain power. The schemes were compiled with FMA generation disabled and using GCC's autovectorizing capabilities. Figures 3.10, 3.11 and 3.12 show these 11 944 measurements as dots. Vertical lines split the set of polynomial evaluation schemes into subsets having the same number of multiplications. For example, evaluation schemes in the first subset (on the left) involve 17 multiplications, while in the last subset (on the right) they involve 25 multiplications.

In Figure 3.10, we note that all lowest-latency schemes have a greater throughput than Horner and Horner-2 schemes, but only about ten are at least as efficient as Estrin's scheme. These are the schemes that use the fewest multiplications. Quite rationally, thresholds indicating lower throughputs can be observed as the number of multiplications for a scheme increases, although they have the same theoretical latency on unbounded parallelism. Figure 3.11 confirms that Estrin's evaluation scheme can be a lot more efficient than most

[1] See `http://cgpe.gforge.inria.fr/`.

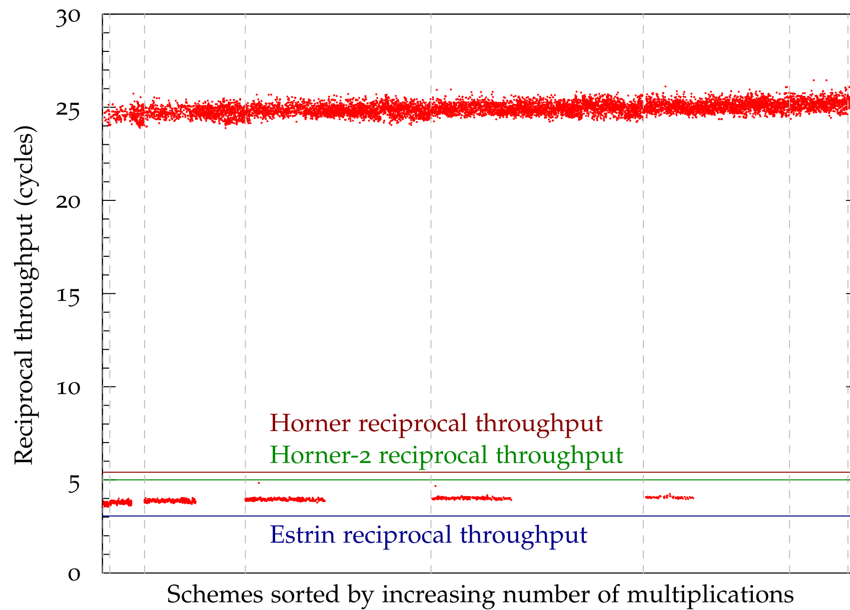


Figure 3.11 – Mean reciprocal throughputs of lowest-latency schemes of degree 12, sorted by increasing number of multiplications. x^n is subnormal when $n = 12$.

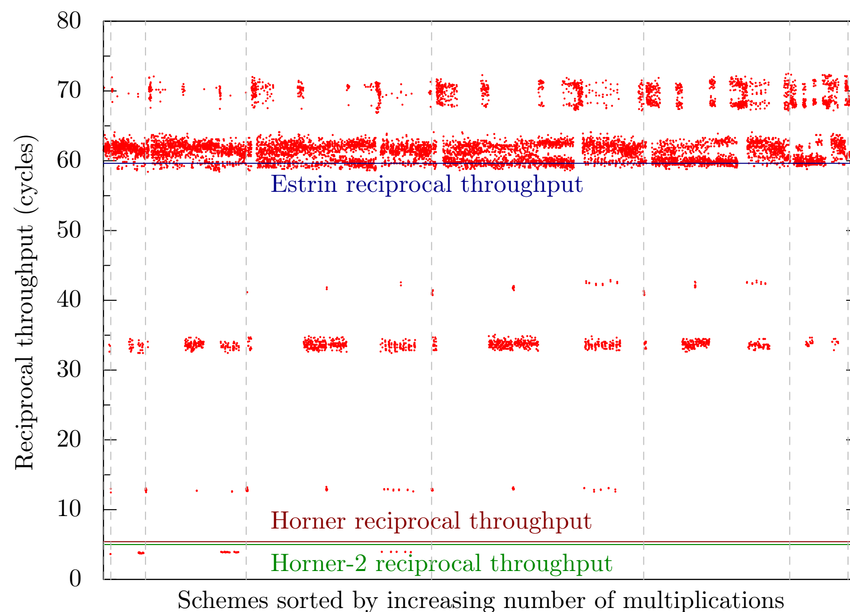


Figure 3.12 – Mean reciprocal throughputs of lowest-latency schemes of degree 12, sorted by increasing number of multiplications. x^n is subnormal when $n = 8$.

lowest-latency schemes, because it is able to produce fewer subnormal intermediate results, especially when the polynomial degree is not a power of two, like 12 here. On the other hand, Figure 3.12 shows that Estrin's scheme is about 12 times as less efficient as Horner and Horner-2 schemes when the production of subnormal results occurs starting from x^8. Although Horner and Horner-2 schemes are more efficient than most lowest-latency schemes, there are about 50 schemes that stand better than them, among those that involve few multiplications. This goes in favor of our conclusion drawn in Section 3.3.2, i.e. that although not trivial, it is important to ensure that the scheme we choose does not produce subnormal results on its evaluation interval if we want to use it on architectures that have huge performance penalties for such results.

A deeper study of the lowest-latency schemes generated by CGPE may allow us to spot common patterns, that would be reusable in the context of elementary functions approximations by univariate polynomials. Otherwise, autotuning by benchmarking each scheme on a given architecture could allow us to select a scheme that has a better throughput than the usual schemes. *A posteriori*, bounds on the accuracy of the final result could be computed with Gappa to verify if the selected evaluation scheme meets the requirements.

3.4 CONCLUSION AND PERSPECTIVES

We draw several conclusions about code generation for polynomial evaluations from the observations made in this chapter. In the context of elementary functions evaluation, we have shown that the production of subnormal results can have a huge impact on performance, which makes it a key factor when selecting an evaluation scheme that is efficient over all its evaluation interval. Finally, we have seen that highly parallel schemes, i.e. high ILP schemes, like Estrin or those generated by CGPE have often a better throughput than the more sequential schemes like Horner.

We believe that this study could be developed in three directions. First, a deeper analysis of the best schemes generated by CGPE may allow us to directly build efficient schemes without having to perform an expensive search. Second, CGPE could use new criteria to discard evaluation schemes that may produce unwanted kinds of intermediate or final results for a given input interval and a given set of coefficients. For example, smart interval arithmetic using the Gappa library could be used to implement such a heuristic. Finally, it might be interesting to study if an efficient evaluation scheme has a noticeable impact on the performance of elementary functions evaluations.

In the long term, an efficient code generator for polynomial evaluation in the context of elementary functions approximation would be desirable. Then it would be interesting to deal with performance/accuracy tradeoffs, as well as formal guarantees.

4

THE POWER OF META-IMPLEMENTATIONS

> "I've done this because, philosophically, I am sympathetic to your aim. I can tell you with no ego, this is my finest sword. If on your journey, you should encounter God, God will be cut."
>
> Hattori Hanzo, in *Kill Bill: Vol.1* by Quentin Tarantino.

We have seen several steps of the design of elementary functions implementations in the previous chapters. First, we developed a generator of exact lookup tables for trigonometric and hyperbolic functions (Chapter 2). Then, based on multiple benchmarks, we defined guidelines for the choice of efficient polynomial evaluation schemes. In order to generate efficient code, code generation frameworks should be able to leverage any source of performance enhancement, such as SIMD instruction sets extensions, e.g. SSE, AVX, or AVX-512, which provide support for vector instructions. Such instructions enable to treat packed inputs — currently up to 16 binary32 or 8 binary64 words for AVX-512 — in parallel, at the same announced latency and throughput as the equivalent scalar instruction. Hence, they allow for better sharing out of latency and multiplication of throughput compared to the same scalar instructions. In this chapter, we focus on automatic and efficient code generation for vectorized elementary functions. Such functions leverage the aforementioned SIMD capabilities to receive a short vector of inputs and return an output vector of same length. With the democratization of parallel computing units such as SIMD instruction sets or GPUs, this kind of functions have been gaining in popularity for efficient libm implementations [73, 100, 138].

This chapter presents a joint collaboration with Nicolas Brunie from Kalray.[1] He created and has been maintaining the Metalibm-lugdunum code generation framework, which has been described in Chapter 1 (p.28). In particular, we present:

- a classic range reduction in a branchless fashion so as to use at best recent micro-architecture features, like the fast hardware reciprocal instruction `rcp`, and to treat all inputs in the same flow;

[1] https://kalray.eu

- a detailed "meta-design" of faithfully-rounded implementations of the natural logarithm;
- a meta-implementation within the Metalibm-lugdunum framework, including contributions to the SSE/AVX and AVX2 backends;
- how our meta-implementation allows to achieve high throughput implementations for the binary32 and binary64 formats in a fully automated way.

4.1 INTRODUCTION

Nowadays, most modern micro-architectures embrace new hardware support for vector instructions in floating-point arithmetic, besides scalar instructions. They enable to treat packed inputs in a single instruction, thus potentially increasing the performance of floating-point programs. By taking into account these vector instructions during the design process, it is possible to achieve efficient implementations of vectorized mathematical functions. These are widely used in scientific programs manipulating floating-point computations, for which performance improvement is crucial [73].

Here, we deal with vectorized implementations of the natural logarithm function $\log(x)$ in floating-point arithmetic, with a particular focus on their automatic generation through the Metalibm framework [20, 22]. This framework enables to *describe* implementations of a function using a meta-language and to generate C source codes optimized for, e.g. different target architectures. For performance reasons, we target faithfully-rounded implementations, that is, with a maximum error of 1 ulp. Indeed, even if, the IEEE 754-2008 standard recommends the correct rounding for scalar elementary functions like $\log(x)$, no output accuracy is required nor recommended for functions with vector arguments [76]. Furthermore, faithful rounding is often sufficient in high-performance computing contexts, for example in physics [138].

We provide high throughput faithfully-rounded implementations of $\log(x)$, in the binary32 and binary64 formats, and with a guarantee on the output accuracy. We propose also a meta-implementation of this function in the Metalibm framework, enabling to generate various optimized binary32 versions. This work being parametrized by the precision, and code generation being based on Metalibm backends, this can easily be extended to any other precision. For performance reasons, we do not consider input arguments possibly requiring special treatments, e.g. NaNs, $\pm\infty$, or nonpositive numbers. However, we do consider subnormal inputs in the main flow. This results in branchless programs, which avoids an expensive fallback treatment for these inputs (like most of the vector libraries presented above do) at a small overhead.

This chapter is organized as follows: Section 4.2 details the range reduction that we use to achieve the targeted accuracy. Section 4.3

gives some guidelines on how to build accurate-enough programs. Then Section 4.4 provides some implementation details. Section 4.5 presents our meta-implementation within the Metalibm framework. Finally, Section 4.6 discusses some experimental results compared to existing implementations, and highlights advantages and disadvantages of meta-implementations, before a conclusion in Section 4.7.

4.2 ALGORITHM FOR $\log(x)$ WITH FAITHFUL ROUNDING

We consider here that the input x is neither a special input (NaN, $\pm\infty$, or $x \leqslant 0$), nor the particular input $x = 1$. All floating-point numbers manipulated, as well as the floating-point operations carried out, are in precision $p \geqslant 2$. This section presents the range reduction we use to implement the natural logarithm.

4.2.1 *Range reduction*

Let $x \neq 1$ be a positive floating-point number as defined in [76] with $s = 0$:

$$x = m \cdot 2^e, \tag{4.1}$$

where $m \in [1, 2 - 2^{1-p}]$ and $e \in [e_{min} \mathinner{..} e_{max}]$. Many evaluation schemes have already been proposed for the implementation of the natural logarithm. They are generally based on table lookups and/or polynomial evaluations [34, 68, 69, 143, 149, 163, 178–180]. Let r_i denote an approximation of $1/m$. Then, using $\log(x) = e \cdot \log(2) + \log(m)$, the logarithm function is computed as follows:

$$\log(x) = (e + \tau) \cdot \log(2) - \log(2^\tau \cdot r_i) + \log(1 + u),$$

where $u = r_i \cdot m - 1$, and $\tau = [m > \sqrt{2}]$. τ is introduced to avoid the catastrophic cancellation that may occur while computing $e \cdot \log(2) + \log(m)$ when $e = -1$ and $m \approx 2$ as in [143]. Note that this rewriting enables to avoid code branches, and to take advantage of the `rcp` instruction, which provides a fast reciprocal approximation, available on recent vector micro-architectures.

Furthermore when x is a subnormal floating-point number, a usual way consists in rescaling x in the normal range [34, 143]. Hence let x' be the rescaled floating-point number defined as:

$$x' = x \cdot 2^\lambda \tag{4.2}$$

where $\lambda \in \{0, \cdots, p - 1\}$ is the number of leading zeros in the binary representation of m. It follows that $x' = 2^{e'} \cdot m' \geqslant 2^{e_{min}}$ and

$$\log(x) = (e' + \tau - \lambda) \cdot \log(2) - \log(2^\tau \cdot r_i) + \log(1 + u), \tag{4.3}$$

with $m' \in [1, 2 - 2^{1-p}]$, represented exactly with only one precision-p floating-point number. The quantity r_i is thus an approximation of $1/m'$ obtained with the `rcp` instruction truncated on i bits, and

$u = r_i \cdot m' - 1$. In addition the value $-\log(2^\tau \cdot r_i)$ is retrieved from a lookup table using the $i > 0$ most significant fraction bits of the significand of r_i.

Note that instead of extracting m' and approximating $1/m'$, we could have preferred computing directly an approximation of $1/x'$. However when the result of $1/x'$ falls into the subnormal range, that is, $x' > 2^{e_{\max}-1}$, the `rcp` instruction flushes the result to 0. To overcome this issue, a rescaling similar to the one used here for handling subnormal numbers should have been performed: but it would have been as costly as the extraction of m', or even more.

Note also the way subnormal inputs are handled here. It is different from the one usually seen in vector function implementations, for which calling a fallback routine for these inputs is often preferred. However, treating subnormal inputs in the general flow has no great impact on performance as discussed in Section 4.6.3.

4.2.2 *How to determine* r_i*,* u*,* τ*, and* $-\log(2^\tau \cdot r_i)$*?*

Let us now detail the computation of r_i, u, τ, and $-\log(2^\tau \cdot r_i)$.

Determination of r_i

Let us assume an `rcp` instruction available, that returns an approximation of $1/m'$ with a relative error bounded by κ:

$$\mathtt{rcp}(m') = 1/m' \cdot (1 + \varepsilon_{\mathrm{rcp}}), \quad \text{with} \quad |\varepsilon_{\mathrm{rcp}}| \leqslant \kappa. \tag{4.4}$$

The quantity r_i is then computed by truncating $\mathtt{rcp}(m')$ on $i > 0$ fraction bits. In order to improve and to re-center error enclosure, we first add one half-ulp on i bits: thus we have

$$r_i = \mathtt{rcp}(m') \cdot (1 + \varepsilon_{r_i}), \quad \text{with} \quad |\varepsilon_{r_i}| \leqslant 2^{-i-1},$$

since $m', r_i > 0$. It follows that $r_i = 1/m' \cdot (1 + \varepsilon_{\mathrm{rcp}}) \cdot (1 + \varepsilon_{r_i})$. Hence:

$$r_i = 1/m' \cdot (1 + \varepsilon) \quad \text{with} \quad \varepsilon = \varepsilon_{\mathrm{rcp}} + \varepsilon_{r_i} + \varepsilon_{\mathrm{rcp}} \cdot \varepsilon_{r_i},$$

where

$$|\varepsilon| \leqslant \kappa + 2^{-i-1} + \kappa \cdot 2^{-i-1}. \tag{4.5}$$

Using the Intel Intrinsics Guide [79], we know that SSE/AVX instruction sets provide a binary32 `rcp` instruction with a maximum relative error less than $1.5 \cdot 2^{-12}$, that is, with $|\varepsilon_{\mathrm{rcp}}| \leqslant 1.5 \cdot 2^{-12}$. On more recent micro-architectures though, the AVX-512 instruction set provides also a binary64 `rcp` instruction with $|\varepsilon_{\mathrm{rcp}}| \leqslant 2^{-14}$. Note that on SSE/AVX-supporting micro-architectures, the binary32 instruction can also be used to approximate $1/m'$ for a binary64 input: m' must first be cast to the binary32 format, thus slightly increasing the approximation error: $|\varepsilon_{\mathrm{rcp}}| \leqslant 1.5 \cdot 2^{-12} + 2^{-24} + 1.5 \cdot 2^{-36}$.

Table 4.1 – Suitable Values for i (Satisfying Property 1) for Different Combinations of (e_{max}, κ).

e_{max}	κ	$i \in \cdots$
127	$1.5 \cdot 2^{-12}$	$[1, 10]$
1023	2^{-14}	$[1, 12]$
1023	$1.5 \cdot 2^{-12} + 2^{-24} + 1.5 \cdot 2^{-36}$	$[1, 10]$

Determination of u

Since $u = r_i \cdot m' - 1$, we deduce that $u = \varepsilon$ in (4.5). Therefore the quantity u in (4.3) is the error occurring when approximating $1/m'$ by r_i.

Property 1. *Let $x \neq 1$ be a positive floating-point number as in (4.1). The quantity u in (4.3) can be represented exactly with only one precision-p floating-point number as long as the parameter i is chosen so that:*

$$i \leqslant e_{max} - 2 \quad \text{and} \quad \kappa \cdot (2^i + 2^{-1}) < 2^{-1}. \tag{4.6}$$

Proof. Let M and R be the integer significands of m' and r_i, respectively. Since r_i is obtained after truncating $\mathtt{rcp}(m')$ on $i > 0$ fraction bits, its significand can be stored on at most $i + 1$ bits. Hence

$$m' = M \cdot 2^{1-p} \quad \text{and} \quad r_i = R \cdot 2^{\delta - i},$$

with $\delta \in \{-1, 0\}$. Using $u = r_i \cdot m' - 1$, we rewrite u as:

$$\begin{aligned} u &= R \cdot M \cdot 2^{\delta - i + 1 - p} - 1 \\ &= 2^{\delta - i + 1 - p} \cdot \left(R \cdot M - 2^{p - 1 + i - \delta}\right). \end{aligned}$$

Since $p \geqslant 2$ and $i > 0$, then if $i \leqslant e_{max} - 2$, we deduce that $\delta - i + 1 - p \in [e_{min} + 1 - p, 1 - p]$ is the exponent of a floating-point number. Moreover the value U defined as:

$$U = R \cdot M - 2^{p + i - 1 - \delta}$$

is an integer. Using (4.5), since $u = \varepsilon$, we deduce that:

$$|U| \leqslant \kappa \cdot 2^p \cdot (2^i + 2^{-1}) + 2^{p-1}.$$

Therefore if i is chosen so that:

$$\kappa \cdot (2^i + 2^{-1}) < 2^{-1},$$

then $|U| < 2^p$, and u is a precision-p floating-point number, which concludes the proof. □

From Property 1, we know that for certain values of i, the quantity u fits exactly in only one floating-point number in precision p. These values are given in Table 4.1, for different combinations (e_{max}, κ). As a consequence, in order to implement the sequence computing u, we

choose to use an fma, returning exactly u in a single instruction, and available on recent micro-architectures. If it had not been available, an alternative would have been to use multi-word arithmetic, like double-double arithmetic in [53, 101].

Determination of τ and $-\log(2^\tau \cdot r_i)$

The value $-\log(2^\tau \cdot r_i)$ is an approximation of $-\log(2^\tau/m')$, and it is retrieved from a lookup table, indexed only by the i most significant fraction bits of r_i, and not by τ. It follows that this table is built such that for the entries corresponding to $\tau = 0$, it contains approximations of $-\log(1/m')$, while for the others ($\tau = 1$), it contains approximations of $-\log(2/m')$.

As a consequence, each cell in the table must represent m' values for which τ is the same. In this sense, τ cannot be decided using m', but it must be decided using the i most significant fraction bits of r_i, as well, denoted by idx below. And since r_i is a floating-point number, τ is computed as follows:

$$\tau = \begin{cases} 1 & \text{if } \texttt{idx} \leqslant \lfloor (2/\sqrt{2} - 1) \cdot 2^i \rfloor, \\ 0 & \text{otherwise.} \end{cases} \tag{4.7}$$

This way, when $m' \approx 1$, we may have $\texttt{idx} = 0$ and $\tau = 1$, while τ should be 0. Hence in this particular case, the value of τ must be explicitly set to 0.

Notice that τ in (4.7) may not correspond exactly to the definition of τ in Section 4.2.1: but it is not an issue since we have a certain latitude for the value involved in the test. Usually $\sqrt{2}$ is chosen to re-center the fraction around 1, but other values around $\sqrt{2}$ do also work.

4.2.3 *How to ensure faithful rounding?*

Let r be an approximation of $\log(x)$, with $x \neq 1$ a positive floating-point number. For accuracy purpose, the value r is represented as the unevaluated sum of two floating-point numbers r_h and r_ℓ in precision p, that is

$$r = r_h + r_\ell \quad \text{with} \quad |r_\ell| < \tfrac{1}{2}\,\mathrm{ulp}(r_h) \leqslant 2^{-p} \cdot |r_h|\,. \tag{4.8}$$

Our goal is to compute r so that r_h is a faithful rounding of $\log(x)$, that is, either RD $(\log(x))$ or RU $(\log(x))$. Following [122], a sufficient condition is:

$$|r_h - \log(x)| < \mathrm{ulp}(\log(x)). \tag{4.9}$$

Using the triangular inequality, we have:

$$|r_h - \log(x)| \leqslant |(r_h + r_\ell) - \log(x)| + |r_\ell|\,.$$

Hence from (4.8) and assuming $\mathrm{ulp}(\log(x)) = \mathrm{ulp}(r_h)$, we deduce that if

$$|(r_h + r_\ell) - \log(x)| \leqslant \tfrac{1}{2}\,\mathrm{ulp}(r_h), \tag{4.10}$$

then (4.9) holds. When $\log(x) \in]2^{k-1} \cdot (2 - 2^{-p}), 2^k[$, with $k \in \mathbb{Z}$, we have $\mathrm{ulp}(\log(x)) = \frac{1}{2}\,\mathrm{ulp}(r_h)$: these cases are ignored at implementation time, and the accuracy of the result for these particular inputs is checked *a posteriori*.

Now let us distinguish three cases:

1. When $e' + \tau - \lambda = 0$, and $x' \in \mathcal{X}$ with

   $$\mathcal{X} = \left[\frac{1+\kappa}{1 + 2^{-i-1} - 2^{1-p}}, \frac{1-\kappa}{1 - 2^{-i-2}}\right],$$

   we can show that $\log(2^\tau \cdot r_i) = 0$, and $\log(x) = \log(1+u)$. In this case, since $|\log(x)| > 2^{-p}$, we have $|r_h| \geqslant 2^{-p}$ and $\mathrm{ulp}(r_h) \geqslant 2^{-2p+1}$. (See [143, Prop. 3] for details.)

2. When $e' + \tau - \lambda = 0$ and $x' \notin \mathcal{X}$:

   $$|\log(x)| > \min$$

   with $\min = \min(|\log(\inf(\mathcal{X}))|, |\log(\sup(\mathcal{X}))|)$. Then

   $$|r_h| \geqslant \min \quad \text{and} \quad \mathrm{ulp}(r_h) \geqslant \mathrm{ulp}(\min).$$

3. Otherwise, $e' + \tau - \lambda \neq 0$: thus we have $|\log(x)| > \log(\sqrt{2})$. Hence without loss of generality, we deduce that $|\log(x)| > 2^{-2}$, then $|r_h| \geqslant 2^{-2}$ and $\mathrm{ulp}(r_h) \geqslant 2^{-p-1}$.

It follows that if

$$|r - \log(x)| \leqslant \theta \text{ with } \theta = \begin{cases} 2^{-2p} & \text{in Case 1,} \\ \mathrm{ulp}(\min)/2 & \text{in Case 2,} \\ 2^{-p-2} & \text{in Case 3,} \end{cases} \tag{4.11}$$

then (4.10) holds, and r_h is a faithful rounding of $\log(x)$. For example, for $(p, i) = (24, 7)$, we have:

$$\min \approx 3.5 \cdot 10^{-3} \quad \text{and} \quad \mathrm{ulp}(\min) = 2^{-32}.$$

Distinguishing these three cases will help us in certifying the error entailed by the evaluation of the program in finite precision as explained in Section 4.4.2.

4.3 GUIDELINES TO BUILD ACCURATE ENOUGH PROGRAMS

This section gives some guidelines to build a program that returns a faithful rounding of $\log(x)$. It is based on the evaluation of a particular polynomial. Then, this section presents how to constrain

some coefficients of this polynomial, so that the special input 1 can also be handled in the general flow.

4.3.1 *Program to handle the general flow*

Let $x \neq 1$ be a floating-point number. Our goal is now to build a program that computes $r = r_h + r_\ell$ as in (4.11). Following [143], we use a 3-step process:

- We consider $\log(x)$ as the exact result of the function F defined as in (4.3):

  $$F(x) = (e' + \tau - \lambda) \cdot \log(2) - \log(2^\tau \cdot r_i) + \log(1 + u).$$

- Since F cannot be evaluated directly, we approximate the function F by another function P:

  $$P(x) = (e' + \tau - \lambda) \cdot L^{(2)} + \text{log_tbl}(i) + a(u).$$

  Here $L^{(2)}$ is an approximation of $\log(2)$ stored as the unevaluated sum of two floating-point numbers:

  $$L^{(2)} = L_h^{(2)} + L_\ell^{(2)} \quad \text{with} \quad \left| L^{(2)} - \log(2) \right| \leqslant \theta_1.$$

  Then log_tbl(i) returns $-\log(2^\tau \cdot r_i)$ with an error no greater than θ_2. And finally $a(u)$ is a polynomial approximant of the function $\log(1 + u)$ over the interval $\mathcal{I}$ defined in (4.5) for ε, and with the approximation error defined as:

  $$\max_{u \in \mathcal{I}} |\log(1 + u) - a(u)| \leqslant \theta_3.$$

- We finally evaluate P by a finite-precision evaluation program $\mathcal{P}$, that computes $r_h + r_\ell$ and returns r_h.

Using the triangular inequality, we have:

$$\begin{aligned} |r - \log(x)| &= |F(x) - \mathcal{P}(x)| \\ &\leqslant |F(x) - P(x)| + |P(x) - \mathcal{P}(x)| \\ &\leqslant \left| e' + \tau - \lambda \right| \cdot \theta_1 + \theta_2 + \theta_3 + \theta_4, \end{aligned}$$

where θ_4 is a bound on the evaluation error of $\mathcal{P}$. Assuming the three cases of Section 4.2.3, in order to ensure that the condition in (4.11) holds, we must build a program so that this sum verifies:

$$\left| e' + \tau - \lambda \right| \cdot \theta_1 + \theta_2 + \theta_3 + \theta_4 \leqslant \theta. \tag{4.12}$$

This is detailed in Section 4.4.2.

4.3.2 *How to handle the special input* 1 *in the general flow?*

Let us now consider the case $x = 1$. Hence we have $x' = 1$, and $m' = 1$. And from (4.4), we know that

$$\mathtt{rcp}(m') \in [1 - \kappa, 1 + \kappa].$$

Recall that the quantity r_i equals $\mathtt{rcp}(m')$ plus one half-ulp on i bits, then truncated on i fraction bits. Therefore if the parameter i is chosen such that

$$2^{-i-2} \geqslant \kappa \quad \text{and} \quad 2^{-i-1} \leqslant 2^{-i} - 2^{1-p} - \kappa, \tag{4.13}$$

then, we can deduce that $r_i = 1$. In this case, the computation of $\log(x)$ is reduced to the evaluation of $\log(1 + u)$. For our implementations, this appears for all the values i in Table 4.1, but for $i = 10$ when $(p, \kappa) = (24, 1.5 \cdot 2^{-12})$ and $(p, \kappa) = (53, 1.5 \cdot 2^{-12} + 2^{-24} + 1.5 \cdot 2^{-36})$.

As a consequence, since u is the error entailed by the approximation of $1/m'$ by r_i, we deduce also that $u = 0$. It follows that in order to handle the special input 1 in the general flow, i must be chosen to satisfy (4.13), and the polynomial approximant $a(u)$ in Section 4.3.1 must be built so that the first coefficient (a_0) is zero, to ensure that $a(u) = 0$ in this case.

4.4 IMPLEMENTATION DETAILS

This section gives some details on our implementation process, for the example of the binary32 format. After some elements on the way used to write branchless programs, it presents how to certify the accuracy of the output programs.

4.4.1 *Branchless program*

For performance purpose, a key point consists in writing branchless programs. First τ is a boolean introduced to avoid the catastrophic cancellation that may occur while reconstructing the result. It is actually an integer $\in \{0, 1\}$ determined in a branch-free fashion using comparison instructions. Then the quantity λ in (4.2) is determined in a similar way, using comparison and logical instructions.

Second, once λ is known, we must compute $x' = x \cdot 2^{\lambda}$. We could have performed this by using a floating-point multiplication. However, as observed in [99], multiplying two floating-point numbers may lead to a huge overhead, when one of both operands is a subnormal number which may be the case. An alternative is to work on the bit string encoding x using the standard binary interchange

format encoding [76, § 3.4]. Let the following bit string encode a subnormal number x in binary32 format:

$$0\;\;00000000\;\;\underbrace{000000000}_{\lambda=9}10101000000000.$$

Computing $x \cdot 2^{\lambda}$ just consists in shifting this bit string by $\lambda + 1$ bits to the left. With $\lambda = 9$, we obtain:

$$0\;\;00000001\;\;0101000000000\underbrace{0000000000}_{\lambda+1=10}$$

which encodes $x' \geqslant 2^{e_{\min}}$. This sequence of operations relies on arithmetic and logical instructions, but it requires also a means to determine λ, and more particularly a routine that returns λ if x is a subnormal number, and 0 otherwise. The instruction set we target does not embrace the `nlz` instruction, enabling to count the number of leading zeros in a bit string. Various techniques can be implemented in software for computing this, but either they are costly or they use branches [176]. In our implementation, we use the following piece of code, presented here for the binary32 arithmetic, but that can be adapted to any other formats. It works on an integer X, that represents the bit string of a floating-point x.

Listing 4.1 – Vectorizable Algorithm to Compute λ.

```
20 typedef union { uint32_t i; float f; } cfloat32;
21
22 uint32_t lambda_or_zero(uint32_t X) {
23   cfloat32 Z;
24   Z.i = X | 0x3f800000; // Exponent mask of 1.0f
25   Z.f -= 1.f;
26   uint32_t mask = 0xffffffff + ((X >> 23) != 0);
27   uint32_t value = ((Z.i >> 23) - 127) & mask;
28   return -value;
29 }
```

1. If x is a normal number, its exponent field (`X >> 23`) is nonzero, and `mask` = 0. Hence the routine returns 0.

2. Otherwise if x is a subnormal number:

$$x = 2^{-126} \times 0.\underbrace{000\cdots000}_{\lambda}1\cdots.$$

Then at Line 24, we have:

$$Z = 2^{0} \times 1.\underbrace{000\cdots000}_{\lambda}1\cdots.$$

At Line 25, the subtraction is exact due to Sterbenz lemma. Thus as long as $1 - p \geqslant e_{\min}$, we have

$$Z = 2^{-\lambda} \times 1.\cdots.$$

The exponent field of x (`X >> 23`) being zero, we deduce that `mask` $= 2^{32} - 1 = 1111\cdots1111$, and `value` $= -\lambda$. Hence the routine returns λ.

This is apparently a well-known technique to compute the number of leading zeros in the bit string of a floating-point number. But to our knowledge, it has never been published, and was brought to our knowledge during a discussion with Christoph Lauter.

4.4.2 *Certified evaluation program*

Recall that in this section, we target the binary32 arithmetic, that is, for $p = 24$. For accuracy purpose, we must build an evaluation program, so that the bound in (4.12) is satisfied. Note that in the design process, as mentioned in Section 4.2.3, we must distinguish three cases. Indeed, the error bound $\theta = 2^{-48}$, which is the tightest of the three cases in (4.11), could not have been proven on the whole input interval.

From now we start by determining the polynomial degree and coefficients. To avoid branches for deciding in which cases the inputs fall, we choose to implement the $\log(x)$ function with a single degree-d polynomial. Since $\theta_3 \geqslant 0$, we know from (4.11) and (4.12) that the approximation error bound must satisfy $\theta_3 \leqslant 2^{-48}$. Remark that the lookup table index size i influences the approximation interval, and thus the degree d: once the couple (i, d) is chosen, the polynomial approximant together with the certified error bounds θ_1, θ_2 and θ_3 are computed using Sollya [26]. This way, we obtain $\theta_1 \approx 2^{-53.33}$. In addition Table 4.2 shows the polynomial degree d, the error bounds θ_2 and θ_3, as well as the memory occupancy in bytes for lookup table and polynomial coefficients, for different table index size i. In the rest of the example, we choose to implement $\log(x)$ using a table indexed by 7 bits and a degree-5 polynomial.

The remaining part is to compute the program evaluation error bound θ_4, and to check if the bound (4.12) holds. This error corresponds mainly to the error due to the evaluation of the polynomial approximant $a(u)$ in finite precision arithmetic, combined with the error entailed by the reconstruction. Note that the way used to evaluate $a(u)$ may be determined by using the software tool CGPE [120], which enables to build several schemes to evaluate a given polynomial. Then, bounding the evaluation error is carried out using the Gappa tool [115]. Notice that our implementation works with double-single arithmetic, that is, with numbers represented as the unevaluated sum of two binary32 floating-point numbers. Thus we first need to adapt the error bounds computed in [102, § 4] for double-

Table 4.2 – Implementation Parameters and Table and Polynomial Coefficient Sizes (# Bytes) for Various Values of i When $p = 24$.

i	d	θ_2	θ_3	table	coefficient
1	15	$2^{-52.69}$	$2^{-50.40}$	16	116
2	11	$2^{-52.69}$	$2^{-50.27}$	32	84
3	9	$2^{-52.69}$	$2^{-52.01}$	64	68
4	7	$2^{-52.29}$	$2^{-49.62}$	128	52
5	6	$2^{-52.10}$	$2^{-49.82}$	256	44
6	5	$2^{-51.36}$	$2^{-48.88}$	512	36
7	5	$2^{-51.20}$	$2^{-54.50}$	1024	36
8	4	$2^{-51.04}$	$2^{-48.98}$	2048	28
9	4	$2^{-51.03}$	$2^{-52.93}$	4096	28

double addition and multiplication to our context, and to pass them to Gappa: we found a relative error bound of 2^{-44} and 2^{-45} for double-single addition and multiplication, respectively. Using them, we thus find that the absolute evaluation error of the polynomial is not greater than $\approx 2^{-50.86}$, which is actually less than 2^{-48}. Then according to the case, we obtain:

$$\theta_4 \leqslant \begin{cases} 2^{-49.86} & \text{in Case 1,} \\ 2^{-45.45} & \text{in Case 2,} \\ 2^{-36.21} & \text{in Case 3,} \end{cases}$$

and in all cases, the condition in (4.12) holds.

4.5 TOWARDS AUTOMATED IMPLEMENTATIONS

This section presents insights on how to automate the implementation of the logarithm function. It starts with a description of the Metalibm framework, before some details on its extension to our context.

4.5.1 *The Metalibm-lugdunum framework*

So far this framework has been actively developed as a fast and efficient code generation tool for two kinds of final users. The first kind is experienced software developers who aim at implementing elementary mathematical function libraries optimized for different architectures. The second kind is non-expert numerical software users, interested in customized mathematical code generation for their applications. Among the main features, it enables:

- To develop multi-architecture-specific meta-codes, and to transparently support different accuracies (improved, normal, degraded or customized), producing automatically different code versions,

- To factor common code across software and hardware architectures, for various I/O precisions and micro-architectures, thus reducing the development time,
- To vectorize meta-code according to different micro-architectures, to fit at best the underlying architecture.

The latter mainly consists in removing code branches, and inserting tests and data selections, instead. This technique implies speculative execution, and thus increases the code size, and eventually the evaluation latency. However for throughput purpose, this may be quite efficient as shown in the Cephes library [73, 138]. Our programs are already written in a branchless fashion: thus this vectorization step will not have any impact on the produced optimized code. But this remark reinforces our choice to write branchless codes, including the handling of subnormal inputs in the general flow.

In this work, an effort has been made to enhance the backend of Metalibm, with all the needed instructions unavailable so far, but also to provide the possibility to use CGPE to build efficient polynomial evaluation schemes, as detailed in Section 4.5.2 below.

4.5.2 *The case of polynomial evaluation schemes*

As shown in Section 4.2.1, our implementations of $\log(x)$ function rely on the evaluation of a polynomial, whose degree varies from 4 up to 15 (Table 4.2). Various schemes may be used to evaluate a given degree-d polynomial. In order to achieve high throughputs, we chose to use CGPE (Code Generation for Polynomial Evaluation) to generate an efficient polynomial evaluation scheme on the targeted architecture. Given the polynomial coefficients and a bound on the evaluation error, it enables to automatically write and certify programs to evaluate this polynomial. At first developed for VLIW integer processors to provide fixed-point computation abilities to this kind of hardware, it has recently been extended to handle floating-point computations. But above all, one of its main features is its capability in computing polynomial evaluation schemes, exposing more or less instruction-level parallelism. To do this, it is based on exhaustive and heuristic algorithms that look for low latency polynomial evaluation schemes on abstract customizable architectures.

CGPE is written in C++ and is available as a command-line tool. Since Metalibm is written in Python, a first step was to develop Python bindings for CGPE (called PythonCGPE). This package provides a non-exhaustive interface to CGPE features in Python that gives means to automatically generate efficient polynomial schemes on customizable architectures. More precisely, multiplier and adder latencies are customizable, but the use of heuristics has been automatized for polynomials of degree greater than 6 to avoid blocking Metalibm whenever the search space grows too large. Note that even with these heuristics, the search space may get huge when the degree increases. Hence as soon as $d \geqslant 12$, we choose to skip the schemes

computation step, and to rely only on the classic Estrin's rule. This is a well-known limit of the CGPE software tool, and solving this issue is out of the scope of this thesis.

4.6 NUMERICAL RESULTS

Using the process presented before, we have generated different versions of $\log(x)$, in order to evaluate their performance. This section presents some numerical results. It compares our implementations to existing solutions. Then it studies the impact of the table index size and the cost of handling subnormal in the general flow.

4.6.1 *Experimental protocol*

We measure the reciprocal throughput of handmade and generated routines, and evaluate them against the Libmvec routines. All benchmarks are run on an Intel® Xeon® Processor E5-2650v4, which features the AVX2 instruction set extensions. The operating system is CentOS Linux release 7.4.1708, running a Linux kernel version 3.10.0-693.2.2.el7.x86_64. Although this CPU has 12 physical cores, we make sure that no other compute-intensive jobs are running at the same time as our benchmarks to reduce potential noise. All source code is compiled using GCC 7.2.0, linked against glibc 2.26, both compiled from source. Shared compiler flags include `-O3 -mtune=native`. For Libmvec benchmarks, we also have to enable `-ftree-loop-vectorize -ffast-math` and link against GNU libm. The former option is used to vectorize loops, while the latter enables mathematical simplifications, flushes subnormal arguments and results to zero, among other optimizations, and is required to activate the Libmvec. For all benchmarks, to enable AVX2 code generation we simply use the `-march=core-avx2` flag, while we use `-march=corei7` to restrict GCC to emitting SSE4 code.

To evaluate these performances, we use either automated micro-benchmarks provided by Metalibm or custom micro-benchmarks for the handmade version and Libmvec routines. These custom micro-benchmarks try to reproduce plausible workloads by using pseudo-random inputs, but also rare workloads by using only subnormal inputs. The benchmarks are designed to warm up the L1 data and instruction caches before the main measurement, so that cache-miss penalties may be kept to a minimum. Also, we choose to take the minimum measured reciprocal throughput for each micro-benchmark. We claim that this represents the best the CPU can *actually* compute when there is the least noise perturbing the measures. This can be further justified by the fact that we benchmark separately pseudo-random input vectors versus subnormals-only vectors, which may yield lower performance. We also benchmark constants-only vectors, which might yield better throughputs for vectorized memory gather.

Table 4.3 – Measured Performances of our Implementations Compared to the GNU Libmvec, in Cycles per Element.

Version	ISA	`fast-math`	Workload	Rcpr. throughput in binary32 (CPE)	Rcpr. throughput in binary64 (CPE)
Libmvec 128 bits	SSE4	yes	PRNG	1.2	5.9
Libmvec 256 bits	AVX2	yes	PRNG	0.3	1.3
Libmvec 128 bits	SSE4	yes	Subnormals	6.7	40.5
Libmvec 256 bits	AVX2	yes	Subnormals	2.6	15.2
Handmade 128 bits	AVX2	no	Any	20.7	88.2
Handmade 256 bits	AVX2	no	Any	11.4	44.9
ML-generated 128 bits	AVX2	no	Any	24.0	n/a
ML-generated 256 bits	AVX2	no	Any	16.0	n/a

4.6.2 *Performance of the* $\log(x)$ *function*

In this section, we measure the reciprocal throughput of the generated routines, and compare them to SSE4 and AVX2 reference implementations coded by hand and to SSE4 and AVX2 versions of Libmvec `logf/log` routines. In this experiment, our implementations use tables indexed by $i = 7$ bits, and we use Metalibm and custom micro-benchmarks, for generated and handmade codes, respectively. Performance results are presented in Table 4.3, in cycles per element (CPE).

We observe that using our approach, we achieve a binary32 implementation with a reciprocal throughput of 20.7 CPE for vector size = 4 and 11.4 CPE for vector size = 8. In binary64, the measured performances are 88.2 CPE for vector size = 2 and 44.9 CPE for vector size = 4. This is greater than binary32 versions by a factor close to 4. Obviously, our implementations are slower than Libmvec implementations: for example, in binary32 their reciprocal throughputs vary from 0.3 up to 6.7 CPE with respect to the vector sizes and the input ranges. This might be due to the fact that binary32 Libmvec implementation relies on the evaluation of a small degree polynomial, done using single precision arithmetic only (binary32), while in our case, polynomials are evaluated using double-single arithmetic. A direct consequence is the accuracy of the output. Indeed, our implementations are correct within 1 ulp (i.e. faithful), while Libmvec provides functions with as much as 4 ulp absolute error. Furthermore Libmvec has a scalar fallback for special inputs such as NaNs, infinities or zero. Since subnormal numbers are flushed to zero with `-ffast-math`, the Libmvec routines are less efficient when dealing with a subnormals-only vector, by a factor of ≈ 5 in binary32 and ranging from 6.8 up to 11.7 in binary64. Conversely, as our scheme unifies normal and subnormal handling, performance is not affected by a subnormals-only vector. (The same remarks hold for the binary64 format.)

An interesting observation is that the multiple-indices vectorized accesses to tabulated values are *not* penalizing, although data is accessed at non-contiguous locations. This may be explained by the fact that all tables are small — typically less than 2 kB for binary32 precision and less than 4 kB for binary64 — therefore fitting easily in the 32-kB L1d cache of the targeted processor.

Note finally that, as mentioned above, our routines are guaranteed faithfully rounded. However, to gain in confidence, the binary32 versions have been exhaustively verified and compared with the values returned by the MPFR library.[1]

4.6.3 *Impact of the table size and subnormal handling*

In this section, we measure the impact on performance of the table size and the cost of supporting subnormals in the main flow. To do

[1] `http://www.mpfr.org/` and [67].

Table 4.4 – Measured Performances (CPE) on AVX2 of our Generated Implementations, in Binary32 Arithmetic and for Vectors of Size 4 (v4) or 8 (v8), With (sub) or Without (no-sub) Subnormal Handling.

i	3	4	5	6	7	8	9
v4 / sub	60.7	51.0	46.0	24.0	24.0	21.6	22.5
v8 / sub	32.3	27.5	16.8	15.7	16.0	14.1	14.0
v4 / no-sub	58.4	49.5	24.5	22.2	22.0	20.6	21.3
v8 / no-sub	31.2	26.8	15.9	14.9	14.7	19.6	19.7

this, we generate routines in the binary32 format with the Metalibm framework, for a parameter i ranging from 3 up to 9, for different vector sizes (typically 4 and 8), and with and without subnormal handling. To unplug subnormal handling, it suffices to remove λ from all the computations. Hence we simply adapt our meta-code in this sense. Table 4.4 shows the performance of these routines in CPE.

As expected, the greater the table size, the better the routines performance. Using this table we may conclude that, for performance purpose, a table indexed by 8 bits seems to be a good choice.

Furthermore, an interesting observation is that treating subnormal floating-point numbers in the main flow has no great impact on the performance of the whole function. Indeed, except for $i = 5$ and vector size = 4, the overhead due to this technique remains no greater than 2.3 CPE, which is acceptable. This reinforces our choice to avoid branches and fallback routines to treat subnormals separately.

4.7 CONCLUSION AND PERSPECTIVES

In this chapter we have addressed the implementation of logarithm functions on vector micro-architectures, with a particular focus on its automation through the Metalibm framework. We achieve high throughput implementations with 1-ulp accuracy, optimized for SSE, AVX and AVX2 micro-architectures. For example, on AVX2 micro-architectures, we measure throughputs of 11.4 and 44.9 CPE for binary32 and binary64, respectively, with a relatively small overhead due to handling subnormals in the main flow (≈ 2 CPE).

In Table 4.3, "n/a" indicates that Metalibm was not able to generate these functions. This is actually due to the lack of certain meta-instructions in the x86 backend. Ongoing work focuses on integrating these requirements. In a near future, this could enable automatic generation of binary64 implementations as well. More generally, the IEEE 754 standard defines two other formats, namely, the binary16 and binary128. Since our design does not depend upon the precision, generating implementations for these formats is reachable and of interest. This would require efforts to eventually emulate the underlying required missing arithmetic (SIMD support for binary128, for example), and to adapt the Metalibm backend in consequence.

Furthermore we could extend our efforts to the implementation of certain elementary functions. Following [143], a direct extension would be the optimized implementations of $\log_2(x)$ and $\log_{10}(x)$. In addition, we could concentrate on the design of efficient exponential or trigonometric functions, which are also widely used in high-performance computing.

GENERAL CONCLUSION AND PERSPECTIVES

> "This is it. Let me give you one piece of advice: be honest. He knows more than you can imagine."
>
> Trinity, in *The Matrix*,
> by the Wachowski brothers.

This thesis aimed at developing more automatic code generation algorithms and techniques for the efficient evaluation of elementary functions. In Chapter 2 we introduced a novel and elegant method to automatically generate exact lookup tables for trigonometric and hyperbolic functions, a goal that had never been reached for decades. This technique could overtake improvements such as those that were brought by Gal and Bachelis or Stehlé and Zimmermann at least for the trigonometric and hyperbolic functions.

In Chapter 3 we analyzed the impact on throughput brought by vectorizing polynomial evaluation schemes. First, we confirmed previous results showing that elementary arithmetic operations producing subnormal floating-point numbers could drastically reduce the throughput of applications such as polynomial evaluation schemes. Second, contrary to the generally assumed idea, we observed that higher-ILP schemes could increase the throughput on Intel Haswell architectures.

Finally, in Chapter 4, we advocated the concept of "metalibms", i.e. frameworks that allow to *describe* rather than to *implement* elementary functions or evaluation schemes for elementary functions, and that are then able to *generate* source codes for different optimization criteria. For example, such criteria include precision, accuracy, target architecture or vectorization. Through our meta-implementation of the natural logarithm, we were able to describe our contributions to Metalibm-lugdunum, a framework for developers and users of mathematical libraries that allows to describe "meta-evaluation schemes" and to generate optimized source code for various targets. Our contributions included improved support for the Intel x86 backend and its vectorized instructions, so that we were able to generate fully-vectorized logarithms in the binary32 and the binary64 floating-point precisions for portable C code *and* the Intel AVX2 architecture. One of the main advantages of this framework is that we only had to describe *one* meta-algorithm to be able to generate *many* optimized implementations, which can bring huge time-saving opportunities in the future.

The AVX2 versions of our meta-logarithm were measured to be competitive on several aspects compared to the GNU Libmvec implementations: First, although the automatically generated codes showed lower measured throughputs than GNU Libmvec, our meta-imple-

mentation is *proven* faithfully-rounded while the GNU Libmvec is only accurate to a tested 4 ulps accuracy. Second, the GNU Libmvec consists in several routines written in x86 assembly, whereas our meta-implementation can generate multiple source codes for several languages and targets. For our example, we used the contributed x86 backend to generate C codes that use the Intel Intrinsics C API.

NEW HORIZONS

In Chapter 2, we presented an exact table lookup generation heuristic using Fässler's algorithms, for the trigonometric functions only. When targeting correct rounding in double precision, we achieved to shrink lookup tables by up to 29%, compared to Tang's and Gal's methods. We mentioned that this benefit could even be higher for greater targeted accuracies since all tabulated values except the corrective terms are exactly stored and, thus, do not depend on the targeted accuracy. Also, we only focused on the trigonometric and hyperbolic functions, as they both benefit from the use of Pythagorean triples. However, the concept remains valid and worthwhile for other functions, provided at least two terms need to be tabulated per row and that exact values can be found for such terms. Therefore, short-to-medium term interesting perspectives could be:

- the implementation of a heuristic for the hyperbolic functions, by generating Pythagorean triples sharing a common leg. A first lead might be to follow the results presented in [147] by Rothbart and Paulsell;

- to investigate sufficient conditions for the optimality of such heuristics, as a first step towards a more formal validation of our approach;

- the integration and benchmarking of our exact lookup tables, e.g. into the high performance implementations of the trigonometric functions in the medium-precision range (4096 bits) detailed in [85]. Indeed, an actual proof-of-concept using our method would probably contribute much to its spreading in real-world code bases.

  Also, a short-term objective could be to integrate our tables to the Metalibm-lugdunum framework, so as to enable meta-implementations to use them for accurate evaluation schemes;

- to investigate whether other elementary functions could benefit from such exact lookup tables. The `sinPi` and `cosPi` routines might be a first lead, although we do not foresee much more than that. Indeed, we must insist that our method is only interesting whenever two or more values must be tabulated. Otherwise, Gal's accurate tables are probably a better option.

As far as evaluating polynomials is concerned, we believe that the study performed in Chapter 3 could be developed in several directions, from short-term to long-term perspectives:

- first, it might be interesting to study if an efficient evaluation scheme has a noticeable impact on the performance of elementary functions evaluations.

  Early tests on classic function evaluation schemes tend to show negligible impacts. However, further investigations, e.g. within the Metalibm-lutetia open-ended generation framework, might demonstrate the contrary, because it relies heavily on a domain-splitting algorithm [22];

- next, CGPE could be developed to use new criteria to discard polynomial evaluation schemes. For instance, schemes that may produce subnormal floating-point results for a given input interval and a given set of coefficients. The Gappa tool could probably be used to implement such a heuristic quite efficiently. As a result, CGPE would be able to certify such property when targeting high performance polynomial evaluation;

- a medium to long-term goal would be to merge CGPE features into the different Metalibm projects so as to benefit from an efficient code generator for polynomial evaluations. Such a project has already started within the Metalibm-lugdunum framework, through the PythonCGPE bindings presented in Chapter 4;

- finally, a long-term research opportunity would be to do an in-depth analysis of the lowest-latency schemes generated by CGPE. It may allow us to directly build efficient schemes without having to perform an exponentially expensive search. This study should probably be combined with theoretical research on “polynomial chains” [90].

The paradigm shift towards more meta- and generative programming presented in Chapter 1, Section 1.3 and in Chapter 4, looks very promising to produce tailored mathematical libraries for high performance software. In addition, it could reduce libms maintenance costs and development time by pushing for generic code reuse. This is why we believe that, along with other complementary concepts of “metalibms” such as MetaLibm-lutetia, meta-implementations are the modern way to implement, optimize, maintain, and generate elementary functions evaluation schemes. Hence, we think that potentially interesting future research topics could include:

- merging similar meta-implementations, e.g. multiple logarithms into an efficient, generic, base-β logarithm. This should be a rather straightforward application of [143] in the context of Metalibm-lugdunum;

- providing support for other IEEE 754 formats such as the binary16 and binary128 formats, or decimal floating-point for-

mats. Indeed, these formats may be relevant choices for specific applications. Metalibm-lugdunum already supports several mixed-precision FMA operators for binary16 and low precision fixed-point implementations, both useful for e.g. machine learning applications [21]. In this sense, it could be interesting to list and implement what is missing in the different MetaLibm projects to support code generation for these formats;

- making an inventory of generic, reusable meta-blocks. For instance, in Metalibm-lugdunum, there is a somewhat generic Payne and Hanek range reduction [137], that could be further abstracted.

  Also, following the specific hardware support for the fast hardware reciprocal approximation that we implemented in Metalibm-lugdunum, we could provide new meta-blocks or optimization passes for specific hardware features, e.g. for NVIDIA Tensor Cores [129]; These units basically provide floating-point $4 \times 4 \times 4$ matrix-matrix FMA in mixed-precision.[1] According to NVIDIA, Tensor Cores can increase throughput for GEMM[2] by a factor up to 12x compared to GP100 accelerators. Therefore, more support of such hardware specificities through higher-level meta-code and optimization passes within the Metalibm-lugdunum framework could bring decisive advantages over existing solutions like CUDA programming. Metalibm-lugdunum already has an OpenCL backend, hence it should not be difficult to develop a CUDA backend too. Then, BLAS[3] could also benefit from tailored code generation within a unified code generation framework.

- finally, Metalibm-lutetia has been supporting automatic black-box detection of periodicity and symmetry properties [22]. A medium-term research problem could be to provide support for automated asymptotic analysis, too. First, a white-box approach seems more realistic as it could reason on algebraic properties. Such analysis might allow for an improved domain-splitting algorithm whenever clear asymptotic behaviors are detected.

---

[1] Concretely, binary16 $\times$ binary16 $+$ binary32 $\rightarrow$ binary16/binary32.

[2] GEneral Matrix Multiplication, BLAS Level-3 and a core operation for training Deep Neural Networks and Convolutional Neural Networks.

[3] Basic Linear Algebra Subroutines.

BIBLIOGRAPHY

[1] Georges Aad et al. "Measurements of the Higgs boson production and decay rates and coupling strengths using pp collision data at $\sqrt{s} = 7$ and 8 TeV in the ATLAS experiment". In: *The European Physical Journal C* 76.1 (Jan. 5, 2016), p. 6. ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-015-3769-y (cit. on p. 1).

[2] Georges Aad et al. "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC". In: *Physics Letters B* 716.1 (2012), pp. 1–29. ISSN: 0370-2693. DOI: 10.1016/j.physletb.2012.08.020 (cit. on pp. vii, 1).

[3] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables*. 10th ed. Vol. 55. Applied Mathematics Series. Washington, D.C. 20402: United States Department of Commerce, Dec. 1972 (cit. on p. 18).

[4] William S. Anglin. "Using Pythagorean Triangles to Approximate Angles". In: *American Mathematical Monthly* 95.6 (June 1988), pp. 540–541. URL: http://www.jstor.org/stable/2322760 (cit. on p. 44).

[5] Tom M. Apostol. *Introduction to analytic number theory*. New-York: Springer-Verlag, 1976. ISBN: 0387901639 (cit. on p. 49).

[6] George A. Baker, Jr. and Peter Graves-Morris. *Padé Approximants*. 2nd ed. Vol. 59. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1996, pp. XII, 746. ISBN: 9780521450072 (cit. on p. 20).

[7] Freek J. M. Barning. "Over Pythagorese en bijna-Pythagorese driehoeken en een generatieproces met behulp van unimodulaire matrices [*On Pythagorean and quasi-Pythagorean triangles and a generation process with the help of unimodular matrices*]". Dutch. In: *Mathematisch Centrum Amsterdam Afdeling Zuivere Wiskunde ZW-011* (1963) (cit. on pp. xv sq., 43, 45).

[8] Guy Beadie et al. "Refractive index measurements of poly(methyl methacrylate) (PMMA) from $0.4 - 1.6$ µm". In: *Applied Optics* 54.31 (Nov. 2015), F139–F143. DOI: 10.1364/AO.54.00F139 (cit. on p. 2).

[9] Albert H Beiler. *Recreations in the theory of numbers: The queen of mathematics entertains*. Dover Publications, 1964 (cit. on p. 50).

[10] B. Berggren. "Pytagoreiska trianglar [*Pythagorean triangles*]". Swedish. In: *Tidskrift för elementär matematik, fysik och kemi* 17 (1934), pp. 129–139 (cit. on p. 31).

[11] Matthias Bernegger and J. E. Meyer. *Manuale mathematicum: darinn begriffen, die tabulae sinuum, tangentium, secantium ; so wol die Quadrat- und Cubictafel, sambt gründlichem Unterricht, wie solche nützlich zugebrauchen*. German. Meyer, 1612. URL: https://books.google.fr/books?id=VstZAAAAcAAJ (cit. on p. 18).

[12] Frank Bernhart and H. Lee Price. "Pythagoras' garden, revisited". In: *Australian Senior Mathematics Journal* 26.1 (2012), pp. 29–40 (cit. on p. 31).

[13] Wolfgang Boehm and Andreas Müller. "On de Casteljau's algorithm". In: *Computer Aided Geometric Design* 16.7 (1999), pp. 587–605. ISSN: 0167-8396. DOI: 10.1016/S0167-8396(99)00023-0 (cit. on p. 21).

[14] Sylvie Boldo. "Preuves formelles en arithmétiques à virgule flottante". French. PhD thesis. École Normale Supérieure de Lyon, Nov. 2004. URL: http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2004/PhD2004-05.pdf (cit. on pp. xxi, 75).

[15] Sylvie Boldo and Marc Daumas. "A Simple Test Qualifying the Accuracy of Horner's Rule for Polynomials". In: *Numerical Algorithms* 37.1–4 (Dec. 2004), pp. 45–60. ISSN: 1017-1398 (print), 1572-9265 (electronic) (cit. on pp. 21, 74).

[16] Nicolas Brisebarre and Sylvain Chevillard. "Efficient polynomial L^∞-approximations". In: *18th IEEE Symposium on Computer Arithmetic*. 2007, pp. 169–176 (cit. on pp. x, 20, 27, 35).

[17] Nicolas Brisebarre, Milos D. Ercegovac, and Jean-Michel Muller. "(M, p, k)-Friendly Points: A Table-Based Method for Trigonometric Function Evaluation". In: *IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*. 2012, pp. 46–52 (cit. on pp. 20, 39).

[18] Nicolas Brisebarre et al. "A New Range-Reduction Algorithm". In: *IEEE Transactions on Computers* 54 (2005) (cit. on pp. xiii, 22, 35).

[19] Geoff Brumfiel. "High-energy physics: Down the petabyte highway". In: *Nature News* 469.7330 (2011), pp. 282–283 (cit. on pp. vii, 1).

[20] Nicolas Brunie. "Contributions to computer arithmetic and applications to embedded systems". PhD thesis. École normale supérieure de lyon, May 2014. URL: https://tel.archives-ouvertes.fr/tel-01078204 (cit. on pp. xi sq., xxvi, 26, 28–30, 88).

[21] Nicolas Brunie. "Modified Fused Multiply and Add for Exact Low Precision Product Accumulation". In: *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*. July 2017, pp. 106–113. DOI: 10.1109/ARITH.2017.29 (cit. on pp. xxx, 108).

[22] Nicolas Brunie et al. "Code Generators for Mathematical Functions". In: *22nd IEEE Symposium on Computer Arithmetic, ARITH 2015, Lyon, France, June 22-24, 2015*. June 2015, pp. 66–73. DOI: 10.1109/ARITH.2015.22 (cit. on pp. viii, xi, xxvi, xxix sq., 4, 26, 28, 88, 107 sq.).

[23] Sergey Chatrchyan et al. "Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC". In: *Physics Letters B* 716.1 (2012), pp. 30–61. ISSN: 0370-2693. DOI: 10.1016/j.physletb.2012.08.021 (cit. on pp. vii, 1).

[24] Sergey Chatrchyan et al. "Observation of a new boson with mass near 125 GeV in pp collisions at $\sqrt{s} = 7$ and 8 TeV". In: *Journal of High Energy Physics* 2013.6 (June 20, 2013), p. 81. ISSN: 1029-8479. DOI: 10.1007/JHEP06(2013)081 (cit. on p. 1).

[25] Sergey Chatrchyan et al. "Study of the Mass and Spin-Parity of the Higgs Boson Candidate via Its Decays to Z Boson Pairs". In: *Phys. Rev. Lett.* 110 (8 Feb. 2013), p. 081803. DOI: 10.1103/PhysRevLett.110.081803 (cit. on p. 1).

[26] Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter. "Sollya: An Environment for the Development of Numerical Codes". In: *Mathematical Software – ICMS 2010*. Ed. by K. Fukuda et al. Vol. 6327. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, Sept. 2010, pp. 28–31 (cit. on pp. x, xxii, 27, 41, 78, 97).

[27] Sylvain Chevillard et al. "Efficient and accurate computation of upper bounds of approximation errors". In: *Theoretical Computer Science* 412.16 (2011), pp. 1523–1543 (cit. on pp. x, 20, 27, 32).

[28] William J. Cody, Jr. and William Waite. *Software manual for the elementary functions*. New Jersey: Prentice Hall, 1980 (cit. on pp. xiii, 22).

[29] William J. Cody. "Implementation and Testing of Function Software". In: *Problems and Methodologies in Mathematical Software Production*. Ed. by Paul C. Messina and Almerico Murli. Vol. 142. Lecture Notes in Computer Science. Springer-Verlag , Berlin, 1982, pp. 24–47 (cit. on pp. xiii, 22).

[30] John Horton Conway and Richard Guy. *The Book of Numbers*. Copernicus Series. Springer New York, 1998. ISBN: 9780387979939 (cit. on p. 42).

[31] Mike F. Cowlishaw. "Decimal Floating-Point: Algorism for Computers". In: *Proceedings 2003 16th IEEE Symposium on Computer Arithmetic*. June 2003, pp. 104–111. DOI: 10.1109/ARITH.2003.1207666 (cit. on pp. 7 sq.).

[32] Hongyun Dai and Wei-Ping Zhu. "Compensation of Loudspeaker Nonlinearity in Acoustic Echo Cancellation Using Raised-Cosine Function". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 53.11 (Nov. 2006), pp. 1190–1194. ISSN: 1549-7747. DOI: 10.1109/TCSII.2006.882344 (cit. on p. 26).

[33] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. "Improving the numerical accuracy of programs by automatic transformation". In: *International Journal on Software Tools for Technology Transfer* 19.4 (Aug. 2017), pp. 427–448. ISSN: 1433-2787. DOI: 10.1007/s10009-016-0435-0 (cit. on pp. xii, 29 sq.).

[34] Catherine Daramy-Loirat et al. *CR-Libm, A library of correctly rounded elementary functions in double-precision*. URL: http://lipforge.ens-lyon.fr/www/crlibm/ (cit. on pp. xiii, xviii, xxvi, 17, 32, 37, 64, 89).

[35] Eva Darulova and Viktor Kuncak. "Towards a Compiler for Reals". In: *ACM Transactions on Programming Languages and Systems* 39.2 (Mar. 2017), 8:1–8:28. ISSN: 0164-0925. DOI: 10.1145/3014426 (cit. on pp. xii, 29 sq.).

[36] Eva Darulova et al. "Synthesis of fixed-point programs". In: *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. Sept. 2013, pp. 1–10. DOI: 10.1109/EMSOFT.2013.6658600 (cit. on pp. xii, 29 sq.).

[37] Marc Daumas and Guillaume Melquiond. "Certification of Bounds on Expressions Involving Rounded Operators". In: *ACM Transactions on Mathematical Software* 37.1 (Jan. 2010), 2:1–2:20. ISSN: 0098-3500. DOI: 10.1145/1644001.1644003 (cit. on pp. xi, 27).

[38] Marc Daumas et al. "Modular Range Reduction: a New Algorithm for Fast and Accurate Computation of the Elementary Functions". In: *Journal of Universal Computer Science*. Ed. by Hermann Maurer, Cristian Calude, and Arto Salomaa. Springer Berlin Heidelberg, 1995, pp. 162–175. ISBN: 978-3-642-80350-5. DOI: 10.1007/978-3-642-80350-5_15 (cit. on p. 35).

[39] Paul Davies. *The Goldilocks enigma: why is the universe just right for life?* Ed. by First Mariner Books. Houghton Mifflin Company, 2008, pp. XVIII, 318. ISBN: 978-0-618-59226-5 (cit. on p. 48).

[40] Florent De Dinechin et al. "Code generation for argument filtering and argument reduction in elementary functions". In: *SCAN 2010: 14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*. Revol, Nathalie and de Dinechin, Florent and Jeannerod, Claude-Pierre and Lefèvre, Vincent and Louvet, Nicolas and Morin, Sèverine and Nguyen, Hong Diep. Lyon, France, Sept.

2010. URL: https://hal.inria.fr/inria-00544808 (cit. on p. 26).

[41] Florent De Dinechin et al. "On Ziv's rounding test". In: *ACM Transactions on Mathematical Software* 39.4 (2013), p. 26. DOI: 10.1145/2491491.2491495 (cit. on p. 18).

[42] David Defour. *Cache-Optimised Methods for the Evaluation of Elementary Functions*. Research report. Laboratoire de l'informatique du parallélisme, 2002 (cit. on pp. 18, 37).

[43] David Defour. "Elementary functions : algorithms and efficient implementation for correct rounding for the double precision". PhD thesis. Ecole normale supérieure de lyon, Sept. 2003. URL: https://tel.archives-ouvertes.fr/tel-00006022 (cit. on pp. xii, 31).

[44] David Defour, Florent de Dinechin, and Jean-Michel Muller. "A new scheme for table-based evaluation of functions". In: *36th Asilomar Conference on Signals, Systems, and Computers*. 2002, pp. 1608–1613 (cit. on pp. 18, 32).

[45] Florent de Dinechin. "Matériel et logiciel pour l'évaluation de fonctions numériques: précision, performance et validation". French. Habilitation à diriger des recherches. Université Claude Bernard - Lyon I, June 2007. URL: https://tel.archives-ouvertes.fr/tel-00270151 (cit. on p. 26).

[46] Florent de Dinechin, Alexey V. Ershov, and Nicolas Gast. "Towards the Post-Ultimate Libm". In: *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*. ARITH '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 288–295. ISBN: 0-7695-2366-8. DOI: 10.1109/ARITH.2005.46 (cit. on p. 17).

[47] Florent de Dinechin and Matei Istoan. "Hardware Implementations of Fixed-Point Atan2". In: *2015 IEEE 22nd Symposium on Computer Arithmetic*. IEEE, June 2015, pp. 34–41. DOI: 10.1109/ARITH.2015.23 (cit. on p. 4).

[48] Florent de Dinechin, Mioara Joldeş, and Bogdan Pasca. "Automatic generation of polynomial-based hardware architectures for function evaluation". In: *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*. July 2010, pp. 216–222. DOI: 10.1109/ASAP.2010.5540952 (cit. on pp. xii, 26, 29 sq.).

[49] Florent de Dinechin, Cristian Klein, and Bogdan Pasca. "Generating high-performance custom floating-point pipelines". In: *2009 International Conference on Field Programmable Logic and Applications*. Aug. 2009, pp. 59–64. DOI: 10.1109/FPL.2009.5272553 (cit. on pp. xii, 29 sq.).

[50] Florent de Dinechin and Christoph Lauter. "Optimizing polynomials for floating-point implementation". In: *Proceedings of the 8th Conference on Real Numbers and Computers*. 2008, pp. 7–16 (cit. on pp. x, 27, 35).

[51] Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. "Assisted Verification of Elementary Functions Using Gappa". In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. SAC '06. Dijon, France: ACM, 2006, pp. 1318–1322. ISBN: 1-59593-108-2. DOI: 10.1145/1141277.1141584 (cit. on pp. xi, 27).

[52] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. "Certifying the Floating-Point Implementation of an Elementary Function Using Gappa". In: *IEEE Transactions on Computers* 60.2 (Feb. 2011), pp. 242–253. ISSN: 0018-9340. DOI: 10.1109/TC.2010.128 (cit. on pp. xi, 27).

[53] Florent de Dinechin, Christoph Lauter, and Jean-Michel Muller. "Fast and correctly rounded logarithms in double-precision". In: *RAIRO - Theoretical Informatics and Applications* 41.1 (Apr. 2007), pp. 85–102 (cit. on pp. xxvii, 92).

[54] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo". In: *IEEE Design Test of Computers* 28.4 (July 2011), pp. 18–27. ISSN: 0740-7475. DOI: 10.1109/MDT.2011.44 (cit. on pp. xii, 29 sq.).

[55] Florent de Dinechin and Arnaud Tisserand. "Multipartite table methods". In: *IEEE Transactions on Computers* 54.3 (Mar. 2005), pp. 319–330. ISSN: 0018-9340. DOI: 10.1109/TC.2005.54 (cit. on pp. 18, 20, 32).

[56] Debjit Das Sarma and David W. Matula. "Faithful Bipartite ROM Reciprocal Tables". In: *12th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society. 1995, pp. 17–28 (cit. on p. 32).

[57] David Dugdale. *Essentials of Electromagnetism*. Macmillan physical science. American Institute of Physics, 1997. ISBN: 9781563962530 (cit. on p. 8).

[58] Marat Dukhan. "PeachPy: A Python Framework for Developing High-Performance Assembly Kernels". In: *Workshop Python for High Performance and Scientific Computing (PyHPC 2013)*. Denver, Colorado, USA, 2013, pp. 1–7 (cit. on pp. xii, 25, 29 sq.).

[59] Marat Dukhan. "PeachPy Meets Opcodes: Direct Machine Code Generation from Python". In: *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing*. PyHPC '15. Austin, Texas, USA: ACM, 2015, 3:1–3:6. ISBN: 978-1-4503-4010-6. DOI: 10.1145/2835857.2835860 (cit. on pp. xii, 25, 29 sq.).

[60] Marat Dukhan and Richard Vuduc. “Methods for High-Throughput Computation of Elementary Functions”. In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 86–95. ISBN: 978-3-642-55224-3. DOI: 10.1007/978-3-642-55224-3_9 (cit. on pp. xii, 25, 29 sq.).

[61] Gerald Estrin. “Organization of Computer Systems: The Fixed Plus Variable Structure Computer”. In: *Papers Presented at the Western Joint IRE-AIEE-ACM Computer Conference*. San Francisco, California: ACM, 1960, pp. 33–40. DOI: 10.1145/1460361.1460365 (cit. on pp. xxi, 21, 75).

[62] James Eve. “The evaluation of polynomials”. In: *Numerische Mathematik* 6.1 (Dec. 1964), pp. 17–21. ISSN: 0945-3245. DOI: 10.1007/BF01386049 (cit. on pp. xx, 21, 74).

[63] “Evidence for the spin-0 nature of the Higgs boson using ATLAS data”. In: *Physics Letters B* 726.1 (2013), pp. 120–144. ISSN: 0370-2693. DOI: 10.1016/j.physletb.2013.08.026 (cit. on p. 1).

[64] Albert Fässler. “Multiple Pythagorean number triples”. In: *American Mathematical Monthly* 98.6 (June 1991), pp. 505–517. ISSN: 0002-9890. URL: http://www.jstor.org/stable/2324870 (cit. on pp. xvii–xix, 59).

[65] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. May 2, 2017. URL: http://agner.org/optimize/instruction_tables.pdf (cit. on p. 3).

[66] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. Jan. 16, 2016. URL: http://www.agner.org/optimize/microarchitecture.pdf (cit. on pp. xxii sq., 3, 23, 77, 81 sq.).

[67] Laurent Fousse et al. “MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding”. In: *ACM Transactions on Mathematical Software* 33.2 (June 2007). DOI: 10.1145/1236463.1236468 (cit. on p. 102).

[68] Shmuel Gal. “Computing elementary functions: A new approach for achieving high accuracy and good performance”. In: *Proceedings of the Symposium on Accurate Scientific Computations*. Ed. by Willard L. Miranker and Richard A. Toupin. Vol. 235. Lecture Notes in Computer Science. Springer-Verlag, 1986, pp. 1–16. ISBN: 3-540-16798-6. URL: http://dl.acm.org/citation.cfm?id=646650.699497 (cit. on pp. xiv, 19, 37, 89).

[69] Shmuel Gal and Boris Bachelis. “An accurate elementary mathematical library for the IEEE floating point standard”. In: *ACM Transactions on Mathematical Software* 17 (Mar. 1991), pp. 26–45. DOI: 10.1145/103147.103151 (cit. on pp. xiv, 19, 37, 89).

[70] David Goldberg. "What Every Computer Scientist Should Know About Floating-point Arithmetic". In: *ACM Computing Surveys* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163 (cit. on pp. 13, 19).

[71] Mourad Gouicem. "Conception et implantation d'algorithmes efficaces pour la résolution du dilemme du fabricant de tables sur architectures parallèles". French. PhD thesis. Université Pierre et Marie Curie – Paris 6, Oct. 2013. URL: https://www.theses.fr/2013PA066468 (cit. on p. 17).

[72] A. Hall. "232. Genealogy of Pythagorean Triads". In: *The Mathematical Gazette* 54.390 (1970), pp. 377–379. ISSN: 00255572. URL: http://www.jstor.org/stable/3613860 (cit. on pp. 43, 45).

[73] Thomas Hauth, Vincenzo Innocente, and Danilo Piparo. "Development and Evaluation of Vectorised and Multi-Core Event Reconstruction Algorithms within the CMS Software Framework". In: *Journal of Physics: Conference Series* 396.5 (2012), p. 052065. DOI: 10.1088/1742-6596/396/5/052065 (cit. on pp. vii, xxvi, 1, 87 sq., 99).

[74] John Leroy Hennessy and David Andrew Patterson. *Computer Architecture. A Quantitative Approach*. 6th ed. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, Nov. 23, 2017, p. 936. ISBN: 9780128119051 (cit. on pp. 21, 23).

[75] IBM. *A Programmer's Introduction to IBM System/360 Assembler Language*. International Business Machines Corporation. Aug. 1970 (cit. on pp. 8, 12).

[76] *IEEE Standard for Floating-Point Arithmetic*. 2008. DOI: 10.1109/IEEESTD.2008.4610935 (cit. on pp. vii sq., xx, 8 sq., 88 sq., 96).

[77] *IEEE Standard for Binary Floating-Point Arithmetic*. 1985. DOI: 10.1109/IEEESTD.1985.82928 (cit. on pp. 8, 12).

[78] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. Intel Corporation. Sept. 2016. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html (cit. on p. 3).

[79] *Intel Intrinsics Guide*. Version 3.4. Intel Corporation. July 2017. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (cit. on pp. 74, 90).

[80] Arnault Ioualalen and Matthieu Martel. "Synthesis of arithmetic expressions for the fixed-point arithmetic: The Sardana approach". In: *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*. Oct. 2012, pp. 1–8 (cit. on pp. xii, 29 sq.).

[81] Arnault Ioualalen and Matthieu Martel. "Synthesizing accurate floating-point formulas". In: *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*. June 2013, pp. 113–116. DOI: 10.1109/ASAP.2013.6567563 (cit. on pp. xii, 29 sq.).

[82] *ISO/IEC 9899:2011*. 3rd ed. International Organization for Standardization. Dec. 2011, p. 683 (cit. on p. 11).

[83] Matei Istoan. "High-performance Coarse Operators for FPGA-based Computing". PhD thesis. Université de Lyon – INSA Lyon, 2017 (cit. on pp. xii, 29 sq.).

[84] Anastasiia Izycheva and Eva Darulova. "On Sound Relative Error Bounds for Floating-Point Arithmetic". In: *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design (FMCAD 2017)*. Oct. 2017, pp. 15–22 (cit. on pp. xii, 29 sq.).

[85] Fredrik Johansson. "Efficient Implementation of Elementary Functions in the Medium-Precision Range". In: *22nd IEEE Symposium on Computer Arithmetic, ARITH 2015, Lyon, France, June 22-24, 2015*. June 2015, pp. 83–89. DOI: 10.1109/ARITH.2015.16 (cit. on pp. xxix, 72, 106).

[86] William Kahan. "Why do we need a floating-point arithmetic standard?" Retypeset by David Bindel, March 2001. Feb. 12, 1981 (cit. on p. 12).

[87] Vladimir Khachatryan et al. "Constraints on the spin-parity and anomalous HVV couplings of the Higgs boson in proton collisions at 7 and 8 TeV". In: *Phys. Rev. D* 92 (1 July 2015), p. 012004. DOI: 10.1103/PhysRevD.92.012004 (cit. on p. 1).

[88] Vladimir Khachatryan et al. "Precise determination of the mass of the Higgs boson and tests of compatibility of its couplings with the standard model predictions using proton collisions at 7 and 8 TeV". In: *The European Physical Journal C* 75.5 (May 14, 2015), p. 212. ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-015-3351-7 (cit. on p. 1).

[89] Donald E. Knuth. "Ancient Babylonian Algorithms". In: *Communications of the ACM* 15.7 (July 1972), pp. 671–677. ISSN: 0001-0782. DOI: 10.1145/361454.361514 (cit. on p. 3).

[90] Donald E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Third. Reading, MA, USA: Addison-Wesley, 1998. ISBN: 0-201-89684-2 (cit. on pp. xxi, 21, 73, 75, 107).

[91] Ker-I Ko. *Complexity Theory of Real Functions*. 1st ed. Progress in Theoretical Computer Science. Birkhäuser Basel, 1991, pp. x, 310. ISBN: 978-1-4684-6804-5. DOI: 10.1007/978-1-4684-6802-1 (cit. on p. 7).

[92] Olga Kupriianova. "Towards a modern floating-point environment". PhD thesis. Université Pierre et Marie Curie - Paris VI, Dec. 2015. URL: https://tel.archives-ouvertes.fr/tel-01334024 (cit. on pp. x, 4, 26).

[93] Olga Kupriianova and Christoph Lauter. "A domain splitting algorithm for the mathematical functions code generator". In: *2014 48th Asilomar Conference on Signals, Systems and Computers*. Nov. 2014, pp. 1271–1275. DOI: 10.1109/ACSSC.2014.7094664 (cit. on pp. xi, 28).

[94] Olga Kupriianova and Christoph Lauter. "Metalibm: A Mathematical Functions Code Generator". In: *Mathematical Software – ICMS 2014: 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*. Ed. by Hoon Hong and Chee Yap. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 713–717. ISBN: 978-3-662-44199-2. DOI: 10.1007/978-3-662-44199-2_106 (cit. on p. 4).

[95] Hugues de Lassus Saint-Geniès, Nicolas Brunie, and Guillaume Revy. "Meta-implementation of vectorized logarithm function in binary floating-point arithmetic". Submitted, under review. Feb. 2018 (cit. on pp. xx, 4, 6).

[96] Hugues de Lassus Saint-Geniès, David Defour, and Guillaume Revy. "Exact Lookup Tables for the Evaluation of Trigonometric and Hyperbolic Functions". In: *IEEE Transactions on Computers* 66.12 (May 2017), pp. 2058–2071. ISSN: 0018-9340. DOI: 10.1109/TC.2017.2703870 (cit. on pp. xii, 5, 32, 49, 51, 59).

[97] Hugues de Lassus Saint-Geniès, David Defour, and Guillaume Revy. "Range reduction based on Pythagorean triples for trigonometric function evaluation". In: *Proceedings of the IEEE 26th International Conference on Application-specific Systems, Architectures and Processors*. 2015, pp. 74–81. DOI: 10.1109/ASAP.2015.7245712 (cit. on pp. xii, 5, 32, 47, 49).

[98] Hugues de Lassus Saint-Geniès, David Defour, and Guillaume Revy. "Réduction d'argument basée sur les triplets pythagoriciens pour l'évaluation de fonctions trigonométriques". French. In: *ComPAS: Conférence en Parallélisme, Architecture et Système*. July 2015 (cit. on pp. xii, 5).

[99] Hugues de Lassus Saint-Geniès and Guillaume Revy. "Performances de schémas d'évaluation polynomiale sur architectures vectorielles". French. In: *ComPAS: Conférence en Parallélisme, Architecture et Système*. July 2016 (cit. on pp. xx, 5, 95).

[100] Christoph Quirin Lauter. "A new open-source SIMD vector libm fully implemented with high-level scalar C". In: *2016 50th Asilomar Conference on Signals, Systems and Computers*. 2016, pp. 407–411. DOI: 10.1109/ACSSC.2016.7869070 (cit. on pp. 4, 25, 87).

[101] Christoph Quirin Lauter. "Arrondi correct de fonctions mathématiques - Fonctions univariées et bivariées, certification et automatisation". French. PhD thesis. École Normale Supérieure de Lyon, Oct. 2008 (cit. on pp. xviii, xxvii, 26, 64, 92).

[102] Christoph Quirin Lauter. *Basic building blocks for a triple-double intermediate format*. Research Report RR-5702. Inria, Sept. 2005, p. 67. URL: https://hal.inria.fr/inria-00070314 (cit. on pp. xviii, 64, 97).

[103] Christoph Lauter and Marc Mezzarobba. "Semi-Automatic Floating-Point Implementation of Special Functions". In: *2015 IEEE 22nd Symposium on Computer Arithmetic*. June 2015, pp. 58–65. DOI: 10.1109/ARITH.2015.12 (cit. on p. 26).

[104] Vincent Lefèvre. *Hardest-to-Round Cases*. Tech. rep. École Normale Supérieure de Lyon, 2010 (cit. on pp. 32, 64).

[105] Vincent Lefèvre and Jean-Michel Muller. "Worst cases for correct rounding of the elementary functions in double precision". In: *15th IEEE Symposium on Computer Arithmetic*. 2001, pp. 111–118. DOI: 10.1109/ARITH.2001.930110 (cit. on pp. 17, 37).

[106] Vincent Lefèvre and Jean-Michel Muller. "Worst cases for correct rounding of the elementary functions in double precision". 2003. URL: http://perso.ens-lyon.fr/jean-michel.muller/TMDworstcases.pdf (cit. on pp. 17, 37).

[107] Vincent Lefèvre, Jean-Michel Muller, and Arnaud Tisserand. "Toward Correctly Rounded Transcendentals". In: *IEEE Transactions on Computers* 47.11 (1998), pp. 1235–1243. ISSN: 0018-9340. DOI: 10.1109/12.736435 (cit. on pp. 14, 16 sq.).

[108] Ren-Cang Li. "Near Optimality of Chebyshev Interpolation for Elementary Function Computations". In: *IEEE Transactions on Computers* 53.6 (June 2004), pp. 678–687. DOI: 10.1109/TC.2004.15 (cit. on p. 20).

[109] Benoit Lopez. "Optimal implementation of linear filters in fixed-point arithmetic". French. PhD thesis. Université Pierre et Marie Curie - Paris VI, Nov. 2014. URL: https://tel.archives-ouvertes.fr/tel-01127376 (cit. on pp. xi, 29).

[110] Julien Le Maire et al. "Computing floating-point logarithms with fixed-point operations". In: *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. July 2016, pp. 156–163. DOI: 10.1109/ARITH.2016.24 (cit. on p. 4).

[111] Peter Markstein. *IA-64 and elementary functions: speed and precision*. Hewlett–Packard professional books. Prentice-Hall, 2000, pp. xix + 298. ISBN: 0-13-018348-2 (cit. on pp. xii, 23, 31, 34).

[112] Matthieu Martel, Amine Najahi, and Guillaume Revy. "Toward the synthesis of fixed-point code for matrix inversion based on Cholesky decomposition". In: *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*. Oct. 2014, pp. 1–8. DOI: 10.1109/DASIP.2014.7115609 (cit. on pp. xii, 29).

[113] Kiyoshi M. Maruyama. "On the Parallel Evaluation of Polynomials". In: *IEEE Transactions on Computers* C-22.1 (Jan. 1973), pp. 2–5. ISSN: 0018-9340. DOI: 10.1109/T-C.1973.223593 (cit. on pp. xxi, 21, 76).

[114] Pramod K. Meher et al. "50 years of CORDIC: Algorithms, architectures, and applications". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 56.9 (2009), pp. 1893–1907 (cit. on p. 21).

[115] Guillaume Melquiond. "De l'arithmétique d'intervalles à la certification de programmes". French. PhD thesis. France: ÉNS Lyon, 2006. URL: https://www.lri.fr/~melquion/doc/06-these.pdf (cit. on pp. xi, 27, 97).

[116] Daniel Menard et al. "Design of fixed-point embedded systems (DEFIS) French ANR project". In: *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*. Oct. 2012, pp. 1–2 (cit. on pp. xii, 29).

[117] Peter A. Milder et al. "Computer Generation of Hardware for Linear Digital Signal Processing Transforms". In: *ACM Transactions on Design Automation of Electronic Systems* 17.2 (2012) (cit. on pp. xii, 29).

[118] Gordon E. Moore. "Lithography and the future of Moore's law". In: vol. 2440. 1995, pp. 2–17. DOI: 10.1117/12.209244 (cit. on p. 23).

[119] *MOTOROLA M68000 FAMILY Programmer's Reference Manual*. Motorola, Inc. 1992 (cit. on p. 8).

[120] Christophe Mouilleron and Guillaume Revy. "Automatic Generation of Fast and Certified Code for Polynomial Evaluation". In: *20th IEEE Symposium on Computer Arithmetic*. 2011, pp. 233–242. DOI: 10.1109/ARITH.2011.39 (cit. on pp. xxiii, 21, 35, 83, 97).

[121] Jean-Michel Muller. *Elementary Functions. Algorithms and Implementation*. 3rd ed. Birkhäuser Basel, 2016, pp. XXV, 283. ISBN: 978-1-4899-7981-0. DOI: 10.1007/978-1-4899-7983-4 (cit. on pp. xiii, 20, 32, 34 sq., 39).

[122] Jean-Michel Muller. "On the definition of ulp(x)". In: *ACM Transactions on Mathematical Software* RR-5504 (Feb. 2005), p. 16. URL: https://hal.inria.fr/inria-00070503 (cit. on p. 92).

[123] Jean-Michel Muller et al. *Handbook of Floating-Point Arithmetic*. 1st ed. Birkhäuser Boston, 2010, p. 572. DOI: 10.1007/978-0-8176-4705-6 (cit. on pp. xx, 5, 8, 10, 12 sq., 16 sq., 20 sq., 64, 73 sq.).

[124] Ian Munro and Michael Paterson. "Optimal algorithms for parallel polynomial evaluation". In: *Journal of Computer and System Sciences* 7.2 (1973), pp. 189–198. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/S0022-0000(73)80043-1 (cit. on pp. xx sq., 21, 74, 76).

[125] Yoichi Muraoka. "Parallelism Exposure and Exploitation in Programs". PhD thesis. Dept. of Computer Science, University of Illinois at Urbana-Champaign, Feb. 1971 (cit. on pp. xxi, 76).

[126] Mohamed Amine Najahi. "Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks". PhD thesis. Université de Perpignan Via Domitia, Dec. 2014. URL: https://tel.archives-ouvertes.fr/tel-01158310 (cit. on pp. xii, 29).

[127] Rafi Nave. "Implementation of Transcendental Functions on a Numerics Processor". In: *Microprocessing and Microprogramming* 11.3–4 (Mar. 1983), pp. 221–225. ISSN: 0165-6074 (print), 1878-7061 (electronic) (cit. on p. 21).

[128] Noam Nisan and Shimon Schocken. *The elements of computing systems: building a modern computer from first principles*. MIT press, 2005 (cit. on p. 7).

[129] *NVIDIA Tesla V100 GPU architecture. the world's most advanced data center GPU*. Version WP-08608-001_v1.1. NVIDIA. Aug. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (cit. on p. 108).

[130] Stuart Oberman, Greg Favor, and Fred Weber. "AMD 3DNow! technology: architecture and implementations". In: *IEEE Micro* 19.2 (1999), pp. 37–48. DOI: 10.1109/40.755466 (cit. on pp. xxvi, 23).

[131] Georg Ofenbeck et al. "Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries". In: *SIGPLAN Not.* 49.3 (Oct. 2013), pp. 125–134. ISSN: 0362-1340. DOI: 10.1145/2637365.2517228 (cit. on pp. xii, 29).

[132] John F. Palmer, Bruce W. Ravenel, and Rafi Nave. "Numeric data processor". U.S. pat. 4338675. Intel Corporation. 1982 (cit. on p. 21).

[133] V. Ya. Pan. "Methods of computing values of polynomials". In: *Russian Mathematical Surveys* 21.1 (1966), p. 105. URL: http://stacks.iop.org/0036-0279/21/i=1/a=R03 (cit. on p. 21).

[134] Bogdan Pasca. "High-performance floating-point computing on reconfigurable circuits". PhD thesis. Lyon, France: École Normale Supérieure de Lyon, Sept. 2011 (cit. on pp. xii, 29 sq.).

[135] Michael S. Paterson and Larry J. Stockmeyer. "On the Number of Nonscalar Multiplications Necessary to Evaluate Polynomials". In: *SIAM Journal on Computing* 2.1 (1973), pp. 60–66. DOI: https://dx.doi.org/10.1137/0202007 (cit. on pp. xx, 21, 74).

[136] Alex Peleg and Uri Weiser. "MMX Technology Extension to the Intel Architecture". In: *IEEE Micro* 16.4 (Aug. 1996), pp. 42–50. DOI: 10.1109/40.526924 (cit. on pp. xxvi, 23).

[137] Mary H. Payne and Robert N. Hanek. "Radian Reduction for Trigonometric Functions". In: *SIGNUM Newsletter* 18 (1983), pp. 19–24 (cit. on pp. xiii, xxx, 22, 35, 108).

[138] Danilo Piparo, Vincenzo Innocente, and Thomas Hauth. "Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions". In: *Journal of Physics: Conference Series* 513.5 (2014), p. 052027. DOI: 10.1088/1742-6596/513/5/052027 (cit. on pp. vii, xxvi, 1, 4 sq., 25 sq., 73, 87 sq., 99).

[139] H. Lee Price. "The Pythagorean Tree: A New Species". In: *ArXiv e-prints* (Sept. 2008). arXiv: 0809.4324v2 [math.HO] (cit. on pp. xv, 43, 45).

[140] Richard Tyler Ranf. "Damage to Bridges During the 2001 Nisqually Earthquake". In: *Proceedings of the 2001 Earthquake Engineering Symposium for Young Researchers*. Ed. by Andrea Dargush, Phillip Gould, and Gerard Pardoen. Multidisciplinary Center for Earthquake Engineering Research, 2001, pp. 273–293 (cit. on p. 26).

[141] George W. Reitwiesner. "Binary Arithmetic". In: *Advances in Computers*. Ed. by A. D. Booth and R. E. Meager. Vol. 1. Academic Press, 1960, pp. 231–308 (cit. on p. 39).

[142] Eugène Yakovlevich Remez. "Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation". French. In: *C.R. Académie des Sciences, Paris* 198 (1934), pp. 2063–2065 (cit. on p. 20).

[143] Guillaume Revy. "Automated Design of Floating-Point Logarithm Functions on Integer Processors". In: *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. July 2016, pp. 172–180. DOI: 10.1109/ARITH.2016.28 (cit. on pp. xxvi, xxx, 26, 89, 93 sq., 104, 107).

[144] Guillaume Revy. "Implementation of binary floating-point arithmetic on embedded integer processors – Polynomial evaluation-based algorithms and certified code generation". PhD thesis. Université de Lyon - École Normale Supérieure de Lyon, Dec. 2009 (cit. on pp. x, xxi, xxiii, 21, 26 sq., 74, 76, 83).

[145] Tiark Rompf. "Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming". PhD thesis. École polytechnique fédérale de Lausanne, 2012 (cit. on pp. xii, 29).

[146] Tiark Rompf et al. "Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems". In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by Thomas Ball et al. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 238–261. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.238 (cit. on p. 29).

[147] Andrea Rothbart and Bruce Paulsell. "Pythagorean Triples: A New, Easy-to-Derive Formula With Some Geometric Applications". In: *The Mathematics Teacher* 67.3 (1974), pp. 215–218. ISSN: 00255769. URL: http://www.jstor.org/stable/27959629 (cit. on pp. xix, xxix, 64, 106).

[148] Michael J. Schulte and James E. Stine. "Approximating elementary functions with symmetric bipartite tables". In: *IEEE Transactions on Computers* 48.8 (Aug. 1999), pp. 842–847. ISSN: 0018-9340. DOI: 10.1109/12.795125 (cit. on pp. 18, 20).

[149] Michael J. Schulte and Earl E. Swartzlander, Jr. "Exact rounding of certain elementary functions". In: *Proc. of the 11th IEEE Symposium on Computer Arithmetic (ARITH'11)*. 1993, pp. 138–145. DOI: 10.1109/ARITH.1993.378099 (cit. on p. 89).

[150] Larry L. Schumaker and Wolfgang Volk. "Efficient evaluation of multivariate polynomials". In: *Computer Aided Geometric Design* 3.2 (1986), pp. 149–154. ISSN: 0167-8396. DOI: 10.1016/0167-8396(86)90018-X (cit. on p. 21).

[151] Andrew Senkevich. *Libmvec*. Glibc Wiki. 2015. URL: https://sourceware.org/glibc/wiki/libmvec (cit. on p. 24).

[152] Jonathan Richard Shewchuk. "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates". In: *Discrete & Computational Geometry* 18 (1997), pp. 305–363 (cit. on pp. 36, 64).

[153] Naoki Shibata. "Efficient evaluation methods of elementary functions suitable for SIMD computation". In: *Computer Science-Research and Development* 25.1-2 (2010), pp. 25–32. DOI: 10.1007/s00450-010-0108-2 (cit. on p. 25).

[154] Peter Shiu. "The Shapes and Sizes of Pythagorean Triangles". In: *The Mathematical Gazette* 67.439 (1983), pp. 33–38. ISSN: 00255572. URL: http://www.jstor.org/stable/3617358 (cit. on pp. 43 sq.).

[155] Wacław Sierpiński. *Elementary Theory of Numbers: Second English Edition*. Ed. by Andrzej Schinzel. Vol. 31. Elsevier, 1988 (cit. on p. 50).

[156] Wacław Sierpiński. *Pythagorean triangles*. Graduate School of Science, Yeshiva University, 1962 (cit. on p. 42).

[157] Albert M. Sirunyan et al. "Measurements of properties of the Higgs boson decaying into the four-lepton final state in pp collisions $\sqrt{s} = 13$ TeV". In: *Journal of High Energy Physics* 2017.11 (Nov. 9, 2017), p. 47. ISSN: 1029-8479. DOI: 10.1007/JHEP11(2017)047 (cit. on p. 1).

[158] Karl-Georg Steffens. *The history of approximation theory: from Euler to Bernstein*. Springer Science & Business Media, 2007 (cit. on pp. 3, 20).

[159] Damien Stehlé and Paul Zimmermann. "Gal's Accurate Tables Method Revisited". In: *17th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 2005, pp. 257–264. ISBN: 0-7695-2366-8 (cit. on pp. xiv, 19, 38).

[160] Simon Stevin de Bruges and Albert Girard. *L'arithmétique de Simon Stevin de Bruges, revue, corrigée et augmentée de plusieurs traités et annotations, par Albert Girard, Samielois Mathématicien*. Old French. Leyde: Imprimerie des Elzeviers, 1625 (cit. on p. 61).

[161] Alen Stojanov et al. "Abstracting Vector Architectures in Library Generators: Case Study Convolution Filters". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: ACM, 2014, 14:14–14:19. ISBN: 978-1-4503-2937-8. DOI: 10.1145/2627373.2627376 (cit. on pp. xii, 29).

[162] Alen Stojanov et al. "SIMD Intrinsics on Managed Language Runtimes". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: ACM, 2018, pp. 2–15. ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168810 (cit. on pp. xii, 29).

[163] Ping Tak Peter Tang. "Table-Driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic". In: *ACM Transactions on Mathematical Software* 16.4 (Dec. 1990), pp. 378–400 (cit. on pp. 18, 89).

[164] Ping Tak Peter Tang. "Table-Lookup Algorithms for Elementary Functions and Their Error Analysis". In: *10th IEEE Symposium on Computer Arithmetic*. Ed. by Peter Kornerup and David W. Matula. IEEE Computer Society Press, 1991, pp. 232–236 (cit. on pp. xiii, 18 sq., 21, 36).

[165] Laurent Thévenoux. "Code Synthesis to Optimize Accuracy and Speed in Floating-Point Arithmetic". French. PhD thesis. Université de Perpignan Via Domitia, July 2014. URL: https://tel.archives-ouvertes.fr/tel-01143824 (cit. on pp. xii, 29).

[166] David B. Thomas. "A General-Purpose Method for Faithfully Rounded Floating-Point Function Approximation in FPGAs". In: *2015 IEEE 22nd Symposium on Computer Arithmetic*. June 2015, pp. 42–49. DOI: 10.1109/ARITH.2015.27 (cit. on pp. xii, 29 sq.).

[167] Serge Torres. "Tools for the Design of Reliable and Efficient Functions Evaluation Libraries". PhD thesis. Université de Lyon, Sept. 2016. URL: https://tel.archives-ouvertes.fr/tel-01396907 (cit. on p. 18).

[168] *UNIVAC 1107 CENTRAL COMPUTER*. Remington Rand Univac, division of Sperry Rand Corporation. Nov. 1961 (cit. on p. 12).

[169] Dominic Vella and Alfred Vella. "When is n a member of a Pythagorean triple?" In: *The Mathematical Gazette* 87.508 (2003), pp. 102–105. ISSN: 00255572. URL: http://www.jstor.org/stable/3620572 (cit. on p. 50).

[170] Jack E. Volder. "The CORDIC trigonometric computing technique". In: *IRE Transactions on electronic computers* 3 (1959), pp. 330–334 (cit. on p. 21).

[171] Anastasia Volkova. "Towards reliable implementation of digital filters". PhD thesis. Université Pierre et Marie Curie, 2017 (cit. on pp. xi, 29).

[172] Anastasia Volkova, Thibault Hilaire, and Christoph Q. Lauter. "Reliable evaluation of the Worst-Case Peak Gain matrix in multiple precision". In: *2015 IEEE 21nd Symposium on Computer Arithmetic (ARITH)*. June 2015, pp. 96–103. DOI: 10.1109/ARITH.2015.14 (cit. on p. 4).

[173] Anastasia Volkova, Christoph Lauter, and Thibault Hilaire. "Reliable verification of digital implemented filters against frequency specifications". In: *2017 IEEE 24nd Symposium on Computer Arithmetic (ARITH)*. July 2017, pp. 180–187. DOI: 10.1109/ARITH.2017.9 (cit. on p. 4).

[174] Anastasia Volkova et al. "Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study". Dec. 2017. URL: http://hal.upmc.fr/hal-01561052 (cit. on p. 4).

[175] Dong Wang et al. "(M, p, k)-Friendly Points: A Table-Based Method to Evaluate Trigonometric Functions". In: *IEEE Transactions on Circuits and Systems* 61-II.9 (2014), pp. 711–715 (cit. on pp. 18, 20, 39).

[176] Henry S. Warren, Jr. *Hacker's Delight*. Addison-Wesley, 2003 (cit. on p. 96).

[177] Thomas Williams, Colin Kelley, and many others. *gnuplot 4.6 – An Interactive Plotting Program*. 2014. URL: http://gnuplot.info/docs_4.6/gnuplot.pdf (cit. on p. 59).

[178] Weng Fai Wong and Eiichi Goto. "Fast Evaluation of the Elementary Functions in Double Precision". In: *Twenty-Seventh Annual Hawaii International Conference on System Sciences*. 1994, pp. 349–358 (cit. on p. 89).

[179] Weng Fai Wong and Eiichi Goto. "Fast Evaluation of the Elementary Functions in Single Precision". In: *IEEE Transactions on Computers* 44.3 (Mar. 1995), pp. 453–457. DOI: 10.1109/12.372037 (cit. on p. 89).

[180] Weng Fai Wong and Eiichi Goto. "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers". In: *IEEE Transactions on Computers* 43.3 (Mar. 1994), pp. 278–294 (cit. on p. 89).

[181] Abraham Ziv. "Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit". In: *ACM Transactions on Mathematical Software* 17.3 (Sept. 1991), pp. 410–423. ISSN: 0098-3500. DOI: 10.1145/114697.116813 (cit. on pp. 17, 32).

On retrouve des fonctions mathématiques élémentaires dans de nombreux codes de calcul hautes performances. Or, bien que les bibliothèques de fonctions mathématiques (libm) auxquelles font appel ces codes proposent en général plusieurs variétés d'une même fonction, ces dernières sont figées lors de leur implémentation. Cette caractéristique *monolithique* des libms représente donc un frein à la performance des programmes qui les utilisent, car elles sont conçues pour être polyvalentes au détriment d'optimisations spécifiques. De plus, la duplication de modèles partagés dans le code rend la maintenance de ces libms plus difficile et sujette à l'introduction de bugs. Un défi actuel est de proposer des "méta-outils" visant la génération automatique de codes performants pour l'évaluation des fonctions élémentaires. Ces outils doivent permettre la réutilisation d'algorithmes efficaces et génériques pour différentes variétés de fonctions ou architectures matérielles. Il devient alors possible de générer des libms optimisées pour des besoins très spécifiques avec du code générateur factorisé, ce qui facilite sa maintenance. Dans un premier temps, nous proposons un algorithme original permettant de générer des tables de correspondances sans erreur d'arrondi pour les fonctions trigonométriques et hyperboliques. Puis, nous étudions les performances de schémas d'évaluation polynomiale vectorisés, premier pas vers la génération de fonctions vectorisées efficaces. Enfin, nous proposons une méta-implémentation d'un logarithme vectorisé, factorisant la génération de code pour différents formats et architectures. Nous montrons que ces contributions sont compétitives comparées à des solutions libres ou commerciales, justifiant le développement de ce nouveau paradigme.

MOTS-CLÉS Fonctions élémentaires, performance, génération de code, évaluation polynomiale.

Elementary mathematical functions are pervasive in many high performance computing programs. However, although the mathematical libraries (libms), on which these programs rely, generally provide several flavors of the same function, these are fixed at implementation time. Hence this *monolithic* characteristic of libms is an obstacle for the performance of programs relying on them, because they are designed to be versatile at the expense of specific optimizations. Moreover, the duplication of shared patterns in the source code makes maintaining such code bases more error prone and difficult. A current challenge is to propose "meta-tools" targeting automated high performance code generation for the evaluation of elementary functions. These tools must allow reuse of generic and efficient algorithms for different flavours of functions or hardware architectures. Then, it becomes possible to generate optimized tailored libms with factorized generative code, which eases its maintenance. First, we propose an novel algorithm that allows to generate lookup tables that remove rounding errors for trigonometric and hyperbolic functions. The, we study the performance of vectorized polynomial evaluation schemes, a first step towards the generation of efficient vectorized elementary functions. Finally, we develop a meta-implementation of a vectorized logarithm, which factors code generation for different formats and architectures. Our contributions are shown competitive compared to free or commercial solutions, which is a strong incentive to push for developing this new paradigm.

KEYWORDS Elementary functions, performance, code generation, polynomial evaluation.